Diss. ETH No. 12434

A Generalisation Approach to Temporal Data Models and their Implementations

A dissertation submitted to the
SWISS FEDERAL INSTITUTE OF TECHNOLOGY
ZÜRICH

for the degree of
Doctor of Technical Sciences

presented by
ANDREAS STEINER
Dipl. Informatik-Ing. ETH
born June 12, 1964
citizen of Oberiberg (SZ)

accepted on the recommendation of
Prof. Dr. M.C. Norrie, examiner
Prof. Dr. C.A. Zehnder, co-examiner

1998

To my parents Frieda and Ernst Steiner-Bisig!

Abstract

Non-temporal data models and their implementations as database management systems (DBMS) capture a single state of the real world, usually the current one. They support modification operations which facilitate the transition from one consistent database state to another. For queries, they assume that the data is synchronous, meaning that all the facts stored in the database are valid at the time instant a query is evaluated.

There exist many application domains, however, where it is necessary to reconstruct earlier database states or even store future database states (e. g. for planning, budgets) in parallel. The different database states are stored as *temporal data*. Such temporal data arises, for example, in financial and insurance applications, in reservation systems and in medical information management. Of course, it is also possible in practice to store timestamps in classical DBMS and model the temporal aspects mentioned above in this way. However, such an approach does not cater for the special semantics of time.

Thus, there are many proposals for both relational and object-oriented models as to how the non-temporal data models can be enhanced to support the management of temporal data. Their focus is mainly on *extending* the data structures and/or the query language. Hardly any of these temporal data models were implemented, even in the form of prototype systems.

A more systematic way to define temporal data models is based on *generalising* a non-temporal data model into a temporal one. Using generalisation means that *all* constructs of the underlying non-temporal data model – its data structures, operations and integrity constraints – are enhanced to support the management of time-varying data.

To show the power of the generalisation approach, this thesis investigates three approaches to managing temporal data, along with the corresponding prototype implementations. The first approach timestamps data by *extending* the data structures with special timestamp attributes, but, in contrast to existing proposals, uses a *generalised* query, data definition and data manipulation language. The second approach *fully generalises* a non-temporal object data model into a temporal one. The resulting temporal object data model TOM does not extend the data structures, but rather uses the notion of temporal object identifiers to timestamp data. In TOM, not only the user data can be timestamped, but also constructs supported by the data model such as collections of objects, types, integrity constraints and so on, since they are also considered to be objects. This temporal data model was implemented as a single-user prototype system. The third approach demonstrates how the extensible nature of object-oriented DBMS can be used directly to support temporal applications through the use of abstract data types. It is shown that while temporal data structures and operations can be accommodated in this way, support for generalised data models and query languages is restricted.

These approaches show that a *generalised* temporal data model is better suited to the modeling and management of temporal data than an *extended* one, and that generalised data models are implementable. By presenting an evolutionary path from temporal first normal form relations to temporal nested relations, temporal complex objects and temporal object-oriented data models, it is shown that the temporal object data model TOM actually subsumes the extended temporal data models.

Zusammenfassung

Nicht-temporale Datenmodelle und ihre Implementationen als Datenbankverwaltungssysteme (DBMS) speichern einen einzigen Zustand der realen Welt, normalerweise den momentan Gültigen. Sie unterstützen Operationen zur Datenmodifikation, welche einen konsistenzerhaltenden Übergang von einem Datenbankzustand zu einem nächsten ermöglichen. Bei Datenbankanfragen wird angenommen, dass die Daten synchron sind, oder in anderen Worten, dass alle Fakten, die in der Datenbank gespeichert sind, zum Zeitpunkt der Anfrageauswertung gültig sind.

Es gibt jedoch viele Anwendungen, in denen es nötig ist, frühere Datenbankzustände rekonstruieren oder gar zukünftige Datenbankzustände (z. B. für Planungen, Budgets) parallel speichern zu können. Zur Darstellung der verschiedenen, zeitabhängigen Zustände werden sogenannte *temporale Daten* verwendet. Temporale Daten entstehen zum Beispiel in Finanz- und Versicherungsanwendungen, in Reservationssystemen und in der Verwaltung medizinischer Daten. Selbstverständlich lassen sich auch mit klassischen DBMS Zeitangaben speichern und so die genannten temporalen Aspekte darstellen. Diese Lösung wird aber der besonderen Bedeutung des Zeitaspektes nicht wirklich gerecht.

Daher gibt es viele Vorschläge, nicht-temporale relationale und objekt-orientierte Datenmodelle anzupassen, um die Verwaltung von temporalen Daten problemgerecht zu unterstützen. Sie konzentrieren sich hauptsächlich darauf, einzelne Datenstrukturen und/oder die Abfragesprache zu *erweitern*. Sehr wenige dieser temporalen Datenmodelle wurden allerdings implementiert, nicht einmal in der Form von Prototypen.

Ein systematischerer Weg, temporale Datenmodelle problemgerecht zu definieren, basiert auf der *Generalisierung* eines nicht-temporalen in ein temporales Datenmodell. Bei der Generalisierung werden *alle* Konstrukte des zugrundeliegenden nicht-temporalen Datenmodells – seine Datenstrukturen, Operationen und Integritätsbedingungen – angepasst, um die Verwaltung zeitabhängiger Daten zu unterstützen.

Um die Stärken der Generalisierung zu zeigen, untersucht diese Dissertation drei Ansätze zur problemgerechten Verwaltung temporaler Daten bis und mit den entsprechenden Implementationen als Prototypen. Der erste Ansatz versieht Daten mit Zeitstempeln, indem die Datenstrukturen mit speziellen Zeitstempelattributen *erweitert* werden. Es wird jedoch im Gegensatz zu anderen Vorschlägen eine *generalisierte* Abfrage-, Datendefinitions- und Datenmanipulationssprache verwendet. Der zweite Ansatz *generalisiert* ein nicht-temporales Objektdatenmodell *vollständig* in ein temporales. Das resultierende temporale Objektdatenmodell TOM erweitert nicht mehr die Datenstrukturen, sondern verwendet temporale Objektidentifikatoren, um die Daten mit Zeitstempeln zu versehen. In TOM lassen sich nicht nur Benutzerdaten, sondern auch Konstrukte des Datenmodells wie Kollektionen von Objekten, Typen, Integritätsbedingungen etc. mit Zeitstempeln versehen, da diese ebenfalls Objekte sind. Dieses temporale Datenmodell wurde als Einbenutzersystem implementiert. Der dritte Ansatz demonstriert, wie die Erweiterbarkeit von objekt-orientierten DBMS direkt verwendet werden kann, um temporale Anwendungen mit abstrakten Datentypen zu unterstützen. Damit wird gezeigt, dass auf diese Weise zwar temporale Datenstrukturen und Operationen zur Verfügung gestellt werden können, die Unterstützung von generalisierten Datenmodellen und Abfragesprachen aber eingeschränkt ist.

Diese Ansätze zeigen, dass mit einem *generalisierten*, temporalen Datenmodell zeitabhängige Daten besser beschrieben und verwaltet werden können als in erweiterten Modellen, und dass

generalisierte Modelle implementierbar sind. Mit einem Evolutionspfad von temporalen Relationen in erster Normalform über temporale verschachtelte Relationen zu temporalen komplexen Objekten und temporalen objekt-orientierten Datenmodellen wird zusätzlich gezeigt, dass das temporale Objektdatenmodell TOM die erweiterten temporalen Datenmodelle subsumiert.

Acknowledgements

My odyssey through different research groups here at ETH now is finally ending up in this thesis. It was an adventurous journey in all aspects, and having reached its final destination and getting ready for a new one, I know all the experiences made will be invaluable.

I am deeply indebted to Professor Dr. Moira C. Norrie, my supervisor, for giving me the chance to finish this thesis, for her extraordinary support at all times, for broadening my data model horizon and providing the good weather for the last and important part of this journey.

I thank Professor Dr. Carl A. Zehnder for acting as a co-examiner. His valuable comments and the discussions with him helped improving this thesis.

I am grateful to Dr. Robert Marti for giving me the opportunity to start this journey and leading me into a very interesting research area.

Additionally, I would like to thank Roman Gross, Urs Badertscher, Thomas Schumacher, Antonia Erni, Adrian Kobler, Epaminondas Kapetanios, Gabrio Rivera and Alain Würgler, my colleagues here at ETH, for interesting and valuable discussions and for contributing to the comfortable and intellectually stimulating ambience.

Last but not least, I would like to thank all those people behind the scenes, especially Maya Ruckli and Lydia Steiner, for their support and help to keep on going.

Contents

Chapter 1

Introduction

Non-temporal data models and their implementations as database management systems capture a single state of the real world, usually the current one. They support modification operations which facilitate the transition from one database state to another, thereby replacing old values by new ones. For queries, it is assumed that the data is synchronous, meaning that all the facts stored in the database are valid at the time instant a query is evaluated.

There exist many application domains, however, where it is necessary to keep the old database states or even store future database states. While it is not a major problem to find ways to add validity time periods to data to keep the history of it, it is a difficult task to query time-varying data and specify integrity constraints which hold over several database states if using a non-temporal data model and query language.

Research in the area of temporal databases has come up with quite a diversity of temporal data models addressing these and other issues. Most of the work has been done with respect to query languages and their extension to support time-varying data. One question often discussed is which additional operations need to be supported in order to be able to query time-varying data. Another one is how the semantics of already supported operations should be changed in order to use them on time-varying data. While most proposals focus on these issues, we believe that, in order to find a general and orthogonal solution, all parts of a data model need to be considered – the data structures, the operations and the constraints.

This chapter first gives a short introduction to non-temporal data models and their implementations as database management systems. Then, the questions why *temporal* databases are needed to handle time-varying data and what approaches are available to provide such support are discussed. The contributions and the structure of this thesis conclude this chapter.

1.1 Non-Temporal Data Models and Database Management Systems

This section gives a short introduction to non-temporal database management systems and the non-temporal data models whose temporal counterparts will be discussed in this thesis. It also shows how these models are used and why they are necessary when large amounts of data have to be managed.

1.1.1 Database Management Systems

In the early 60's, data was stored in files and organised with respect to the application that used the data. By using file systems, data redundancy was introduced. Additionally, data was neither independent from the devices it was stored on nor from changes in applications. In the early 70's, systems were developed to overcome the drawbacks of approaches based on file systems. The

trend was to get away from application specific software solutions to a general framework, allowing different application domains to be implemented using a general-purpose software system. This software system is called a *database management system (DBMS)*. One of the main ideas was to introduce *data independence*, preventing changes to the physical storage management from affecting the application programs and vice versa.

A DBMS is a general-purpose software system which helps users to store, access and maintain *data* in a uniform way. Data can be anything from simple values such as integers and character strings to composite values to more complex data such as texts, images, sounds and videos.

A DBMS is based on a *data model*, which defines the constructs and formalisms available to define, modify and access the data. The data model provides users with data structures which hide many of the details of how the data is stored physically. It uses logical concepts such as objects, their properties and interrelationships, which are easier for most users to understand than the computer storage concepts [EN94]. It also supplies operations to access and modify data, and integrity constraints to express what is considered to be a *consistent* database state. Data defined using a data model is treated in a uniform way and stored in a *database*. This has the advantage that *application programs* such as payroll, accounting and inventory applications, can access the same data through a *common interface*, namely the DBMS, as depicted in figure 1.1. This prevents data redundancy and allows simpler application development and maintenance.

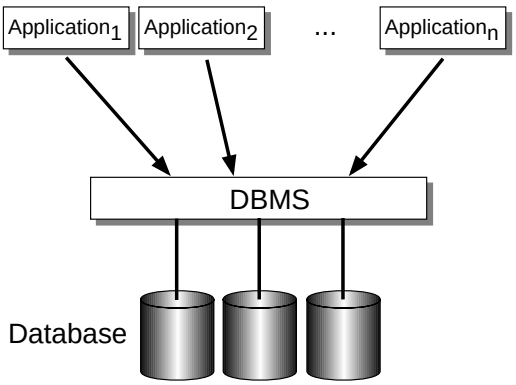

Figure 1.1: Application programs accessing a database through a DBMS

Modern DBMS have to support a variety of tasks and requirements [EN94, HS95]. A DBMS manages *persistent* data, meaning that data exists longer than tasks run by application programs. *(Declarative) languages* and *operations* need to be supported which allow the definition, modification and retrieval of data. A DBMS should manage huge amounts of data *efficiently*. It should provide, for example, means and techniques to optimise data retrieval. It should allow multiple users to access the same data simultaneously, avoiding any problems which might evolve. An important concept to achieve this are *transactions* which are sequences of database modifications and queries treated as single atomic operations. Additionally, a modern DBMS should support concepts to provide *data consistency*, *recovery* from system crashes and *database security* and *authorisation*. Commercial DBMS support most of these tasks and requirements in one way or another.

1.1.2 The Relational Data Model and Relational DBMS

The relational data model was introduced in 1970 by [Cod70]. Nowadays, it is the most widespread data model used for database applications, and several commercial DBMS implementing the relational data model are available.

A relational database contains a set of *relations*. A relation, sometimes also called a *table*, has a name and a well-defined structure, called a *relation schema*. The set of all relation schemas of a database is called the *database schema*. A relation schema consists of a set of *attributes* which represent the common properties of a real world object called an *entity*. Each attribute has

associated a *name* which is unique within the relation schema and a scalar *domain* such as integer, real, string or date. A relation has to be in *first normal form (1NF)*, meaning that the domains of the attributes in its schema may only be of scalar data types. In its simplest form, a relation can be considered as a subset of the Cartesian product of all the attribute domains contained in its schema. A single element of a relation is called a *tuple*. A relation is a *set* of tuples. The only data structure supported by the relational data model is the relation.

Example 1.1 *The following table* `Employees` *contains data about employees of a university*

EmpID	Name	Salary	Dept
100	Moira	20000	IS
101	Adrian	9000	IS
102	Alain	9000	IS
103	Antonia	11000	IS
104	Gabrio	10000	IS
105	Andreas	9000	IS

and corresponds to the relation schema

```
R = (EmpID  : INTEGER PRIMARY KEY,
     Name   : CHAR(20),
     Salary : INTEGER,
     Dept   : CHAR(5))
```

Each tuple describes the properties of an employee. For example, the tuple `(100, Moira, 20000, IS)` *contains data about employee Moira, including her employment number, salary and the department she works in.*

The relational data model also defines operations which work on relations. A basic set of six generic operations is sufficient to express all relational algebra operations. These basic operations are the set union ($\cup$), set difference ($\Leftrightarrow$) and cross product ($\times$) of two relations, and selection (σ), projection (π) and rename (β) operations on a single relation. The result of each of these operations is another relation. The set union operation adds all tuples of both relations involved into a single relation. The set difference returns all tuples of the first relation which do not occur in the second. The cross product combines all tuples of the first with all tuples of the second relation. The schema of the resulting relation contains all attributes of the first and the second relation. Selection returns a subset of rows of a relation, and projection a subset of columns. The rename operation allows the renaming of attributes in a relation. Other operations can be defined in terms of these six operations. For example, a natural join ($\bowtie$) can be expressed using a combination of a cross product, a selection, a projection and a rename operation.

Example 1.2 *We would like to find the name and employment number of those employees earning more than 10000. Using the relational algebra, this can be expressed using a selection and a projection operation:*

$$\pi[\texttt{EmpID}, \texttt{Name}](\sigma[\texttt{Salary} > 10000](\texttt{Employees}))$$

The selection operation first selects those tuples in relation `Employees` *whose value in attribute* `Salary` *is more than 10000. The resulting relation is projected to a relation containing only the columns* `EmpID` *and* `Name`*:*

EmpID	Name
100	Moira
103	Antonia

Most of the relational DBMS support the standard language SQL [Dat89, DD93, MS93]. SQL can be divided into three parts: the *query language*, the *data modification language* and the *data definition language*. The query language allows the retrieving of data. It is based on the relational algebra, but additionally supports features such as aggregation functions, grouping and ordering of tuples. The data modification language is used to insert, update and delete data. Base tables, indexes and views are created with the data definition language. A base table is a relation that is physically stored in the database. Indexes are access paths to base tables which, for example, allow a faster lookup of specific data in the table. In contrast to a base table, a view is a virtual table. It is not physically stored in the database, but looks to the user as if it were [Dat89]. The data contained in it is specified by a query.

SQL is a declarative and set-oriented language. SQL queries specify what data is desired, but not how this data is retrieved. The query language compiler translates an SQL statement into low-level operations, accessing the data on the storage-management level. *Optimisation* is performed to select the most efficient execution of a query. The user specifies the desired result in the query language, and the DBMS is free to select any execution plan that achieves this result. Query optimisation is an important aspect of providing an efficient DBMS.

Example 1.3 *The algebra expression given in example 1.2 written using SQL is as follows:*

```
SELECT EmpID, Name
FROM   Employees
WHERE  Salary > 10000
```

A relational DBMS supports several integrity constraints, for example, *primary keys*, *foreign keys* and *referential integrity*. A primary key is an attribute (or a set of attributes) of a relation whose value uniquely identifies a tuple in a relation. In the relation schema `R` given in example 1.1, attribute `EmpID` is denoted to be the primary key. A foreign key is an attribute (or a set of attributes) of one table that contains only values of the primary key of another (referenced) table.

The referential integrity constraint for a relation demands that each of its foreign key values must exist also as a primary key value in the referenced table. In the relational data model, *relationships* among data are expressed using foreign keys and verified using referential integrity constraints.

A major drawback of the relational data model is the requirement that relations have to be in 1NF. This restricts the expressive power of the data structures. For example, hierarchical data structures need to be flattened to be implementable in the relational model. To overcome this drawback, the relational model has been extended to support, for example, non first normal form (NFNF) or nested relations [SS91], or other complex attribute values.

1.1.3 Object Data Models and Object-Oriented DBMS

While the relational data model and its implementations nowadays are well-established, its model-inherent restrictions – for example, the single data structure supported, the 1NF restriction on relations and the impossibility to extend the functionality – lead to the development of new data models. On one hand, the relational model was extended to overcome these deficiencies, while on the other hand, the new paradigm of object data models evolved. Object data models overcome the restrictions both on data structures and functionality. They provide for storing not only complex values, but also behaviour in a database, thereby allowing applications to share code as well as data.

Systems based on object-oriented data models originated with the object-oriented programming paradigm. During the last decades, a variety of object data models have been proposed. An overview is given, for example, in [Cat94].

The relational model is value-based. An entity is identified via a primary key. Object-oriented data models are identity-based. An object can be referenced via a unique system-generated number

which is independent of the value of its primary key (if there is any) [Cat94]. This number is called the *object identifier (OID)*. It differs from a key by the fact that a key may change, and the tuple gets a new identity, although it still represents the same real-world object. Additionally, an OID has no semantics and usually is not visible to the user. A key is either generated by a program written by an application programmer, or human-meaningful names are used. The same entity may have different keys in different relations.

In a relational DBMS, information about a complex object is often scattered over many relations, which leads to a loss of direct correspondence between a real world object and its database representation. Object-oriented data models allow the storage of all information concerning an entity as a *single* database object. As depicted in figure 1.2, an *object* consists of an *object identifier*, its *state* and *behaviour*. All three components are stored in the database.

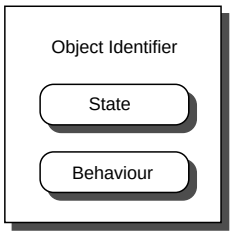

Figure 1.2: Components of an Object

The *state of an object* is specified by a data structure of arbitrary complexity. State values of objects are constructed from a combination of simple attributes, reference attributes and complex attributes. Simple attributes contain values of some atomic data type, for example, integers, reals or strings. Reference attributes store the OID of another object. An object thus can comprise other objects. Complex attributes store, for example, sets, bags or lists of values. The *behaviour of an object* is specified by operations. These operations are also called *methods*. The code of these methods is stored in the database as well.

An *object* actually is an *instance* of a type which defines both the state and behaviour of the object. In this thesis, the term *class* denotes the set of all objects of the same type, or in other words, a class shall be the extent of a type. So all objects in a class share a common definition but differ in the values assigned to their attributes. Types may be reused in the sense that new types can be derived from already defined types. This leads to a *type hierarchy*, where the dependencies – which type is derived from which – are depicted. A more specialised type inherits all attributes and methods of the type it is derived from, maybe overwriting some of the definitions.

Associations between objects can be expressed in two different ways. First, a reference to another object may be implemented as a reference attribute. The OID of the referenced object is stored *implicitly* within the state of the object. Second, associations may be regarded as a concept of the data model used. This means that the model allows the specification of associations between objects *explicitly*. The former approach leaves the burden of keeping the associations consistent up to the application programmer, whereas the latter approach leads to a semantically richer data model and DBMS, supporting means and ways to create these associations, to specify integrity constraints over them, to check their consistency and execute operations over them.

1.1.3.1 The Object Data Model OM

There is no single object data model as there is a relational data model. Several different proposals for object data models exist, differing, for example, in their semantic richness. Some models do not support constructs to express constraints over objects or associations between them. They have to be implemented by the application programmer using methods. In this thesis, the OM model [Nor92, Nor93] is used to investigate temporal object-oriented databases. There are several reasons for this.

First, the OM model is *generic*. It strictly separates *typing* from *classification* such that *classification structures* model the *roles* of objects rather than their representation. The model is therefore independent of any particular type system and programming language environment. This supports our goal to specify a *generic* temporal data model. Second, the model is *orthogonal* with respect to how the constructs of the model may be applied. As an example of its orthogonality, collections are themselves objects and this enables arbitrary nesting of structures. Third, the model supports role modeling, allowing an object to be member in several collections (or roles) simultaneously. Fourth, the OM model is a *semantically rich* object data model. For example, it has explicit support for integrity constraints and associations. Fifth, an algebra and query language was defined. Last but not least, there exists a DBMS for this model.

As seen before, *types* describe the structure of objects, and instances of the same type are collected in classes. Types may evolve over time, which is denoted as *type evolution*. Types can be reused and refined which leads to type hierarchies describing the dependencies between supertypes and subtypes. The OM model allows the same type definition to be used for several different collections. This contrasts with the use of schemas in relational DBMS, where a schema is directly attached to a relation. To create two relations with the same schema using SQL means that the schemas have to be declared twice.

In the OM model, *collections* are used to group objects *semantically*. An entity may play several roles during its existence. For example, a person studies computer science, then gets employed in a company and later is promoted to a project manager. So, this person plays the roles of a student, an employee and a manager. The same person, however, may also be a tennis player simultaneously. This means that the same entity may even play several roles at the same time. Collections in the OM model group objects with respect to the role they play. An object may be a member of several collections at the same time. Each collection has a *member type*, which denotes the type the objects contained in it must have. Since an object can be member of several collections at the same time, it can be dressed with several different types simultaneously.

Classifications are represented by the bulk type constructor *collection*. *Classification structures* are built from collections linked by means of *subcollection*, *disjoint*, *cover* and *intersection* constraints over these collections. Explicit support for the representation of relationships between objects is given by association constructs based on a special form of collection referred to as a *binary collection*. As a result, classification structures may be used to model relationship roles as well as entity roles.

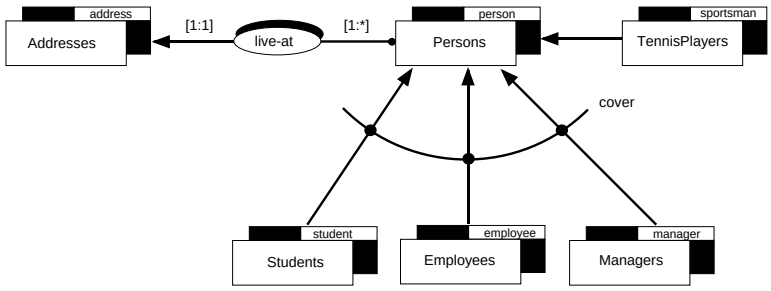

Figure 1.3: Mini-World modeled in OM

Figure 1.3 depicts a mini-world modeled using the OM model. Shaded boxes are used to denote collections with the name of the collection in the unshaded region and the type of the member values in the shaded region.

A *classification structure* relates different collections with each other using a *subcollection constraint*. For example, all students, employees, managers and tennis players are persons. This means, that collections `Students`, `Employees`, `Managers` and `TennisPlayers` are subcollections of collection `Persons`. As time goes by, objects change their membership in collections. As mentioned before, a student eventually becomes an employee. The corresponding student object thus

migrates from collection `Students` to collection `Employees`, changing its type due to their different member types. This is denoted as *object evolution*.

The OM model supports *disjoint*, *cover* and *intersection constraints* over collections. These constraints restrict the form of relationship between supercollections and their subcollections. The disjoint constraint demands that a set of subcollections do not share a single object of their common supercollection, whereas the cover constraint demands that each object of the supercollection appears in at least one of the subcollections involved in the constraint. The intersection constraint relates a subcollection with its supercollections such that all the common objects of the supercollections are also members of the subcollection. In figure 1.3, a cover constraint demands that collections `Students`, `Employees` and `Managers` cover collection `Persons`.

Relationships between objects can be expressed explicitly in the OM model using *associations*. An association has a source and a target collection, relating objects of the source collection with objects of the target collection. In an association, cardinality constraints specify how often an object may be related with other objects. For example, persons have addresses where they live. For each person, at least one address shall be stored in the database. This means that an object in collection `Persons` must be related at least once with an object in collection `Addresses`, denoted using an interval $[1 : *]$ where the asterisk expresses that there is no upper limit. On the other hand, it is demanded that each address must be related to exactly one person, denoted as $[1 : 1]$. An association `live-at` with corresponding cardinality constraints relates the source collection `Persons` with the target collection `Addresses`.

The OM model also defines a set of generic operations over unary and binary collections, called the collection algebra. Additionally, it supports object and relationship evolution [NSWW96] and defines restrictions over how they can evolve.

1.2 Why Temporal Databases ?

As mentioned earlier, non-temporal data models and DBMS only support functionality to access a single state of the real world, usually the most recent one, and to change from one database state to another thereby giving up the old state. There exist, however, many application domains which need to have access not only to the most recent state, but also to past and even future states, and the notion of data consistency must be extended to cover all of these database states. [Sno95b] lists sixteen different application domains requiring the storage and retrieval of time-varying data. Examples of such application domains are financial applications (e. g. the history of stock market data), insurance applications (e. g. when were which policies in effect), reservation systems (e. g. when is which room in a hotel booked), medical information management systems (e. g. patient records) and decision support systems (e. g. planning for future contingencies).

[Sno95b] mentions that in fact, it is difficult to identify applications that do *not* involve the management of time-varying data. One reason why the management of time-varying data is not considered for most applications is the lack of appropriate support by commercial DBMS. Nowadays, if time-varying data has to be managed, special application programs must be developed.

1.3 Handling Temporal Data

The first step when building a database application is analysing the part of the real world which shall be modeled and captured by the database. On one hand, the real world objects, their relationships among each other, together with restrictions on these objects and relationships, have to be determined. On the other hand, the required operations performed on these objects and relationships need to be specified. This leads to *database* and *functional* requirements [EN94].

The *database requirements* lead to the specification of data types, associations and integrity constraints. The *functional requirements* consist of the user-defined transactions that will be applied to the database, including both retrieval and modification of data. In the case that the DBMS can be extended with functionality, data types (schemas) are defined according to both *functional*

and *database* requirements. Otherwise, only the database requirements influence the definition of data types, and the functional requirements are built into application programs. This step from a mini-world to the different requirements is depicted in figure 1.4.

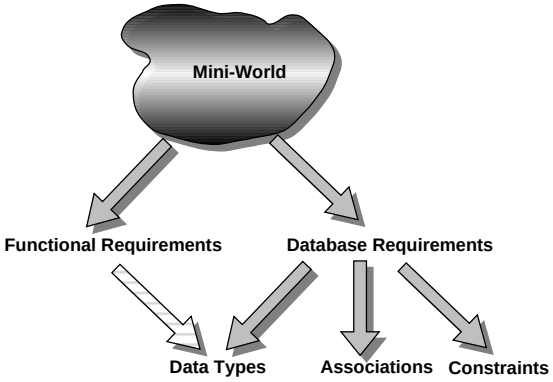

Figure 1.4: Database Design

Figure 1.4 shows that the specifications for a database application do not only consist of data types. Operations, associations and integrity constraints also play important roles when building a database application. Of course, this also holds for application domains using time-varying data. Thus, adding time attributes to data types in order to record validity time periods of data is not sufficient for *temporal database applications*. Means and ways are needed to specify operations to retrieve and modify temporal data and express integrity constraints which hold over several database states.

1.3.1 How are Data Models affected when Time is added?

As mentioned before, a DBMS is based on a data model which defines constructs and formalisms with which all of the data can be described, modified and accessed in a uniform way. A data model supplies a set of concepts to describe the data structures, integrity constraints and retrieval and update operations. A data model $\mathtt{M} = (\mathtt{DS}, \mathtt{OP}, \mathtt{C})$ thus consists of three components – the data structures DS, operations OP and integrity constraints C.

A *temporal* data model $\mathtt{M}^T = (\mathtt{DS}^T, \mathtt{OP}^T, \mathtt{C}^T)$ should enhance all the concepts contained in the three components of a data model with respect to time. Data structures should be adapted such that they can store time-varying data, leaving it up to the user to choose the level of what data units shall be timestamped. Algebra *and* modification operations should be redefined using temporal semantics, for example, using the notion of *snapshot reducibility* [Sno87]. Snapshot reducibility defines the temporal semantics of an operation using the semantics of its non-temporal counterpart applied at each time point.

Additionally, for each expressible constraint in the non-temporal data model M, the temporal data model $\mathtt{M}^T$ should provide a temporal version. The semantics of temporal constraints may also be defined using the notion of snapshot reducibility.

1.3.2 How can Support for Temporal Databases be implemented?

Modifying or extending a data model, for example, to support time-varying data, means that the implementation of the model – the DBMS – must be changed as well. Since most DBMS can be considered as black boxes, the corresponding change of the software has to be done by the DBMS supplier. Other approaches to achieve some support to handle time-varying data are building the temporal functionality into the database applications or using the extensibility inherent to some DBMS to support special temporal data structures and behaviour. In fact, four different ways to implement temporal database applications can be identified:

1. Use type `date` supplied in a non-temporal DBMS and build temporal support into applications.

2. Implement an abstract data type (ADT) for time.

3. Extend a non-temporal data model to a temporal data model.

4. Generalise a non-temporal data model to a temporal data model.

The first two approaches do not involve any changes to the DBMS. The system is taken as-is, and all implementations need to be done by the database designers and/or application programmers. The last two approaches can only be achieved by changing the DBMS itself.

The four approaches of course differ in what kind of support for time-varying data can be achieved and how difficult this is. At present, only the first two approaches can be used in practice. In this thesis, approaches 2, 3 and 4 are investigated and the corresponding prototype systems presented. This way, the advantages and disadvantages of each approach are identified.

Unfortunately, there exist no temporal DBMS and hardly any prototype systems. Many temporal data models have been proposed, but almost none of them have actually been implemented. We believe that such prototype systems are necessary for the verification of a specific approach and the investigation of whether or not the approach is implementable. They are useful to show the power (or weakness) of a model. [Böh95] provides a list of thirteen prototypes dealing with time-varying data in one way or another. They state that the lack of such prototypes is one reason why current commercial DBMS provide only limited temporal functionality.

Now, the four approaches introduced above are discussed in more detail. First, the DBMS used is assumed to be a black box which cannot be changed. Then, the question how DBMS software itself may be enhanced to support temporal database applications is discussed.

1.3.2.1 Temporal DBMS without Changes to existing Database Technology

Use Type `date` in a Non-Temporal DBMS For this approach, the only assumption is a data type `date`. Type `date` can either be supported directly by the DBMS or dates could be mapped, for example, to another system supported data type. With a type `date`, user-defined time attributes such as birthdates can be specified. Additionally, it is possible to timestamp data by adding special time attributes to schemas. A time interval, for example, can be mapped to two attributes of type `date`, denoting the lower and the upper bound of the time interval. Any temporal semantics, however, have to be built into the application program itself. Temporal operations and temporal integrity constraints have to be provided by the application programmer.

This means that the burden is on the database application programmer as he has no specific support whatsoever from the DBMS. This approach is error prone, and a lot of time needs to be invested into the additional coding of the temporal framework used for the applications. This part of the code is usually reinvented by each company which deals with time-varying data. There is no standardised way of implementing temporal database applications, which eventually will lead to more problems, for example, when parts of the applications need to be modified or are replaced. The lack of direct support such as a temporal query language, leads to complex applications which will be difficult to maintain. Since the temporal semantics are completely unknown to the underlying DBMS, we argue that this approach also leads to inefficient application programs.

In our opinion, the use of a commercial non-temporal DBMS and a type `date` might be one of the solutions for the moment, however this approach to handling temporal data is not adequate in the long term.

Implement an Abstract Data Type (ADT) for Time For this approach, it is assumed that the DBMS used supports facilities to implement abstract data types. Object-relational and object-oriented DBMS allow users not only to specify data structures, but also to extend the functionality of the system and to store it in the database. This is not possible using a pure relational DBMS.

We can implement an ADT for time, including, for example, data structures for sets of time intervals and operations to calculate the union, intersection and difference of sets of time intervals as proposed in [SN97b]. Similar approaches using an ADT for time are, for example, described in [GÖ93, DW92, WD93]. This ADT then can be used to build temporal semantics into application programs. To relieve the database users from having to implement this ADT themselves and to guarantee a common basis for different applications, a library would be helpful.

The greatest disadvantage of an ADT for time is the fact that the DBMS cannot make use of the special semantics time has, for example, to optimise the retrieval of temporal data. Additionally, neither writing temporal queries or updates using the ADT nor migrating legacy code is straightforward.

Extending the functionality of a DBMS using an ADT makes sense where the additional functionality only concerns part of the data or is very specific. However, time-varying data and temporal operations usually are neither restricted to a small part of data nor very specific. In fact, the class of application domains dealing with time-varying data is huge, and usually, not only a small part of the data is time-varying. Thus it is necessary to build temporal support directly into the data model and the corresponding DBMS.

1.3.2.2 Temporal DBMS with Changes to existing Database Technology

Extend a Non-Temporal Data Model Extending a non-temporal data model to a temporal data model in our terms means that the concepts already supported by the non-temporal models are used to specify the temporal extension. New concepts are introduced where it is awkward or impossible to express them in the non-temporal model.

For example, timestamping data is usually achieved by extending the non-temporal schemas or types with special time attributes. Query languages and algebras are extended with additional operations to express a temporal join or temporal selection operations. More expressive temporal data models define a temporal algebra which refer to the special time attributes for time calculations. This approach of extending schemas and query languages has been chosen for the relational data model (e. g. [Sar93, NA93, Sno93, Sno95b]) as well as for semantic data models (e. g. [EW90, EWK93a]) and object data models (e. g. [RS91, KS92b, BFG96]).

In fact, this schema extension approach is the most widely used approach to define temporal data models. The advantage is that only parts of the model must be changed, for example, the query language and integrity constraints. With respect to an existing DBMS, this would mean that only part of it would have to be changed. Access methods and storage structures, for example, are not affected.

This advantage, however, also directly leads to the disadvantages of this approach. By reusing the existing concepts of the non-temporal data model, the temporal data model automatically inherits the restrictions of the non-temporal parent and additionally, due to the reuse of existing concepts, the temporal concepts turn out to be not as general and orthogonal as desired. By adding additional time attributes to relations, only specific data structures – for example tuples – can be timestamped. Since collections of entities, types or integrity constraints are created and eventually dropped, it can also make sense to keep track of the history of each of these constructs.

Proposed *extended* temporal data models neglect issues like these. They usually concentrate on special features such as temporal data structures, query language design, temporal algebra or temporal integrity constraints.

Since existing DBMS have to be adapted anyway if the underlying data model is modified, we argue that this approach only goes half of the way.

Generalise a Non-Temporal Data Model A more promising way is to generalise the whole data model to support temporal data. All three components – data structures, operations and integrity constraints – have to be generalised. With respect to data structures, this means that the type or schema of objects is not simply extended, but a new, simple and orthogonal concept needs to be found which does not rely on any type specific assumptions. Simple means that it should

be easily implementable, but expressive enough to timestamp different units of data. Orthogonal in this context means that this concept is not restricted to specific constructs of the data model, for example, to tuples or attributes. It should be left to the user to decide which granularity of data, and even which constructs of the model (for example, types, collections, constraints or even databases) shall be timestamped. Temporal operations (including updates) and integrity constraints shall then refer to this new concept. This approach is more promising to find the key concepts needed to support the handling of temporal data.

With respect to generalising temporal algebra operations, a step has been made by defining *temporal completeness* [BM94, BJS95] which introduced useful requirements with respect to temporal data and temporal queries. This proposal lacks, however, to define requirements for constraints and focuses on timestamping data in 1NF relations.

1.4 Contribution

The main contribution of this thesis is the introduction and discussion of the generalisation approach for temporal data models. Generalisation can be used as a guideline when developing a temporal data model which is based on an existing non-temporal data model. As mentioned earlier, the generalisation approach considers all aspects of the non-temporal data model when turning it into a temporal one. The result is a more general and orthogonal temporal data model than those based on the extension approach.

The results presented in this thesis cover research in the area of both temporal relational and temporal object-oriented data models. In this context, three different projects were initiated and successfully completed by the author of this thesis: The implementation of TimeDB, the definition of the temporal object data model TOM and different implementations of TOM. Additionally, the various proposals of temporal data models introduced or defined in this thesis are compared with each other.

Implementing TimeDB

The *first project* was the implementation of the *bitemporal relational DBMS TimeDB*. TimeDB is based on the generalisation approach, however only to a limited extent. Temporal relations, for example, are extensions of the non-temporal relations with timestamp attributes. TimeDB supports the temporal query language ATSQL2 [SBJS96b, SBJS96a]. ATSQL2 is an extension to SQL to support the management of time-varying data. It was designed by an international group of researchers (Michael Böhlen, Aalborg University; Christian Jensen, Aalborg University; Richard Snodgrass, University of Arizona; Andreas Steiner, ETH Zürich). TimeDB was implemented during the design process of ATSQL2. It influenced the design of ATSQL2, for example, the orthogonal treatment of valid time and transaction time, and helped to identify and eliminate weaknesses of the language. TimeDB is built as a front-end to the commercial DBMS Oracle. With this layered approach, important concepts of the underlying commercial DBMS such as transactions, persistence, recovery, security and authorisation could be used. Implementing them from scratch would have been too time-consuming. TimeDB is the only temporal relational DBMS providing an extended SQL which supports temporal queries, a temporal data definition and data modification language and temporal integrity constraints. It is used at universities for demonstration purposes and is currently extended in another research group with respect to physical storage management of temporal data.

The Temporal Object Data Model TOM

The *second project* was to define and apply the generalisation approach with respect to object-oriented data models. This project was based on the non-temporal object data model OM due to its genericity, orthogonality, semantic richness, support for role modeling and the existence of a corresponding DBMS. Applying the generalisation approach to this data model has led to

the generic, general and orthogonal *temporal object data model TOM*. TOM is *generic* in the sense that it is independent of any specific type system. *General* means that data structures, operations and constraints of the non-temporal data model OM are generalised into *temporal* data structures, *temporal* operations and *temporal* constraints, using the notion of *snapshot reducibility* to define their semantics. The model is *orthogonal* in two senses. First, anything which is an object (entity, collection, constraint, type or even database) may be timestamped. Second, valid time and transaction time are treated as orthogonal time lines, having the *same set of operations* defined on them. The constructs and formalisms to achieve such a temporal data model are described in this thesis.

Implementing TOM

The *third project* was to implement the temporal object data model TOM. Two different approaches were investigated. First, the temporal data model TOM was implemented as a single-user DBMS called *TOMS* using Prolog. As will be shown, the intrinsic generality of TOM and its orthogonality are major contributing factors allowing a simple implementation. Second, the data model TOM is implemented as an ADT for the commercial object-oriented DBMS O_2. This thesis sketches both approaches and compares them with each other.

Comparison

This thesis also describes an evolutionary path from temporal relational to temporal object data models which once more shows the generality of the temporal object data model TOM. This evolutionary path allows the comparison of the different temporal data models based on the extension approach with the generalised data model TOM and the identification of the drawbacks of the extension approach. Additionally, ideas are presented how the general concepts found in TOM could be achieved in temporal data models which are based on the extension approach.

The generalised temporal object data model TOM presented in this thesis also provides insight into how a more general form of an extensible DBMS – based on the ideas of having *extensible* object identifiers, algebra operations and constraints – could be provided which also supports other specialised applications managing, for example, spatial data or versions of objects.

1.5 Structure of the Thesis

Chapter 2 presents the different concepts and notions proposed in various temporal data models which either help to distinguish temporal data models or are essential to all temporal data models.

Chapter 3 describes different extensions of relational, extended relational, semantical, functional and object data models for the management of temporal data. It also shows that proposed temporal data models are based on *extending* the underlying data model on the type level.

Chapter 4 describes the prototype temporal relational DBMS TimeDB which implements ATSQL2. ATSQL2 extends the query language SQL with temporal features such as a temporal query language, a temporal data definition and modification language, temporal views and temporal constraints. The implementation of TimeDB as a front-end to the commercial DBMS Oracle is presented.

Chapter 5 introduces the temporal object data model TOM which is a *temporal generalisation* of the OM model. The TOM data model defines temporal objects, temporal collections, temporal associations, a temporal collection algebra and temporal constraints. Due to the timestamping approach used in TOM, anything considered to be an object may be timestamped. This leads to the insight that also meta-data can and should be timestamped, a feature which automatically evolved in the TOM data model.

Chapter 6 describes how the temporal object data model TOM is implemented. Two different approaches are shown. The first approach is a direct implementation of the TOM data model. The

second one is based on building an ADT for time using the commercial object-oriented DBMS O_2. Additionally, the drawbacks of the ADT approach are discussed.

Chapter 7 finally compares the different temporal data models introduced and discussed in this thesis. It shows that the temporal data models based on the extension approach can be viewed as special cases of the temporal object data model TOM. Additionally, the issue of timestamping meta-data is considered in more detail and motivated, and a solution for temporal data models based on the extension approach is presented.

Chapter 2

Key Concepts used in Temporal Data Models

This chapter introduces the key concepts used in the various proposed temporal data models. As seen in the previous chapter, a data model `M` consists of data structures `DS`, operations `OP` on these data structures, and constraints `C`, namely `M` = (`DS`, `OP`, `C`). A *temporal* data model thus should support temporal data structures which allow the storage of temporal data, temporal operations which access or modify instances of these temporal data structures and are able to use their special semantics, and temporal constraints.

The first question discussed is how time itself can be modeled in a temporal data model or DBMS. Having decided on how to represent time in a data model, the question arises whether there exist different time dimensions with respect to which data shall be stored. For example, the time point when data is updated in a database does not necessarily need to be equal to the time point when the part of the real world represented by this data has changed. This distinction of time dimensions leads to several kinds of temporal databases. Then, the decision has to be made on which level data shall be timestamped. With respect to temporal operations, this chapter introduces the notions of coalescing and snapshot reducibility.

2.1 Modeling Time

The temporal database community has used mainly three basic models for time. They see time either as continuous, dense or discrete. Another question is how time points and time intervals are represented, and which operations and comparison predicates are defined on them. This section explains the various notions of time and how they affect the specifications of temporal data models.

2.1.1 Different Models of Time

What is time? Saint Augustine said he knew precisely what time is, provided no one asked him to explain it. While we do not want to go into this philosophical discussion, we still need to think about how we measure and model time in temporal data models.

Time seems to be continuous by nature. Scientists assume that our universe started with the *Big Bang*, and they assume this to be the start of time. They are not sure, though, whether or not our universe (and with it time) will have an end [Haw88].

For daily life, ways to quantify time and to specify certain points in time with different granularities such as years, months, days, hours, minutes and so on were introduced. Calendars are used to structure time.

Modeling time in a DBMS demands the exact specification of what time is assumed to be. This does not necessarily mean that all real world aspects of time have to be captured. A more

promising way is to model time such that it can easily be integrated into the data model and that it supports most aspects of time needed in different application domains.

The temporal database community has come up with three basic models for time: the *continuous model*, the *dense model* and the *discrete model*. These models have been exhaustively discussed in the literature, for example, in [GV85, GY91, Sno95b]. Time is usually mapped to a *set of numbers* which are totally ordered with respect to the comparison predicate $<$. Thus time can be represented as an axis in a coordinate system.

The *continuous model* views time as being isomorphic to the real numbers. Each real number corresponds to a point in time. So between any two time points on the time axis, there exists another time point. This approach models time most accurately. Since temporal data models are used on digital machines, it is not possible to map this model of time losslessly to computers. The *dense model* views time as being isomorphic to the rational numbers. The *discrete model* maps time to integers [CT85]. A successor function can be defined in the discrete model of time. Between a time point and its successor, no other time points exist.

[Sno95b] views the choice of these three alternatives as largely unimportant. They argue that for a time model being implemented on a discrete computing device, the view of time must be necessarily discrete. We follow this view. While discussion of these alternatives might be interesting on a philosophical level, we feel it not to be of any practical importance for the work presented here.

2.1.2 Chronons

The previous section has shown that time can be mapped to an underlying time axis. The time axis can be viewed as a *set of time points*. The smallest non-decomposable time unit on the time axis is called a *chronon* [TCG+93]. So a particular chronon is a subinterval of fixed duration on the time axis. As stated above, a smallest non-decomposable time unit is needed, since digital computers only support a limited granularity for real numbers. For the continuous and dense model of time, a chronon covers a small part on a time axis, whereas a time point is a single point on the time axis. For the discrete model, chronons and time points can be assumed to be the same. They can be mapped to integer numbers.

2.1.3 Time Instants and Events

A *time instant* is in fact a time point on an underlying time axis, whereas an event is an instantaneous fact, for example, something occurring at a time instant [TCG+93, Jea93]. As seen above, a coarser granularity of time units is used in the models of time, called chronons. Modeling time instants or events thus means that they have to be mapped to chronons. An event thus is said to occur at a chronon `t` if it occurs at any time instant during the chronon represented by `t`.

2.1.4 Time Intervals and Temporal Elements

A temporal database stores facts and, for example, the time when they are true in the real world. Thus, from a theoretical viewpoint, all the time instants a fact is true have to be stored in the temporal database. In the case that this set of time instants contains only continuous instants, it can be written as a time interval. So time intervals can be used to model the time period during which a fact was true in the real world.

A time interval represents a period of time having a *starting time instant* S and an *ending time instant* E as its lower and upper bound respectively. Time instants S and E are the smallest and the highest value in the set of continuous time instants. Lower and upper bounds can be compared with each other with comparison predicates such as $<$ and $\leq$. [Gad88] lists the following possibilities of specifying temporal intervals:

$$\begin{array}{rcl} [S \Leftrightarrow E] & = & \{\ t\ |S \leq t \leq E\ \} \\ (S \Leftrightarrow E) & = & \{\ t\ |S < t < E\ \} \\ [S \Leftrightarrow E) & = & \{\ t\ |S \leq t < E\ \} \\ (S \Leftrightarrow E] & = & \{\ t\ |S < t \leq E\ \} \end{array}$$

where t denotes a time instant.

Thus a time interval can be represented, for example, as an interval $[S \Leftrightarrow E)$ having a closed lower and an open upper bound.

For intervals, the set-theoretic operations union, intersection and difference are defined. These set operations are important to do reasoning about temporal facts. Set intersection is used, for example, to calculate the time period during which two facts were simultaneously true. The overall validity time period of several facts can be determined using set union. The time period a fact is true while another one is not can be calculated using set difference of the corresponding time periods.

These operations however are not closed with respect to intervals. The union of two non-overlapping intervals is defined to return a *set* of intervals. The difference of two intervals may return zero, one or two intervals. The same holds for *time intervals*.

Example 2.1 *Assume three time intervals* $I_1 = [1980 \Leftrightarrow 1990)$, $I_2 = [1992 \Leftrightarrow 1994)$ *and* $I_3 = [1975 \Leftrightarrow 1996)$. *The union of* I_1 *and* I_2 *returns a set of intervals:*

$$I_1 \cup I_2 = \{[1980 \Leftrightarrow 1990), [1992 \Leftrightarrow 1994)\}$$

The difference of the time intervals I_1 *and* I_3 *results in an empty set of time intervals*

$$I_1 \Leftrightarrow I_3 = \{\}$$

whereas the difference of the time intervals I_3 *and* I_2 *returns a set of intervals:*

$$I_3 \Leftrightarrow I_2 = \{[1975 \Leftrightarrow 1992), [1994 \Leftrightarrow 1996)\}$$

To overcome this drawback, [GV85] introduced *temporal domains* which are finite unions of time intervals. [Gad86] called these sets of intervals *temporal elements*. The results of the union and difference operations given in example 2.1 are temporal elements.

In TimeDB, time intervals are used, whereas the temporal object data model TOM uses temporal elements.

2.1.5 Comparison Predicates for Time Intervals

[All83] introduced a set of thirteen temporal comparison predicates for time intervals. These predicates are exhaustive in the sense that they describe all possible relationships between two time intervals. They are shown in table 2.1. $\mathtt{I}_1$ and $\mathtt{I}_2$ are time intervals, and `begin(I)` and `end(I)` represent the starting time instant `S`, and, the ending time instant `E`, respectively, of time interval `I` = [`S` - `E`).

A smaller set of temporal comparison predicates for time intervals has been proposed in [SDJ+93]. They extend the semantics of some of these comparison predicates such that they can also be used to compare time instants with time instants and even time intervals with time instants. [SDJ+93] show that their comparison predicates for time intervals and time instants together with the functions `begin` and `end` have the same expressive power as the comparison predicates proposed in [All83]. The difference between their proposal of temporal comparison predicates and those proposed in [All83] is that they need the functions `begin` and `end` to access the starting and ending points of time intervals in order to achieve the same expressive power. Their set of comparison predicates for time intervals is shown in table 2.2.

Comparison Predicate	Equivalent Predicates on Endpoints
I_1 `before` I_2	$end(I_1) < begin(I_2)$
I_1 `after` I_2	$end(I_2) < begin(I_1)$
I_1 `during` I_2	$(begin(I_1) > begin(I_2) \wedge end(I_1) \leq end(I_2)) \vee$ $(begin(I_1) \geq begin(I_2) \wedge end(I_1) < end(I_2))$
I_1 `contains` I_2	$(begin(I_2) > begin(I_1) \wedge end(I_2) \leq end(I_1)) \vee$ $(begin(I_2) \geq begin(I_1) \wedge end(I_2) < end(I_1))$
I_1 `overlaps` I_2	$begin(I_1) < begin(I_2) \wedge end(I_1) > begin(I_2) \wedge end(I_1) < end(I_2)$
I_1 `overlapped_by` I_2	$begin(I_2) < begin(I_1) \wedge end(I_2) > begin(I_1) \wedge end(I_2) < end(I_1)$
I_1 `meets` I_2	$end(I_1) = begin(I_2)$
I_1 `met_by` I_2	$end(I_2) = begin(I_1)$
I_1 `starts` I_2	$begin(I_1) = begin(I_2) \wedge end(I_1) < end(I_2)$
I_1 `started_by` I_2	$begin(I_1) = begin(I_2) \wedge end(I_2) < end(I_1)$
I_1 `finishes` I_2	$begin(I_1) > begin(I_2) \wedge end(I_1) = end(I_2)$
I_1 `finished_by` I_2	$begin(I_2) > begin(I_1) \wedge end(I_1) = end(I_2)$
I_1 `equals` I_2	$begin(I_1) = begin(I_2) \wedge end(I_1) = end(I_2)$

Table 2.1: Allen's thirteen temporal comparison predicates [All83]

Comparison Predicate	Equivalent Predicates on Time Instants
I_1 `precedes` I_2	$end(I_1) < begin(I_2)$
I_1 `overlaps` I_2	$\exists t : t \in I_1 \wedge t \in I_2$
I_1 `meets` I_2	$end(I_1) = begin(I_2)$
I_1 `contains` I_2	$begin(I_1) \leq begin(I_2) \wedge end(I_1) \geq end(I_2)$
$I_1 = I_2$	$begin(I_1) = begin(I_2) \wedge end(I_1) = end(I_2)$

Table 2.2: The set of temporal comparison predicates for time intervals used in *MultiCal* [SDJ+93]

2.2 Notions of Time

In our daily life, events are usually scheduled with respect to a single time line. There exist different calendars and thus different ways to refer to the same time instants or time intervals. The underlying time line, however, is more or less the same for everybody on this earth.

For temporal databases, the existence of more than one time line was proposed in order to capture different aspects of the relationship between time and data. The following subsections introduce different notions of time denoting different time lines. These are then used to distinguish different forms of temporal databases in terms of their ability to model the various forms of temporal properties of data.

2.2.1 User-defined Time

A distinction can be made whether or not the timestamp added to data is interpreted by the DBMS (for example, during query evaluation). An uninterpreted timestamp, for example, a value in an attribute `Birthday`, is called *user-defined time* [SA86], because the user himself interprets the given time information, whereas the DBMS treats this temporal data as just another attribute.

Data models support user-defined time by offering, for example, a type `date` or `timestamp` for attribute values. Values can be any time instant referring to past, present or future time points. User-defined time values are supplied by the user and may be updated.

The distinction of interpreted or uninterpreted time attributes arises because proposed temporal

data models use the data structures of the underlying non-temporal data model to store temporal data by extending the schemas with special *time* attributes. In case a query is evaluated using temporal semantics, the algebra operations access these special time attributes and thus interpret them. User-defined attributes are never accessed this way.

2.2.2 Valid Time

One approach when proposing a temporal data model is to be able to denote when facts are true with respect to the real world. For example, we would like to know when we deposited money on our bank account, and when we withdrew it again. Sometimes, we also would like to record future events, for example, when reserving tickets for a play in a theatre – we know when the play will be given and when we would like to see it.

This notion of time, recording data with respect to when it was, is or will be valid in the real world, is called *valid time*. A valid-time interval thus records the time period when a fact is true. It is interpreted by a DBMS supporting valid time, for example, during query evaluation or constraint checking. Valid time must be supplied by the user when adding or modifying data. Valid-time values can be updated.

2.2.3 Transaction Time

Besides recording data reflecting the history of the real world, another history of data has been detected to be relevant for temporal databases. Often, data cannot be recorded in a database in real time, for example, due to a delay in the processing of information. So there might be a time gap between data being valid in the real world and recording the data in a database. Sometimes, it is also necessary to keep track during which time periods facts are *stored* in a database. This notion of time is called *transaction time*.

We have seen that valid time is used, for example, to record the history of bank account deposits and withdrawals. This can be viewed as the history of value changes. Assume that a clerk records a wrong date of withdrawal or a wrong amount. In this case, a correction of the wrong values has to be made. This must be recorded, for example, since the bank also wants to know when values of bank accounts were *corrected* (and not only when they were changed). When the bank clerk updates the values, the system automatically sets the transaction-time interval of the old values to end at the current time, and the transaction-time interval of the corrected values to start at the current time. Thus, recording transaction time can also be viewed as recording the history of corrections. This means that data timestamped with transaction time provides for querying the history of data manipulations and errors, whereas data timestamped with valid time allows the querying of value changes. Classical non-temporal updates do not differentiate between a correction and a change of data [GY88, GN93].

It makes perfect sense to query the history of corrections just as the history of the real world is queried. A temporal DBMS thus should be able to interpret transaction-time timestamps during query evaluation or constraint checking in the same way as it interprets valid-time timestamps. So, valid time and transaction time can be viewed as *orthogonal* time lines.

Values for transaction time cannot be later than the current time, since transaction time reflects the time when a database operation is actually executed. The DBMS itself records transaction time. It also does not make sense to update transaction time, since a database operation of a committed transaction can never be undone. The only way to change a committed database operation is to do an inverse transaction, which, however, is executed at a later time point and thus leads to another transaction-time record.

2.2.4 Other Time Lines

So far, three different notions of time distinguished in temporal databases were introduced. There exist, however, other time lines which might be interesting to be recorded. For example, it might

be necessary to record when facts were believed to be true [DLHC95]. Such notions of time have hardly been considered or supported in temporal database proposals.

Data models supporting several notions of time need a more flexible approach than those used in temporal database research so far. In *Bitemporal ChronoSQL* [Pul95], valid time and transaction time were treated orthogonally. This means, that all retrieval operations and constraints referring to valid time can also be used such that they refer to transaction time. Exceptions are the modification operations. Since modifications referring to transaction time are handled by the DBMS, the modification language for transaction time updates is different from updates with respect to valid time. The user may update the valid time of a fact, he is not allowed to update its transaction time, however. The idea of treating these time lines orthogonally was later also used for ATSQL2 [SBJS96b, SBJS96a].

We argue, however, that this orthogonality not only holds for valid and transaction time. All time lines should be treated orthogonally. This means that actually only the functionality with respect to one time line needs to be specified. Then, all other notions of time shall have the same functionality.

2.3 Temporal Databases

The previous section has introduced different notions of time. In most proposals of temporal data models, one or two of them are interpreted by a system. Based on these interpreted time lines, [SA85] introduced several kinds of databases differentiated by their ability to represent temporal information.

According to the taxonomy of temporal databases presented in [SA85], four categories of databases are now discussed with respect to the valid- and transaction-time lines introduced above.

2.3.1 Snapshot Databases

Non-temporal databases capture the real world, as it changes dynamically, by recording a certain state, for example, the current one. Modifying the state of a database is done using update operations such as *insertion*, *deletion* or *update*. Past states of the database are overwritten. This is what currently available commercial DBMS support.

This sort of database is called a *snapshot database*, since it only captures a single snapshot of the real world, usually the current one. Figure 2.1 depicts a snapshot database with respect to valid time and transaction time, where it is assumed that the current state is stored. Only a single database state is recorded, denoted by a container, with respect to the current time. When executing modification statements on such a database state, this state is deleted and the new state is stored. So, after a successful transaction, there is no way to refer to a previous state.

In snapshot databases storing the current state of the real world, it cannot be distinguished between when a fact is true and when it is recorded. Usually, it is assumed that a fact stored in a snapshot database is also true in the real world. If this does not hold anymore, the fact in the database is modified.

The relational data model [Cod70] and the Object Model [Nor92, Nor93], introduced in section 1.1.2 and section 1.1.3.1 respectively, are examples for data models supporting snapshot databases.

2.3.2 Historical Databases

Historical databases record the history of data with respect to the real world. As we have seen, the dynamics of the real world is captured along the valid-time axis. Thus, a historical database records database states along the valid-time line. Past, present and future database states may be recorded. This can also be viewed as storing the history of value changes with respect to the real world. This is depicted in figure 2.2, where each container denotes a different database state

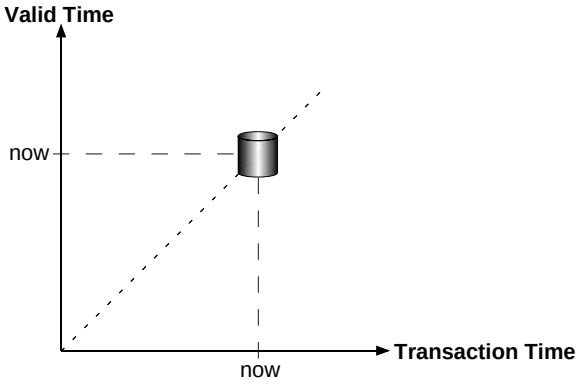

Figure 2.1: A snapshot database in context of valid time and transaction time

in the historical sequence of states. Each change in the real world leads to a new container. As mentioned before, the user must supply the valid time of a fact.

Historical databases require a more sophisticated query language, supporting reasoning about time, selecting data of specific database states and so on. Modification statements need to specify which database states shall be affected by the update operation. Note that updating the valid time of a fact cannot be recorded. This is only possible if the time when a fact is stored in a database – the transaction time – is also recorded, enabling the different valid times of a fact to be distinguished. Thus, in a historical database, data is still physically deleted when corrected.

Historical databases are the most common ones and many of the applications listed in section 1.2 are designated to be implemented using historical databases.

Relational models supporting historical databases are proposed, for example, in [JMS79, Cli82, CW83, Ari86, LJ88, NA88, Sar90a, Lor93, CT85, GV85, Tan86, CC87, Gad88, GY88, TAO89, TG89]. Object data models supporting valid time are proposed in [EW90, SC91, Wuu91, CG92, KS92b, BFG96].

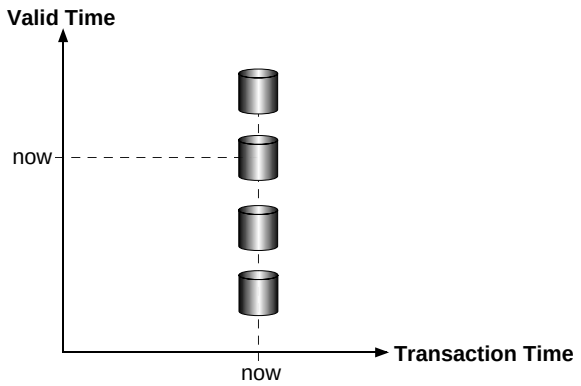

Figure 2.2: A historical database in context of valid time and transaction time

2.3.3 Rollback Databases

A database recording the changes to the database itself is called a *rollback database*. A rollback database thus records data along the transaction time line and can be viewed as an *append-only database*. Rollback databases do not record future database states, since the system itself keeps track of transaction time and does not know anything about future events.

Whenever a modification statement is executed, the system records a new database state with

respect to the time when the modification was done in the system while keeping the old states, as shown in figure 2.3. No data is ever physically deleted. A deletion of data, for example, a tuple, is done by setting its transaction-time interval to end at the execution time of the corresponding statement. A transaction-time interval thus can be viewed as the time period data would have been stored in a snapshot database.

Rollback databases can be used for linear versioning. The time of modification of an object can be used to distinguish the different versions of an object.

Proposals for relational data models supporting rollback databases can be found in [SRH90, JMR91]. Object data models supporting transaction time are proposed in [BM88, KGBW90, SRH90, KS92a].

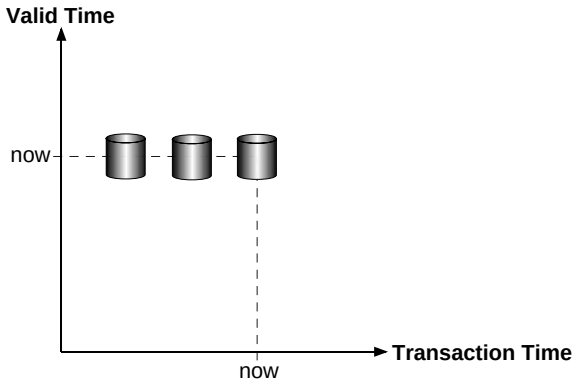

Figure 2.3: A rollback database in context of valid time and transaction time

2.3.4 Bitemporal Databases

A *bitemporal database* is a combination of a historical and a rollback database. Figure 2.4 shows a bitemporal database, recording database states with respect to both valid time and transaction time.

A bitemporal database thus has the properties of both historical and rollback databases. As already mentioned, it is now possible to record updates of valid time in this kind of database.

Bitemporal relational data models are described in [SA85, SA86, Sno87, Sno95b, SBJS96b, SBJS96a]. Object data models supporting bitemporal databases in [CS88, KRS90, RS91, Sci91, EdOP93, GÖ93, WD93, Sci94].

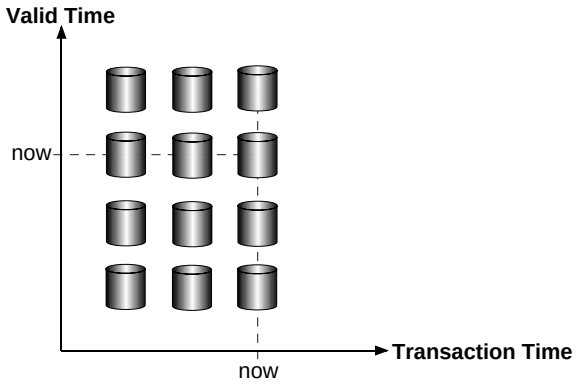

Figure 2.4: A bitemporal database in context of valid time and transaction time

2.4 Timestamping Data

In temporal data models, facts are represented as data units having a timestamp which expresses during which time period they were *valid* in the real world and/or *stored* in the database. The question now is what a data unit is considered to be. Is it a single value? Is it the combination of several property values belonging to the same entity? Is it the set of several entities which are grouped in some way? Or is it that part of the real world which is modeled in the database?

The main questions when timestamping data are *at what level of granularity data units are stamped* and *what kind of timestamp is used*. The following two subsections discuss these issues.

2.4.1 What is Timestamped ?

As mentioned before, temporal databases store facts stamped with time periods of different semantics. The proposed temporal data models can also be distinguished by the level of granularity which is assumed when timestamping data units. A data unit can be anything from an attribute value to a tuple or object to a collection of tuples or objects to a database. Additionally, even schemas or constraints could be considered to be timestamped.

For full generality, all of these possibilities of timestamping should be supported in a temporal data model, although we accept that this full generality is not always needed nor wanted. Our aim is, though, to provide a *general* temporal data model, where the decision of which constructs and what granularity of data units should be timestamped is left to the user.

Research papers in the area of temporal relational databases discuss two levels of timestamping data. None of the previously proposed temporal data models discusses timestamping data on all levels listed above. The following three sections introduce the approaches appearing in literature and review their advantages and disadvantages.

2.4.1.1 Tuple and Object Timestamping

Tuple timestamping is usually applied in temporal *relational* data models supporting only 1NF relations. Data models applying *tuple timestamping* add timestamps to each tuple in a relation. In case of historical data, each tuple is stamped with a validity time period, denoting when the tuple as a whole was valid in the real world. In a rollback database, tuples are stamped with transaction-time periods. Tuples in a bitemporal database are stamped with both valid- and transaction-time periods. Timestamping is usually achieved using the extension approach introduced in section 1.3.2, where special time attributes are added to a non-temporal schema. Temporal data models applying tuple timestamping are proposed, for example, in [JMS79, Cli82, CW83, Ari86, LJ88, NA88, Sar90a, SRH90, Lor93].

The main disadvantage of tuple timestamping is the fact that *information about a real world entity is spread over several tuples* where each tuple represents a state the real world entity was in during a certain time period. [GV85, Gad88] called this the *vertical temporal anomaly*. Additionally, tuple timestamping introduces data redundancy. If, for example, a tuple containing data on an employee is modified by changing the salary, all other information in the tuple, such as the name, the employment number and the department the employee is working in, has to be repeated.

Example 2.2 *We use tuple timestamping to express that Moira has worked in the department of information systems since 1994, having an employment number* 100*. In 1994, she earned* 15000*, and got a salary raise in 1996 to* 20000*. This information is stored in two tuples, one valid from 1994 to 1996, the other one valid since 1996. The term* `until changed` *denotes that it is not yet known when the validity time period of the tuple finishes.*

```
< 100, Moira, 15000, IS > from 1994 until 1996
< 100, Moira, 20000, IS > from 1996 until changed
```

The notion of *object timestamping* appeared with the proposals of temporal object data models. Object timestamping in most proposed temporal object data models, however, is akin to tuple timestamping. The object state is stored in a record containing atomic or complex fields (or attributes). This data structure can be viewed as a tuple. So the data structure storing an object state corresponds to a tuple in NFNF. Temporal data models applying object timestamping are proposed, for example, in [BM88, EW90, KGBW90, SC91, Wuu91, KS92b].

2.4.1.2 Attribute Timestamping

Attribute timestamping overcomes the disadvantage of data redundancy introduced when applying tuple timestamping. Attribute timestamping adds timestamps to each attribute value. Values in a tuple which are not affected by a modification do not have to be repeated. So, the history of values is stored separately for each attribute.

Using the schema extension approach, attribute timestamping demands that the underlying data model supports NFNF relations or complex objects, since all timestamped attributes are of a complex type, storing the attribute value together with its timestamp.

Temporal relational data models applying attribute timestamping are described in [CT85, GV85, Tan86, CC87, Gad88, GY88, TAO89, TG89] and proposals of temporal object data models in [CS88, EW90, RS91, Sci91, CG92, EWK93a, GÖ93].

Example 2.3 *Using attribute timestamping, we can store the information given in example 2.2 in a single tuple:*

```
< {|100 from 1994 until changed|},
  {|Moira from 1994 until changed|},
  {|15000 from 1994 until 1996|, |20000 from 1996 until changed|},
  {|IS from 1994 until changed|} >
```

The tuple contains the history of each attribute. An attribute history is a set of value-timestamp *pairs.*

2.4.2 How is it Timestamped ?

We have seen that temporal data models can be distinguished by whether they apply tuple or attribute timestamping. Using the schema extension approach, the choice of what to timestamp is restricted by the underlying data model.

The same holds if we look at *how* data is timestamped. Typically, data may be timestamped either with a *time instant*, a *time interval* or a *temporal element*. Temporal relational data models staying within 1NF are restricted to use timestamps which can be mapped to additional scalar attribute values of a tuple. Data models supporting NFNF relations or objects permit complex attribute values, allowing timestamps of higher complexity to be used.

The following three sections discuss the different usage of timestamps and how they can be implemented using the schema extension approach. Time intervals are assumed to be closed at the lower and open at the upper bound.

2.4.2.1 Timestamping with Time Instants

Data timestamped with a time instant is usually assumed to be valid only at the specified instant. Relations containing data timestamped with time instant are called *event tables* [Sno95b]. *Tuple timestamping* with time instants can be modeled by extending the schema with an (implicit) attribute of type `date`. The relation stays within first normal form.

Example 2.4 *The non-temporal table* `Employees` *introduced in section 1.1.2 is turned into an* event table *using tuple timestamping with time instants:*

EmpID	Name	Salary	Dept	At
100	Moira	15000	IS	1994
101	Adrian	9000	IS	1996
102	Alain	9000	IS	1995
103	Antonia	11000	IS	1996
104	Gabrio	10000	IS	1996
105	Andreas	9000	IS	1993

Event tables timestamp data with the time instant the corresponding event took place. The schema is extended, *for example, with an additional attribute* `At` *of type* `date`. *The event table* `Employees` *contains the dates when people were hired, their name, employment number and initial salary.*

NFNF relations allow information to be timestamped on the attribute level. *Attribute timestamping* with time instants can be implemented accordingly. Each attribute is turned into a (complex) attribute with the initial type extended with an attribute `At` of type date.

Temporal data models stamping data with time instants are proposed in [CW83, Ari86]. Object data models are described in [BM88, CS88, Sci94].

2.4.2.2 Timestamping with Time Intervals

As seen before, the timestamp associated with a tuple identifies, for example, when the combination of values in the tuple was valid. Using the schema extension approach, timestamping with time intervals can be applied to 1NF relations. The lower and the upper bound of the time interval are mapped to two additional attributes of type `date`. Bitemporal tables are extended with four attributes, two to capture valid time and two for transaction time.

Tuple timestamping with time intervals introduces *redundancy*. As mentioned before, updating values in a tuple leads to a new tuple in the relation, where all attribute values not concerned by the modification are repeated. Additionally, a tuple which is valid during several non-overlapping time periods is stored separately for each time period, spreading the history of a real world object over several tuples. Since time intervals are not closed under the set-theoretic operations intersect, difference and union, a similar effect can occur when calculating a query result.

Example 2.5 *Table* `Employees` *is extended applying tuple timestamping. Two additional attributes* `From` *and* `To`*, which, for example, store the starting and ending time instant of a valid-time interval, are added:*

EmpID	Name	Salary	Dept	From	To
100	Moira	15000	IS	1994	1996
100	Moira	20000	IS	1996	∞
101	Adrian	9000	IS	1996	∞
102	Alain	9000	IS	1995	∞
103	Antonia	11000	IS	1996	∞
104	Gabrio	10000	IS	1996	∞
105	Andreas	9000	IS	1993	∞
106	Martin	9000	IS	1989	1996
107	Tom	4500	Math	1985	1991
107	Tom	4500	Math	1993	1995

Time intervals are assumed to be closed at the lower and open at the upper bound, and ∞ is used to represent the special value `until changed`. *Note that besides the tuples seen in the non-temporal version of this table, it is now possible to store data about employees who are no longer working in the university. Martin, for example, left the university in 1996. Due to Moira's salary raise in 1996, she is listed twice in the table. Tom also appears twice, since he was employed during non-overlapping time periods.*

Applying *attribute timestamping*, each attribute in a relation is extended to a (complex) attribute by adding two or four additional attributes. Sets of such complex attributes contain the history of each attribute in a tuple. Redundancy is introduced if the same value appears several times during non-overlapping time periods for an attribute in a tuple. For example, since Tom was employed during non-overlapping time periods, the corresponding attribute values have to be repeated for each time interval.

Proposals for temporal data models using time intervals for timestamping can be found in [Tan86, Sno87, LJ88, NA89, TAO89, Sar90a, Wuu91, KS92b, Lor93, Tan93, BFG96].

Example 2.6 *Table* `Employees` *is extended using attribute timestamping with time intervals. Two additional attributes* `From` *and* `To` *are added to each attribute. Moira's salary raise in 1996 does not introduce redundancy anymore.*

EmpID	Name	Salary	Dept
{<100, 1994, ∞>}	{<Moira, 1994, ∞>}	{<15000, 1994, 1996>, <20000, 1996, ∞>}	{<IS, 1994, ∞>}
{<101, 1996, ∞>}	{<Adrian,1996, ∞>}	{<9000, 1996, ∞>}	{<IS, 1996, ∞>}
...	...	...	...
{<107, 1985, 1991>, <107, 1993, 1995>}	{<Tom, 1985, 1991>, <Tom, 1993, 1995>}	{<4500, 1985, 1991>, <4500, 1993, 1995>}	{<IS, 1985, 1991>, <IS, 1993, 1995>}

2.4.2.3 Timestamping with Temporal Elements

Timestamping data with temporal elements allows modeling the fact that data was valid during several non-overlapping time periods. Additionally, it has the advantage that temporal elements are closed under the set theoretic operations intersect, difference and union, as seen in section 2.1.4.

Extending the data structures of a non-temporal data model to support data timestamped with temporal elements may only be done if the data model used supports non-atomic attribute values.

Temporal models using temporal elements to timestamp data are introduced, for example, in [GV85, Gad88, GY88, TG89, EW90, GN93].

Example 2.7 *Table* `Employees`*, extended with temporal elements using tuple timestamping with valid time, looks the following way:*

EmpID	Name	Salary	Dept	Validity
100	Moira	15000	IS	{<1994, 1996>}
100	Moira	20000	IS	{<1996, ∞>}
101	Adrian	9000	IS	{<1996, ∞>}
102	Alain	9000	IS	{<1995, ∞>}
103	Antonia	11000	IS	{<1996, ∞>}
104	Gabrio	10000	IS	{<1996, ∞>}
105	Andreas	9000	IS	{<1993, ∞>}
106	Martin	9000	IS	{<1989, 1996>}
107	Tom	4500	Math	{<1985, 1991>, <1993, 1995>}

Moira's salary update still causes redundancy. Employee Tom now is only listed once, since it is possible to store the different time periods of his employment in a temporal element.

To reduce the data redundancy, *attribute timestamping* with temporal elements can be applied. Each attribute in the schema is extended with a temporal element.

Example 2.8 *Table* `Employees` *is extended applying attribute timestamping with temporal elements:*

EmpID	Name	Salary	Dept
{<100, {<1994, ∞>}>}	{<Moira, {<1994, ∞>}>}	{<15000, {<1994, 1996>}>, <20000, {<1996, ∞>}>}	{<IS, {<1994, ∞>}>}
{<101, {<1996, ∞>}>}	{<Adrian, {< 1996, ∞>}>}	{<9000, {<1996, ∞>}>}	{<IS, {<1996, ∞>}>}
...	...	...	...
{<107, {<1985, 1991>, <1993, 1995>}>}	{<Tom, {<1985, 1991>, <1993, 1995>}>}	{<4500, {<1985, 1991>, <1993, 1995>}>}	{<IS, {<1985, 1991>, <1993, 1995>}>}

As already seen in example 2.6, attribute timestamping omits the data redundancy introduced by updating Moira's salary. Employee Tom is also only listed once. This approach of timestamping data eliminates all redundancy.

Attribute timestamping with temporal elements does not introduce redundancy anymore. It leads, however, to rather unreadable relations and it is more difficult to write queries dealing with the different time periods.

2.5 Temporal Operations

As shown in chapter 1, extending the data structures of a non-temporal database schema to support temporal data is not enough. The data model also has to support operations which allow the *temporal* data to be *queried* and *updated* with respect to time.

Different approaches have been taken to give support for temporal reasoning and querying [MS91, Cho94, Sno95b]. For example, the operational part of some temporal data models contains special operations to do temporal selections or temporal joins. Other approaches propose a temporal algebra.

The following subsections describe three important notions concerning the operational part of temporal data models. These notions hold not only in the context of temporal relational data models in which they were introduced, but also for other data models.

2.5.1 Vertical Temporal Anomaly and Coalescing

As mentioned before, the history of a real world entity is spread over several tuples if tuple timestamping is applied. Each tuple contains a state the real world entity once was in, together with the time information when this state was valid (or recorded). [GV85, Gad88] called this forced splitting of a logical unit of information into more than one tuple *vertical temporal anomaly*.

Temporal data models using tuple timestamping thus represent temporal data inappropriately. A user wants to view all data about an employee as a unit, for example, in a single tuple. As we have seen, the vertical temporal anomaly cannot be omitted in 1NF temporal data models. The consequence is that the overall time a real world entity plays a specific role, for example, when the entity is an employee, is split up in several smaller time periods, due to the changes in attribute values. Often, however, it is of interest during which time period or how long an employee was working in a company. In order to calculate maximal time intervals for such tuples, a special operation – *coalescing* – was proposed [NA93, Sar93, Böh94, Sno95b, SBJS96b].

The idea to calculate the overall time period a real world entity played a specific role, for example, when he was an employee, is the following: Attributes containing time-varying values must first be projected away[1] since they cause the temporal data about the entity to be split up into several tuples. This leads to *value-equivalent tuples*. Value-equivalent tuples are tuples having identical non-timestamp attribute values. The next step is to calculate the overall time period for each set of value-equivalent tuples having overlapping or consecutive valid-time periods. This is done using the coalescing operations. So, a table is said to be *coalesced* if it does not contain two or more value-equivalent tuples with overlapping or consecutive valid-time periods. Some proposals of temporal data models assume implicitly coalesced table, others support an explicit coalescing operation.

[1] Projecting away an attribute means that a new relation without that attribute is calculated using the projection operation

Example 2.9 *Assume we want to find the overall time period employees were working for a company. We use table* `Employees` *shown in example 2.5. First, we have to project away all time-varying attributes. The employment number and the name of the employee shall be time invariant. This step results in the following table:*

EmpID	Name	From	To
100	Moira	1994	1996
100	Moira	1996	∞
101	Adrian	1996	∞
102	Alain	1995	∞
103	Antonia	1996	∞
104	Gabrio	1996	∞
105	Andreas	1993	∞
106	Martin	1989	1996
107	Tom	1985	1991
107	Tom	1993	1995

The table shown above contains two sets of value-equivalent tuples. We see that the two tuples containing data on employee Moira have identical non-timestamp attribute values. The same holds for the two tuples containing data about employee Tom. Since the time periods of the former are overlapping, a maximal time period can be calculated, whereas the time periods of the latter are disjoint. To find the maximal employment time periods for each employee, the above table is coalesced which leads to

EmpID	Name	From	To
100	Moira	1994	∞
101	Adrian	1996	∞
102	Alain	1995	∞
103	Antonia	1996	∞
104	Gabrio	1996	∞
105	Andreas	1993	∞
106	Martin	1989	1996
107	Tom	1985	1991
107	Tom	1993	1995

There is a major problem in using this operation, however. For a user, it might not be clear which of the attributes are time-varying and which are time-invariant. In fact, this might even change during the life time of a temporal database. This means, that for a specific query, it is not straightforward to know which attributes have to be projected away for the desired result.

For example, using the table given in example 2.5, two scenarios are possible. First, assume a user wants to find the maximal time periods employees were working for a specific department. Looking at the relation instance given, he assumes attributes `EmpID` and `Name` to be time invariant. So, he removes attribute `Salary` from the relation using the projection operation and coalesces the resulting table. Second, assume the user wants to find the maximal time periods employees were working for the university. This means that prior to coalescing, he additionally has to get rid of attribute `Dept`.

Thus, in order to write the correct queries, the user must look at the table to find out which attributes are time-varying. This is not satisfactory. Additionally, being forced to do a projection operation prior to finding out during which time period an entity had a certain role, is neither natural nor straightforward. Chapter 5 will show that the concept of temporal object role modeling eliminates these deficiencies.

2.5.2 Snapshot Reducibility

When enhancing a non-temporal algebra to a temporal algebra, the question is what the semantics of the temporal operations should be. The same, of course, holds when defining a temporal query language. [Sno87, BJS95] propose a reduction proof to specify the semantics of temporal queries. It relates the databases and queries of a *temporal* data model $\mathtt{M}^T = (\mathtt{DS}^T, \mathtt{QL}^T)$ with a non-temporal data model $\mathtt{M} = (\mathtt{DS}, \mathtt{QL})$, where temporal relations and queries are instances in $\mathtt{DS}^T$ and $\mathtt{QL}^T$, respectively, and non-temporal relations and queries are instances in DS and QL. In the following, the focus is on historical (valid-time) databases. The notion of snapshot reducibility, however, can be used also with respect to rollback or bitemporal databases.

The reduction proof given in [Sno87, BJS95] states that the snapshot at time instant t of the result of a valid-time query $\mathtt{q}^v$ in $\mathtt{QL}^v$, executed on a valid-time database $\mathtt{db}^v$, must be equal to the result of the corresponding non-temporal query q in QL, executed on the snapshot of the valid-time database at time instant t, $\tau_t^{M^v,M}(\mathtt{db}^v)$, where valid-time database $\mathtt{db}^v$ is an instance in $\mathtt{DS}^v$.

The valid-time slice operation τ takes a valid-time database $\mathtt{db}^v$ of data model $\mathtt{M}^v$ and returns the state of $\mathtt{db}^v$ at a time instant t, preserving value-equivalent tuples as duplicates. The resulting snapshot of a valid-time database, $\mathtt{db} = \tau_t^{M^v,M}(\mathtt{db}^v)$ of data model M, is the state the database was in at the specified time instant, without the valid time.

Example 2.10 *Assume a valid-time database* $\mathtt{db}^v$ *consisting of a single valid-time table* Employees *as given in example 2.5, and assume that this database is specified using a valid-time relational data model* $\mathtt{RM}^v$.

The snapshot of this historical database, taken at January 1, 1994,

$$\tau_{January1,1994}^{RM^v,RM}(\mathtt{db}^v),$$

returns a non-temporal relational database db *defined in the non-temporal relational data model* RM*, which consists of the following snapshot relation* Employees*:*

EmpID	Name	Salary	Dept
100	Moira	15000	IS
105	Andreas	9000	IS
106	Martin	9000	IS
107	Tom	4500	Math

With respect to the relational model, the time-slice operation τ returns non-temporal relations containing the tuples which were valid at a specific time point. The notion of snapshot reducibility can now be defined using the valid-time slice operation.

Definition 2.1 (Snapshot Reducibility) *[BJS95] Let* $\mathtt{M} = (\mathtt{DS}, \mathtt{QL})$ *be a non-temporal relational data model, and let* $\mathtt{M}^v = (\mathtt{DS}^v, \mathtt{QL}^v)$ *be a valid-time relational data model. Also, let* $\mathtt{db}^v$ *be a database instance in* $\mathtt{DS}^v$*. A valid-time query* $\mathtt{q}^v$ *in* $\mathtt{QL}^v$ *is* snapshot reducible *with respect to a non-temporal query* q *in* QL *if and only if*

$$\forall \mathtt{db}^v \forall \mathtt{t} : \tau_t^{M^v,M}(\mathtt{q}^v(\mathtt{db}^v)) = \mathtt{q}(\tau_t^{M^v,M}(\mathtt{db}^v))$$

Snapshot reducibility implies that for all valid-time databases $\mathtt{db}^v$ and for all time instants t, the commutativity diagram depicted in figure 2.5 must hold [BJS95].

Snapshot reducibility has been used to define the semantics of queries in [Sno87, Böh94, BJS95]. Their definitions are not as general as possible, though. First, they explicitly refer to a non-temporal *relational* data model and second, they do not specify exactly what the query language covers. [Sno87] mentions that snapshot reducibility can also be used to define the semantics of modification statements.

In section 1.3.1, a data model M was specified as consisting of three parts – the data structures DS, operations OP and integrity constraints C. The operations do not only cover queries for data

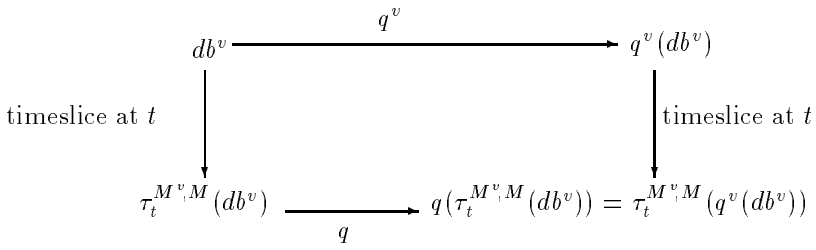

Figure 2.5: Snapshot reducibility of query $\mathtt{q}^v$ with respect to query $\mathtt{q}$ at time instant t

retrieval, but also modification statements. Additionally, constraints are specified as an explicit part of a data model. In this thesis, a more general (or clearer) definition of snapshot reducibility will be used which is defined as:

Definition 2.2 (General Version of Snapshot Reducibility) *Let* $\mathtt{M} = (\mathtt{DS}, \mathtt{OP}, \mathtt{C})$ *be a non-temporal data model, and let* $\mathtt{M}^T = (\mathtt{DS}^T, \mathtt{OP}^T, \mathtt{C}^T)$ *be a temporal data model. Also, let* $\mathtt{db}^T$ *be a database instance in* $\mathtt{DS}^T$*. A temporal operation* $\mathtt{op}^T$ *in* $\mathtt{OP}^T$ *is* snapshot reducible *with respect to a non-temporal operation* $\mathtt{op}$ *in* $\mathtt{OP}$ *if and only if*

$$\forall \mathtt{db}^\mathtt{T} \forall \mathtt{t} : \tau_t^{M^T,M}(\mathtt{op}^\mathtt{T}(\mathtt{db}^\mathtt{T})) = \mathtt{op}(\tau_t^{M^T,M}(\mathtt{db}^\mathtt{T}))$$

A temporal constraint $\mathtt{c}^T$ *in* $\mathtt{C}^T$ *is* snapshot reducible *with respect to a non-temporal constraint* $\mathtt{c}$ *in* $\mathtt{C}$ *if and only if*

$$\forall \mathtt{db}^\mathtt{T} \forall \mathtt{t} : \tau_t^{M^T,M}(\mathtt{c}^\mathtt{T}(\mathtt{db}^\mathtt{T})) = \mathtt{c}(\tau_t^{M^T,M}(\mathtt{db}^\mathtt{T}))$$

where $\mathtt{c}^T(\mathtt{db}^T)$ *and* $\mathtt{c}(\mathtt{db})$ *denote the evaluation of the constraints over a database.*

Definition 2.2 does not make any assumption on the underlying non-temporal data model and specifies clearly that not only temporal queries but also temporal modification operations and temporal constraints are defined using snapshot reducibility. Additionally, it is not restricted to valid-time data models. The semantics of operations on transaction time may also be defined using the notion of snapshot reducibility. The temporal data model TOM is based on this general definition of snapshot reducibility.

2.5.3 Temporal Completeness

[BM94, Böh94] introduced the notion of *temporal semi-completeness* and *temporal completeness* for query languages. In [BJS95], slightly changed versions of these definitions are proposed, adding syntactical requirements for temporal query languages to these definitions.

Temporal semi-completeness essentially states that a temporal data model must contain temporal generalisations of *all* non-temporal queries and non-temporal instances of data structures. Temporally generalised queries must be *syntactically similar* to the snapshot queries they generalise. Temporal completeness adds further functionality and syntactical requirements to temporal semi-completeness, such as temporal comparison operators and the possibility to override snapshot reducible semantics of queries with non-temporal semantics. In the following, the definitions given in [BJS95] are discussed, since these definitions seem to replace earlier ones.

Definition 2.3 (Temporal Semi-Completeness) *[BJS95] Let* $\mathtt{M} = (\mathtt{DS}, \mathtt{QL})$ *be a non-temporal data model, and let* $\mathtt{M}^v = (\mathtt{DS}^v, \mathtt{QL}^v)$ *be a valid-time data model. Data model* $\mathtt{M}^v$ *is* temporally semi-complete *with respect to model* $\mathtt{M}$ *if and only if all three of the following conditions hold:*

1. *For every relation* r *in* DS*, there exists a valid-time relation* r^v *in* DS^v *and a time instant* t *such that* $r = \tau_t^{M^v,M}(r^v)$.

2. *For every query* q *in* QL*, there exists a query* q^v *in* QL^v *that is* snapshot reducible *with respect to* q.

3. *There exist two (possibly empty) text strings* S_1 *and* S_2 *such that for all pairs* (q, q^v) *of queries, where* q^v *is snapshot reducible with respect to* q*, query* q^v *is syntactically identical to query* S_1 q S_2.

The first condition in definition 2.3 states that all relation instances in the non-temporal data model can be produced by taking a snapshot of a temporal relation instance. The consequence of this is that if the non-temporal data model M supports duplicates, duplicates must also be supported in the temporal model M^v. Temporal data models assuming that relations are implicitly coalesced hence are not temporally complete, since duplicates are eliminated in coalesced tables.

The second condition demands that each non-temporal query q expressible in data model M must have a temporal counterpart q^v in data model M^v. Hence, if data model M supports negation, then data model M^v must also support it for temporal queries. This accounts for the fact that most temporal data models proposed do not support temporal negation, for example, in form of a temporal set difference.

The last condition specifies a syntactical condition for the query language of temporal data models. The idea is that a non-temporal query and its temporal counterpart shall be *syntactically similar*. So the syntax of a legal non-temporal query may only be extended by adding keywords in front of or after it. The same two strings S_1 and S_2 must apply to all pairs (q, q^v). This condition allows an easy migration of non-temporal queries to temporal queries since non-temporal queries can be reused for the definition of their temporal counterparts.

Temporal semi-completeness covers only those queries in the temporal data model which are snapshot reducible to a non-temporal query in the non-temporal data model. A temporal data model should provide for other forms of temporal queries, however. For example, it should support queries using temporal comparison predicates as introduced in section 2.1.5. These aspects are covered in the definition of *temporal completeness*.

Definition 2.4 (Temporal Completeness) *[BJS95] A valid-time data model* M^v = (DS^v, QL^v) *is* temporally complete *with respect to a non-temporal data model* M = (DS, QL) *if and only if all five of the following conditions hold:*

1. M^v *is temporally semi-complete with respect to* M.

2. *For every snapshot reducible query* q^v *in* QL^v*, it is possible to override snapshot reducibility, either by dropping the syntactic extensions that enforce snapshot reducibility or by modifying* q^v *syntactically to* S_1 q S_2*, where* S_1 *and* S_2 *are (possibly empty) text strings that depend on* QL^v *but not on* q^v*. Overriding snapshot reducibility means to evaluate a query without interpreting valid times.*

3. *The name of a valid-time relation within a statement can be syntactically substituted (perhaps with other syntactic modifications and additions, such as parentheses) with a query* q^v *in* QL^v *that defines the respective valid-time relation without changing the semantics of the statement. The syntactic modifications must depend on* QL^v *only, not on* q^v.

4. *Allen's temporal comparison predicates [All83] can be used between (a) temporal attributes of stored valid-time tables (for example, valid-time attributes and explicit temporal attributes), (b) implicitly computed valid times associated with temporally semi-complete (sub-)queries, and (c) temporal constants.*

5. *It is possible to retrieve and constrain (a) maximal continuous valid-time periods and (b) valid times as specified by the user.*

A temporally complete language must be temporally semi-complete. It must be possible to override snapshot reducible semantics in a query and treat the elements of a database as *uninterpreted* objects. This is, for example, necessary to compare the objects *temporally* with each other. Additionally, it must be possible to substitute a temporal relation within a temporal query by another temporal query. The temporal comparison predicates defined in [All83] (or an equivalent set) have to be supported. Finally, the data model has to support maximal continuous time periods, for example, by providing a coalescing operation, and it must be possible to retrieve and constrain valid times as specified by the user.

Temporal completeness provides requirements for temporal data models, specified with respect to a non-temporal reference model. This supports what was called the generalisation approach in section 1.3.2.2. However, as already mentioned there, temporal completeness only refers to a *query language* and does not explicitly include constraints. It is obvious that these definitions were made with respect to the relational data model and the query language SQL. Additionally, the mixture of syntax and semantics in these definitions seems to be unnatural. These definitions were customised for a specific language rather than intended to be general requirements.

Interpreted in a general way, these definitions supply nevertheless valuable requirements applicable also, for example, to temporal object data models.

2.6 Summary

This chapter has introduced important concepts appearing in several places in literature on temporal database research. On one hand, these concepts are used in several temporal data models, for example, the notions of valid time and transaction time and the way data is timestamped. On the other hand, this chapter also has introduced ideas which have not yet been widely used, but in our opinion are essential for temporal data models, for example, snapshot reducibility and temporal completeness.

In chapter 1, it was stated that the goal of this thesis is to come up with a *general*, *generic* and *orthogonal* temporal data model. In order to achieve this, some of the concepts introduced in this chapter will be used, because they support the ideas of a generalised temporal data model. Other concepts introduced in this chapter, however, will be questioned.

As a summary, a wish list for a temporal data model will now shortly be discussed with respect to the concepts introduced in this chapter. This chapter has shown how time can be modeled and the proposed notions of time. Additionally, an overview of the different ways to timestamp data was given, and what kind of timestamps have been used in the literature. With respect to the functionality of a temporal data model, the important notions of snapshot reducibility and temporal completeness and the *coalescing* operation were discussed.

In this thesis, the vision of a temporal data model is based on the idea that a data model consists of data structures, operations and integrity constraints. A *general* temporal data model should include the general concepts already proposed – for example, snapshot reducibility and temporal completeness.

On the other hand, many weaknesses found in other proposals should be eliminated. One source of restrictions is the fact that data structures are extended instead of generalised. This means, that considering only tuple and attribute timestamping using schema extension is too restrictive. *First*, it does not allow for a *generic* data model, since assumptions on special attributes in the schema or type are made and temporal algebra operations are based on these special attributes. *Second*, the problem of the vertical temporal anomaly was mentioned. One solution is a special coalescing operation, necessary to calculate the overall time period a real world entity played a role. The use of this operation is not straightforward. In general, a logical unit of information should be prevented from being spread over several objects of the temporal data model used. The history of an entity should be stored within a single object of the model. *Third*, tuple and attribute timestamping are used to timestamp real world entities. Timestamping other constructs such as sets of entities (in the form of relations or collections of objects), constraints or even databases as a whole were never discussed in temporal database research.

Another restriction is that most proposals focus on temporal query languages. It is not clear, however, what they consider the query language to be. Are modification operations included when not explicitly stated ? And what about constraints? The requirements for a *general*, *generic* and *orthogonal* temporal data model thus are the following:

A temporal data model should support

1. valid time and/or transaction time, possibly other time lines
2. temporal data structures which do not make any assumptions about a specific type system or the presence of specific time attributes
3. temporal data structures which allow the timestamping of *all* constructs supported by the model in an orthogonal way
4. temporal data structures which overcome the vertical temporal anomaly
5. a temporally complete query language or algebra with snapshot reducible semantics
6. temporal constraints with snapshot reducible semantics

Additionally, it would be preferable to have a semantically rich temporal data model supporting, for example, temporal associations and the conceptual modeling of temporal databases without having to map the conceptual model to a data model supported by a DBMS.

Chapter 3

Temporal Data Models

Having discussed the general concepts used for temporal databases in the previous chapter, this chapter now gives an overview of proposed temporal data models. The data models are on one hand classified according to the underlying data model, and on the other hand, according to the kind of timestamping they use. Interestingly, most of the models use the schema extension approach.

Additionally, proposals of abstract data types for time and a functional data model which is extended to handle time-varying data will be discussed.

3.1 Temporal Relational Data Models

First, temporal *relational* data models proposed in the literature are introduced. Since most of the work in the research area of temporal databases has been done with respect to the relational data model, numerous proposals can be found (with only few implementations). This section looks at the most important ones with respect to the work presented in this thesis. As discussed in the previous chapter, data may be timestamped using tuple or attribute timestamping. Temporal relational data models using tuple timestamping are introduced first. Then, NFNF relational data models using attribute timestamping are discussed. A temporal relational data model using a combination of both tuple and attribute timestamping concludes this section.

3.1.1 Tuple Timestamping

Many proposals can be found extending the relational model supporting 1NF relations for time-varying data. In the following, the relevant proposals are classified with respect to what kind of timestamp is used. As already mentioned, temporal data models applying tuple timestamping suffer from the vertical temporal anomaly. Different solutions which address this issue in one way or another will also be discussed. Additionally, models assuming *implicitly* coalesced relations will be identified. As stated before, these models are not temporally complete since they do not support duplicates.

3.1.1.1 Timestamp is a Time Instant

In the *Historical Data Model (HDM)* [Cli82, CW83], a *historical database* consists of a collection of historical relations over the same set of states. A *historical relation* is viewed as a sequence of relation instances, indexed by valid time, each one representing a different *state* of the historical relation. A historical relation thus is a three dimensional object – the two dimensions *attributes* and *tuples* plus a *time dimension*. This is called the cubic view of a historical relation. Each plane on the time dimension is a static relation instance.

All tuples are assumed to appear in *all* states of the relation. Such *completed relations* contain a tuple in each state for every entity that appears in a relation at any state of the database history.

In case a tuple does not exist in a state, it contains *null* values in all attributes not belonging to the primary key of the relation.

An attribute `STATE` containing time instants and a boolean-valued attribute `EXISTS` are added to each relation. These attributes are an intrinsic part of their temporal data model. `STATE` denotes the relation state the tuple belongs to. The `EXISTS` attribute is used to specify whether the tuple exists in this relation state or not. A non-temporal key is extended by the attribute `STATE`.

Example 3.1 *Assume attribute* `EmpID` *to be the key of the non-temporal relation* `Employees`*. The historical relation given in example 2.5, modeled using HDM, then has a key* {`EmpID, STATE`}*. Each tuple valid at a time instant is repeated in all relation states. The values 1 or 0 denote whether the tuple exists in a state or not. Value* $\perp$ *represents the null value.*

EmpID	Name	Salary	Dept	STATE	EXISTS?
100	$\perp$	$\perp$	$\perp$	1985	0
101	$\perp$	$\perp$	$\perp$	1985	0
102	$\perp$	$\perp$	$\perp$	1985	0
103	$\perp$	$\perp$	$\perp$	1985	0
104	$\perp$	$\perp$	$\perp$	1985	0
105	$\perp$	$\perp$	$\perp$	1985	0
106	$\perp$	$\perp$	$\perp$	1985	0
107	Tom	4500	Math	1985	1
100	$\perp$	$\perp$	$\perp$	1989	0
101	$\perp$	$\perp$	$\perp$	1989	0
102	$\perp$	$\perp$	$\perp$	1989	0
103	$\perp$	$\perp$	$\perp$	1989	0
104	$\perp$	$\perp$	$\perp$	1989	0
105	$\perp$	$\perp$	$\perp$	1989	0
106	Martin	9000	IS	1989	1
107	Tom	4500	Math	1989	1
100	$\perp$	$\perp$	$\perp$	1991	0
101	$\perp$	$\perp$	$\perp$	1991	0
102	$\perp$	$\perp$	$\perp$	1991	0
103	$\perp$	$\perp$	$\perp$	1991	0
104	$\perp$	$\perp$	$\perp$	1991	0
105	$\perp$	$\perp$	$\perp$	1991	0
106	Martin	9000	IS	1991	1
107	$\perp$	$\perp$	$\perp$	1991	0
100	$\perp$	$\perp$	$\perp$	1993	0
101	$\perp$	$\perp$	$\perp$	1993	0
102	$\perp$	$\perp$	$\perp$	1993	0
103	$\perp$	$\perp$	$\perp$	1993	0
104	$\perp$	$\perp$	$\perp$	1993	0
105	Andreas	9000	IS	1993	1
106	Martin	9000	IS	1993	1
107	Tom	4500	Math	1993	1
100	Moira	15000	IS	1994	1
101	$\perp$	$\perp$	$\perp$	1994	0
102	$\perp$	$\perp$	$\perp$	1994	0
103	$\perp$	$\perp$	$\perp$	1994	0
104	$\perp$	$\perp$	$\perp$	1994	0
105	Andreas	9000	IS	1994	1
106	Martin	9000	IS	1994	1
107	Tom	4500	Math	1994	1
...	...	...	...	...	...

Since completed relations contain a tuple for each entity and state, regardless if the entity existed in that state, a historical relation in HDM is highly redundant. [CW83] mention that the

picture of each historical relation as a cube is an idealisation and a direct implementation highly redundant. However, they do not provide better ideas for its implementation.

The *temporally oriented data model (TODM)* proposed by [Ari86] tries to overcome this deficiency. A *data cube*, also consisting of the three dimensions *attributes*, *tuples* and *time*, supports an explicit and inherent order of the tuples contained in it. This preserves the temporal context of the data and the identity of the tuples. The ordering of objects over time and interpolation are necessary to know how long a state prevailed or what the state was at any time.

3.1.1.2 Timestamp is a Time Interval

[JMS79] timestamp tuples in their data model *LEGOL 2.0* with two implicit time attributes `Start` and `Stop`. These two additional attributes correspond to the starting and ending point of the time period an entity exists in the real world. `Start` and `Stop` actually represent a closed valid-time interval [`Start` ⇔`Stop`].

Example 3.2 *The historical relation of example 2.5 would be modeled in LEGOL in the following way:*

EmpID	Name	Salary	Dept	Start	Stop
100	Moira	15000	IS	1994	1995
100	Moira	20000	IS	1996	∞
101	Adrian	9000	IS	1996	∞
102	Alain	9000	IS	1995	∞
103	Antonia	11000	IS	1996	∞
104	Gabrio	10000	IS	1996	∞
105	Andreas	9000	IS	1993	∞
106	Martin	9000	IS	1989	1995
107	Tom	4500	Math	1985	1990
107	Tom	4500	Math	1993	1994

[NA88, NA89, NA93] introduce the *Temporal Relational Model (TRM)*. Similar to the LEGOL approach, every time-varying relational schema in TRM has two mandatory timestamp attributes – time-start (T_S) and time-end (T_E) – which represent a *closed* valid-time interval [$T_S \Leftrightarrow T_E$]. A valid-time relation contains a time-invariant key and time-varying attributes. They distinguish between *synchronous* and *asynchronous* time-varying attributes. Synchronous attributes in a relation change their values always at the same time, whereas asynchronous attributes change their values independently from the other attributes in the relation. For example, attributes `Salary` and `Dept` in the example relation `Employees` do not necessarily change their values at the same time for a particular employee.

An attribute is not allowed to have multiple values at a particular instant of time. This means that a relation in TRM is always *coalesced*. Every time-varying relation schema has two candidate keys – the time-invariant key plus either T_S or T_E.

As already seen, the fragmentation of an entity over several tuples causes incomplete information about the lifespan of attributes to be stored in a tuple. The tuples in a relation are semantically dependent. This causes update and retrieval anomalies. To overcome these problems, due to the asynchronous variations of attribute values within a tuple over time, they proposed a *time normal form (TNF)*. Time normalisation ensures that the lifespans of a tuple *and* its attribute values are the same, or, in other words, that all attributes in a relation change synchronously. This is achieved by decomposing relations into sub-relations where all time-varying attributes change their values *simultaneously*. However, in the case that synchronous attributes are not available, TNF degenerates to tuple timestamping over relations containing, for example, a key attribute which is invariant over time, plus a time-varying attribute and a timestamp. The TNF addresses problems similar to the vertical temporal anomaly, as the next example shows.

Example 3.3 *The following relation* `Employees` *is not in TNF, since attributes* `Salary` *and* `Dept` *are asynchronous:*

EmpID	Name	Salary	Dept	T_S	T_E
108	John	8000	IS	1990	1993
108	John	8000	Math	1994	1995
108	John	9000	Math	1996	now

We assume attributes `EmpID` *and* `Name` *to be time invariant. The relation needs to be decomposed in two relations* `Salary` *and* `Employees` *which are in TNF:*

EmpID	Name	Salary	T_S	T_E
108	John	8000	1990	1995
108	John	9000	1996	now

EmpID	Dept	T_S	T_E
108	IS	1990	1993
108	Math	1994	now

The time information, for example, the salary time periods, are not fragmented anymore. The vertical temporal anomaly, however, is not removed since data about real world entity John is still spread over several tuples.

3.1.1.3 Timestamp is a Time Instant or a Time Interval

The *Historical Data Model (HDM)* described in [Sar90b, Sar93] supports *historical relations*. A historical relation contains either tuples stamped with a time instant (event relation) or stamped with a time interval (state relation). Entities are modeled as objects described by attributes. The values of these attributes define the state of the object. A state prevails over an interval of time, during which *none of the attributes* change their values.

In HDM, historical relations are mapped to relations having *visible attributes* – attributes of a corresponding non-temporal relation – and automatically added *timing attributes* `FROM` and `TO`, modeling a closed time interval [`FROM` - `TO`]. Conventional relations may be transformed into historical relations by assuming a valid time [0 - NOW].

HDM only supports valid time. They view a temporal database as being updated in real time, which means that valid time and transaction time are the same. All historical relations are automatically coalesced.

For a corresponding historical DBMS (HDBMS), they propose that historical relations would be created by listing the visible attributes and specifying a granularity for the timestamps used for the relations. A hierarchy of different granularities would be provided by the system. HDBMS separates each historical relation in two union compatible relations, one containing the currently valid tuples, the other one the history of the tuples. Future states cannot be recorded in this model. The separation of current and historical tuples is transparent to the user and allows faster access to the current data.

Example 3.4 *Relation* `Employees`*, given in example 2.5, would be created using the following SQL statement:*

```
CREATE STATE TABLE Employees (EmpID  INTEGER,
                              Name   CHAR(10),
                              Salary INTEGER,
                              Dept   CHAR(10))
WITH TIME GRANULARITY Date
```

This historical relation then is separated into the segment `CURRENT_Employees`

EmpID	Name	Salary	Dept	FROM	TO
100	Moira	20000	IS	1996	now
101	Adrian	9000	IS	1996	now
102	Alain	9000	IS	1995	now
103	Antonia	11000	IS	1996	now
104	Gabrio	10000	IS	1996	now
105	Andreas	9000	IS	1993	now

and the segment `HISTORY_Employees`

EmpID	Name	Salary	Dept	FROM	TO
100	Moira	15000	IS	1994	1996
106	Martin	9000	IS	1989	1996
107	Tom	4500	Math	1985	1991
107	Tom	4500	Math	1993	1995

Another temporal data model supporting both time instant and state timestamps is proposed in [Sno84, Sno87, Sno93]. They present a temporal query language *TQuel* which is based on Quel [HSW75]. Their temporal data model supports *event* and *bitemporal* relations where tuples are timestamped with either time instants or time intervals. A time instant is mapped to a single attribute, and valid- and transaction-time intervals to pairs of attributes. Temporal relations are always in a coalesced state, and temporal operations always return coalesced results. TQuel also supports a temporal data type to support user-defined time. [Sno87] states that since the additional temporal attributes are an artifact of embedding a temporal relation in a snapshot one, the users must be constrained in how they use these attributes. This is a general problem when using the schema extension approach. Their main focus, though, is on the specification and semantics of a temporal query language.

Example 3.5 *The table of example 2.5 can be modeled in TQuel as a* bitemporal *relation. Attributes* `From` *and* `To` *denote the valid-time interval,* `Start` *and* `Stop` *the transaction-time interval.*

EmpID	Name	Salary	Dept	From	To	Start	Stop
100	Moira	15000	IS	1994	1996	1995	1996
100	Moira	20000	IS	1996	∞	1996	∞
101	Adrian	9000	IS	1996	∞	1997	∞
...	...	...	...	...	...	...	...
107	Tom	4500	Math	1985	1991	1990	1993
107	Tom	4500	Math	1984	1993	1993	1995
107	Tom	4500	Math	1985	1991	1995	∞
...	...	...	...	...	...	...	...

Transaction time denotes the time when tuples were inserted into or deleted from the database. The first tuple was stored in the database in 1995 and deleted in 1996, when the salary was updated and a new tuple had to be inserted. The third tuple was inserted into the database in 1997. The fourth tuple was corrected twice. First, it was stored that employee Tom was working for the first time at the university from 1985 to 1991. This was believed to be true and thus stored in the database from 1990 until 1993. From 1993 to 1995, it was believed that Tom worked for the university during 1984 to 1993. In 1995, it was set back to valid-time interval [1985 $\Leftrightarrow$1991).

3.1.1.4 Timestamp is a Temporal Element

[Sno95b] propose a conceptual data model, the *Bitemporal Conceptual Data Model (BCDM)*. This conceptual model is used as a basis for the definition of the temporal query language *TSQL2* and allows for multiple representation data models.

TSQL2 supports user-defined, valid time and transaction time. Data is timestamped either with sets of time instants or temporal elements. A relation in BCDM consists of a set of ordinary tuples, consisting of *explicit* and *implicit* attributes. User-defined time is recorded as an explicit

attribute. Valid time and transaction time are recorded as implicit attribute values of a tuple, specifying when the data represented by the tuple is true in the real world and stored in the database, respectively. The implicit valid-time attribute has either a valid-time instant set or a valid-time element as its value. Transaction-time attributes are recorded as temporal elements. Tuples in bitemporal relations are timestamped with implicit attributes containing bitemporal elements or bitemporal instant sets. Temporal relations in BCDM are inherently coalesced.

Example 3.6 *In example 3.5, Tom's correction history with respect to the time period of his first employment was described. This is depicted in figure 3.1. The shaded area covers those time instants this specific fact about employee Tom was valid and stored in the database.*

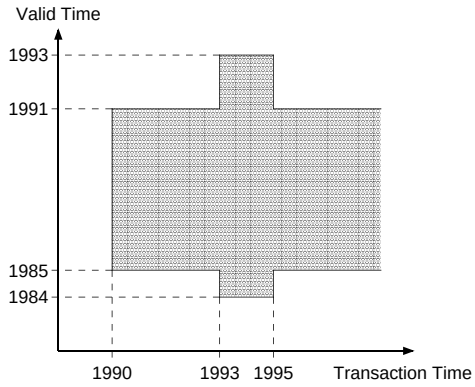

Figure 3.1: A bitemporal element in BCDM

The timestamp of the bitemporal element shown in figure 3.1 is the following set of pairs (valid-time, transaction-time) *of bitemporal chronons, assuming a chronon to have the granularity of a year:*

$$
\begin{aligned}
\{\quad & (1985, 1990), \ldots, (1991, 1990), \\
& (1985, 1991), \ldots, (1991, 1991), \\
& (1985, 1992), \ldots, (1991, 1992), \\
& (1984, 1993), \ldots, (1993, 1993), \\
& (1984, 1994), \ldots, (1993, 1994), \\
& (1985, 1995), \ldots, (1991, 1995), \\
& (1985, 1996), \ldots, (1991, 1996), \\
& \qquad\qquad \ldots \qquad\qquad\quad \}
\end{aligned}
$$

The BCDM is a unifying model in that it can be mapped to several existing bitemporal representational data models. BCDM obviously is neither appropriate to present stored data to the user nor to physically store data. [Sno95b] claim, however, that it is a most appropriate basis for expressing time-varying data. They describe mappings to several representational data models, for example, to the temporal data model described in [Sno87].

3.1.2 Attribute Timestamping

This section discusses various temporal data models based on attribute timestamping. Compared to temporal data models using tuple timestamping, these models reduce the redundancy introduced when storing time-varying data. These models also overcome the vertical temporal anomaly. We have not found a proposal using time instants to timestamp attributes.

3.1.2.1 Timestamp is a Time Interval

The temporal data model presented in [CT85, Tan86] supports *historical relations*. A historical relation may have four types of attributes: *atomic attributes*, *triplet-valued attributes*, *set-valued*

attributes and *set-triplet-valued attributes*. Atomic attributes contain atomic values such as integers, reals and character strings. Triplet-valued attributes consist of a valid-time interval $[l \Leftrightarrow u)$, closed at the lower and open at the upper bound, together with an atomic value, $< [l \Leftrightarrow u), value >$. If the upper bound of the valid-time interval is *now*, a closed interval $[l \Leftrightarrow now]$ is used. Set-valued attributes are sets of atomic values, whereas set-triplet-valued attributes are sets of triplets.

These different types allow the modeling of non-time-varying or time-varying attributes which are either atomic or set-valued. The histories of two attributes of the same tuple need not cover exactly the same time period. The nesting depth of a historical relation is at most one – attribute values can only be sets of atomic values or triplets. This allows the storage of *entity histories*, but not of (hierarchical) relationships.

Time intervals of an attribute value history may or may not overlap. Allowing time-intervals to overlap supports the modeling of the acquisition of new skills or the winning of prizes for persons.

Example 3.7 *The relation given in example 2.5 can be modeled the following way: attributes* `EmpID` *and* `Name` *shall be non-time-varying, atomic values, attribute* `Department` *a time-varying, atomic value whereas attribute* `Salary` *is a time-varying set-valued attribute.*

EmpID	Name	Salary	Dept
100	Moira	{<[1994-1996), 15000>, <[1996-now], 20000>}	<[1994-now], IS>
101	Adrian	{<[1996-now], 9000>}	<[1996-now], IS>
102	Alain	{<[1995-now], 9000>}	<[1995-now], IS>
...	...	...	...
107	Tom	{<[1985-1991), 4500>}	<[1985-1991), Math>
107	Tom	{<[1993-1995), 4500>}	<[1993-1995), Math>

3.1.2.2 Timestamp is a Temporal Element

The temporal data model described in [GV85, Gad86, Gad88, GN93] overcomes the vertical and horizontal temporal anomalies they detected in temporal data models using tuple timestamping. As described in chapter 1, *vertical temporal anomaly* denotes the fact that the history of a real world entity is spread over several tuples. There is no way to omit this in temporal data models staying within 1NF (and thus using tuple timestamping). *Horizontal temporal anomaly* refers to the problem that attributes of the same relation change their values at different time instants. For example, the salary of an employee does not necessarily change at the same time as the employee changes from one department to another. One way to deal with this is by decomposing a temporal relation into relations in time normal form (TNF) [NA88, NA89, NA93]. This need of decomposition is called the horizontal temporal anomaly. The horizontal anomaly is usually avoided by increasing the vertical anomaly.

The basic idea behind their model is to store the history of a real world entity in a single tuple. In order to do that, they store the histories of each attribute separately using a non-first-normal-form relational data model, timestamping attribute values with temporal elements.

Their basic temporal data model is *homogeneous*. Homogeneity relates the timestamps of attributes within a tuple with each other. A tuple is called homogeneous if all timestamps of its attributes values cover the same time period. A relation is called *homogeneous* if all of its tuples are homogeneous. This requirement guarantees that a snapshot of a temporal relation will be a relation without null values. This, however, is not always needed nor wanted. So, in [Gad86, GY88], several ways are described to relax this requirement.

In contrast to the data model introduced in section 3.1.2.1, this model does not combine temporal and non-temporal attributes within a tuple. The values of a key are required to be time invariant. Since tuples are homogeneous, all timestamps of attributes cover the same time period. Thus, the timestamp of a key represents the lifespan of the entity stored in the corresponding tuple.

Example 3.8 *The example relation* `Employees` *is a homogeneous relation. Attributes* `Salary` *and* `Dept` *change their values asynchronously. Their value history is stored in a set-valued attribute. So, the history of each employee can be stored in a single tuple.*

Dept	EmpID	Name	Salary
{<{[1994 – *now*]}, IS>}	{<{[1994 – *now*]}, 100>}	{<{[1994 – *now*]}, Moira>}	{<{[1994 – 1996)}, 15K>, <{[1996 – *now*]}, 20K>}
...	...	...	...
{<{[1985 – 1991), [1993 – 1995)}, Math>}	{<{[1985 – 1991), [1993 – 1995)}, 107>}	{<{[1985 – 1991), [1993 – 1995)}, Tom>}	{<{[1985 – 1991), [1993 – 1995)}, 4.5K>}

[TG89, Tan93] also extend a non-first-normal-form relational data model to support the handling of historical data. They do not assume homogeneity in their model. *Temporal sets* of disjoint time intervals are used to timestamp values. A *temporal atom* `<t, v>`, the fundamental construct of their model, contains a temporal set `t` and a value `v`. The history of an attribute is represented as a set of temporal atoms, where attribute values are either atomic values or relations whose tuple components are made up again of temporal atoms.

In their model, the *schema of a historical relation can be viewed as a tree*, where each node either is a leaf of the tree or may contain another subtree. Only leaves carry time information. The time information of all other nodes is defined to be the union of the time information of the nodes of their subtrees. In contrast to their previous work (see section 3.1.2.1) where they restricted the nesting depth to one, such nested historical relations now allow the modeling of the history of (hierarchical) relationships. Additionally, relations contained in another relation can be seen as *timestamped with a temporal element* which corresponds to the union of the timestamps of all tuples contained in it.

Example 3.9 *Figure 3.2 shows the schema tree of a relation* `Departments`*, which contains – among others – the attributes* `DeptName` *and* `Employees`*. Node* `DeptName` *is a leaf in the schema tree and contains temporal atoms. Node* `Employees` *is composed of nodes* `EmpID`, `Name` *and* `Salary`*. This means that attribute* `Employees` *contains relations consisting of attributes* `EmpID`, `Name` *and* `Salary`*.*

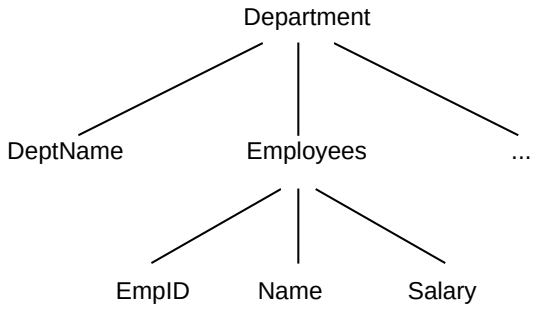

Figure 3.2: Schema tree for relation `Departments`

Department			
DeptName	Employees		
	EmpID	Name	Salary
{<{[1985-now]}, IS>}	{<{[1996-now]}, 100>}	{<{[1996-now]}, Moira >}	{<{[1994-1996)}, 15K>, <{[1996-now]}, 20K>}
	...	...	...
	{<{[1993-now]}, 105>}	{<{[1993-now]}, Andreas >}	{<{[1993-now]}, 9K>}
{<{[1970-now]}, Math>}	{<{[1985-1991), [1993-1995)}, 107>}	{<{[1985-1991), [1993-1995)}, Tom>}	{<{[1985-1991), [1993-1995)}, 4.5K>}

Note that with respect to the previous examples, the relation given above models the existences of the different departments independently of the employees. For example, department IS exists since 1985 and department Math since 1970.

3.1.3 Tuple and Attribute Timestamping

[CT85, CC87, CC93] introduce the *Historical Relational Data Model (HRDM)*. In HRDM, data is timestamped with temporal elements which they call *lifespans*. A lifespan of an object denotes those periods of time during which the database models the properties of that object. For example, the lifespan of an employee object is the time during which data about the employee is stored in the database.

In their papers, they discuss the interesting question *what an appropriate object would be with which to associate such lifespans*. They state that this can be done on the database level, on relation level, on tuple level or on attribute level. By associating lifespans on database level they mean that all relations in the database and with them all tuples and attribute values have the same valid-time period. In other words, they timestamp a database and assume *homogeneity* throughout all the levels. This does not buy very much, but only because they assume homogeneity.

In our opinion, it makes perfect sense to timestamp a database, however *without* assuming homogeneity. In this way, it is possible to model, for example, that a database was created at a certain time instant and may be replaced at a later point in time, letting the data contained in it have their own lifespans which of course have to be within the lifespan of the database. The database is not no longer physically deleted.

The same holds for timestamping on the relation level. Again, their interpretation is that the lifespan associated with a relation also denotes the time period during which all tuples in the relation were valid. In this case, assuming homogeneity again restricts the expressiveness in an unnatural way. We argue that timestamping relations is a very powerful concept. Since a relation is also created at some point in time and possibly dropped later, it has a lifespan itself. The tuples contained in it, however, do not necessarily need to have the same lifespan, but are restricted to be contained in it.

With the view of associating lifespans with objects at different levels, assuming homogeneity, their conclusion is that they should associate lifespans on the tuple level for more flexibility. Additionally, they timestamp attributes within the relation schemas. This provides for the possibility of evolving schemas. The lifespan of an attribute in a schema denotes the time period during which the attribute was part of the schema. The lifespan of a particular attribute value in a relation then is limited both by the lifespan of the tuple and the lifespan of the attribute in the schema.

3.2 Temporal Entity Relationship Data Models

The entity-relationship model (ER model) [Che76] is a semantic data model. Where the relational data model describes a database interface, the ER model provides concepts and formalisms for the description of the semantics of a mini-world in discourse.

In the ER model, application domains are viewed as consisting of *entities* and *relationships* among entities. *Cardinality constraints* allow the specification of how often two entities may be related with each other.

An entity is described by its attributes. The type of an entity thus consists of the specification of the entity's attributes. Attributes may be atomic or composite. A *single-valued* attribute has at most one value at a time instant, where as *multi-valued* attribute contains a set of values of the same type. An *entity set* contains entities of the same type.

The ER model is used to model application domains on a higher level. This is usually called the *conceptual modeling* of a database. The resulting design is mapped to a data model of a specific DBMS, for example, to the relational data model.

This section has a look at the temporal extensions proposed for the ER model. First, temporal ER models which timestamp entities are discussed, then, models which timestamp attributes of entities.

3.2.1 Entity Timestamping

Entity timestamping is similar to tuple timestamping. When mapping an entity to the relational data model, it is modeled as a tuple. The temporal ER model introduced next thus also suffers from the vertical temporal anomaly. We have not found temporal ER models which use time instants or temporal elements to timestamp data.

Timestamp is a Time Interval

[Wuu91] enhance an *extended entity relationship model (EER)* to support temporal data in future planning databases. In their model, entities are timestamped using time intervals. They call this approach *object versioning*. A *temporal entity* is represented by a collection of sequenced versions. Each of these versions has an attribute `valid_time` containing a time interval. An *entity version* stores a state of the temporal entity during a specific valid-time period.

Versions of the same entity cannot overlap in their valid-time intervals. The set of versions spreads the data about a real world entity over several version entities. This corresponds to what was earlier introduced as the vertical temporal anomaly.

Each temporal entity has an attribute called *lifespan*. The union of the valid-time intervals of versions of the same temporal entity needs to be contained in the lifespan of the temporal entity. Homogeneity is not required. Their model supports the concept of temporal relationships between entities, where each relationship may be versioned in the same way as temporal entities.

3.2.2 Attribute Timestamping

Similar to the relational model, there exist proposals for ER models, extended to handle time-varying data, which timestamp data on the attribute level. The only kind of timestamp for which we have found proposals is the temporal element. To the best of our knowledge, there are no temporal ER models applying attribute timestamping with either time instants or time intervals.

Timestamp is a Temporal Element

[EW90, EWK93a] also propose a temporal extension for an EER model (TEER). They incorporate the concept of *lifespan for entities and relationships* in the ER model which is a set of disjoint time intervals. A lifespan of an entity thus is a temporal element $T(entity) \subseteq [0, now]$. Each entity has a system-defined *surrogate attribute* whose value is unique for every entity in the database. The temporal element of the surrogate attribute defines the entity lifespan. Thus each real world object is member of an entity set and has a surrogate attribute specifying the object's lifespan.

A subclass can either be specified via a predicate or explicitly by the user. In the former case, each entity in the superclass that satisfies a defining predicate will be a member of the subclass. An entity of the superclass belongs to the subclass throughout all time intervals when the predicate evaluates to true for that entity. In the latter case, the user assigns an entity from the superclass to become a member of the subclass. Additionally, he specifies the time points the entity is to be made a member. In both cases, an entity may only be member of a subclass when it is also member of the superclass.

The temporal value of each attribute of an entity is a partial function from temporal element $T(entity)$ to the domain of the attribute.

Example 3.10 *Assume* $\texttt{ID}_i$ *to be a surrogate. A relation* `Employees`*, similar to the one given in example 2.5, then can be modeled in TEER in the following way:*

SURROGATE	EmpID	Name	Salary
$\{\{[1994 - now]\} \rightarrow ID_0\}$	$\{\{[1994 - now]\} \rightarrow 100\}$	$\{\{[1994 - now]\} \rightarrow$ Moira$\}$	$\{\{[1994 - 1995]\} \rightarrow 15000,$ $\{[1996 - now]\} \rightarrow 20000\}$
$\{\{[1996 - now]\} \rightarrow ID_1\}$	$\{\{[1996 - now]\} \rightarrow 101\}$	$\{\{[1996 - now]\} \rightarrow$ Adrian$\}$	$\{\{[1996 - now]\} \rightarrow 9000\}$
$\{\{[1995 - now]\} \rightarrow ID_2\}$	$\{\{[1995 - now]\} \rightarrow 102\}$	$\{\{[1995 - now]\} \rightarrow$ Alain$\}$	$\{\{[1995 - now]\} \rightarrow 9000\}$
...	...	...	...
$\{\{[1985 - 1990],$ $[1993 - 1994]\} \rightarrow ID_7\}$	$\{\{[1985 - 1990],$ $[1993 - 1994]\} \rightarrow 107\}$	$\{\{[1985 - 1990],$ $[1993 - 1994]\} \rightarrow$ Tom $\}$	$\{\{[1985 - 1990],$ $[1993 - 1994]\} \rightarrow 4500\}$

In [EEAK90], the notion of *conceptual objects* are introduced as an extension to TEER. In their approach, an entity is a persistent object that, once its existence becomes known, is never deleted. Each conceptual entity has an *existence time* which is unrelated to the concept of lifespan. The upper bound of an entity's existence time is always infinity. The lower bound is the time the entity is materialised. Non-temporal attributes can only be properties of conceptual entity types, since they hold over the entire existence time of the object. The temporal aspect of the entity – the history of its attribute values – is modeled as *entity roles*. A *role type* is a set of entity roles of the same type. An entity role has all the time-varying attributes.

Their motivating example for conceptual objects is the following. An employee exists in the real world as a person. An employee is of interest to a company only when they hire him. In this case, they may want to record previous information about this person, or if the employee leaves the company, the employee remains an object of interest. This is depicted in figure 3.3. Person objects have the non-temporal attributes which are the name of a person and his social security number. Employees are modeled as entity roles, having the salary and the department as time-varying attributes.

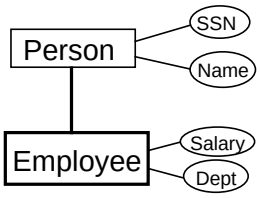

Figure 3.3: Conceptual entity type `Person` and role type `Employee` in TEER

We feel that the approach of conceptual objects is a rather cumbersome way to store temporal data about entities playing different roles. It is also not clear, why the upper bound of the existence time is always infinity. However the idea of objects playing different roles over time is important.

3.3 Temporal Object Data Models

Proposed temporal object data models mainly take the same approach as the temporal relational data models and temporal ER models introduced in the previous sections – data may be time-stamped on object or attribute level. Many proposals use the schema extension approach. They add special timestamp attributes to the types of objects. Additionally, however, the intrinsic possibility of object-oriented data models to extend their functionality can be used. None of the models is generic. Several approaches specify abstract data types for time and use them to model temporal application domains.

Since there is no single object-oriented data model, categorising the proposals is quite difficult. So they are discussed according the kind of timestamp they use.

3.3.1 Object Timestamping

Object timestamping is similar to tuple timestamping and thus has the same problems. Flat relations spread the history of a single real world entity over several tuples, as seen before. In the following, temporal object data models are described which try to overcome these deficiencies – either by using a new data structure or by reducing the data redundancy.

3.3.1.1 Timestamp is a Time Instant

[KRS90] describe a temporal data model which overcomes the vertical temporal anomaly. Their idea is to map *time sequences* [SK86] to a complex object data model. A time sequence represents

the history of an entity of the real world by preserving the the different states of the entity over time in a time-ordered sequence.

Example 3.11 *The following time sequence represents Moira's employment history in the university. Attribute* `Valid` *specifies the time point when the values in the tuple become valid. Attribute* `Timestamp` *contains the transaction time, for example, when a transaction modified the tuple.*

Valid	EmpID	Name	Salary	Dept	Timestamp
< 1994,	100,	Moira,	15000,	IS,	1994 >
< 1996,	100,	Moira,	20000,	IS,	1996 >

A time sequence belongs to exactly one time sequence relation and is the unit for modification and data retrieval as the tuple is in the relational model. In [KRS90], time sequences are used on a conceptual level only. They propose to map them to a data model which allows an easy and efficient implementation. They identify the *complex object data model MAD* (molecule-atom data model) [Mit88, Mit89] to be a well-suited candidate.

The MAD model supports *atoms* and *atom types*. Compared with the relational model, an *atom* corresponds to a tuple, whereas an *atom type* contains atoms and thus is similar to a relation. Additional to the usually supported data types, the MAD model supports the data types `TIME`, `IDENTIFIER` and `REFERENCE`. An attribute of type `TIME` contains a time instant of a specified granularity. Data type `IDENTIFIER` represents a system defined surrogate which uniquely identifies an atom. Data type `REFERENCE` is needed to link atoms together. A value of type `REFERENCE` is a duplicate-free list of `IDENTIFIER` values, all pointing to atoms of the same type [KRS90]. In the MAD model, a time sequence then is modeled as a sequence of tuples linked by special attributes of type `REFERENCE`.

Example 3.12 *The time sequence given in example 3.11 is translated into the following atom type:*

```
ATOM_TYPE Employee (
  ID        : IDENTIFIER;
  Valid     : TIME(YEAR);
  EmpID     : INTEGER;
  Name      : STRING;
  Salary    : INTEGER;
  Dept      : STRING;
  future    : REFERENCE(Employee.past) [0,1];
  past      : REFERENCE(Employee.future) [0,1];
  alive     : BOOLEAN;
  timestamp : TIME(SECOND));
```

Attributes `ID`, `future`, `past`, `alive` *and* `timestamp` *are added automatically and are not visible for the user. The references* `future` *and* `past` *model the sequence of the history. In the case that attribute* `alive` *is false, the valid time denotes the time of death of the time sequence. Otherwise,* `valid` *is the starting time of the validity of the other attributes.* `YEAR` *and* `SECOND` *specify the granularity used for the time attributes. Cardinality constraints* $[0,1]$ *restrict the historical references to at most one other atom, namely the previous and the next state of the object.*

3.3.1.2 Timestamp is a Time Interval

Instead of using time sequences, [KS92b] propose a direct extension for the data model MAD. They use the same ideas already presented in section 3.3.1.1. This new temporal complex object data model TMAD also consists of atoms and atom types. The difference is that now a valid-time interval is used instead of a time instant to timestamp the atoms.

Example 3.13 *In TMAD, a schema extension of an atom type is used in order to capture historical data. In contrast to example 3.12, a valid-time interval is used for timestamping which makes attribute* `alive` *obsolete.*

```
ATOM_TYPE Employee (
  ID          : IDENTIFIER;
  EmpID       : INTEGER;
  Name        : STRING;
  Salary      : INTEGER;
  Dept        : STRING;
  future      : REF_TO(Employee.past) [0,1];
  past        : REF_TO(Employee.future) [0,1];
  valid_from  : TIME(YEAR);
  valid_until : TIME(YEAR);
  timestamp   : TIME(SECOND));
```

[SC91] extend the object-oriented data model OSAM* [SLK89] with time. In OSAM*/T, they timestamp objects with a time interval [`Start_time`⇔`End_time`]. The data redundancy introduced when timestamping objects is avoided using the so called delta file concept [Roc75, SL76]. The most recent version of an object contains all attribute values, whereas the predecessors only store those attribute values that changed. So, accessing the current version of the object does not cause any overhead. Historical data of the object is searched from the current object instance to the historical data area. So, similar to the approach presented in HDM [Sar90b, Sar93], they separate current from historical data thereby neglecting the possibility to store future object states.

Attributes `Start_time` and `End_time` are the only two time notions they support. Other time notions such as transaction time or any other user-defined time are added directly to single objects in form of *temporal rules*. This has the advantage over adding additional time attributes to the type of a class of objects, that storage space can be saved where these notions of time are not relevant for objects in the class. The disadvantage, however, is that the query response time will increase due to the additional execution of the rules. Each rule has a timestamp denoting the time when the rule is active.

In OSAM*/T, an object may be in any number of different classes simultaneously. An instance is the representation of an object in a specific class and contains the attribute values that characterise that class. Each object instance has its own history. Additionally, OSAM*/T supports association histories.

Example 3.14 *The historical data given in example 2.5 would be modeled in OSAM*/T the following way:*

	EmpID	**Name**	**Salary**	**Dept**	**Start_time**	**End_time**	
<	100,	Moira,	20000,	IS,	1996,	now	>
<	#,	#,	15000,	#,	1994,	1995	>
	...	...	...	...	...	...	
<	107,	Tom,	4500,	Math,	1985,	1990	>
<	107,	Tom,	4500,	Math,	1993,	1994	>

The special value # is used instead of repeating redundant values. In Moira's employment history, only the salary has changed. The most recent object contains values for all attributes.

3.3.2 Attribute Timestamping

[GÖ93] use the extensibility of the objectbase management system *TIGUKAT* [PÖS92] to implement a *type lattice for time*. This type lattice specifies an extensible set of ADT and a rich set of behaviours to model time. For example, three different kinds of timestamps are supported – time instants, time intervals and durations of time – and types to model discrete, continuous and

dense time. These types then can be subtyped to model, for example, timestamps with different granularities.

TIGUKAT is behavioural in the sense that all access and manipulation of objects is based on application of behaviours to objects. The primitive objects of the model include atomic entities, types, behaviours, functions, classes and collections. *Atomic entities* are values of the generally supported types real, integer, string and so on. *Types* are used to define the common features of objects. *Behaviours* specify the semantics of operations that may be performed on objects. *Functions* are the implementations of behaviours over types. *Classes* classify objects according to their type and are used during object creation. They are controlled by the system. *Collections* support general groupings of objects. They are under the control of the user. An object may be in several collections but only member in the shallow extent of one class.

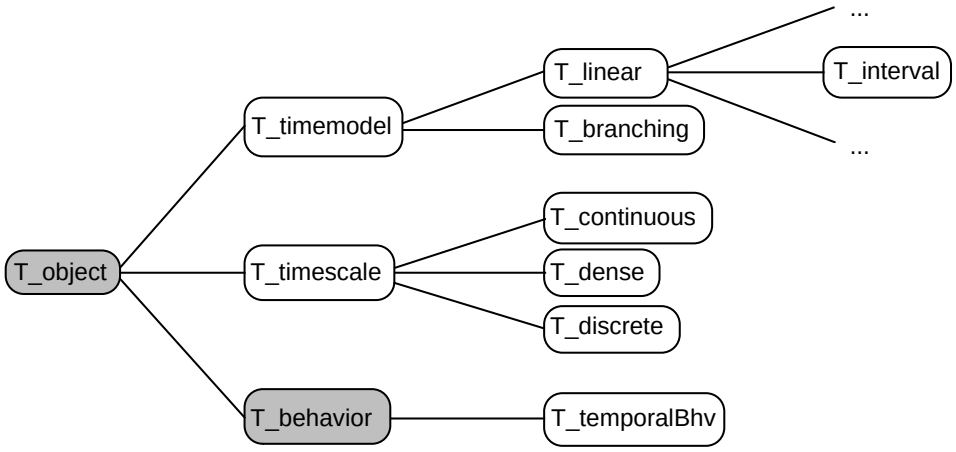

Figure 3.4: Part of the type lattice for time in TIGUKAT

Figure 3.4 shows part of the type lattice to support time in TIGUKAT. The shaded types belong to the primitive type lattice, whereas the unshaded ones are abstract time types.

TIGUKAT supports different time models. For example, type `T_linear` is a subtype of `T_timemodel`. Type `T_timemodel` has a behaviour `B_timescale` which returns a collection of `T_timescale` objects. This behaviour is overwritten in `T_linear` to return a *list* of `T_timescale` objects, since the linear model of time represents a total ordering of time, representing the time flowing from the past to the future in a totally ordered way. Branching time is supported with type `T_branching`, and allows the representation of several futures states branching out at time instant *now*. Prior to *now*, time is linear.

Type `T_interval` contains behaviours which allow the comparison of an object's time interval with another time interval using one of the comparison predicates introduced in section 2.1.5. An object then can be timestamped, for example, using objects of type `T_interval`.

Type `T_timescale` defines the different models of time – continuous, dense and discrete time. Instances in `T_temporalBhv` maintain a history of updates with respect to a particular object. Behaviour `B_history` is a behaviour in the interface of type `T_temporalBhv`. An instance of `T_temporalBhv` is called a *temporal behaviour*.

Example 3.15 *Assume object Moira to be a member in class* `C_employee` *of type* `T_employee`. *Additionally assume* `B_salary` *to be a temporal behaviour defined in the interface of* `T_employee`. *Then the expression* `B_salary.B_history(Moira)` *returns the salary history of employee Moira.*

While TIGUKAT allows the modeling of almost any important concept found in the temporal database literature, it lacks of an intuitive and comprehensible notation. Additionally, it is not clear how expressive this model is with respect to queries.

Another approach using time sequences is presented in [RS91, RS93]. [RS91, RS93] extend a basic object-oriented data model with new types, as depicted in figure 3.5. The shaded types are the system-defined ones. An object is either extended to a class, a value of a primitive type or a collection. A collection is, for example, a set or a sequence of objects of a type `T`.

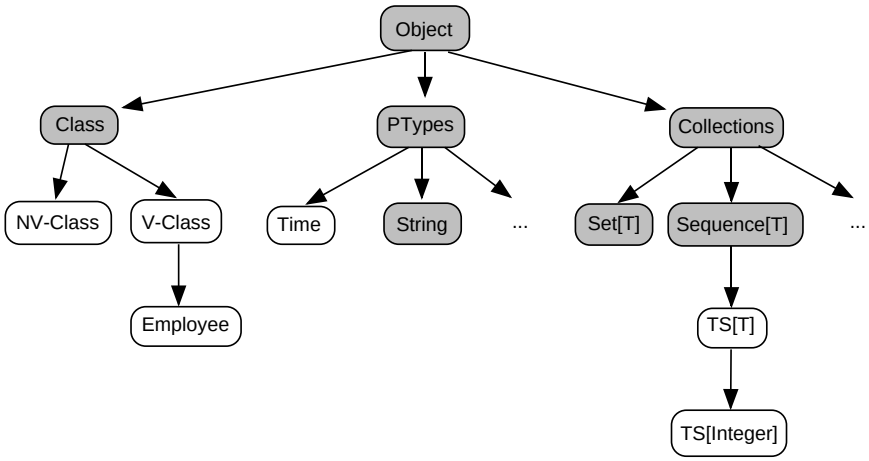

Figure 3.5: Type lattice in TOODM

In their temporal object-oriented data model *TOODM*, the system-defined types are subtyped with the new classes `NV-Class`, `V-Class`, `Time` and `TS[T]` for temporal support. Types `NV-Class` and `V-Class` represent *non-versionable* and *versionable* classes. A user-defined type can be a subclass of either. A history of the class definitions is kept of all classes defined as subtypes of `V-Class`. The special collection `TS[T]` is called a *time sequence* [SK86, SS87]. A time sequence object contains a sequence of pairs of values or instances of type `T` and a time value having type `Time` or one of its specialisations.

Example 3.16 *Class* `Employees` *in TOODM is defined as a subclass of class* `V-Class`*. A possible class definition for employees would be*

```
Define Employee as subclass of V-Class
  surrogate      : OID
  empid          : Integer
  name           : String
  salary-history : TS[Integer]
  ...
```

The next temporal object data model to be discussed is called *T_Chimera* [BFG96]. It is based on the idea to extend all data types of the underlying model Chimera [GBB94] into temporal data types. [BFG96] introduce the notion of *temporal types* to handle in a uniform way temporal and non-temporal domains. For each type in Chimera, a temporal type and a set of legal temporal values is defined in T_Chimera.

The value of a variable of temporal type `temporal(T)` can be represented as a set of pairs `(t, f(t))` where `f` is a partial function and `t` is a time instant. Since values do not change at each time instant, the variable's value can be represented more concisely as a set of pairs

$$\{\texttt{<I}_1\texttt{, V}_1\texttt{>, ..., <I}_n\texttt{, V}_n\texttt{>}\}.$$

where $\texttt{I}_1$, ..., $\texttt{I}_n$ are time intervals, closed on the lower and upper bound, and $\texttt{V}_1$, ..., $\texttt{V}_n$ are values of type `T`.

Classes in T_Chimera may be static or historical. In case one of the class attributes has a temporal type, the class is considered to be historical. Otherwise it is static. A class contains information for the use of the class and its instances. For example, the set of attributes of the class instances is given as a class attribute. Each attribute is specified as a (name, type) pair. Each class also contains a *history* class-attribute containing the history of class membership of class instances. Additionally, each class has a lifespan as a class attribute.

Similar to classes, objects are historical if at least one of its attributes is of a temporal type. Objects also have a lifespan attribute and contain the history of their class memberships.

Example 3.17 *The salary history of an employee can be modeled in T_Chimera as a temporal integer type* `temporal(integer)`*. His employment number and name are modeled as attributes of a non-temporal integer and string type, respectively. The class* `Employees` *containing instances of employee objects is then described as follows:*

```
c = Employees
type = historical
lifespan = [1985, now]
attr = { (EmpID, string), (Name, string), (Salary, temporal(integer)), ... }
...
history = (number_of_employees : { <[1985,1988], 1>, <[1989,1990], 2>, ...},
           ext : { <[1985,1990], Tom>, <[1989,1995], Martin>, ...},
           ... )
```

Attribute `c` *specifies the name of the class. Attribute* `type` *denotes that the class is historical since one of its class attributes, namely* `number_of_employees`*, is temporal. Attribute* `attr` *lists the properties an instance of class* `Employees` *has. Attribute* `history` *contains the class attribute* `number_of_employees`*, and the membership history of class instances in* `ext`*.* `Tom` *and* `Martin` *are actually object identifiers referencing two instances of this class.*

3.3.3 Tuple and Attribute Timestamping

[DW92, WD93] also model time as an ADT and use it to extend object-oriented data model OODAPLEX [Day89] which is based on the functional data model DAPLEX [Shi81]. In their approach, time is modeled as a generic type that carries most general semantics of time. The different notions of time such as discrete, dense or continuous are modeled as subtypes of type time.

In OODAPLEX, properties of objects, relationships among objects and operations on objects are all uniformly modeled by *functions* which are applied to objects. Objects that have similar properties and behaviour are grouped into types. So a type specifies a set of functions that can be applied to instances of the type.

Time-varying properties of objects, relationships among objects or behaviour are modeled in OODAPLEX by functions that return another function that maps time elements into snapshot values. OODAPLEX has the flexibility to support both attribute and object timestamping.

Example 3.18 *We model temporal employee objects having – among others – a name, a salary and a department property, in OODAPLEX. Attribute timestamping using the time ADT can be achieved the following way:*

```
type Employee is object
  function name(e: Employee -> n: String)
  function salary(e: Employee -> f:(t: Time -> s: Money))
  function dept(e: Employee -> f:(t: Time -> d: Department))
  ...
```

Function `name` *models an attribute which is constant over time. Functions* `salary` *and* `dept` *model time-varying attribute values – the salary history of an employee and the history of memberships in different departments.*

It is also possible to use object timestamping in OODAPLEX. In this case, temporal employee objects are modeled with a state function that maps from time to snapshot states of employees. A snapshot state is modeled as a conventional, non-temporal employee type:

```
type Employee is object
  function state(e: Employee -> f: (t: Time -> s : Snapshot_Employee))

type Snapshot_Employee
  function name(e: Snapshot_Employee -> n: String)
  function salary(e: Snapshot_Employee -> s: Money)
  function dept(e: Snapshot_Employee -> d: Department)
  ...
```

They also support a *lifespan* function for objects. It is a polymorphic function accepting both a type `T` and a database `DB` as input parameters. The function `lifespan(T)` returns the *union of lifespans of all objects in the extent* of type `T`. The function `lifespan(DB)` return the *union of lifespans of all types* in DB.

Using this lifespan function, they define several basic constraints between the lifespans of objects, types and a database. A first constraints demands that

$$\texttt{lifespan(o/T)} \subseteq \texttt{lifespan(T)} \subseteq \texttt{lifespan(DB)}$$

where `lifespan(o/T)` denotes the lifespan of an object `o` as an instance of type `T`. This constraint is actually superfluous since the definition of the `lifespan` function always guarantees that this constraint holds. For temporal attributes, a second constraint is defined, demanding that their values histories need to be restricted to the lifespan of the object. A third constraint requires that an object's lifespan as an instance of a subtype is contained in the lifespan the object is also an instance of the corresponding supertype. They define more constraints which, however, follow from the three constraints discussed above.

3.4 Summary

[Cli82, CW83] describe a temporal data model which supports historical relations. A historical relation is viewed as a sequence of relation instances indexed by valid time. [JMS79] use schema extension for tuple timestamping with time intervals. [NA88, NA89, NA93] introduced the Time Normal Form in their temporal data model TRM. They propose to decompose relations containing asynchronous attributes. TQuel [Sno84, Sno87, Sno93] defines the semantics of queries and update operations using snapshot reducibility. HDM [Sar90b, Sar93] separates current and historical data physically into two relations. BCDM [Sno95b] is a conceptual data model which can be mapped to other temporal data models. [CT85, Tan86] store entity histories in NFNF relations but restrict the nesting to one level. This is given up in the model proposed in [TG89, Tan93], where they allow arbitrary nesting of historical relations. [GV85, Gad86, Gad88, GN93] discuss the vertical and horizontal temporal anomalies and propose a temporal data model which overcomes these deficiencies. [CT85, CC87, CC93] discuss different levels of timestamping data, but under the assumption of homogeneity. Their temporal data model HRDM provides for the possibility of evolving schemas. [Wuu91] stores the histories of entities as sequences of versions. In [EEAK90], the concept of conceptual objects is introduced which allows a simple form of temporal role modeling in an temporal EER model. [KRS90, KS92b] extend the complex object data model MAD to support historical data. Their model using references to past and future states allows the modeling of the history of a real world entity as a single complex object. In OSAM*/T [SC91], an object can

appear in different classes simultaneously, where each instance has its own history. Additionally, temporal rules are supported which allow the specification of other time notions for specific objects. OSAM*/T uses object timestamping and a special form of storage to reduce the data redundancy. [GÖ93] specify a type lattice of abstract data types for time in the object data model TIGUKAT. These types and behaviours can be used to model different kinds of timestamps, for example, time instants and time intervals, and different models of time, namely dense, continuous and discrete time. TOODM [RS91, RS93] also extend a type lattice. Historical attributes are modeled as time sequences containing objects of a specific type. T_Chimera [BFG96] propose a temporal object data model by introducing temporal types. [DW92, WD93] propose an ADT for time for the functional data model OODAPLEX.

All models described in this section use – in one way or another – the schema extension approach. Additionally, proposals were introduced which discuss the timestamping of data on different levels. HDM [Cli82, CW83] timestamps relations with time instants. The several other approaches use tuple and object or attribute timestamping. The most flexible approach is proposed in [TG89, Tan93], where data is timestamped on attribute level, and each attribute may contain again a relation using attribute timestamping. This provides for timestamping on different levels of nesting. The ideas used for temporal object data models are similar to those already used in 1NF and NFNF relational data models.

An important topic is the *vertical temporal anomaly*. Several solutions were proposed to overcome this problem. One is to support a special coalescing operation or assume relations to be automatically coalesced. Another one is using a NFNF relational data model, where the history of an entity can be stored within a single tuple using sets of timestamped attributes. Some temporal object data models use a similar approach called time sequences. The temporal data models proposed in [Gad88, GY88, Sno87, Tan86, NA88, NA89] are examples which do not represent the history of a real world entity in a single tuple.

The question whether or not to *extend a data model* with temporal semantics or to implement it as an *abstract data type* is another issue. This chapter also introduced approaches proposing ADT instead of changing the underlying data model. A question not discussed in these proposals is efficiency.

Chapter 4

A Temporal Relational DBMS : TimeDB

This chapter describes the prototype temporal DBMS *TimeDB* [Ste95] and the translation algorithm developed to map *temporal queries*, *temporal data definition and modification statements* and *temporal constraints* into *standard SQL statements*. First, an overview of the features of TimeDB, its historical background and a short introduction to the concepts and ideas behind ATSQL2, the language implemented in TimeDB, are given. Next, the translation of valid-time queries, specified using ATSQL2, into standard SQL statements is described. Then the semantics of bitemporal algebra operations are discussed where they are different from the unitemporal ones. Finally, the approach to check temporal constraints used in TimeDB is described, and the architecture of TimeDB is presented.

4.1 Features of TimeDB

On one hand, TimeDB uses the extension approach with respect to the data structures, on the other hand, however, it generalises the query, modification, data definition and constraint specification language based on SQL. According to the classification introduced in this thesis, the implementation of an extension or generalisation of a language such as SQL is based on changes done to the DBMS software. The temporal DBMS TimeDB, however, uses a *layered* approach which means that it was built as a front-end to a commercial DBMS which translates temporal statements into standard SQL statements. This way, it is possible to support features such as persistence, concurrency, recovery without having to implement them from scratch, since there is no access to the source code of a commercial DBMS.

The translation algorithm implemented in TimeDB is a general one in the sense that it can be used to translate different temporal query and modification languages into standard SQL statements. It was developed for the bitemporal query language ChronoSQL (see e. g. [Pul95]), and then was reused in TimeDB for the language ATSQL2 [BJS95, SBJS96b, SBJS96a]. ATSQL2 is a temporally complete query language which was designed collaboratively by an international group of researchers. It is based on SQL and supports temporal updates, temporal views and temporal assertions, table- and column-constraints.

TimeDB is a *bitemporal* DBMS. It can handle valid-time operations, transaction-time operations and operations referring to both time lines simultaneously. Operations referring to only one time line are denoted as *unitemporal*. The implementation of the unitemporal algebra operations and the translation of the valid-time ATSQL2 queries to standard SQL statements will be discussed in the following sections. Transaction-time queries can be evaluated using the same algorithms due to the orthogonal treatment of valid time and transaction time. Bitemporal queries are translated using the same translation algorithm, however need different implementations of some algebra operations.

As the reader will see, the concepts used in the query translation process of TimeDB can also be used for translating modification statements and temporal constraint checking operations.

4.2 The History of TimeDB and ATSQL2

During the last 15 years, many temporal query languages for relational DBMS have been proposed. However, almost none of these languages have been implemented. [Böh95] lists 13 implementations of systems which either qualify as temporal DBMS or are somehow related to temporal databases (for example, a temporal database generator). Out of these, only two are available prototypes which implement temporal query languages based on SQL. A closer look at these implementations reveals two important restrictions found in most of the temporal data models and systems. First, most of the implementations focus on the query language, neglecting updates, rules (views) and integrity constraints. Second, with respect to the query language, a lot of work has been done on temporal selections and joins, but almost nothing on temporal negation.

The bitemporal relational DBMS TimeDB implements the language ATSQL2 [BJS95, SBJS96b, SBJS96a] which is based on the standard query language SQL[Dat89, MS93]. ATSQL2 includes not only a bitemporal query language, but also a bitemporal modification, data definition and constraint specification language. It is the result of integrating three different approaches, namely

- TSQL2 [Sno95b], a temporal query language based on SQL,
- ChronoLog [BM94, Böh94], introducing the concept of temporal completeness
- Bitemporal ChronoSQL, developed by the author of this thesis (see, for example, [Pul95]), featuring a bitemporal query language.

TimeDB thus supports a temporal complete query language based on SQL, temporal modification statements, temporal views and temporal integrity constraints. Since the query language is temporally complete, it features temporal versions of all algebra operations, including negation. So, TimeDB can be considered the first complete temporal DBMS implementation.

In chapter 3, the data model of TSQL2 was introduced. TSQL2 was a first attempt to specify a standard of a temporal query language. Many researchers from the area of temporal databases have contributed to this language. The idea was to consolidate different approaches to temporal data models, to achieve a consensus query language and an associated data model upon which future research could be based.

In 1994, Richard Snodgrass started working with the ANSI and ISO SQL3 committee to propose a new part to SQL3, termed SQL/Temporal. This was formally approved in July, 1995. The idea was to use the TSQL2 for the SQL/Temporal standard. There were several concerns voiced about the TSQL2 design, however. For example, in TSQL2, duplicates were not allowed due to the implicit enforcement of coalescing, the syntax was confusing, there was no formal semantics and no implementation.

The notions of temporal semi-completeness and completeness introduced in [BM94, Böh94] were used to evaluate the completeness of TSQL2 [BJS95] which lead to a major redesign of the language. This work was started by Michael Böhlen, Christian Jensen and Richard Snodgrass. The new language which evolved was called ATSQL2 (Applied TSQL2).

In 1994, the author of this thesis developed a bitemporal DBMS called bitemporal ChronoSQL which, on one hand, treated valid-time queries and transaction-time queries orthogonally, and, on the other hand, supported bitemporal queries. In 1995, the author of this thesis was invited to participate in the ATSQL2 project. During this collaboration, he migrated his prototype system ChronoSQL to TimeDB. Besides the influence of bitemporal ChronoSQL on the treatment of valid time and transaction time in temporal statements of ATSQL2, the implementation also helped refining and clarifying the language ATSQL2 as a whole and supplied the temporal database community with a complete temporal DBMS.

In 1996, ATSQL2 has been proposed for the SQL/Temporal standard. The two change proposals [SBJS96b, SBJS96a] were unanimously accepted by ANSI and forwarded to ISO. These proposals have not yet been voted on by the ISO committee.

4.3 ATSQL2

In this section, a short introduction to the main ideas and concepts behind ATSQL2 [BJS95, SBJS96b, SBJS96a] is given. TimeDB implements ATSQL2 as it was proposed for the SQL/Temporal standard in [SBJS96b, SBJS96a]. First, the requirements which served as the basis for the definition of ATSQL2 are listed. Then it is shown how SQL [Dat89, MS93] was extended to ATSQL2 to fulfill these requirements.

4.3.1 Requirements for ATSQL2

As mentioned previously, ATSQL2 is an extension of SQL. It defines syntax and semantics of the temporal query, modification and constraint specification language.

When extending SQL to ATSQL2, one focus was to give maximal support for migrating a non-temporal database to a temporal database. The notion of *upward compatibility* [BJS95] and *temporal upward compatibility* [SBJS96b] describe the requirements for ATSQL2 with respect to database migration. Upward compatibility demands that SQL is a subset of ATSQL2. So every legal SQL statement may also be used with the same semantics in ATSQL2. Temporal upward compatibility specifies requirements to the query and modification language of ATSQL2 after a non-temporal database has been migrated to a corresponding temporal database. It demands that each legal SQL query and modification statement, executed on a temporal database, leads to the same result as if it were executed on the corresponding non-temporal database.

Another focus when designing ATSQL2 was to make the transition from SQL to ATSQL2 also easy for programmers. Non-temporal and temporal queries and modification statements must be *syntactically similar*. According to the definition of *temporal completeness*, two queries q and q^t are syntactically similar if there exist two possibly empty strings S_1 and S_2 such that $q^t = S_1\ q\ S_2$. A programmer thus can write a non-temporal SQL statement and turn it into a temporal one by simply adding a keyword in front of the non-temporal query, for example.

In ATSQL2, temporal statements are classified into two classes – *sequenced* and *non-sequenced* statements [SBJS96b]. Sequenced statements have snapshot reducible semantics. So, the result of a sequenced query is equivalent to the sequence of the results of a corresponding non-temporal query evaluated on each database state. This means that a particular state of the result is derived solely from the database state at the same time instant. This is depicted in figure 4.1.

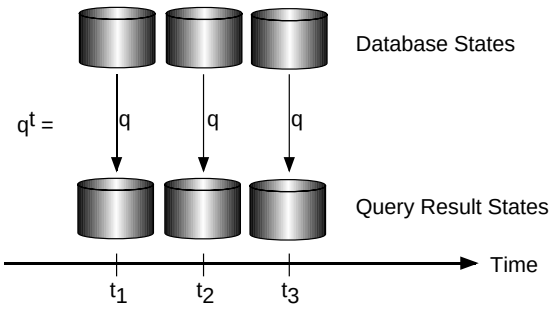

Figure 4.1: A sequenced query q^t uses corresponding database states for each resulting state

There exists another important class of statements which requires several database states to be examined, for example, when comparing different database states with each other or when querying database state transitions. This kind of statement is called *non-sequenced*. Figure 4.2

shows possible references to different database states to calculate a query result. A single statement may be composed of both sequenced and non-sequenced statements.

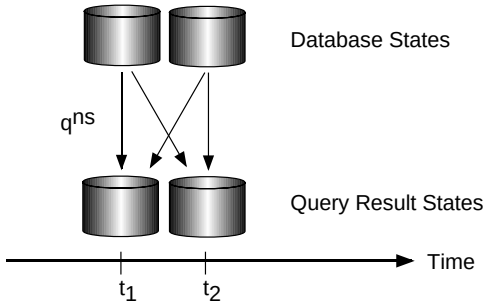

Figure 4.2: A non-sequenced query q^{ns} may use several database states for a single resulting state

Since ATSQL2 supports both valid time and transaction time, the step from a non-temporal to a temporal statement may be done either with respect to valid time or transaction time or both. Thus, another important requirement is that these different time lines are treated orthogonally. *Orthogonality on valid time and transaction time* demands that any query may be evaluated with respect to valid time, transaction time or both. So, for each valid-time query, a transaction-time or bitemporal query exists. This approach was first used for bitemporal ChronoSQL [Pul95].

4.3.2 The Query Language of ATSQL2

The requirements of upward compatibility, temporal upward compatibility and orthogonality on valid time and transaction time together with the requirements listed in the definition of temporal completeness – syntactical similarity, sequenced and non-sequenced semantics of statements, substitutability of a relation in a query by another query and the support of both Allen's comparison operators and coalescing – served as guidelines during the design of ATSQL2. These goals were achieved mainly by adding four extensions to SQL.

First, flags were introduced to be able to express which one of four possible semantics – *upward compatibility*, *temporal upward compatibility*, *sequenced* and *non-sequenced* semantics – should be used to evaluate an ATSQL2 statement. Second, ATSQL2 supports the concept of *derived tables*. Derived tables, as defined in SQL3 [SQL93], allow a table in the `FROM`-clause to be substituted by a query. In ATSQL2, any table (temporal or non-temporal) may be substituted by a query. Third, a set of temporal comparison operators is supported having the same expressive power as those proposed by [All83]. Fourth, a keyword is added to express if a (intermediate) result should contain only tuples with maximal time periods or not (*coalescing*). With these extensions, it is possible to fulfill all the requirements described above.

The following subsections discuss in more detail the four different levels of functionality supported in ATSQL2. These levels are introduced following a migration process going from non-temporal SQL statements to temporal ones stated in ATSQL2.

4.3.2.1 Upward Compatible Queries

As mentioned before, *upward compatibility* demands that any non-temporal SQL statement and any non-temporal table or view can still be used in ATSQL2. Thus, TimeDB contains SQL as a subset. The advantage of supporting upward compatibility is the fact that legacy data and code can still be used when migrating a non-temporal database application to a temporal DBMS. The following example uses standard SQL statements which are part of TimeDB.

Example 4.1 *Assume that we want to store data about employees in a company. We migrate the data and code from a non-temporal relational DBMS to TimeDB. First, we create a non-temporal*

relation containing the employment number, the name, the salary and the employment number of the manager of each employee and then insert some data:

```
CREATE TABLE Employees (EmpID INTEGER, Name CHAR(30), Salary INTEGER, MgrID INTEGER);

INSERT INTO Employees VALUES (111, 'John', 8700, 111);
INSERT INTO Employees VALUES (112, 'Paul', 9100, 111);
INSERT INTO Employees VALUES (113, 'George', 8300, 111);
```

This results in the following table :

EmpID	Name	Salary	Manager
111	John	8700	111
112	Paul	9100	111
113	George	8300	111

We now can query this data for employees earning more than 9000, together with the names of their managers:

```
SELECT e1.EmpID, e1.Name, e1.Salary, e2.Name Manager
FROM   Employees e1, Employees e2
WHERE  e1.MgrID = e2.EmpID AND
       e1.Salary > 9000;
```

This query returns the following table:

e1.EmpID	e1.Name	e1.Salary	Manager
112	Paul	9100	John

4.3.2.2 Temporal Upward Compatible Queries

When data is migrated from a non-temporal relational DBMS to a temporal relational DBMS, the non-temporal (snapshot) tables can be altered, for example, to valid-time tables. This means that after migration, the data is kept track of with respect to valid time. The user however should still be allowed using the legacy queries and should get the same results as if he was working with the non-temporal relational DBMS. This can be achieved by evaluating legal SQL queries on snapshots of the temporal tables at time instant *now*. This is called *temporal upward compatibility*.

Temporal upward compatibility thus demands that any legal *non-temporal* query `q` executed on a *temporal* database $\texttt{db}^t$ yields the same result as if executed on the corresponding snapshot database $\texttt{db} = \tau_{now}(\texttt{db}^t)$ of a non-temporal DBMS. A non-temporal DBMS stores only one database state, usually the one representing the most recent state of the real world. With respect to a temporal database $\texttt{db}^t$, this corresponds to selecting the database state at time instant *now*.

The advantage of temporal upward compatibility is the fact that code of a non-temporal application can still be used on *migrated* temporal databases, which means that application programs still run without further changes, and users may query the database in the same manner as before migration.

Example 4.2 *On October 21, 1996, we migrate the table* `Employees` *given in example 4.1 to a valid-time table. This means that we will keep track of the history of employees with respect to valid time from October 21, 1996 on.*

```
ALTER TABLE Employees ADD VALID;
```

This statement changes the schema and content of table `Employees` *such that additionally the valid time of the tuples is recorded. The valid time of the tuples contained in* `Employees` *is then set to* `[1996/10/21 -` ∞`)`*. It is still possible to use queries and update statements of a non-temporal application on migrated tables. Assume that we give John a salary raise October 29, 1996. To store this fact in the database, we execute the following legal SQL modification statement at this date:*

```
UPDATE Employees SET Salary = 9300 WHERE Name = 'John';
```

After this database modification, the valid-time table `Employees` *looks like*

VALID	EmpID	Name	Salary	MgrID
$[1996/10/21 - \infty)$	112	Paul	9100	111
$[1996/10/21 - \infty)$	113	George	8300	111
$[1996/10/21 - 1996/10/29)$	111	John	8700	111
$[1996/10/29 - \infty)$	111	John	9300	111

The non-temporal query of example 4.1,

```
SELECT e1.EmpID, e1.Name, e1.Salary, e2.Name Manager
FROM   Employees e1, Employees e2
WHERE  e1.MgrID = e2.EmpID AND
       e1.Salary > 9000;
```

executed on the valid-time table `Employees` *at October 29, 1996, then returns the following result:*

e1.EmpID	e1.Name	e1.Salary	Manager
112	Paul	9100	John
111	John	9300	John

4.3.2.3 Sequenced Queries

Sequenced semantics of queries expresses that the queries are evaluated using *temporal* algebra operations having *snapshot reducible semantics*. This means that a temporal algebra operation interprets the timestamps of tuples – be it valid time and/or transaction time – and uses them for the calculation of the tuple timestamps of the resulting relation.

In ATSQL2, syntactical similarity of non-temporal and temporal statements is achieved by introducing keywords `VALID` and `TRANSACTION`, which can be written in front of any legal SQL query. These keywords express which temporal dimension shall be used for temporal evaluation of the query. If the query should be evaluated with respect to both valid time and transaction time, the two keywords are combined as `VALID AND TRANSACTION`.

Example 4.3 *We formulate the query given in example 4.1 as a sequenced query with respect to valid-time by simply adding the keyword* `VALID` *in front of the corresponding non-temporal query:*

```
VALID
 SELECT e1.EmpID, e1.Name, e1.Salary, e2.Name Manager
 FROM   Employees e1, Employees e2
 WHERE  e1.MgrID = e2.EmpID AND
        e1.Salary > 9000;
```

This query returns the history of employees whose salaries are higher than 9000 along with the valid-time intervals:

VALID	e1.EmpID	e1.Name	e1.Salary	Manager
[1996/10/21 – 1996/10/29)	112	Paul	9100	John
[1996/10/29 – ∞)	112	Paul	9100	John
[1996/10/29 – ∞)	111	John	9300	John

The sequenced query in example 4.3 returns two tuples with adjacent valid-time periods for employee Paul. This is caused by the temporal cross product of table `Employees` with itself which is needed to find the manager's name for each employee. John is Paul's manager, and since John has got a salary raise October 29, 1996, he is listed twice in table `Employees`. So the splitting of the information about Paul is caused by John's salary raise.

The temporal cross product calculates the common time period of both tuples involved and thus returns a single timestamp for each resulting tuple. The result is independent of the order of execution of the algebra operations.

4.3.2.4 Non-Sequenced Queries

As mentioned before, there is a second class of temporal statements which need to examine different database states for a single resulting state. These statements are called *non-sequenced*. Non-sequenced semantics of queries express that the queries are evaluated using algebra operations which treat the time information like any other attribute. Such algebra operations have the same semantics as the corresponding non-temporal operations and *do not* interpret the timestamps of tuples – be it valid time and/or transaction time – but allow these timestamps to be referred to as if they were user-defined attributes. This way, it is possible to access and compare different database states in a query and retrieve, for example, time points of specific database state transitions.

Queries comparing different database states usually calculate the cross product of the temporal relations involved and then use join-conditions for comparison. If the cross product operation with sequenced semantics were used, the resulting table would only contain a single timestamp for each time line involved – the timestamp specifying the common time period of the tuples composed to a new tuple. This does not allow the comparison of the different time periods of the tuples involved. Instead, the cross product operation with non-sequenced semantics has to be used which treats the timestamps as user-defined attributes. This means that for each relation involved in the cross product, a separate timestamp appears in the resulting table.

Example 4.4 *We want to find out when data about employees has changed with respect to valid-time, for example, due to salary raises. This means that we have to find tuples containing information about the same employee having adjacent (meeting) valid-time intervals. In ATSQL2, the valid-time intervals in a relation* `R` *can be referenced using* `VALID(R)`*.*

```
NONSEQUENCED VALID
  SELECT BEGIN(VALID(a2)), a2.Name
  FROM   Employees a1, Employees a2
  WHERE  a1.EmpID = a2.EmpID AND
         VALID(a1) meets VALID(a2);
```

BEGIN(VALID(a2))	a2.Name
1996/10/29	John

In example 4.4, the cross product of tables a_1 and a_2 is calculated using non-temporal semantics, and valid-time is treated as a user-defined attribute, accessible using the expressions `VALID(`a_1`)` and `VALID(`a_2`)`.

4.4 Translation of Temporal Queries to Standard SQL Statements

The following subsections describe how ATSQL2 queries are translated into standard SQL statements. Additionally, the implementation of the unitemporal algebra operations is sketched and

the bitemporal algebra operations are described.

4.4.1 The Basic Idea of the Translation Algorithm

The main problem when evaluating temporal queries using standard SQL statements is that time intervals have to be calculated during the evaluation of the query. This means that a way needs to be found to integrate these time calculation statements somewhere in the standard SQL statements.

The idea of the translation algorithm used in TimeDB is the following: a temporal query is translated into a temporal algebra expression using the temporal set operations *union* ($\cup^t$), *intersect* ($\cap^t$) and *difference* ($\Leftrightarrow^t$). Each argument to these set operations is either a simple algebra expression using a *temporal select* (σ^t), project (π^t) and cross product ($\times^t$) operation or the result of another temporal set operation. Each simple algebra expression using only a combination of a select, project and cross product operation can then be evaluated separately using a *standard* `SELECT-FROM-WHERE` statement. These intermediate results are stored in temporary tables and then used to calculate other parts of the expression.

This way, it is possible to coalesce intermediate results prior to further evaluation of the algebra expression or to resolve temporal queries with correlated subqueries. Also, temporal calculations such as temporal negation can be done for intermediate results.

In TimeDB, a valid-time interval `I = [vts_#$ - vte_#$)`, closed at the lower bound and open at the upper, is mapped internally to two attributes `vts_#$` (valid-time start) and `vte_#$` (valid-time end). Thus, each valid-time relation will have two additional (hidden) attributes. Transaction-time tables are extended accordingly. Bitemporal tables contain attributes for both valid time and transaction time.

Time instants such as 1996/6/12 are stored as integer values, expressing the amount of chronons gone by with respect to a reference date. Recall that a chronon is the smallest non-decomposable time unit used, for example, a second. The reference date and granularity of a chronon depends on the calendar used. TimeDB supports several different calendars and allows the specification of new ones.

4.4.2 Temporal Upward Compatible Queries

Due to the temporal upward-compatibility requirement, legal non-temporal SQL queries have to return the same results when executed on temporal tables as if they were executed on corresponding non-temporal tables. This is done by first selecting the data valid at time instant *now* in the tables referenced by the query and leaving away any timestamp attributes prior to further evaluation.

Example 4.5 *The temporal upward compatible query used in example 4.2 is translated to the following standard SQL query*

```
SELECT a#$_0.EmpID, a#$_0.Name, a#$_0.Salary, a#$_1.Name Manager
FROM   Employees a#$_0, Employees a#$_1
WHERE  |now| >= a#$_0.vts_#$ AND |now| < a#$_0.vte_#$ AND
       |now| >= a#$_1.vts_#$ AND |now| < a#$_1.vte_#$ AND
       a#$_0.MgrID = a#$_1.EmpID AND
       a#$_0.Salary > 9000;
```

where `|now|` *is an integer-value representing time instant* now *with respect to the calendar used. The cryptic aliases of the form* `a#$_0` *and so on are system generated and should not interfere with names in the query chosen by the user.*

4.4.3 Implementing the Temporal Algebra

TimeDB implements the temporal algebra operations union ($\cup^t$), difference ($\Leftrightarrow^t$), intersection ($\cap^t$), selection (σ^t), projection (π^t) and cross product ($\times^t$) using standard SQL statements. Due to the

orthogonal treatment of valid time and transaction time, only one set of these algebra operations has to be implemented for *unitemporal* operations. When calling these generic operations, a parameter specifies which of the timestamp attributes (`vts_#$` and `vte_#$` for valid-time intervals and `tts_#$` and `tte_#$` for transaction-time intervals, respectively) should be used to calculate the resulting timestamps. Bitemporal algebra operations are implemented separately.

The following subsections describe how the unitemporal operations can be implemented using standard SQL, without focusing on efficiency. However, TimeDB was built such that it allows an easy replacement of the implementation of any of these operators by a more efficient one. In fact, it would also be possible to have several different implementations of the same algebra operation and have a query optimiser choose the best implementation for a specific query. The algorithm for unitemporal set difference presented here is based on the one used in ChronoLog [BM92, Böh94].

4.4.3.1 Temporal Set Union Operation

Assume two union-compatible valid-time relations $\mathtt{R}_1$ and $\mathtt{R}_2$ with the attributes

$$\mathtt{R}_1 = < A_1, A_2, ..., A_n, vts\_\#\$, vte\_\#\$ >$$
$$\mathtt{R}_2 = < B_1, B_2, ..., B_n, vts\_\#\$, vte\_\#\$ >$$

The valid-time union operation $\cup^v$ of $\mathtt{R}_1$ and $\mathtt{R}_2$ is the same as the non-temporal union:

$$\mathtt{R}_1 \cup^v \mathtt{R}_2 \equiv \mathtt{R}_1 \cup \mathtt{R}_2$$

The temporal set union operation does not perform any coalescing on value-equivalent tuples. The result of a temporal set union of two relations thus may lead to a relation containing value-equivalent tuples with overlapping time periods.

4.4.3.2 Temporal Set Difference Operation

Unitemporal set difference of two relations is more complicated to calculate. Assume again two union-compatible valid-time relations $\mathtt{R}_1$ and $\mathtt{R}_2$, having attributes

$$\mathtt{R}_1 = < A_1, A_2, ..., A_n, vts\_\#\$, vte\_\#\$ >$$
$$\mathtt{R}_2 = < B_1, B_2, ..., B_n, vts\_\#\$, vte\_\#\$ >$$

The valid-time difference of $\mathtt{R}_1$ and $\mathtt{R}_2$,

$$\mathtt{R}_1 := \mathtt{R}_1 \Leftrightarrow^v \mathtt{R}_2,$$

can be translated into the following standard SQL statements:

```
INSERT INTO R1
   SELECT a0.vts_#$, a1.vts_#$, a0.A1, a0.A2, ..., a0.An
   FROM   R1 a0, R2 a1
   WHERE  a1.vts_#$ > a0.vts_#$ AND
          a1.vts_#$ < a0.vte_#$ AND
          a0.A1 = a1.B1 AND
          a0.A2 = a1.B2 AND
          ...           AND
          a0.An = a1.Bn;

INSERT INTO R1
   SELECT a1.vte_#$, a0.vte_#$, a0.A1, a0.A2, ..., a0.An
   FROM   R1 a0, R2 a1
   WHERE  a1.vte_#$ > a0.vts_#$ AND
          a1.vte_#$ < a0.vte_#$ AND
          a0.A1 = a1.B1 AND
```

```
            a0.A2 = a1.B2 AND
            ...           AND
            a0.An = a1.Bn;

DELETE FROM R1 a0
WHERE EXISTS (SELECT a1.*
              FROM   R2 a1
              WHERE  ((a0.vts_#$ >= a1.vts_#$ AND a0.vts_#$ < a1.vte_#$) OR
                      (a1.vts_#$ >= a0.vts_#$ AND a1.vts_#$ < a0.vte_#$))
              AND
                     a0.A1 = a1.B1 AND
                     a0.A2 = a1.B2 AND
                     ...           AND
                     a0.An = a1.Bn);
```

The unitemporal set difference `R1` $\Leftrightarrow^{y}$ `R2` returns the tuples in `R1` with time intervals during which no value-equivalent tuple in `R2` can be found. In the case that the resulting time interval is empty, the tuple is abandoned. Assume `t1` to be a tuple in table `R1`, having a valid-time interval as depicted in figure 4.3. Tuple `t2` is a value-equivalent tuple in table `R2`. There are two main cases distinguished in the algorithm given above. The first case considers a possible non-overlapping part of tuple `t1` at the beginning of its valid-time interval, the second case a possible non-overlapping part of tuple `t1` at the end. The first `INSERT`-statement adds tuples to table `R1` for which a value-equivalent tuple in `R2` exists whose valid-time interval overlaps as depicted in case 1. These tuples are timestamped with the non-overlapping time interval of tuple `t1`, in this case with an interval $[t1.vts\_\#\$ \Leftrightarrow t2.vts\_\#\$)$. The second `INSERT`-statement does the same for tuples in `R1` for which a value-equivalent tuple in `R2` exists whose valid-time interval overlaps as depicted in case 2. These tuples are timestamped with the non-overlapping time interval of tuple `t1`, in this case with an interval $[t2.vte\_\#\$ \Leftrightarrow t1.vte\_\#\$)$. This approach also covers the case that the valid-time interval of tuple `t2` is contained in the valid-time interval of tuple `t1`. The next step is to delete all those tuples in table `R1` for which value-equivalent tuples in `R2` exist having overlapping time intervals.

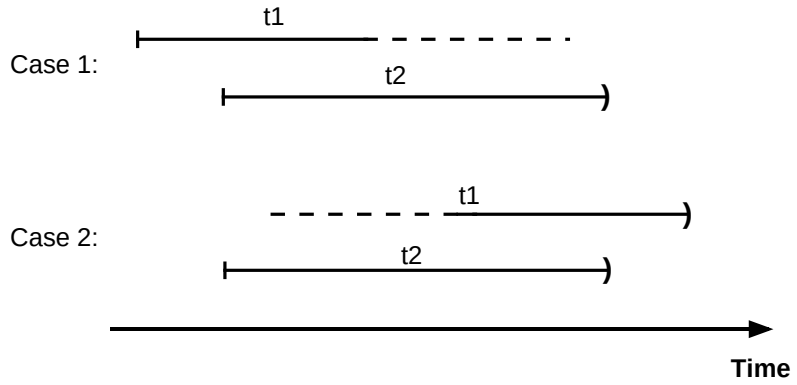

Figure 4.3: The two cases of overlapping time intervals considered in the calculation of temporal set difference

In the algorithm presented above, table `R1` is modified. If `R1` is a user-defined table, the temporal set difference of two tables must be calculated on an auxiliary table. The user-defined table is copied before calculating a temporal set difference. TimeDB stores all intermediate results in auxiliary tables. These auxiliary tables are then used as arguments for other operations.

4.4.3.3 Temporal Set Intersection Operation

The unitemporal set intersection operation can be rewritten by replacing the temporal intersection operation with a pair of temporal set difference operations. Assume again two union-compatible valid-time relations $\mathtt{R}_1$ and $\mathtt{R}_2$, having attributes

$$\mathtt{R}_1 = < A_1, A_2, ..., A_n, vts\_\#\$, vte\_\#\$ >$$
$$\mathtt{R}_2 = < B_1, B_2, ..., B_n, vts\_\#\$, vte\_\#\$ >$$

The valid-time set intersection of $\mathtt{R}_1$ and $\mathtt{R}_2$, $\mathtt{R}_1 \cap^v \mathtt{R}_2$, can be written as

$$\mathtt{R}_1 \Leftrightarrow^v (\mathtt{R}_1 \Leftrightarrow^v \mathtt{R}_2)$$

Another way to evaluate the unitemporal set intersection is to rewrite it as a temporal join:

```
SELECT DISTINCT A1, A2, ..., An,
       GREATEST(R1.vts_#$, R2.vts_#$) vts_#$,
       LEAST(R1.vte_#$, R2.vte_#$) vte_#$
FROM   R1, R2
WHERE  R1.A1 = R2.B1 AND
       R1.A2 = R2.B2 AND
       ...           AND
       R1.An = R2.Bn AND
       GREATEST(R1.vts_#$, R2.vts_#$) < LEAST(R1.vte_#$, R2.vte_#$)
```

The intersection of the valid-time periods is calculated using `GREATEST` and `LEAST`. These functions are available in Oracle[1] and calculate the greatest (least) value in a list of values. Calculating the greatest lower and the least upper bound of the time intervals involved returns the intersection of them if the resulting lower bound is smaller than the resulting upper bound. This condition is checked in the `WHERE`-clause. If the time periods involved do not have a common time interval (`GREATEST(R1.vts_#$, R2.vts_#$)` $\geq$ `LEAST(R1.vte_#$, R2.vte_#$)`), then the resulting tuple is abandoned.

In contrast to the relational data model, SQL supports duplicates in relations. Thus, the temporal algebra should also provide for handling bags of tuples. A problem arises when the implementation of the unitemporal intersection operation as presented above is used for relations containing *duplicates*. Translating a temporal intersection operation into temporal difference operation returns duplicates according to the first argument relation of the intersection operation. For example, if the first relation contains three times a tuple which occurs twice in the second relation during a common time period then it will occur three times in the result. With respect to translating a temporal intersection operation into a temporal natural join of two relations, the same restriction holds as in the non-temporal case. Set intersection and a corresponding natural join are only equivalent if the relations do not contain duplicates. Otherwise, due to the cross product of the two relations involved, too many resulting tuples are returned.

TimeDB calculates the temporal intersection operation using the temporal difference. So, with respect to relations containing duplicates, the semantics of the unitemporal intersection are that the first argument of a set intersection operation determines how often duplicates appear in the resulting relation. This differs from the semantics of set operations in SQL3 [SQL93]. SQL3 defines that intersecting two sets containing duplicates shall return the minimum of the number of occurrences of a duplicate in the relations involved.

[1] These functions are beyond the SQL Standard, however they can be expressed using the `CASE`-expression available in SQL-92 (see [MS93])

4.4.3.4 Temporal Selection Operation

The temporal selection operation actually is the same as the non-temporal selection. However, it is extended with additional predicates for temporal comparison. As proposed in [SDJ+93], TimeDB supports the predicates `=`, `precedes`, `overlaps`, `meets` and `contains`. In TimeDB, these comparison operators are translated into their definitions (see table 2.2). They can be used both for valid time and transaction time.

TimeDB also supports temporal comparison operators for durations of time periods and time instants, following [SDJ+93]. Additionally, functions `begin` and `end` are supported returning the lower and the upper bound of a time interval.

4.4.3.5 Temporal Projection Operation

The temporal projection operation mainly corresponds to its non-temporal counterpart. The only difference is whether or not the timestamp attributes are members of the projection list (the list of columns appearing in the result) by default. This depends on the kind of query stated.

In the case of a (temporal) upward compatible query, no valid- or transaction-time attributes are in the projection list, since they return a result with respect to the time instant *now* only. In the case of a sequenced query, the timestamp attributes are members by default. A valid-time query implicitly returns valid-time timestamps, a transaction-time query transaction-time timestamps, and a bitemporal query both. In the case of a non-sequenced query, the temporal attributes are treated as user-defined attributes and may or may not appear in the result, depending on whether they appear in the projection list specified by the user. The valid- and transaction-time timestamps of tuples in a relation `R` can be referenced by `VALID(R)` and `TRANSACTION(R)`, respectively.

4.4.3.6 Temporal Cross Product Operation

The valid-time cross product combines tuples of two relations during their common valid-time periods. Assume two valid-time relations $\mathtt{R}_1$ and $\mathtt{R}_2$, having attributes

$$\mathtt{R}_1 = < A_1, A_2, ..., A_n, vts\_\#\$, vte\_\#\$ >$$
$$\mathtt{R}_2 = < B_1, B_2, ..., B_m, vts\_\#\$, vte\_\#\$ >$$

The valid-time cross product of $\mathtt{R}_1$ and $\mathtt{R}_2$, $\mathtt{R}_1 \times^v \mathtt{R}_2$, can be translated into the SQL statement

```
SELECT A1, A2, ..., An, B1, B2, ..., Bm,
       GREATEST(R1.vts_#$, R2.vts_#$) vts_#$,
       LEAST(R1.vte_#$, R2.vte_#$) vte_#$
FROM   R1, R2
WHERE  GREATEST(R1.vts_#$, R2.vts_#$) < LEAST(R1.vte_#$, R2.vte_#$)
```

4.4.4 Bitemporal Algebra Operations

So far, the implementation of the unitemporal algebra operations has been discussed. TimeDB however also handles bitemporal queries. For these queries, special operations for set union, set difference, set intersection and cross product need to be implemented. The temporal selection and projection operations presented above are sufficient for bitemporal queries. A bitemporal query is translated to standard SQL the same way as a unitemporal query. It uses, however, bitemporal algebra operations instead.

Bitemporal timestamps can be depicted as areas in a coordinate system having two axes – valid time and transaction time. For the following examples, two value-equivalent tuples are assumed which are members of two different bitemporal relations $\mathtt{R}_1$ and $\mathtt{R}_2$, having timestamps as shown in figure 4.4. Rectangle S_1 defines the timestamp of a tuple in relation $\mathtt{R}_1$ and rectangle S_2 of a tuple in $\mathtt{R}_2$.

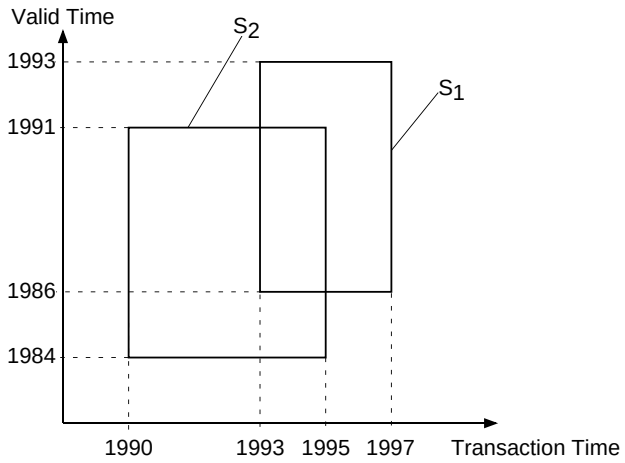

Figure 4.4: Bitemporal timestamps of two value-equivalent tuples

4.4.4.1 Bitemporal Set Union Operation

The bitemporal union operation $\cup^{vt}$ of two union-compatible bitemporal relations $\mathtt{R}_1$ and $\mathtt{R}_2$, having attributes

$$\mathtt{R}_1 = < A_1, A_2, ..., A_n, vts\_\#\$, vte\_\#\$, tts\_\#\$, tte\_\#\$ >$$
$$\mathtt{R}_2 = < B_1, B_2, ..., B_n, vts\_\#\$, vte\_\#\$, tts\_\#\$, tte\_\#\$ >$$

is the same as the non-temporal union, $\mathtt{R}_1 \cup^{vt} \mathtt{R}_2 \equiv \mathtt{R}_1 \cup \mathtt{R}_2$.

Example 4.6 *Assume that the bitemporal relation* $\mathtt{R}_1$ *contains the tuple*

TRANSACTION	VALID	Name	Dept
$[1993 - 1997)$	$[1986 - 1993)$	John	Sales

whereas relation $\mathtt{R}_2$ *contains the tuple*

TRANSACTION	VALID	Name	Dept
$[1990 - 1995)$	$[1984 - 1991)$	John	Sales

The timestamp of the tuple in relation $\mathtt{R}_1$ *thus corresponds to the area* S_1 *and the timestamp of the tuple in relation* $\mathtt{R}_2$ *to the area* S_2*. The bitemporal set union returns a bitemporal relation containing the two tuples. So, the query*

```
VALID AND TRANSACTION
  (SELECT * FROM R1
   UNION
   SELECT * FROM R2);
```

returns the bitemporal relation

TRANSACTION	VALID	Name	Dept
$[1993 - 1997)$	$[1986 - 1993)$	John	Sales
$[1990 - 1995)$	$[1984 - 1991)$	John	Sales

As with the unitemporal set union operation, the bitemporal operation does not perform any coalescing of overlapping time periods of value-equivalent tuples.

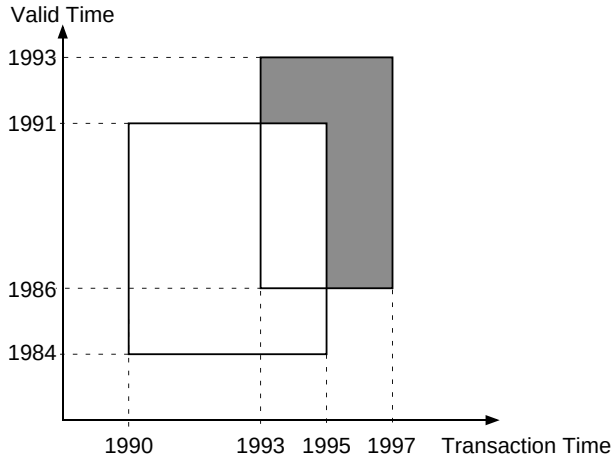

Figure 4.5: Difference of two bitemporal timestamps

4.4.4.2 Bitemporal Set Difference Operation

Assume again two relations $\mathtt{R}_1$ and $\mathtt{R}_2$ containing two value-equivalent tuples having timestamps as depicted in figure 4.4. When calculating the bitemporal difference of $\mathtt{R}_1$ minus $\mathtt{R}_2$, the resulting timestamp of the two tuples is the shaded area depicted in figure 4.5.

The resulting timestamp area may no longer be a rectangle. In the case depicted in figure 4.5, TimeDB returns two value-equivalent tuples where the resulting timestamp of the bitemporal difference is given as a polygon. Depending on the sequence – `VALID AND TRANSACTION` or `TRANSACTION AND VALID` – used in the bitemporal query, the resulting timestamp is set up differently.

Example 4.7 *In TimeDB, the bitemporal difference* $\mathtt{R}_1$ *-*^{vt} $\mathtt{R}_2$ *can be expressed in two different ways. The query*

```
VALID AND TRANSACTION
  (SELECT * FROM R1
   EXCEPT
   SELECT * FROM R2);
```

returns the bitemporal relation

TRANSACTION	VALID	Name	Dept
[1993 – 1995)	[1991 – 1993)	John	Sales
[1995 – 1997)	[1986 – 1993)	John	Sales

whereas the query

```
TRANSACTION AND VALID
  (SELECT * FROM R1
   EXCEPT
   SELECT * FROM R2);
```

results in the following bitemporal relation:

VALID	TRANSACTION	Name	Dept
[1986 – 1991)	[1995 – 1997)	John	Sales
[1991 – 1993)	[1993 – 1997)	John	Sales

4.4.4.3 Bitemporal Set Intersection Operation

Bitemporal set intersection in TimeDB is implemented similarly to the unitemporal set intersection. We again use the fact that set intersection can be expressed using set difference. The bitemporal set intersection of the two relations $\mathtt{R}_1$ and $\mathtt{R}_2$, $\mathtt{R}_1 \cap^{vt} \mathtt{R}_2$, can be written as

$$\mathtt{R}_1 \Leftrightarrow^{vt} (\mathtt{R}_1 \Leftrightarrow^{vt} \mathtt{R}_2)$$

Figure 4.6 shows the resulting timestamp area if each relation contains tuples with bitemporal timestamps as depicted in figure 4.4.

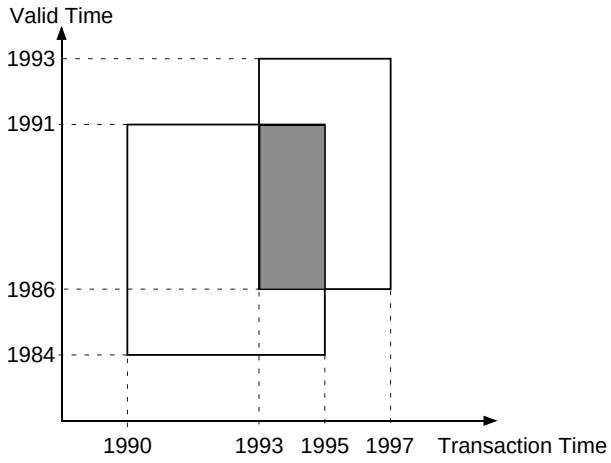

Figure 4.6: Intersection of two bitemporal timestamps

Example 4.8 *The bitemporal set intersection* $\mathtt{R}_1 \cap^{vt} \mathtt{R}_2$ *can be written in TimeDB as*

```
VALID AND TRANSACTION
  (SELECT * FROM R1
   INTERSECT
   SELECT * FROM R2);
```

and returns the bitemporal relation

TRANSACTION	VALID	Name	Dept
[1993 – 1995)	[1986 – 1991)	John	Sales

Using the flag `TRANSACTION AND VALID` *results in the same relation.*

4.4.4.4 Bitemporal Cross Product Operation

The bitemporal cross product of $\mathtt{R}_1$ and $\mathtt{R}_2$, $\mathtt{R}_1 \times^{vt} \mathtt{R}_2$, can be translated into the following SQL statement:

```
SELECT A1, A2, ..., An, B1, B2, ..., Bm,
       GREATEST(R1.vts_#$, R2.vts_#$) vts_#$,
       LEAST(R1.vte_#$, R2.vte_#$) vte_#$,
       GREATEST(R1.tts_#$, R2.tts_#$) tts_#$,
       LEAST(R1.tte_#$, R2.tte_#$) tte_#$
FROM   R1, R2
WHERE  GREATEST(R1.vts_#$, R2.vts_#$) < LEAST(R1.vte_#$, R2.vte_#$)
AND    GREATEST(R1.tts_#$, R2.tts_#$) < LEAST(R1.tte_#$, R2.tte_#$)
```

The resulting timestamp for the two tuples in $\mathtt{R}_1$ and $\mathtt{R}_2$, respectively, is again the intersection of the two areas as depicted in figure 4.6.

Example 4.9 *The bitemporal cross product* $\mathtt{R}_1 \times^{vt} \mathtt{R}_2$ *can be written in TimeDB as*

```
VALID AND TRANSACTION
  (SELECT * FROM   R1, R2);
```

and returns the bitemporal relation

TRANSACTION	VALID	Name	Dept	Name	Dept
[1993 – 1995)	[1986 – 1991)	John	Sales	John	Sales

Using the flag `TRANSACTION AND VALID` *returns the same relation.*

4.4.5 Derived Tables and Views

An important concept of ATSQL2 is the *derived table*. A derived table is a (temporal) query expressions in the `FROM`-clause which is used instead of a table reference. Derived tables are useful, for example, when two states of a temporal database need to be compared to each other with respect to their time.

Example 4.10 *On November 1, 1996, we execute the following* upward compatible *modification statement on valid-time table* `Employees` *as given in example 4.2:*

```
UPDATE Employees SET MgrID = 112 WHERE Name = 'George';
```

Since table `Employees` *is a valid-time table, the valid-time of the updated tuple will be* $[now \Leftrightarrow \infty)$ *due to the upward compatible semantics. This means, that the above update statement stores the fact that from November 1, 1996 on, Paul is George's manager.*

VALID	EmpID	Name	Salary	MgrID
[1996/10/21 – ∞)	112	Paul	9100	111
[1996/10/21 – 1996/10/29)	111	John	8700	111
[1996/10/29 – ∞)	111	John	9300	111
[1996/10/21 – 1996/11/1)	113	George	8300	111
[1996/11/1 – ∞)	113	George	8300	112

Now, we would like to find the employee number of George's manager prior to Paul:

```
NONSEQUENCED VALID
  SELECT a2.MgrID
  FROM   (VALID
             SELECT e1.EmpID
             FROM   Employees e1, Employees e2
             WHERE  e1.Name  = 'George' AND
                    e1.MgrID = e2.EmpID AND
                    e2.Name  = 'Paul') a1,
         Employees a2
  WHERE  VALID(a2) meets VALID(a1);
```

This query returns the employee number 111 *which belongs to John.*

The query in example 4.10 combines a non-sequenced query with a sequenced query. Due to keyword `VALID` in the derived table, it is calculated using temporal semantics. Thus, alias `a1` actually stands for a valid-time relation, containing tuples with George's employee number and the valid-time intervals, during which Paul was his manager. The temporal comparison of table `a2` with the derived table `a1` is done in the context of the `NONSEQUENCED VALID` time flag. Thus, the valid times of both `a1` and `a2` are treated as user-defined attributes and operations have non-temporal semantics, meaning that a *non-temporal cross product* of `a1` and `a2` is calculated. This allows that their valid times can be used for comparison in the `WHERE`-clause.

The next example uses a *temporal view* instead of a derived table. A temporal view is a virtual temporal table. For example, a valid-time view contains data timestamped with valid time.

Example 4.11 *Instead of using a derived table as in example 4.10, we can create a temporal view to find employee number of George's manager prior to Paul:*

```
CREATE VIEW ManagerView AS
   VALID
     SELECT e1.EmpID
     FROM   Employees e1, Employees e2
     WHERE  e1.Name  = 'George'   AND
            e1.MgrID = e2.EmpID AND
            e2.Name  = 'Paul';

NONSEQUENCED VALID
   SELECT a2.MgrID
   FROM   ManagerView a1,
          Employees a2
   WHERE  VALID(a2) meets VALID(a1);
```

TimeDB materialises both derived tables and views referenced in a query before executing the main part of the query. In the above examples 4.10 and 4.11, the derived table `a1` and the view `ManagerView` are materialised in an auxiliary table before the main query itself is evaluated. After execution of the query, the materialised data is removed from the database.

4.4.6 Subqueries

SQL supports the concept of subqueries. Subqueries are query expressions which appear in the body of another query. This is another method by which data of different relations can be related with each other [MS93]. Subqueries are used, for example, together with predicates as `EXISTS` or `IN` or their negated form. *Correlated subqueries* are subqueries which refer to attribute values of tables of some outer query [Dat89].

Example 4.12 *We would like to find those employees for whom employees can be found who earn more. The following ATSQL2 query with a correlated subquery returns the desired result:*

```
VALID
  SELECT *
  FROM   Employees e1
  WHERE  EXISTS (SELECT *
                 FROM Employees e2
                 WHERE e1.Salary < e2.Salary);
```

If the above query is evaluated on table `Employees` *given in example 4.10, the following valid-time relation is returned:*

VALID	EmpID	Name	Salary	MgrID
[1996/10/21 – 1996/10/29)	111	John	8700	111
[1996/10/21 – 1996/11/1)	113	George	8300	111
[1996/11/1 – ∞)	113	George	8300	112
[1996/10/29 – ∞)	112	Paul	9100	111

During the whole time period of his employment, George earns less than Paul. From 1996/10/21 *to* 1996/10/29*, John earns less than Paul. After John's salary raise on* 1996/10/29*, Paul earns less than John.*

When translating temporal queries into standard SQL queries, correlated subqueries need special attention. To be able to do temporal calculations, correlated subqueries have to be translated into corresponding set operations. For example, queries with `EXISTS`-, `NOT EXISTS`-, `IN`- or `NOT IN`-subqueries can be translated into set difference operations of the outer query and the subquery. TimeDB handles temporal queries having any number of subqueries conjuncted by `AND` and `OR` and also subqueries containing other subqueries.

The query in example 4.12, having an `EXISTS`-subquery with correlation $e1.Salary < e2.Salary$, is translated into the algebra expression

$$\texttt{Employees} \Leftrightarrow^v (\texttt{Employees} \Leftrightarrow^v \pi_{\texttt{Employees}_{e1}.*}(\sigma_{e1.\texttt{salary}<e2.\texttt{salary}}(\texttt{Employees}_{e1} \times^v \texttt{Employees}_{e2})))$$

where the projection list $\texttt{Employees}_{e1}.*$ stands for all attributes of table `Employees`, which are `VALID`, `EmpID`, `Name`, `Salary` and `MgrID`. First, the temporal cross product of table `Employees` with itself is calculated returning the combinations of a tuple of the outer table with a tuple of the inner table together with their common validity time period. The subscripts $e1$ and $e2$ are aliases used to distinguish the attributes of the outer and the inner table involved. Second, in the resulting valid-time table, those tuples are selected for which $e1.\texttt{salary} < e2.\texttt{salary}$ is true. Third, all attributes of the inner table are projected away and a table is returned containing those salaries for which – during the resulting valid-time periods – higher salaries exist.

The semantics of the SQL query of example 4.12 demands that the result shall contain duplicates as they appear in the outer `Employees` table. So with respect to duplicates, the same approach as for temporal set intersection is used in TimeDB. The `EXISTS` subquery is translated into a sequence of temporal difference operations.

In general, if the outer query references tables $p_1, ..., p_m$ and the subquery tables $q_1, ..., q_n$, `EXISTS`- or `IN`-subqueries can be translated into

$$(p_1 \times^v ... \times^v p_m) \Leftrightarrow^v ((p_1 \times^v ... \times^v p_m) \Leftrightarrow^v \pi_{p_1.*,...,p_m.*}((p_1 \times^v ... \times^v p_m) \bowtie^v_{joincond_1} q_1 ... \bowtie^v_{joincond_n} q_n))$$

We use $\bowtie^v_{joincond_i}$ as shorthand for a temporal cross product of the tables involved and selection operations testing for condition $joincond_i$. Join condition $joincond_i$ contains – among others – the correlations of table q_i with one of the tables of the outer query. The projection list $p_i.*$ stands for all attributes of table p_i. The valid-time cross product of the tables of the outer query, $p_1 \times^v ... \times^v p_m$, needs only to be calculated once, and copies of the result can be used.

A query with a `NOT EXISTS`- or `NOT IN`-subquery can be translated into an algebra expression of the form

$$(p_1 \times^v ... \times^v p_m) \Leftrightarrow^v \pi_{p_1.*,...,p_m.*}((p_1 \times^v ... \times^v p_m) \bowtie^v_{joincond_1} q_1 ... \bowtie^v_{joincond_n} q_n)$$

where again $p_1, .., p_m$ are the tables referenced in the outer query, $q_1, ..., q_n$ the tables referenced in the subquery, and $joincond_i$ contains the correlation of table q_i with one of the tables of the outer query.

Example 4.13 *We would like to know the history of the highest salaries. We can find it using a* `NOT-EXISTS` *subquery:*

```
VALID
  SELECT *
  FROM   Employees e1
  WHERE  NOT EXISTS (SELECT *
                     FROM   Employees e2
                     WHERE  e1.Salary < e2.Salary);
```

This query returns the following tuples:

VALID	Name	Salary	MgrID
[1996/10/21 – 1996/10/29)	Paul	9100	111
[1996/10/29 – ∞)	John	9300	111

This query is translated into the following algebra expression:

$$\texttt{Employees} \Leftrightarrow^{v} \pi_{\texttt{Employees}_{e1}.*}(\texttt{Employees}_{e1} \bowtie^{v}_{e1.\texttt{salary}<e2.\texttt{salary}} \texttt{Employees}_{e2})$$

This algebra expression actually is very similar to the algebra expression given for the example with the `EXISTS`-subquery. First, the valid-time cross product of table `Employees` with itself is calculated. Next, those tuples are selected for which $e1.\texttt{salary} < e2.\texttt{salary}$ is true. For these two operations, the shorthand notation $\bowtie^{v}_{e1.\texttt{salary}<e2.\texttt{salary}}$ is used. Third, the resulting table shall only contain those attribute values coming from the first argument of the temporal cross product operation. The intermediate result now contains all those tuples for which an employee can be found with a higher salary (with the corresponding valid-time periods). Temporally subtracting these from valid-time table `Employees` returns the desired result.

4.4.7 Coalescing

Coalescing is a special operation which does not have a counterpart in the non-temporal relational algebra. As described in chapter 2, coalescing is used to calculate maximal time intervals for value-equivalent tuples. The following subsections describe how unitemporal coalescing is implemented and discuss the different forms of bitemporal coalescing supported in TimeDB.

4.4.7.1 Unitemporal Coalescing

Coalescing a valid-time relation means to calculate maximal time intervals for tuples having identical non-timestamp attribute values. This operation could be written as a single standard SQL statement, however its complexity would not lead to an efficient evaluation. TimeDB implements unitemporal coalescing the same way as proposed in [BM92, Böh94].

The idea is to update the upper bound of the time interval of tuples for which value-equivalent tuples exist with an overlapping time period having a higher upper bound. This is repeated until there are no further updates. This loop is actually beyond standard SQL. Additional program code is used to achieve it. Last, those tuples are deleted for which other value-equivalent tuples exist which contain their whole valid-time period.

Assume a valid-time relation $\texttt{R}_1$ having attributes

$$\texttt{R}_1 = < A_1, A_2, ..., A_n, vts\_\#\$, vte\_\#\$ >.$$

The coalescing of this relation, $\texttt{R}_1 := \texttt{coalesce(R}_1\texttt{)}$, is performed as follows:

```
** REPEAT **
UPDATE R1 A0
  SET (A0.vte_#$) = (SELECT MAX(A1.VTE_#$)
                     FROM   R1 A1
                     WHERE  A0.C0 = A1.C0 AND
                            A0.C1 = A1.C1 AND
```

```
                          A0.VTE_#$ >= A1.VTS_#$ AND
                          A0.VTE_#$ <  A1.VTE_#$)
  WHERE EXISTS (SELECT *
                FROM   R1 A1
                WHERE  A0.C0 = A1.C0 AND
                       A0.C1 = A1.C1 AND
                       A0.VTE_#$ >= A1.VTS_#$ AND
                       A0.VTE_#$ <  A1.VTE_#$);
** UNTIL no updates **

DELETE FROM R1 A0
  WHERE EXISTS (SELECT *
                FROM   R1 A1
                WHERE  A0.C0 = A1.C0 AND
                       A0.C1 = A1.C1 AND
                       A1.VTS_#$ < A0.VTS_#$ AND
                       A1.VTE_#$ = A0.VTE_#$);
```

This algorithm for coalescing modifies the argument table `R1`. As already mentioned for the valid-time set difference operation, in the case that `R1` is a user-defined table, it first needs to be copied.

In ATSQL2, the keyword `(PERIOD)` is used to denote that a set of tuples shall be coalesced. This keyword can appear after any reference to a (derived) table in the `FROM`-clause or at the end of a temporal ATSQL2 query.

Example 4.14 *Looking at table* `Employees`*, we see that due to the update of John's salary, there are two tuples contained in this table with data concerning him:*

VALID	EmpID	Name	Salary	MgrID
$[1996/10/21 - \infty)$	112	Paul	9100	111
$[1996/10/21 - 1996/10/29)$	111	John	8700	111
$[1996/10/29 - \infty)$	111	John	9300	111
$[1996/10/21 - 1996/11/1)$	113	George	8300	111
$[1996/11/1 - \infty)$	113	George	8300	112

We would like to know the maximal time period during which data is stored about John in table `Employees`*. In order to get value-equivalent tuples for employee John, we first have to project away the salary attribute. Then, we coalesce the remainder of this table:*

```
VALID
  (SELECT EmpID, Name, MgrID
   FROM   Employees
   WHERE  Name = 'John') (PERIOD);
```

This results in the following table:

VALID	EmpID	Name	MgrID
$[1996/10/21 - \infty)$	111	John	111

In example 4.14, the result of the projection- and selection-operation on table `Employees` is first materialised in an auxiliary table. On this intermediate result, coalescing is performed.

4.4.7.2 Bitemporal Coalescing

Bitemporal relations can be coalesced in two different ways. Due to the two time dimensions, maximal time intervals can be calculated either with respect to valid-time first and then coalesce the result with respect to transaction-time, or vice versa.

Example 4.15 *Figure 4.4 shows two bitemporal elements of two value-equivalent tuples. Assume that these bitemporal timestamps belong to the two value-equivalent tuples given in the following table* Employees*:*

VALID	TRANSACTION	Name	Dept
[1984 – 1991)	[1990 – 1995)	John	Sales
[1986 – 1993)	[1993 – 1997)	John	Sales

Due to the overlapping of the two areas, different results of bitemporal coalescing are possible. Figure 4.7 shows the two solutions which can be calculated in TimeDB. The figure on the left hand side first shows the resulting time periods when calculating maximal time intervals first with respect to valid time and then with respect to transaction time. Calculating first maximal time intervals with respect to transaction time and then with respect to valid time leads to a result as shown in the figure on the right. In TimeDB, the desired result is specified using either the flag `VALID AND TRANSACTION` or `TRANSACTION AND VALID`.

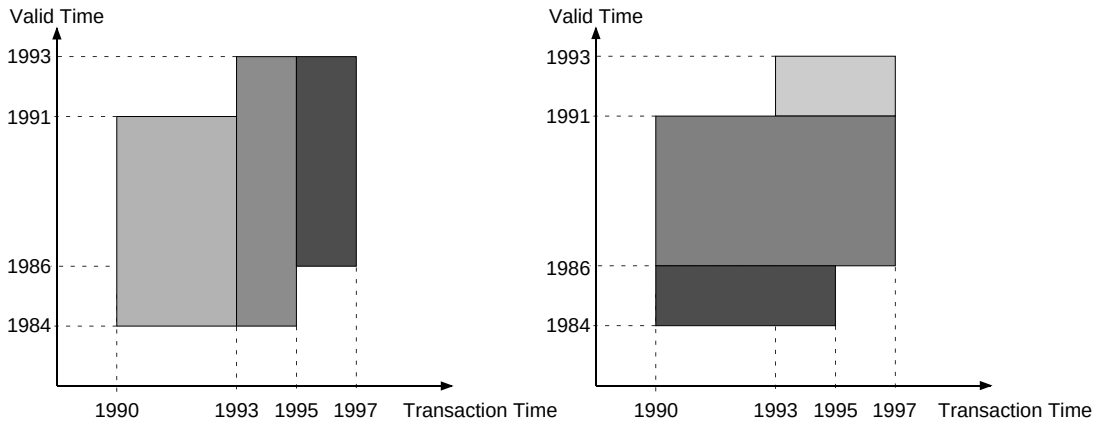

Figure 4.7: Two different results of bitemporal coalescing

Example 4.16 *The bitemporal query*

```
VALID AND TRANSACTION
  (SELECT * FROM Employees) (PERIOD);
```

returns the following table:

TRANSACTION	VALID	Name	Dept
[1990 – 1993)	[1984 – 1991)	John	Sales
[1993 – 1995)	[1984 – 1993)	John	Sales
[1995 – 1997)	[1986 – 1993)	John	Sales

This corresponds to the bitemporal timestamps depicted in the left picture of figure 4.7. The bitemporal query

```
TRANSACTION AND VALID
  (SELECT * FROM Employees) (PERIOD);
```

first coalesces the transaction-time periods and then the valid-time periods which results in the following bitemporal relation:

VALID	TRANSACTION	Name	Dept
[1984 – 1986)	[1990 – 1995)	John	Sales
[1986 – 1991)	[1990 – 1997)	John	Sales
[1991 – 1993)	[1993 – 1997)	John	Sales

4.4.8 An extended Example

The previous sections explained how the basic temporal operations are translated into standard SQL. Additionally, it was shown how TimeDB handles derived tables, views and subqueries. In this section, an extended example is given focusing on the different steps of translating a complex query into standard SQL.

Example 4.17 *We would like to find the history of employees working for manager John having the highest and the history of employees working for manager Paul having the lowest salaries :*

```
VALID
  (SELECT e1.Name, e1.Salary
   FROM   Employees e1, Employees e2
   WHERE  e1.MgrID = e2.EmpID AND
          e2.Name = 'John'    AND
          NOT EXISTS (SELECT *
                      FROM   Employees e3
                      WHERE  e1.Salary < e3.Salary)
  UNION
   SELECT e1.Name, e1.Salary
   FROM   Employees e1, Employees e2
   WHERE  e1.MgrID = e2.EmpID AND
          e2.Name = 'Paul'    AND
          NOT EXISTS (SELECT *
                      FROM   Employees e3
                      WHERE  e1.Salary > e3.Salary)) (PERIOD);
```

The result of this query is the following table:

VALID	Name	Salary
[1996/10/21 – 1996/10/29)	Paul	9100
[1996/10/29 – ∞)	John	9300
[1996/11/1 – ∞)	George	8300

TimeDB first evaluates the two main `SELECT-FROM-WHERE`-blocks separately. Since the keyword `VALID` embraces the whole query, these are then combined using the valid-time set operation `UNION`. The final result has to be coalesced due to the keyword `(PERIOD)` at the end of the query.

Due to the subqueries, each of the two `SELECT-FROM-WHERE`-blocks is translated into a valid-time set difference as described in subsection 4.4.6. Each of these set differences has two auxiliary tables as arguments, the first argument is the main query without any (correlated) subquery, the second argument is the result of joining the tables referenced in the `FROM`-clause of the upper query with the tables in the subquery, which allows the correlation to be evaluated. So the algebra expression of the query in example 4.17 looks like

$$coalesce(\pi[vts\_\#\$, vte\_\#\$, Name, Salary](\texttt{aux1} \Leftrightarrow^v \texttt{aux2}) \cup^v \pi[vts\_\#\$, vte\_\#\$, Name, Salary](\texttt{aux3} \Leftrightarrow^v \texttt{aux4}))$$

TimeDB evaluates this algebra expression on the commercial DBMS in several steps. First, the two auxiliary tables `aux1` and `aux2` are created. Table `aux1` contains the result of evaluating the first argument of the union-operation without the subquery:

```
CREATE TABLE aux1(vts_#$, vte_#$,
                  c#$_0, c#$_1, c#$_2, c#$_3, c#$_4, c#$_5,
                  c#$_6, c#$_7, c#$_8, c#$_9, c#$_10, c#$_11)
AS
SELECT GREATEST(e1.vts_#$, e2.vts_#$) vts_#$, LEAST(e1.vte_#$, e2.vte_#$) vte_#$,
       e1.vts_#$, e1.vte_#$, e1.EmpID, e1.Name, e1.Salary, e1.MgrID,
       e2.vts_#$, e2.vte_#$, e2.EmpID, e2.Name, e2.Salary, e2.MgrID
FROM   Employees e1, Employees e2
WHERE  GREATEST(e1.vts_#$, e2.vts_#$) < LEAST(e1.vte_#$, e2.vte_#$) AND
       e1.MgrID = e2.EmpID AND
       e2.Name = 'John';
```

Next, table `aux2` is created containing the result of joining the tables of the outer query with the tables in the subquery using the correlation.

```
CREATE TABLE aux2(vts_#$, vte_#$,
                  c#$_0, c#$_1, c#$_2, c#$_3, c#$_4, c#$_5,
                  c#$_6, c#$_7, c#$_8, c#$_9, c#$_10, c#$_11)
AS
SELECT DISTINCT GREATEST(e1.vts_#$, e2.vts_#$, e3.vts_#$) vts_#$,
       LEAST(e1.vte_#$, e2.vte_#$, e3.vte_#$) vte_#$,
       e1.vts_#$, e1.vte_#$, e1.EmpID, e1.Name, e1.Salary, e1.MgrID,
       e2.vts_#$, e2.vte_#$, e2.EmpID, e2.Name, e2.Salary, e2.MgrID
FROM   Employees e1, Employees e2, Employees e3
WHERE  GREATEST(e1.vts_#$, e2.vts_#$, e3.vts_#$) < LEAST(e1.vte_#$, e2.vte_#$, e3.vte_#$)
AND    e1.Salary < e3.Salary;
```

Now it is possible to calculate the temporal difference of tables `aux1` and `aux2`,

$$\texttt{aux1} := \texttt{aux1} -^{v} \texttt{aux2},$$

storing the result in table `aux1`. Next, the projection of the result to the timestamp attributes and the attributes `Name` and `Salary` needs to be done. The result of this projection operation is stored in another auxiliary table `aux5`,

$$\texttt{aux5} := \pi[\texttt{vts\_\#\$, vte\_\#\$, c\#\$\_3, c\#\$\_4}](\texttt{aux1}).$$

Table `aux5` is used as a first argument to the union-operation.

The second argument table is calculated accordingly. Two auxiliary tables `aux3` and `aux4` are created containing the result of the outer query and the result of joining the tables of the outer query with the tables in the subquery using the correlation.

```
CREATE TABLE aux3(vts_#$, vte_#$,
                  c#$_0, c#$_1, c#$_2, c#$_3, c#$_4, c#$_5,
                  c#$_6, c#$_7, c#$_8, c#$_9, c#$_10, c#$_11)
AS
SELECT GREATEST(e1.vts_#$, e2.vts_#$) vts_#$,
       LEAST(e1.vte_#$, e2.vte_#$) vte_#$,
       e1.vts_#$, e1.vte_#$, e1.EmpID, e1.Name, e1.Salary, e1.MgrID,
       e2.vts_#$, e2.vte_#$, e2.EmpID, e2.Name, e2.Salary, e2.MgrID
FROM   Employees e1, Employees e2
WHERE  GREATEST(e1.vts_#$, e2.vts_#$) < LEAST(e1.vte_#$, e2.vte_#$) AND
       e1.MgrID = e2.EmpID AND
       e2.Name = 'Paul';

CREATE TABLE aux4(vts_#$, vte_#$,
                  c#$_0, c#$_1, c#$_2, c#$_3, c#$_4, c#$_5,
                  c#$_6, c#$_7, c#$_8, c#$_9, c#$_10, c#$_11)
AS
```

```
SELECT DISTINCT GREATEST(e1.vts_#$, e2.vts_#$, e3.vts_#$) vts_#$,
       LEAST(e1.vte_#$, e2.vte_#$, e3.vte_#$) vte_#$,
       e1.vts_#$,e1.vte_#$, e1.EmpID, e1.Name, e1.Salary, e1.MgrID,
       e2.vts_#$,e2.vte_#$, e2.EmpID, e2.Name, e2.Salary, e2.MgrID
FROM   Employees e1, Employees e2, Employees e3
WHERE  GREATEST(e1.vts_#$, e2.vts_#$, e3.vts_#$) < LEAST(e1.vte_#$, e2.vte_#$, e3.vte_#$)
AND    e1.Salary > e3.Salary;
```

TimeDB calculates the temporal difference of `aux3` and `aux4`, `aux3 := aux3 -`^{v} `aux4` and returns the timestamp attributes plus attributes `Name` and `Salary`. This result is stored again in an auxiliary table `aux6`, `aux6 :=` π`[vts_#$, vte_#$, c#$_3, c#$_4](aux3)`.

Both argument tables for the temporal set union are now ready. TimeDB calculates the valid-time union of tables `aux5` and `aux6` and stores the result in table `aux5`, `aux5 := aux5` $\cup^v$ `aux6`. Last, coalescing is performed on table `aux5` and the result is presented to the user.

4.5 Temporal Constraint Checking

Besides the standard non-temporal constraints, TimeDB supports valid-time assertions and valid-time table- and column-constraints. Valid-time assertions are, according to standard SQL, defined using the `CREATE ASSERTION` statement. ATSQL2 additionally allows the use of the keyword `VALID` to denote that the assertion should be checked using *temporal* semantics. Similarly, valid-time column-constraints such as referential integrity and `CHECK` constraints may be declared as temporal in ATSQL2 by writing `VALID` in front of the column-constraint.

TimeDB translates these temporal assertions and constraints into temporal queries and stores them in the database. To *verify* temporal constraints, TimeDB supports a *constraint checker* which, at commit time, fetches the constraint queries from the database, evaluates them and decides according to their results whether or not any of the constraints are violated. With this approach, the translation algorithm for queries can be reused to evaluate constraints. Of course, only those constraints need to be checked which are related to modified tables. So only the relevant constraint queries are evaluated at commit time.

Example 4.18 *The company's policy is that all employees have a minimal salary of 8500. To make sure that this is always guaranteed, we add a column-constraint to table* `Employees`*:*

```
ALTER TABLE Employees ADD VALID CHECK (salary >= 8500);
```

This valid-time column-constraint is translated into the following temporal query:

```
VALID
  SELECT *
  FROM   Employees
  WHERE  NOT (Salary >= 8500);
```

This query is saved in a meta-table containing all column-constraints. At commit time, the constraint checker gets the relevant queries in the meta-tables and executes them. As soon as the query returns a resulting tuple, the column-constraint specified in example 4.18 is violated. In this case, the constraint checker writes an error message out to the screen, causes the DBMS to perform a rollback and quits checking constraints. Assertions are handled similarly.

Example 4.19 *We want to add an assertion to check that no employee ever earns more than his manager. In ATSQL2, this assertion can be specified as follows:*

```
CREATE ASSERTION Min_Salary
  VALID CHECK (
    NOT EXISTS (SELECT *
                FROM   Employees e1, Employees e2
                WHERE  e1.MgrID = e2.EmpID AND
                       e1.Salary > e2.Salary));
```

The assertion is translated into the temporal query

```
VALID
  SELECT *
  FROM   Employees e1, Employees e2
  WHERE  e1.MgrID = e2.EmpID AND
         e1.Salary > e2.Salary;
```

The temporal query is saved together with the quantifier (specifying that the condition `NOT EXISTS` was chosen) in a meta-table containing all assertions. At commit time, the constraint checker also gets the relevant queries from this meta-table and evaluates them. Depending on the type denoting whether `EXISTS` or `NOT EXISTS` was chosen as a quantifier, the queries have to return at least one tuple or no tuple, respectively. If a constraint is violated, the constraint checker writes an error message to the screen, causes the DBMS to do a rollback and quits.

4.6 The TimeDB DBMS

This section illustrates the general architecture of the DBMS TimeDB [Ste95], shows the steps of the rewriting algorithm, gives a short overview on the additional meta-tables used and describes the query interface.

4.6.1 Architecture of the TimeDB DBMS

The architecture of the DBMS TimeDB is shown in figure 4.8. Since suppliers of commercial DBMS do not make the source code of their DBMS available, TimeDB had to be implemented using a layered approach. It is built as a front-end to the commercial DBMS Oracle. The disadvantage of such an approach is the limitation in performance.

TimeDB was written in Prolog. The communication between the front-end and the DBMS Oracle was implemented using the C interfaces of both Prolog [Swe93] and Oracle [SD94]. Special Prolog predicates were implemented which use the C interface of Prolog to execute SQL statements on the DBMS. Previously, this architecture has been successfully used for deductive DBMS prototypes such as ProQuel [Bur92], the temporal deductive DBMS ChronoLog [Böh94] and the constraint DBMS DeCoR [Gro96].

4.6.2 Steps of the Rewriting Algorithm

Figure 4.8 shows the modules and the data paths of TimeDB. The modules *parser*, *translator* and *evaluator* call operations implemented in modules *scanner*, *checker* and *coalescing*, respectively.

The *scanner* reads an ATSQL2 statement, generates a token list and passes it to the parser. The *parser* then checks if the statement is a legal ATSQL2 statement. In case of a syntax error, an error message is displayed and the translation is stopped. Otherwise a parse tree is generated and passed to the translator. The parse tree reflects the ATSQL2 statement in a symbolic notation which is more convenient to handle in Prolog than strings.

The *translator* checks if all tables and attributes referenced in the statement actually exist. In the case of a temporal query, it checks if the tables are of the right kind (valid-time, transaction-time

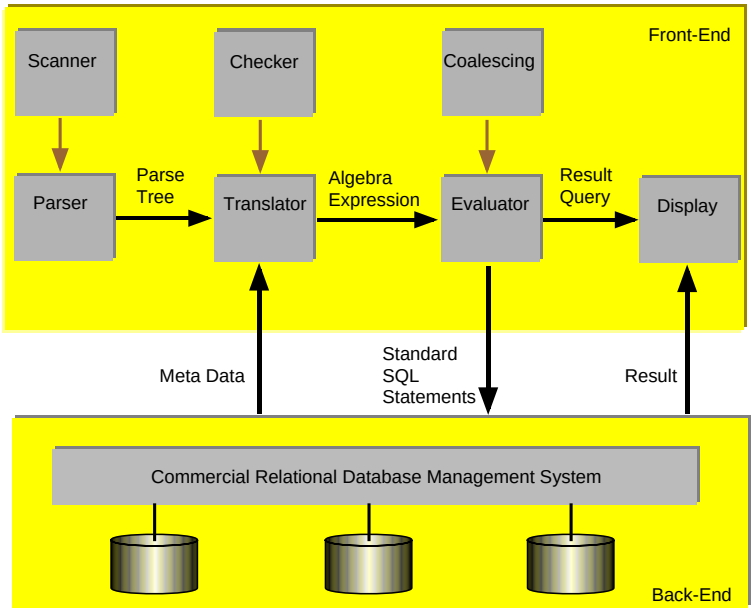

Figure 4.8: Architecture of TimeDB

or bitemporal tables). Additionally, other checks – for example, for type and union compatibility – are done wherever needed. The module *checker* provides operations for these checks. Again, if any error is detected, an error message is displayed and translation is stopped. Otherwise, the parse tree is translated into an algebraic expression which is then sent to the evaluator.

The *evaluator* executes the algebra expression on the commercial relational DBMS, using the algorithms presented in section 4.4. As mentioned previously, intermediate results are stored in auxiliary tables when needed, which are automatically created and dropped by the TimeDB system. The evaluator then sends a system-generated query to the *display* module, which is used to fetch the final result from the database. These tuples are presented in a tabular form on the screen.

4.6.3 Meta-Tables

Due to the new kinds of tables, views, constraints and assertions, TimeDB stores its own meta-data. As already seen before, queries to check temporal table- and column-constraints and temporal assertions are stored in separate meta-tables. This subsection gives a short description of the new meta-tables introduced in TimeDB. These meta-tables are

- `table_types`
- `views`
- `table_column_constraints`
- `refint`
- `assertions`

All meta-tables in TimeDB are snapshot relations. The timestamping of meta-data and its influences on the model have not been considered in TimeDB.

Temporal queries can only be evaluated on temporal tables. To be able to quickly find out the type of a table (snapshot, valid-time, transaction-time or bitemporal table), this information is stored explicitly in meta-table `table_types`. The parser, for example, accesses this data to check whether the tables referenced in a temporal query are of the right kind with respect to time.

Temporal views are stored in meta-table `views`. The meta-data of a temporal view contains the name of the view and the query calculating the view.

There are three additional meta-tables for constraints, one for table- and column-constraints, one for referential integrity constraints and one for assertions. The `table_column_constraints` meta-table has two attributes, namely `table_name` and `query`. Since all table- and column-constraints are stored in the same meta-table, TimeDB uses the table name to access the constraints of a single table. Referential integrity constraints are stored in meta-table `refint`. To distinguish between non-temporal and temporal referential integrity constraints, TimeDB stores table-name and name of the *referencing* column(s) and table-name and name of the *referenced* column(s) together with the information as to whether the constraint has to be evaluated temporally or not. Temporal assertions are stored in meta-table `assertions`. Each assertion has a name and a quantifier specifying if the assertion verifies the existence or non-existence of particular values. Additionally, as seen before, the assertion itself is translated into a (temporal) query which is used to look for these particular values. Meta-table `assertions` thus contains the three attributes `name`, `type` and `query`.

4.6.4 User Interface of TimeDB

TimeDB supports a command-oriented user interface, which means that, after starting TimeDB, a prompt appears where commands can be entered. After opening a database, the system is ready to execute statements written in ATSQL2. In order to see the standard SQL commands generated by TimeDB, the system can be set to a trace mode.

```
{TimeDB 1.07, November 1, 1996}
{Andreas Steiner, ETH Zuerich, Switzerland}

ATSQL2> open 'EmployeeDB';

Database opened


ATSQL2> VALID SELECT * FROM Employees;

                    VALID empid   name salary mgrid
 ----------------------- ----- ------ ------ -----
    [1996/10/21-forever)   112   Paul   9100   111
 [1996/10/21-1996/10/29)   111   John   8700   111
    [1996/10/29-forever)   111   John   9300   111
    [1996/10/21-1996/11)   113 George   8300   111
       [1996/11-forever)   113 George   8300   112

ATSQL2> trace;

...
```

4.7 Summary

This chapter has presented how a bitemporal relational DBMS can be built as a front-end to a commercial non-temporal relational DBMS, translating temporal statements into standard SQL statements. The query translation approach in TimeDB is general in the sense that it can be used for other temporal query languages than ATSQL2. ATSQL2 covers all aspects of a language for DBMS – query language, data definition language, data modification language and constraint specification language.

This approach of building a temporal DBMS is what was introduced in chapter 1 as the *extension approach* with respect to the data structures. ATSQL2 assumes special attributes which store the timestamps, and TimeDB implements them using two attributes for a time interval. This means that in order to store the history of values, the schemas of the non-temporal relations are extended. ATSQL2 however uses the generalisation approach with respect to query and modification language and the constraints. Each of them is generalised into a temporal form. For example, each algebra operation is generalised into a temporal operation, a requirement stated in the definition of temporal completeness.

TimeDB can be used not only to demonstrate the power of ATSQL2, but also to implement and test different query optimisation strategies. An early version of TimeDB supported a semantic query optimiser which used temporal conditions in selection operations to derive new conditions which helped to restrict the amount of data handled during query evaluation [Pul95]. It is possible to substitute the implementations of algebra operations in TimeDB with more efficient ones, or add special indexing techniques for faster query evaluation. TimeDB can also be used to *compile* temporal queries into standard SQL queries which then can be added to application programs running on commercial relational DBMS.

TimeDB is available as public domain software. Several universities use it to introduce students to the area of temporal databases. Other ones are currently developing extensions for TimeDB which add indexing techniques for temporal data.

For better performance, a commercial temporal relational DBMS should incorporate the temporal algebra operations and the temporal constraint checking mechanisms *directly*. TimeDB is used to demonstrate the functional power of temporal DBMS, to supply other researchers with an implementation they can extend with their own ideas and to give a platform for discussion.

Chapter 5

A Temporal Object Data Model : TOM

So far, all the temporal data models introduced in this thesis are based on the schema extension approach. Chapter 3 has introduced several of these proposals which all extend the schemas or types to timestamp data. With respect to query languages, most of these proposals add special operations to the query language, for example, to select historical data or to support temporal joins. As discussed in chapter 4, ATSQL2 goes a step further. It generalises the query and modification language and the constraints into temporal ones. This chapter introduces a new temporal object data model which is not based on the extension approach but rather generalises all concepts of an object data model into temporal ones.

The first section of this chapter summarises the deficiencies of the extension approach with respect to TimeDB and describes the ideas behind generalising the object data model OM into the temporal object data model TOM. The second section recalls the main features of the OM model in terms of a simple example application used throughout this chapter. Next, the basic constructs of the temporal object data model TOM are presented in terms of temporal objects, roles and associations. Then, the temporal generalisation of classification and association constraints are discussed. Finally, the temporal algebra and query language are presented. Additionally, examples of queries, data definition and data modification statements are given which can be run in the prototype DBMS TOMS. TOMS (Temporal Object Model System) is an implementation of the TOM data model.

5.1 Generalising an Object Data Model

ATSQL2 is not generic. One of the strengths of the relational data model is that it is generic in that it does not assume anything about the underlying type system. ATSQL2, though, assumes special attributes which store the time information, and its temporal algebra operations and constraints access them. Additionally, as most other proposals, ATSQL2 suffers from the vertical temporal anomaly. The history of a real world entity is split up into several tuples. In order to find maximal time periods, the user has to apply the coalescing operation.

ATQSL2 is not truly orthogonal. For example, the timestamping of meta-data and its influences on the data model and language have not been considered. It can be useful, however, to store at least the transaction-time of relations, for example. Since relations are created and dropped – which, on a different level, corresponds to an insert and delete – users might want to know when relations were stored in a database. Storing the transaction time of relations means that operations also have to verify whether or not relations exist in the database states relevant for the operation.

This chapter introduces a new temporal object data model which overcomes these deficiencies. The non-temporal generic OM model [Nor92, Nor93] is *generalised* into the temporal object data

model TOM [SN97c, SN97a]. This means that all aspects of the model – constructs, operations and constraints – have temporal generalisations. Further, the temporal dimension applies not only to data but also to meta-data.

To achieve this, a new form of object timestamping is introduced. Instead of adding validity time periods to objects in form of special attributes, denoting when a specific *state* of an object was valid (or stored, respectively), the notion of *temporal object identifiers* is introduced in TOM which timestamp objects with their *overall time period of existence*. This approach does not make any assumptions about the underlying type system. Thus, TOM is a *generic* temporal object data model. An object then can be viewed through the different roles it was, is or will be playing. Depending on the role through which an object is viewed, only that *part of the lifespan* of the object is *visible* during which the object took this role. Additionally, during the objects visibility in a role, the state of the object may vary, yielding the history of the attribute values. Attribute value histories, however, have to be provided for on the type system level.

Since collections are also objects, the approach of temporal object identifiers leads naturally to a representation of the lifespans of both objects and object roles. Further, associations – including the membership associations between objects and the collections to which they belong – are timestamped, allowing both object role histories and association histories to be modeled.

For the operational part of the model, a similar approach to that proposed in ATSQL2 is used. TOM supports a full temporal algebra, a temporal query language and temporal constraints. Additionally, meta-data can have temporal properties allowing the modeling of role, association and constraint lifespans. The collection algebra of OM was generalised into a temporal algebra using as guidelines the requirements specified in the definition of temporal completeness. Additionally, the notions of upward and temporal upward compatibility were also reused.

5.2 The Non-Temporal Object Data Model OM

As we have already mentioned in chapter 1, the OM data model [Nor92, Nor93] allows the specification of semantic groupings of objects and their interrelationships and thus deals with issues of classification and associations between objects at the same level of abstraction. The characteristics of objects in terms of interface and implementation would be specified by the type system associated with a particular DBMS in which the model is used to support object data management. This separation of typing from classification is beneficial, not only in terms of the universality of the model, but also in distinguishing issues of representation from those of data semantics. Further, it allows an Entity-Relationship style of conceptual modeling to be combined with the power and flexibility of object-oriented systems.

The basic construct of the OM model is the collection in that both objects and relationships are classified into collections of a given member type. Unary collections are those which have unary values as elements and represent object roles. Binary collections are those which have pair values as elements and represent relationships between entities. Relationships are actually represented by associations. An association consists of a binary collection together with constraints that specify the roles of objects that may participate in the relationship and the corresponding cardinalities. A collection is itself an object which provides the capability to have collections of collections and so on.

Collections are grouped into classification structures each of which describes related object roles in terms of a generalisation/specialisation graph. This is illustrated by means of the example schema for a property leasing company shown in figure 5.1.

Figure 5.1 shows four classification structures. The classification structure on the right represents real estate properties and consists of the collection `Properties` and its subcollections `Residences`, `Offices`, `Rented`, `Available` and `Renovating`. Recall that shaded boxes are used to denote collections with the name of the collection in the unshaded region and the type of the member values specified in the shaded region.

Subcollections `Residences` and `Offices` are constrained to be *disjoint* meaning that, at any point in time, no property can be categorised for use as both a residence and an office. Subcollec-

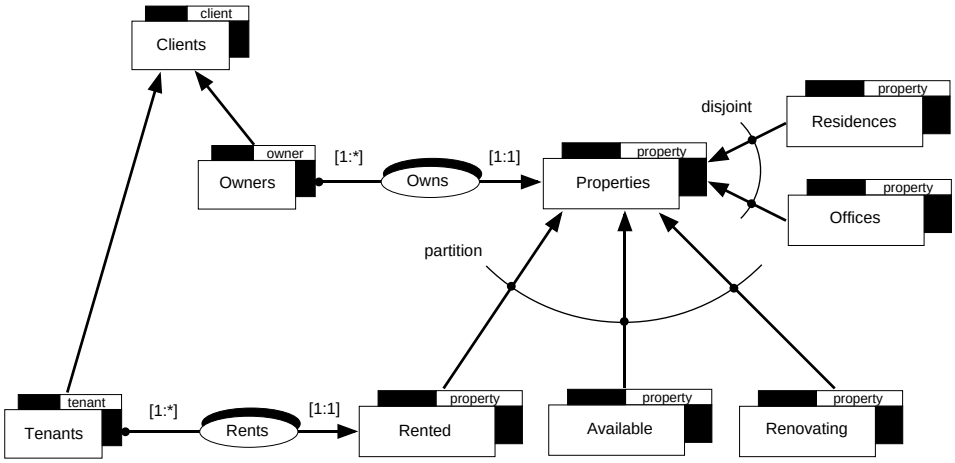

Figure 5.1: An example OM schema diagram

tions `Rented`, `Available` and `Renovating` form a *partition* in that they are pairwise *disjoint* and form a *cover* of `Properties` in that, at any point in time, every property must have exactly one of these three roles.

A second classification structure represents clients and their roles. Collection `Clients` has subcollections `Owners` and `Tenants`. The member type `tenant` of `Tenants` is a subtype of the member type `client` of `Clients`. Likewise, `owner` is also a subtype of `client`. It is possible that a client may be both a tenant and an owner and hence that a `client` object belongs to both `Owners` and `Tenants`.

The third and fourth classification structures consist of the single associations `Owns` and `Rents`, respectively. Associations are represented by oval-shaped boxes, together with links to the collections related by the association and their respective cardinality constraints. In order to specify operations over associations, it is necessary to specify a direction of such a relationship. For example, `Owns` would actually be represented by a collection of pairs of object values such that the first elements of the pairs belong to `Owners` and the second elements belong to `Properties`. Collection `Owners` is referred to as the *source collection* and `Properties` as the *target collection* of `Owns`.

OM supports object evolution in that objects may change their roles over the course of time [NSWW96]. Such forms of evolution require changes in collection membership and this in turn may involve changes in the type of an object which is called *object metamorphosis*. For example, if a `tenant` object becomes an `owner` object, then the object must gain additional owner properties. OM supports object metamorphosis through *dress* and *strip* operations. Further, the model includes mechanisms to control object evolution. For example, objects can only migrate within a classification structure, thereby preventing absurd evolutions such as an object in `Tenants` becoming an object in `Properties`.

The operational model of OM is based on a *collection algebra*. A list of operations supported in OM is given in table 5.1. The algebra includes operations such as *union*, *intersection*, *difference*, *cross product*, *selection*, *map*, *flatten* and *reduce* as well as special operations over binary collections such as *domain*, *range*, *inverse*, *compose* and *closure*. In the remainder of this section, a short overview of these algebra operations is given. Definitions of most of their temporal equivalents together with example queries are given in section 5.5.

The algebra operations are given specifying the input arguments and the results along with their types. Collections which list two types, for example $coll[(Type_1, Type_2)]$, are binary collections, containing objects of the form $<\texttt{oid}_1, \texttt{oid}_2>$, where $\texttt{oid}_1$ refers to an object of type $Type_1$ and $\texttt{oid}_2$ to an object of type $Type_2$.

For set operations, [Nor93, Nor92] use the notion of *least common supertype* and *greatest common subtype* to determine the member type of the resulting collections. The *common super-*

Algebra Operation	Signature
Union	$\cup : (coll[Type_1], coll[Type_2]) \rightarrow coll[Type_1 \sqcup Type_2]$
Intersection	$\cap : (coll[Type_1], coll[Type_2]) \rightarrow coll[Type_1 \sqcap Type_2]$
Difference	$\Leftrightarrow : (coll[Type_1], coll[Type_2]) \rightarrow coll[Type_1]$
Cross Product	$\times : (coll[Type_1], coll[Type_2]) \rightarrow coll[(Type_1, Type_2)]$
Selection	$\sigma : (coll[Type], Type \rightarrow \texttt{boolean}) \rightarrow coll[Type]$
Map	$\texttt{map} : (coll[Type_1], Type_1 \rightarrow Type_2) \rightarrow coll[Type_2]$
Flatten	$\texttt{flatten} : coll[coll[Type]] \rightarrow coll[Type]$
Reduce	$\texttt{reduce} : (coll[Type_1], (Type_1, Type) \rightarrow Type, Type) \rightarrow Type$
Domain	$\texttt{domain} : coll[(Type_1, Type_2)] \rightarrow coll[Type_1]$
Range	$\texttt{range} : coll[(Type_1, Type_2)] \rightarrow coll[Type_2]$
Inverse	$\texttt{inv} : coll[(Type_1, Type_2)] \rightarrow coll[(Type_2, Type_1)]$
Compose	$\circ : (coll[(Type_1, Type_2)], coll[(Type_3, Type_4)]) \rightarrow coll[(Type_1, Type_4)]$
Closure	$\texttt{closure} : coll[(Type_1, Type_2)] \rightarrow coll[(Type_1, Type_2)]$

Table 5.1: Algebra operations supported in the OM model

type (upper bound) of two types $Type_i$ and $Type_j$ is defined to be any type $Type_k$ such that $Type_i \leq_t Type_k$ and $Type_j \leq_t Type_k$, where $\leq_t$ denotes a subtype relationship. If $Type_k$ is a common supertype of $Type_i$ and $Type_j$, such that for any other common supertype $Type_l$ of $Type_i$ and $Type_j$, $Type_k \leq_t Type_l$, then $Type_k$ is the *least common supertype (least upper bound)* of $Type_i$ and $Type_j$ which is written as $Type_k = Type_i \sqcup Type_j$. Similarly, the *greatest common subtype (greatest lower bound)* of $Type_i$ and $Type_j$ is defined as a common subtype $Type_k$ such that for any other common subtype $Type_l$, $Type_l \leq_t Type_k$, which is written as $Type_k = Type_i \sqcap Type_j$. It can be easily shown that both $Type_i \sqcup Type_j$ and $Type_i \sqcap Type_j$ are unique.

The *union* of two collections thus returns a collection whose type is the least common supertype of the two collections involved. The resulting collection contains all the elements belonging to one (or both) of the argument collections. The *intersection* of two collections, on the other hand, returns a collection whose type is the greatest common subtype of the two collections involved, containing the objects which are members of both argument collections. The type of the resulting collection of the *difference* of two collections corresponds to the type of the first argument in the operation. The resulting collection contains exactly those objects in the first argument collection which are not also members of the second one.

The *cross product* of two collections returns a binary collection $coll[(Type_1, Type_2)]$ containing all combinations of an object of the first collection with an object of the second one. The *selection operation* has as arguments a collection and a function which selects objects from the collection. The result is a subset of the argument collection. The *map operator* applies a function with a signature $Type_1 \rightarrow Type_2$ to each object in the argument collection and returns a collection of objects of type $Type_2$. The *flatten operator* takes a collection of collections of the same member type and flattens them to a collection of type $Type$. The *reduce operator* allows the execution of aggregate functions over a collection of values. It has three arguments – the collection $coll[Type_1]$ over which the aggregate function is executed, the aggregate function itself with a signature $(Type_1, Type) \rightarrow Type$ and an initial value of type $Type$.

The OM model also supports special operations over binary collections. Some of them are listed in the second part of table 5.1. The *domain* operation takes a binary collection and forms a collection of all the objects that appear as the first element of a pair of objects $<\texttt{oid}_1, \texttt{oid}_2>$ belonging to the binary collection, whereas the *range* operation returns a collection containing the second elements of such pairs. The *inverse* operation swaps the first and the second element of the pairs contained in the argument binary collection and returns them in a new binary collection. The *compose* operation combines those binary objects of the two argument collections where the second object in the first binary object $<\texttt{oid}_1, \texttt{oid}_2>$ appears as first object of the second binary object,

$<\texttt{oid}_2, \texttt{oid}_3>$. The resulting collection contains binary objects, for example, $<\texttt{oid}_1, \texttt{oid}_3>$. The *closure* of a binary collection is the reflexive transitive closure of a relationship represented by the binary collection.

5.3 Generalised Temporal Data Structures

This section describes how the data structures of the OM model were generalised into temporal ones. Additionally, the notions of temporal objects, temporal collections, temporal object evolution and temporal associations are defined. First, the fundamental ideas of object lifespans and visibility are introduced.

5.3.1 Object Lifespans and Visibility

The temporal object data model TOM is based on *object timestamping*. However, in contrast to the proposals discussed in chapter 3, TOM does not extend the *types* but rather extends the *object identifiers* with a timestamp to give temporal object identifiers of the form

$$\texttt{toid} := \ll \texttt{oid}; \texttt{ls} \gg$$

where `oid` is an object identifier and `ls` is a timestamp referred to as the *lifespan* of an object. It expresses, for example, when an object was *valid* (existent) in the real world. Thus, not the values of an object are timestamped but the object itself with its overall time of existence. The history of its values is kept track of separately. This means that it is not necessary to calculate the union of all timestamps of values of an object in order to establish the time period during which the object existed. Figure 5.2 shows the lifespan of an object. It may exist during several non-overlapping time periods. For example, assume we want to store information about the history of different countries. Poland existed as an independent country during the time periods [1025 $\Leftrightarrow$1795), [1918 $\Leftrightarrow$1939) and again since 1945, so its lifespan contains three non-overlapping time intervals.

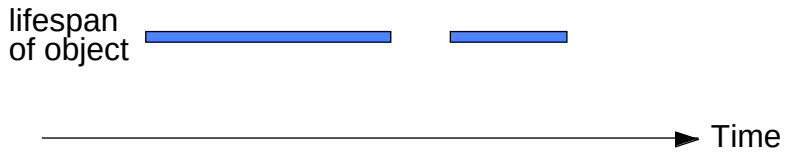

Figure 5.2: Lifespan of an object in TOM

Timestamps may also be associated with relationships between objects which are represented by member pairs of binary collections. In this case, each pair of object identifiers $(\texttt{oid}_1, \texttt{oid}_2)$ is tagged with a timestamp to give elements of the form

$$\ll (\texttt{oid}_1, \texttt{oid}_2); \texttt{ls} \gg$$

where `ls` is a lifespan as before.

Since object roles are represented by collections which are themselves objects, collections may also be timestamped. As a result, the fact that roles also exist for limited lifespans can be modeled and, further, that they may appear and disappear with respect to the current state of an application domain. For example, assume that the property leasing company whose schema is given in figure 5.1 initially only managed properties owned by its parent company. Later, it decided to generalise and also lease properties owned by others. After a while, it ceased to manage other owner's properties while the general leasing market declined. When the market picked up a few years later, it resumed leasing of other people's properties. This can be modeled through the lifespan of the `Owners`

collection. Similarly, associations which represent relationship roles between objects may also be timestamped.

The next stage to consider is how to model the times at which a particular entity has a particular role, for example, that an object is a member of a collection. An object may be in several collections at one time and may migrate between collections. Looking at the example schema in figure 5.1, a tenant is a member of collection `Tenants`. This means that the property leasing company helped him to find a property to rent. It is possible that over some interval, a tenant is also an owner of a property leased by the company. For example, they may own a property in one city and lease another in a different city. In this case, the tenant would appear in both the `Tenants` and `Owners` collections of the company's database during this time period.

An object's *visibility* in a temporal collection is determined by the overall lifespan of the object, the collection's own lifespan and a membership time $\mathtt{t}_{user}$ specified by the user. This is depicted in figure 5.3. Thus, if we look at a `client` object through the `Owners` role, it is only visible during the time period he is actually an owner, even though the object existed before (or after) its membership in `Owners`.

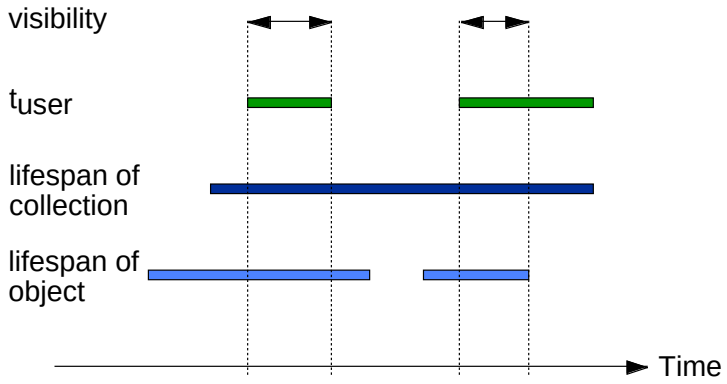

Figure 5.3: Visibility of an object in a collection

The history of the attribute values of a real world object is stored within a single object. The possibility of role modeling allows a real world object to be classified in different roles. The visibility of an object in a collection denotes the time period during which it had a specific role. So, with the approach of lifespans, visibility and roles, it is possible to model, for example, when a person is an employee, and querying the corresponding time period is straightforward. In example 2.9 of chapter 2, it was shown how such a query is done with respect to a relational data model using coalescing. It was stated there that it is up to the user to find out which are the time-varying attributes which need to be projected away prior to coalescing a table for the desired result.

The notion of a lifespan was already used in other temporal data models, for example, in [CC87, CC93], [DW92, WD93] and [EW90, EWK93a]. In [CC87, CC93], the lifespan of an employee denotes those times for which that employee is relevant to the users of the database. In [DW92, WD93], the lifespan of a type is the union of lifespans of all objects in the type extent, and the lifespan of a database is the union of lifespans of all types defined in the database. In [EW90, EWK93a], the temporal element of the surrogate attribute defines the entity lifespan. These notions of lifespans are different from the notion of a lifespan in TOM. In TOM, the lifespan of an object denotes when the object existed in the real world, regardless of any roles it is playing. For example, a human being is born and dies, regardless of any role. When looking at such an object through a specific role, only that part of the lifespan is visible during which the object had that role. The notions of a lifespan in [CC87, CC93], [DW92, WD93] and [EW90, EWK93a] refer to what in TOM is considered to be the visibility rather than the existence of an object.

Adding timestamps to objects leads naturally to a more general model than the usual relational temporal models in that, not only entities and their roles, but also the roles themselves can have

temporal properties. By timestamping objects (and object-pairs in binary collections), a direct comparison can be made between lifespans of objects, relationships, object roles and associations. In the following, these various aspects of the data model TOM are considered in more detail.

5.3.2 Temporal Object Identifiers

As introduced in chapter 2, a *chronon* [TCG+93] is the smallest non-decomposable time unit assumed in a temporal database, for example, a *second*. Despite the different interpretations of lifespans, the same definition for a lifespan is used in TOM as given in [CC87, CC93]. Let $\mathcal{T} = \{t_0, t_1, \ldots\}$ be a set of chronons, at most countably infinite. The linear order $<_\mathcal{T}$ is defined over this set, where $t_i <_\mathcal{T} t_j$ means that t_i occurs before t_j.

A lifespan `ls` is any subset of the set $\mathcal{T}$. As mentioned in chapter 2, [GV85] called this sort of timestamp *temporal elements*. $\mathcal{T}$ is assumed to be isomorphic to the natural numbers. Thus, a lifespan can also be represented as a set of non-overlapping intervals, closed at the lower bound and open at the upper bound.

This definition of *lifespan* reflects the fact that an object may appear and disappear several times during its overall time of existence. A lifespan contains all those time points at which an object existed.

Now, the notion of a temporal object identifier is formally introduced and a snapshot operation defined for it.

Definition 5.1 (Temporal Object Identifier) *Let O be the set of all possible non-temporal object identifiers. A temporal object identifier* `toid` *consists of an object identifier* `oid` $\in O$ *and a lifespan* `ls` *:*

$$\texttt{toid} := \ll \texttt{oid}; \texttt{ls} \gg$$

In the following, the notation $lifespan(\texttt{toid})$ is used to reference the lifespan contained in the temporal object identifier `toid`. O^v shall denote the set of all *temporal object identifiers*, whereas O is the set of non-temporal object identifiers. Value ω represents the *undefined* object identifier.

Definition 5.2 (Snapshot of a Temporal Object Identifier) *Let* `toid` $\in O^v$ *be a temporal object identifier containing* `oid` $\in O$ *as object identifier. Let $t \in \mathcal{T}$ be a time instant. Then $\tau_t(\texttt{toid})$ is the* snapshot of a temporal object identifier *at a time instant t and defined as*

$$\tau_t(\texttt{toid}) := \texttt{IF } t \in lifespan(\texttt{toid}) \texttt{ THEN oid ELSE } \omega$$

The snapshot of a temporal object identifier at time instant t returns the object identifier `oid` if the object exists at t, otherwise the special value ω is returned. Definition 5.2 is required for later definitions.

5.3.3 Valid-Time Objects

In TOM, the term *value* is used to mean any form of data item that can be described by the underlying type system, for example, a base value such as an integer value or a complex value such as an object value. For simplicity of presentation, simply integers and strings are assumed here as base values.

Let V_I be the set of all integer values and V_S the set of all string values. The values $v \in (V_I \cup V_S)$ have an implicit lifespan $[0 \Leftrightarrow \infty)$. The snapshot operation τ_t evaluated on an integer or string value thus always returns the integer or string value itself.

The set of *values* available in the temporal object data model TOM is defined as

$$V^v := V_I \cup V_S \cup O^v$$

With definition 5.2, it is now possible to express what the snapshot of values $v \in V^v$, $\tau_t(v)$, returns. If v is an integer or a string value, then the integer or string value itself is returned. If v is a *temporal* object identifier, then a *non-temporal* object identifier (or ω) is returned.

Valid-time objects are objects having a temporal object identifier. Depending on the role they play (meaning the collection they are member of), they show corresponding property values. In the following, the *temporal object identifier* of a valid-time object obj is referred to by $toid(obj)$, the *lifespan* of this object by $lifespan(obj)$ and the *object identifier* by $oid(obj)$.

Example 5.1 *When creating a valid-time object in the prototype DBMS TOMS, an implementation of the data model TOM, a set of valid-time periods expressing the object's lifespan has to be provided by the user:*

```
create object andreas lifespan { [1964 - inf) };
create object antonia lifespan { [1969 - inf) };
create object moira lifespan { [1970 - inf) };
...
create object herbert lifespan { [1964 - inf) };
...
create object apart1 lifespan { [1980 - 1995) };
create object apart2 lifespan { [1970 - inf) };
...
```

As mentioned before, these objects will be dressed with a type when added to a collection. In TOMS, names such as `andreas` *are used to refer to objects which is easier than using plain object identifiers. Time instant* `inf` *denotes that the object is valid until further notice. Non-temporal objects are created by leaving away the lifespan specification.*

5.3.4 Valid-Time Collections

A collection contains objects which are of the same member type. In fact, in its full generality, a collection can contain values of any type – including object values – but, in the discussion here, the focus is on the case of *collections of objects*. Recall that a collection is itself an object. *Valid-time collections* are collections having a *lifespan* and containing *valid-time objects* which have their own lifespans.

Definition 5.3 (Valid-Time Collection) *A* valid-time collection C *consists of*

- *a temporal object identifier* $\texttt{toid} \in O^v$, $toid(C) = \texttt{toid}$, *and*
- *a collection extent* $ext(C) \subseteq V^v$.

$C = [\texttt{toid}, \texttt{ext}]$ denotes a valid-time collection. Since a valid-time collection is also a valid-time object, the *temporal object identifier of a valid-time collection* C can be referenced by $toid(C)$, the *object identifier* by $oid(C)$ and its *lifespan* by $lifespan(C)$.

Example 5.2 *In order to create the collections depicted in the figure 5.1 as valid-time collections in TOMS, we first have to define the corresponding member types:*

```
create type client(name : string);
create type tenant(profession : string) subtype of client;
create type owner(bank_account : string) subtype of client;

create type property(price : integer; street : string; city : string);
```

Now we can create the main valid-time collections `Clients` *and* `Properties`*. Assume that the property leasing company started to exist in 1980.*

```
create collection Clients type client lifespan { [1980 - inf) };
create collection Properties type property lifespan { [1980 - inf) };
```

The snapshot of a collection extent $ext(C)$ of a valid-time collection C at a time instant t is defined to be the set of those values in the extent of C, which exist at time instant t. These snapshot values have no time information attached.

Definition 5.4 (Snapshot of a Collection Extent) *The* snapshot of the collection extent $ext(C) \subseteq V^v$ *at a time instant,* $\tau_t(ext(C))$*, is defined as*

$$\tau_t(ext(C)) := \{v | \exists v^v \in ext(C) \wedge v = \tau_t(v^v) \wedge v \neq \omega\}$$

Definition 5.4 will be needed to define the temporal subcollection relationship (definition 5.7) and the temporal membership relation (definition 5.9). Now, the snapshot of a valid-time collection at a particular time instant can be defined to be a non-temporal collection which is valid at this time instant.

Definition 5.5 (Snapshot of a Valid-Time Collection) *Let C be a valid-time collection with a temporal object identifier* `toid` *and an extent $ext(C)$. The* snapshot of the valid-time collection C, $\tau_t(C)$*, is defined as*

$$\tau_t(C) = [\tau_t(toid(C)), \tau_t(ext(C))]$$

If the temporal object identifier `toid` of a valid-time collection C is undefined (ω) at time instant t, then the extent of this collection is also undefined at t. This means that the objects in the extension are not visible at t.

5.3.5 Valid-Time Subcollection Relationship

The generalisation approach makes it also necessary to redefine the subcollection relationship between two collections. This section introduces the valid-time subcollection relationship used in TOM. First, the time instant definition of the subcollection relationship is given. The term $C_1 \preceq^t C_2$ shall denote that collection C_1 is a subcollection of collection C_2 at time instant t.

Definition 5.6 (Subcollection Relation at a Time Instant) *Let C_1 and C_2 be valid-time collections. C_1 is a* subcollection *of C_2 at time instant $t \in \mathcal{T}$, $C_1 \preceq^t C_2$, if and only if all of the following conditions hold:*

1. $\tau_t(toid(C_1)) \neq \omega$
2. $\tau_t(toid(C_2)) \neq \omega$
3. $\tau_t(ext(C_1)) \subseteq \tau_t(ext(C_2))$

Using definition 5.6, the subcollection constraint for the temporal object data model TOM can be defined. In the prototype system TOMS, this constraint is used to trigger actions such as update propagations to ensure that database consistency is maintained [NSWW96].

Definition 5.7 (Valid-Time Subcollection Relationship) *Let C_1 and C_2 be valid-time collections. The* valid-time subcollection relationship $C_1 \preceq^v C_2$ *holds if and only if the following holds:*

$$\forall t \in lifespan(C_1) \ : C_1 \preceq^t C_2$$

The valid-time subcollection relationship is defined over the lifespan of the subcollection. The valid-time subcollection relationship demands that for each time instant subcollection C_1 exists, supercollection C_2 also has to exist, but not vice versa. So, in the example, the lifespan of any subcollection of valid-time collection `Clients` must be contained in the lifespan [1980 $\Leftrightarrow\infty$).

Example 5.3 *We show how the subcollections depicted in figure 5.1 can be created. Assume that at first, the property leasing company only dealt with renting properties owned by a parent company. In 1982, the company decided to extend their operations and also lease properties owned by others. During the first five years, the company only dealt with residential properties. In 1985, they started to deal also with properties used as offices.*

```
create collection Tenants
   subcollection of Clients type tenant lifespan{ [1980-inf) };
create collection Owners
   subcollection of Clients type owner lifespan{ [1982-inf) };

create collection Residences
   subcollection of Properties type property lifespan { [1980 - inf) };
create collection Offices
   subcollection of Properties type property lifespan{ [1985-inf) };

create collection Rented
   subcollection of Properties type property lifespan { [1980 - inf) };
create collection Available
   subcollection of Properties type property lifespan { [1980 - inf) };
create collection Renovating
   subcollection of Properties type property lifespan { [1980 - inf) };
```

5.3.6 Adding and Removing Valid-Time Objects to Valid-Time Collections

Adding an object to a valid-time collection restricts the object's *visibility* in the collection in several ways. As stated previously, the object is *visible* only during a certain time period in the collection as determined by the collection's lifespan, the object's own lifespan and a membership time specified by the user. An object's *maximal visibility* in a collection is the collection's lifespan. This contrasts, for example, with the approach proposed in [GÖ93] where an object's lifespan has to be contained in the lifespan of the collection to which it is added.

The resulting visible lifespan $\mathtt{ls}$ of the added object is the intersection of the lifespan $\mathtt{ls}_O$ of the object with the lifespan of the collection $\mathtt{ls}_C$, intersected with the user specified membership time $\mathtt{t}_{user}$:

$$\mathtt{ls} := \mathtt{ls}_C \cap \mathtt{ls}_O \cap \mathtt{t}_{user}$$

Example 5.4 *We now want to add the valid-time objects of example 5.1 to valid-time collections created in example 5.2:*

```
insert object andreas into Tenants during { [1980 - inf) };
Give a value for name: Andreas
Give a value for profession: Assistant

insert object antonia into Tenants during { [1984 - 1996) };
...
insert object moira into Tenants during { [1992 - inf) };
...
insert object herbert into Owners during { [1982 - inf) };
...
insert object apart1 into Rented during { [1980 - 1995) };
...
insert object apart2 into Rented during { [1987 - inf) };
...
```

Andreas is a client of the company since 1980. He found a property with the help of this company and thus is a member of collection `Tenants` *in the company's database. When inserting objects into a collection, the system dresses the object with the corresponding member type (if it is not already dressed with it) and asks for attribute values (for example,* `name` *and* `profession`*). Additionally, objects are propagated automatically to super-collections if needed.*

In TOM, it is possible to add temporal objects to non-temporal collections and non-temporal objects to temporal collections. The idea is that a non-temporal object is assumed to exist at time instant *now*, where *now* actually moves on as time passes. So, if a temporal object is inserted in a non-temporal collection, it is visible if it exists at time instant now. The same idea of visibility as before is used, assuming that the collection's valid time $\mathtt{ls}_C$ corresponds to [now-now]. Deleting the object from the non-temporal collection means to actually remove it. On the other hand, if a non-temporal object is added to a temporal collection, the object's lifespan $\mathtt{ls}_O$ is assumed to be [now-now]. Deleting it from the temporal collection also removes it without leaving a trace. The same ideas can be used for handling transaction-time objects.

5.3.7 Object Evolution

As stated in section 5.2, objects must be allowed to evolve and change roles during their lifespan. This accounts for the fact that entities in the real world change their roles during their life. With respect to the example application used in this chapter, a tenant buys a property in another city which is then leased by the property leasing company. This client plays the role of a tenant and then gains the role of an owner. Such changes and accumulation of roles is reflected in TOM by the possibility that an object can migrate from one collection to another and may also be a member of several collections at the same time.

Each collection has an associated member type. This means that for a given collection C and a given type *Type*, if $member\_type(C) = Type$, then for any value x in the extent of C, x must be an instance of type *Type*. Thus, to change collection membership, an object must also be able to change its type while retaining the same object identity. This is referred to as *object metamorphosis*. Object evolution thus consists of the following steps: First change an object's type (or let it gain a new type) within the type hierarchy (object metamorphosis), possibly adding values for additional attributes, and then add the object to a new collection in the classification structure (object migration), possibly removing it from other collections.

Assume again classification structures as depicted in figure 5.1. If an object in collection `Clients` is also added to subcollection `Owners`, then the object first has to be *dressed* with member type `owner` of collection `Owners`. Then a valid-time period $\mathtt{t}_{user}$ has to be specified by the user which expresses the time the object was a property owner in the real world. The visibility of this object in the valid-time collection `Owners` then results in

$$\mathtt{ls} := \mathtt{ls}_{\mathtt{Owners}} \cap \mathtt{ls}_{object} \cap \mathtt{t}_{user},$$

where $\mathtt{ls}_{\mathtt{Owners}}$ represents the lifespan of collection `Owners` and $\mathtt{ls}_{object}$ corresponds to the lifespan of the client object to be added to collection `Owners`.

Example 5.5 *Assume Moira and Andreas both decided to buy properties, but remained in the properties already rented. They asked the leasing company to find tenants for their own properties. Thus, they also became owners in the company's database.*

```
insert object andreas into Owners during { [1982 - 1995) };
Give a value for bank_account: SBG 123-456

insert object moira into Owners during { [1982 - 1987, 1991 - inf) };
...
```

Andreas bought a property in 1982 and in 1995 he decided to have another company manage his property. Moira actually had her first property managed by the company in 1982. She sold the property in 1987 and bought another one in 1991. When inserting objects, the system again dresses the objects with the corresponding member type (if needed) and asks for attribute values. Object `andreas` *was already dressed with type* `person` *in example 5.4, so the system asks only for values specific for type* `owner`*.*

5.3.8 Temporal Associations

As described previously, relationships between objects are represented by associations. Relationships may also have valid times associated with them and these are represented by temporal associations. A temporal association is a valid-time binary collection together with constraints specifying the source and target collections and their respective cardinality constraints.

Definition 5.8 (Valid-Time Binary Collection) *A* valid-time binary collection C *consists of*

- *a temporal object identifier* $\texttt{toid} \in O^v$, $toid(C) = \texttt{toid}$, *and*
- *a collection extent* $ext(C) \subseteq (V \times V)^v$ *where* V *is the set of non-temporal values* $V_I \cup V_S \cup O$.

The extent of a valid-time binary collection will be a set of object value pairs together with a lifespan. As mentioned previously, given a valid-time binary collection C, then an element of $ext(C)$ may be of the form $\ll (\texttt{oid}_1, \texttt{oid}_2); \texttt{ls} \gg$ where $\texttt{oid}_1, \texttt{oid}_2 \in O$ and `ls` is the lifespan of the relationship.

Example 5.6 *According to the database schema depicted in figure 5.1, we have to create two valid-time associations* `Rents` *and* `Owns`*. The association* `Rents` *exists since 1980, when the company started. Since the company decided in 1982 to extend their activity and find tenants for property owners, the association* `Owns` *exists since 1982. Of course, both source and target valid-time collections have to exist during the lifespan of an association. This is checked by the system.*

```
create association Rents
  source Tenants
  target Rented
  lifespan { [1980 - inf) };

create association Owns
  source Owners
  target Properties
 lifespan { [1982 - inf) };
```

Now we can create associations between tenants and the properties they rent, and between owners and the properties they own:

```
insert binary object (andreas, apart1) to Rents during { [1980 - 1993) };
insert binary object (herbert, apart2) to Owns during { [1987 - inf) };
...
```

In case a temporal association references an object which does not exist at all or during the specified time period, an error message is produced. This means that *temporal referential integrity* needs to be checked for on both the collection and object level.

5.4 Temporal Constraints

This section discusses the issue of the temporal generalisation of the classification constraints in detail. The conditions imposed by the constraints are considered with respect to a particular time instant and then generalised over time. This is done by redefining the membership relation for a time instant. Then, the *valid-time* cover, disjoint and intersection constraints over *valid-time collections* can be defined.

The *non-temporal* membership relation of a value x in a set S is denoted by $x \in_{set} S$. The membership relation *at a time instant* is defined as:

Definition 5.9 (Membership Relation at a Time Instant) *Let C be a valid-time collection of elements of type* `Type`*, $member\_type(C) =$* `Type`*, and let $t \in \mathcal{T}$ be a time instant. Then for any value x :*`Type`*, $x \in V^v$, x is a member of C at time instant t, $x \in^t_{set} C$, if and only if both of the following conditions hold:*

1. $\tau_t(toid(C)) \neq \omega$
2. $\tau_t(x) \in_{set} \tau_t(ext(C))$

According to definition 5.4, ω is never a member of a set of values $\tau_t(ext(C))$. With definition 5.9, the valid-time disjoint, cover and intersection constraints can be defined.

Definition 5.10 (Disjoint Constraint at a Time Instant) *Let $t \in \mathcal{T}$ be a time instant. The* disjoint constraint at time instant t *over a set of valid-time collections CS, $disjoint^t(CS)$, is defined as*

$$disjoint^t(CS) :\Leftrightarrow \forall C_i, C_j \in CS : oid(C_i) \neq oid(C_j) \Rightarrow \neg\exists x : \; x \in^t_{set} C_i \wedge x \in^t_{set} C_j$$

If at least one of the two collections C_i or C_j is undefined at time instant t, then C_i and C_j are *disjoint* at time instant t due to definition 5.9.

Definition 5.11 (Valid-Time Disjoint Constraint) *The* valid-time disjoint constraint *over a set of valid-time collections CS, $disjoint^v(CS)$, is defined as*

$$disjoint^v(CS) :\Leftrightarrow \forall t \in \textstyle\bigcup_{C_j \in CS} lifespan(C_j) : disjoint^t(CS)$$

A set of valid-time collections CS is temporally disjoint, if no pair of member collections has a common member value at any time point.

Definition 5.12 (Cover Constraint at a Time Instant) *Let $t \in \mathcal{T}$ be a time instant, C a valid-time collection and CS a set of valid-time subcollections of C. The* cover constraint at time instant t, *$cover^t(C, CS)$, then is defined as*

$$cover^t(C, CS) :\Leftrightarrow \forall x \in^t_{set} C \; \exists C_j \in CS : x \in^t_{set} C_j$$

With definition 5.12, the valid-time cover constraint can be defined as

Definition 5.13 (Valid-Time Cover Constraint) *Let C be a valid-time collection and CS a set of valid-time subcollections of C. The* valid-time cover constraint, *$cover^v(C, CS)$, is defined as*

$$cover^v(C, CS) :\Leftrightarrow \forall t \in lifespan(C) : cover^t(C, CS)$$

A set of valid-time collections CS is a valid-time cover of a valid-time collection C, if each member of CS is a subcollection of C and each element of C appears in at least one collection of CS during each time instant of its existence.

Temporal partition constraints can be expressed by a combination of a temporal cover and a temporal disjoint constraint. The valid-time intersection constraint is defined following the definitions of the temporal cover and disjoint constraints. The intersection constraint demands that at any point in time, a collection C which is a subcollection of a set of collections CS, corresponds to the intersection of all collections in CS.

Definition 5.14 (Intersection Constraint at a Time Instant) *Let $t \in \mathcal{T}$ be a time instant, C a valid-time collection and CS a set of valid-time collections. C is a subcollection of each $C_j \in CS$. The* intersection constraint at time instant t, *$intersect^t(CS, C)$, then is defined as*

$$intersect^t(CS, C) :\Leftrightarrow \forall x \in^t_{set} C \; \forall C_j \in CS : x \in^t_{set} C_j$$

The valid-time intersection constraint now can be defined as

Definition 5.15 (Valid-Time Intersection Constraint) *Let C be a valid-time collection and CS a set of valid-time collections. The* valid-time intersection constraint, *$intersect^v(CS, C)$, is defined as*

$$intersect^v(CS, C) :\Leftrightarrow \forall t \in lifespan(C) : intersect^t(CS, C)$$

The example schema given in figure 5.1 only uses temporal cover and disjoint constraints. The temporal partition constraint is specified as two separate temporal cover and disjoint constraints.

Example 5.7 *The temporal disjoint constraint used in figure 5.1 demands that at each time point valid-time collection* `Properties` *exists, the subcollections* `Residences` *and* `Offices` *must be disjoint. The valid-time partition constraint demands that the three valid-time collections* `Rented`, `Available` *and* `Renovating` *form a partition at each time point valid-time collection* `Properties` *exists.*

```
create valid constraint disjointRO disjoint([Residences, Offices]);
create valid constraint coverRAR cover(Properties, [Rented, Available, Renovating]);
create valid constraint disjointRAR disjoint([Rented, Available, Renovating]);
```

The constraints specified in example 5.7 do not have lifespans. Previously, however, we stated that all information represented as objects – including constraints – may be timestamped. This means that with the object timestamping approach used in TOM, constraint objects may also be extended to temporal objects having a lifespan. The constraints given in example 5.7 are checked according to the definitions 5.11, 5.13 and 5.15. This means that the lifespans of the collections involved specify the time periods during which the constraints have to hold. Sometimes it is useful, however, to specify explicitly a time period during which a constraint has to hold. For example, the disjoint constraint for collections `Residences` and `Offices` could be changed to a partition constraint in 1990. This could be done the following way:

```
create constraint coverRO
   cover(Properties, [Residences, Offices]) lifespan { [1990 - inf) };
```

In this case, the valid-time cover constraint must have a lifespan which is contained in all lifespans of the collections involved. Then – instead of checking the constraint during all time instants of the lifespan of the supercollection – the constraint is tested during the lifespan of the constraint. The same holds for the other constraints.

5.5 Temporal Collection Algebra

So far, the temporal constructs of TOM have been introduced. Another aspect of the model is the generalisation of the collection algebra of the OM model to give equivalent temporal operations. As seen in section 5.2, all of the algebra operations in the OM model work on collections of objects and return a result collection of objects (except the reduce operation which returns a single value).

Two categories of operations are distinguished in the temporal algebra of TOM. The *first category* contains those operations which calculate a new lifespan for the result collection and new visibilities for the objects contained in it, for example, the *temporal composition operation*, the *temporal cross product* or *temporal set operations*. The *second category* of temporal operations only

works on object identifiers while retaining lifespans and visibilities. Examples are the *temporal inversion* or the *temporal domain* operations.

In section 5.2, the different non-temporal algebra operations have already been introduced. The specification of their input and output arguments also holds for the temporal equivalents. Now, some of the temporal algebra operations are discussed in more detail by considering example queries, explaining how they are evaluated and giving definitions for the temporal operations. In the prototype DBMS TOMS, it is possible to either use algebra expressions or an SQL-like syntax for querying.

Example 5.8 *We would like to know the history of tenants renting one of Herbert's properties. The temporal algebra expression calculating the corresponding result looks like*

$$\texttt{range}^v(\sigma^v_{left.name='Herbert'}(Owns) \circ^v \texttt{inv}^v(Rents))$$

This expression can be run in the prototype system TOMS as a query using either the algebra expression

```
valid range(compose(select (left.name = 'Herbert') Owns, inv(Rents)));
```

or an SQL-like statement

```
valid
  range((select own in Owns where left(own).name = 'Herbert') compose (inv Rents));
```

The extent of the resulting collection, having its own lifespan $[1982 \Leftrightarrow \infty)$*, contains, for example, tenant objects with the following visibilities and property values:*

VALID	Profession	Name
$[1995 - \infty)$	Assistant	Andreas
$[1992 - \infty)$	Professor	Moira

Operations in an algebra expression having a superscript v denote that they are evaluated using temporal semantics with respect to valid time. In TOM, *temporal semantics* is actually a synonym for snapshot reducible semantics. In example 5.8, all of the operations use temporal semantics. According to the approach proposed in [SBJS96b, SBJS96a], the keyword `valid` is used to denote that temporal evaluation semantics should be applied. In the example, the scope of keyword `valid` is the whole query.

In example 5.8, first those binary objects in the temporal association `Owns` are selected which have the object denoting owner Herbert on the left side. The valid-time selection is defined the following way:

Definition 5.16 (Valid-Time Selection in a Valid-Time Collection) *Let* C_1 *be a valid-time collection of type* `Type` *and* P *be a function that maps each element of* C_1 *to one of the Boolean values* `true` *or* `false`*. The* valid-time selection *of* C_1 *using function* P*,* $C = \sigma^v_P(C_1)$*, mapping collection* C_1 *to a collection* C *of type* `Type` *and valid-time* $lifespan(C) = lifespan(C_1)$*, is then defined as*

$$\forall t \in lifespan(C_1), \forall x \in C_1 : P(\tau_t(x)) = \texttt{true} \Leftrightarrow x \in^t_{set} C$$

Next, the temporal result collection of the selection operation is combined with binary collection `Rents`. The valid-time composition operation ($\circ^v$) composes out of two binary collections a new binary collection by taking the objects in the domain of the first and the objects in the range of the second and combining them if they have equal range and domain objects respectively. This operation belongs to the first category of operations where lifespan calculation is done. The formal definition of the temporal composition operation is

Definition 5.17 (Composition of two Valid-Time Binary Collections) *Let B_1 and B_2 be two valid-time binary collections of types* $(\texttt{Type}_1, \texttt{Type}_2)$ *and* $(\texttt{Type}_3, \texttt{Type}_4)$ *respectively. The valid-time composition of B_1 and B_2, $B = B_1 \circ^v_{set} B_2$, returns a binary collection B of type* $(\texttt{Type}_1, \texttt{Type}_4)$ *which has a lifespan equal to $lifespan(B) = lifespan(B_1) \cap lifespan(B_2)$ and is defined as*

$$\forall t \in lifespan(B) : \tau_t(ext(B)) = \{\ll x, z \gg | \exists y : \ll x, y \gg \in^t_{set} B_1 \wedge \ll y, z \gg \in^t_{set} B_2\}$$

Since collections in TOM also have lifespans, it has to be defined what the lifespan of a resulting valid-time collection shall be. A non-temporal DBMS returns an error if one of the arguments of an operation does not exist. In the temporal case, TOM is defined such that a resulting temporal collection only covers those time instants when all of the argument collections exist. Thus the result of a valid-time composition operation is valid only during the intersection of the two lifespans of the valid-time collections involved. This also holds for other temporal operations of the first category.

Definition 5.17 defines that those pairs of objects are combined where the right object of the first pair is the same as the left object of the second pair (during their common time period). In example 5.8, we want to find tenants of properties owned by Herbert. We combine owner objects in `Owns` with tenant objects in `Rents` through their common objects of type `property`. To be able to do that with a temporal composition operation, we first have to invert collection `Rents`. The valid-time inversion operation (inv^v) just switches source and target objects of a binary collection, leaving the timestamp the same. This operation belongs to what was earlier called the second category of operations in the temporal algebra of TOM. The formal definition of this operation is

Definition 5.18 (Inverse of a Valid-Time Binary Collection) *Let B_1 be a valid-time binary collection of type* $(\texttt{Type}_1, \texttt{Type}_2)$*. The valid-time inverse of B_1, $B = \texttt{inv}^v_{set}(B_1)$, returns a binary collection B of type* $(\texttt{Type}_2, \texttt{Type}_1)$ *which has a lifespan equal to $lifespan(B) = lifespan(B_1)$ and is defined as*

$$\forall t \in lifespan(B) : \tau_t(ext(B)) = \{\ll y, x \gg | \ll x, y \gg \in^t_{set} B_1\}$$

The result of the composition operation $\sigma^v_{left.name='Herbert'}(Owns) \circ^v inv^v(Rents)$ is a binary collection containing pairs having an owner object on its left and a tenant object on its right side together with their common time periods. Since we look for the tenant objects of this binary collection, only the range of the binary collection is of interest. The corresponding operation is the temporal range operation ($range^v$), which can be defined similarly to the temporal inversion and also belongs to the second category of temporal operations.

Definition 5.19 (Range of a Valid-Time Binary Collection) *Let B_1 be a valid-time binary collection of type* $(\texttt{Type}_1, \texttt{Type}_2)$*. The valid-time range of B_1, $C = \texttt{range}^v_{set}(B_1)$, returns a unary valid-time collection C of type* $\texttt{Type}_2$ *which has a lifespan equal to $lifespan(C) = lifespan(B)$ and is defined as*

$$\forall t \in lifespan(C) : \tau_t(ext(C)) = \{y | \exists x : \ll x, y \gg \in^t_{set} B_1\}$$

The next example uses a temporal set difference and a temporal cross product operation. The temporal cross product operation is similar to the temporal composition in that it calculates the common lifespan of both collections and objects involved and returns a valid-time binary collection. Its arguments, however, are *unary* valid-time collections.

Example 5.9 *We would like to find those residences and the corresponding time period during which no higher priced offices exist. The corresponding algebra expression looks like*

$$Residences \Leftrightarrow^v \texttt{domain}^v(\sigma^v_{left.price < right.price}(Residences \times^v Offices))$$

We can query the system either with the algebra expression

```
valid Residences -
      domain(select (left.price < right.price) (Residences * Offices));
```

or the SQL-like statement

```
valid
  select r in Residences
  where not exists (select o in offices where r.price < o.price);
```

The result of this query is a valid-time collection with a lifespan $[1985 \Leftrightarrow \infty)$*, and it contains, for example, residence objects with the following visibilities and property values:*

VALID	Price	Street	City
$[1985 - 1991)$	1200	Stegstrasse	Pfaeffikon
$[1987 - 1991)$	900	Hauptplatz	Rapperswil

All of the operations in the algebra expression have temporal semantics. This is denoted by the keyword `valid` at the beginning of the query whose scope again is the whole query. The valid-time cross product *Residences* $\times^v$ *Offices* generates pairs of object identifiers together with their common lifespan. It returns a collection of valid-time binary objects containing pairs of non-temporal object identifiers together with a lifespan which is calculated by the intersection of the lifespans of the objects involved. Formally, the temporal cross product is defined as

Definition 5.20 (Valid-Time Cross Product of Collections) *Let* C_1*,* C_2 *be valid-time collections of types* $\texttt{Type}_1$ *and* $\texttt{Type}_2$ *respectively. The* valid-time cross product *of* C_1 *and* C_2*,* $B = C_1 \times^v_{set} C_2$*, returns a binary valid-time collection* B *of type* $(\texttt{Type}_1, \texttt{Type}_2)$ *which has a lifespan equal to* $lifespan(B) = lifespan(C_1) \cap lifespan(C_2)$ *and is defined as*

$$\forall t \in lifespan(B) : \tau_t(ext(B)) = \{\ll x, y \gg | x \in^t_{set} C_1 \wedge y \in^t_{set} C_2\}$$

In the result of *Residences* $\times^v$ *Offices*, we then select those pairs of residence and office objects where the residence's price was lower than the office's price (together with the time period during which this is true). Last, the valid-time difference of the *domain* of the resulting valid-time binary collection and collection `Residences` is calculated returning those residences with time periods for which no higher priced office can be found. The *temporal domain operation* can be defined similarly to the temporal range operation (definition 5.19). The difference is that the temporal domain returns the objects on the left side in a binary collection.

Definition 5.21 (Domain of a Valid-Time Binary Collection) *Let* B_1 *be a valid-time binary collection of type* $(\texttt{Type}_1, \texttt{Type}_2)$*. The valid-time domain of* B_1*,* $C = \texttt{domain}^v_{set}(B_1)$*, returns a unary valid-time collection* C *of type* $\texttt{Type}_1$*, has a lifespan equal to* $lifespan(C) = lifespan(B)$ *and is defined as*

$$\forall t \in lifespan(C) : \tau_t(ext(C)) = \{x | \exists y :\ll x, y \gg \in^t_{set} B_1\}$$

The valid-time difference, union and intersection operations belong to the first category of operations. Temporal set difference in our model is defined as

Definition 5.22 (Valid-Time Difference of Collections) *Let* C_1 *and* C_2 *be valid-time collections of member types* $\texttt{Type}_1$ *and* $\texttt{Type}_2$ *respectively. The* valid-time difference *of* C_1 *and* C_2*,* $C = C_1 \Leftrightarrow^v_{set} C_2$*, mapping the two collections to a collection* C *of member type* $\texttt{Type}_1$ *and valid-time* $lifespan(C) = lifespan(C_1) \cap lifespan(C_2)$*, is defined as*

$$\forall t \in lifespan(C) : \tau_t(ext(C)) = \{x | x \in^t_{set} C_1 \wedge x \notin^t_{set} C_2\}$$

Valid-time union and intersection can be defined in a similar style to definition 5.22. The type of the result collections of valid-time union and intersection operations are described using the notions of greatest common subtype and least common supertype, as we have introduced them in section 5.2.

Definition 5.23 (Valid-Time Union of Collections) *Let C_1 and C_2 be valid-time collections of member types* $\texttt{Type}_1$ *and* $\texttt{Type}_2$*. The* valid-time union *of C_1 and C_2, $C = C_1 \cup^v_{set} C_2$, mapping the two collections to a collection C of type* $\texttt{Type} = \texttt{Type}_1 \sqcup \texttt{Type}_2$ *and a lifespan equal to $lifespan(C) = lifespan(C_1) \cap lifespan(C_2)$, is defined as*

$$\forall t \in lifespan(C) : \tau_t(ext(C)) = \{x | x \in^t_{set} C_1 \vee x \in^t_{set} C_2\}$$

Definition 5.24 (Valid-Time Intersection of Collections) *Let C_1 and C_2 be valid-time collections of member types* $\texttt{Type}_1$ *and* $\texttt{Type}_2$*. The* valid-time intersection *of C_1 and C_2, $C = C_1 \cap^v_{set} C_2$, mapping the two collections to a collection C of type* $\texttt{Type} = \texttt{Type}_1 \sqcap \texttt{Type}_2$ *and a lifespan equal to $lifespan(C) = lifespan(C_1) \cap lifespan(C_2)$, is defined as*

$$\forall t \in lifespan(C) : \tau_t(ext(C)) = \{x | x \in^t_{set} C_1 \wedge x \in^t_{set} C_2\}$$

The above definitions of valid-time union, intersection and difference ensure that, for example, the valid-time intersect operation can be expressed using the valid-time difference operation:

$$C_1 \cap^v C_2 = C_1 \Leftrightarrow^v (C_1 \Leftrightarrow^v C_2)$$

5.6 A Similar Temporal Object Data Model: TEER

In some aspects, TOM is similar to the TEER model [EW90, EWK93a] which was mentioned already in chapter 3. TEER also uses temporal elements for timestamping entities and introduced notions of temporal relationships and temporal subclass relationships, and it mentions the idea behind temporal disjoint constraints. [EW90, EWK93a] do not discuss all constraints, however. Timestamping meta-data and role modeling is not considered. Additionally, with respect to an algebra, they only support what they call temporal boolean expressions, temporal selection and temporal projection. A temporal boolean expression is a conditional expression on the attributes and relationships of an entity. Instead of returning simply true and false as with non-temporal boolean expressions, a temporal boolean expression returns what they call *true times* which are temporal elements during which the expression returns true. A temporal selection condition compares two temporal elements using set-comparison operators $=$, $\neq$ and $\subseteq$. The temporal elements can be results of a temporal boolean expression. The temporal projection is applied to a temporal entity and restricts the timestamps of the entity's attributes to a specified temporal element.

[EW90, EWK93a] thus *extend* the algebra of their underlying non-temporal data model with the operations described above, they *do not generalise it*. The resulting query language thus is not as powerful as the temporal algebra of TOM. For example, their algebra does not support temporal negation.

5.7 Summary

The temporal object data model TOM not only generalises the data model structures to support temporal data, but also considers all parts of a data model by temporally generalising data structures, constraints and collection algebra.

Rather than extending the data structures with additional properties, the approach of extending the notion of object identifiers is used in TOM, adding timestamps to object identifiers. The underlying model OM is *general* in the sense that entities, collections, associations and even databases are considered as objects. Further, it is *generic* in the sense that it is not based on a specific type system but can be used in a variety of programming language environments and implementation platforms. These advantages carry over to the temporal data model TOM. By generalising the notion of an object identifier to a temporal object identifier, everything considered as an object can be timestamped. In the temporal object data model TOM, for example, collections, constraints, and even types, methods, and so on are objects. Additionally, the possibility that objects may

have several roles at the same time and evolve by changing roles makes both OM and TOM very powerful models.

Experiences in developing the model TOM, together with the prototype implementation TOMS, show that the generalisation approach leads naturally to more general models and systems. The generality and orthogonality of the underlying data model OM are major contributing factors and therefore essential to fully exploit the generalisation approach.

The next chapter describes two different approaches how the temporal object data model TOM can be implemented.

Chapter 6

Implementing the Temporal Object Data Model TOM

This chapter sketches how the temporal object data model TOM presented in the previous chapter can be implemented. Two different possibilities are presented. A first approach implements the data model and a type system from scratch. The second approach is based on specifying an ADT for the commercial DBMS O_2.

With respect to object-oriented DBMS, there is a still unanswered question as to whether the inherent extensibility of these systems is sufficient to build temporal database applications. Thus, to shed some light on this question, the experiences made with building and using the data model TOM as an application in O_2 will also be discussed in this chapter.

6.1 Different Possibilities to implement the Temporal Object Data Model TOM

TOM is generic, which means that its definition does not assume anything particular about the type system. On one hand, this genericity is inherited from its underlying object data model OM. On the other hand, the approach of temporal object identifiers allows the model to stay generic, in contrast to the extension approaches adding time attributes to the data structures.

The TOM model defines temporal data structures, operations and constraints on the collection and object level. It does not specify anything about histories of attribute values, since attributes are defined on the type level. Depending on the type system upon which the temporal object data model TOM is built, different scenarios of attribute history management are possible.

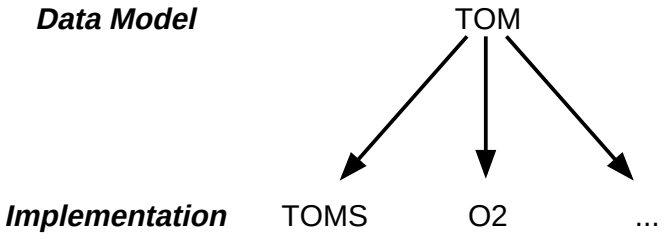

Figure 6.1: Different possibilities to implement the generic temporal object data model TOM

Figure 6.1 shows two different possibilities of implementing the data model TOM. The two approaches actually represent two different levels the temporal data model may be implemented on. The *first possibility* is to implement the data model and a type system from scratch. Implementing a new data model, a type system, a query language, transactions and so on is the work of a *DBMS*

engineer. He implements all these features himself. The prototype DBMS TOMS is an example of such an approach.

The *second possibility* is to use all features of an existing DBMS, its type system, its storage management, its transactions and security and recovery mechanisms and so on. This approach resembles more the work of an *application engineer*. This thesis also describes such an approach, using the object-oriented DBMS O_2 to implement the TOM data model as a layer on top of it. The additional layer actually is an ADT based on the type system of the DBMS O_2.

6.2 The Temporal Object Model System TOMS

This section describes the features and the implementation of the prototype system TOMS which is an implementation of the temporal object data model TOM. It shows that implementing the temporal object data model TOM is straightforward and leads to an efficient system.

6.2.1 Features of TOMS

TOMS (*Temporal Object Model System*) is a one-to-one implementation of the TOM model. The advantage of such an approach is the full support of every feature directly by the system, together with an efficient implementation. There are no restrictions of an underlying system which influence the new system.

TOMS is a single-user system. The idea behind this prototype system was to verify the design of the temporal data model TOM and to show the feasibility to implement it. Thus, the focus was on the implementation of the model's concepts rather than on issues such as user interfaces and storage management aspects. The examples of chapter 5 have shown that unique object names are used to reference objects for convenience. A simple command-oriented interface allows the creation and deletion of objects and the execution of queries and other statements. Query results are displayed as tables. For persistence, files are used which store both data and meta-data in a structured way.

Since the TOM model is generic, a type system has to be specified when implementing a corresponding DBMS. Additionally, the different kinds of objects have to be supported. Objects can be instances of types, or special forms such as collections, associations and constraints. Last but not least, the algebra operations and a query language have to be provided. Examples showing how types, objects, collections and associations are defined and how temporal queries are written in TOMS were given in chapter 5. The following subsections sketch how these parts were implemented in the prototype DBMS TOMS.

6.2.2 A simple Type System

In the TOM model, an object may be dressed and stripped with several types during its existence. Additionally, it may have different types simultaneously. Depending on which role the object is viewed in, different attribute values are displayed. Types are created and dropped in the following way:

```
create type client(name : string);
create type tenant(profession : string) subtype of client;
create type owner(bank_account : string) subtype of client;
create type property(price : integer; street : string; city : string);
...
drop type property;
drop type owner;
drop type tenant;
drop type client;
```

When an object is inserted into a collection, it has to be dressed with the corresponding member type of the collection. Recall the example of a property leasing company used in chapter 5. To insert an object into the collection `Tenants`, the object has to be dressed with the collection's member type `tenant`.

In TOMS, the meta-data describing these type definitions is stored in three different *persistent* data structures. Data structure *subtype* stores the *type hierarchy*, data structure *type* the *type descriptions* and data structure *type_extent* the *extent of the types*. The type hierarchy stores pairs of subtype-supertype relationships. The type description contains the name of the type plus the attribute names with their corresponding types. For subtypes, only the new attributes are stored. Similarly, in the type extent, only the attribute value histories of those attributes defined at this level are stored. This is done in order to avoid data redundancy. The example schema given in picture 5.1 shows a type `client` and its two subtypes `tenant` and `owner`. Type `client` has an attribute `name`. The value of attribute `name` of an object dressed with both type `tenant` and type `owner` is the same, regardless if the object is viewed as a tenant or as an owner. The disadvantage of this horizontal fragmentation is that for each object, its attribute value histories have to be collected by going up the type hierarchy. In the example, an object dressed with type `tenant` has an attribute `profession` and inherits attribute `name` from its supertype. When this object is displayed – for example, when browsing through collection `Tenants` – the attribute values are set together by first taking the attribute values stored in the extent of type `tenant`. Then the system checks whether there exists a supertype, and in the case it does, the attribute values of the supertype are added. This is done until no further supertypes are found.

TOM uses the notion of visibilities. Each object is *visible* in a collection during a certain time period. The result of a query is a collection containing objects with visibilities. The visibility of an object restricts the visibilities of its attribute value histories. TOMS thus uses the visibility of an object in a result collection to restrict its attribute value histories. Assume that the attribute `price` in type `property` stores the history of renting prices a property had during its existence. If the property object is viewed during a time period $[1980 \Leftrightarrow 1985)$, TOMS automatically restricts the price history to this time period as well. When an attribute value is updated, TOMS modifies its value history accordingly.

6.2.3 Implementing Temporal Objects

Chapter 5 has introduced the data model TOM in detail. The model defines different temporal objects – objects which can be dressed with types, collections, associations and constraints. This section describes how these different kinds of objects are implemented in TOMS.

6.2.3.1 Simple Temporal Objects

In its simplest form, an object in TOMS consists of the object identifier (OID) and the lifespan. This was called a temporal object identifier. When a user creates an object, he specifies the lifespan of the object and a name which can be used to refer to the object. An object is created in the following way:

```
create object andreas lifespan { [1964 - inf) };
```

TOMS generates a new OID and stores it together with the name, for example, `andreas`, and the lifespan, for example, $\{[1964 \Leftrightarrow \infty)\}$, in a persistent data structure *object*.

Whenever an object is added to a collection, the system checks whether or not the object is already dressed with the corresponding member type during its membership time period. If not, it is dressed with it and the user is asked for the required attribute values.

6.2.3.2 Temporal Collections

A temporal collection consists of a temporal object identifier and an extent. The extent is a set of temporal objects which are members of the collection. The extent thus contains elements of the

form

$$< \texttt{OID};\ t_{user} >$$

where OID is the object identifier of a collection member and t_{user} is the user-specified membership time. As we have seen, collections are created the following way:

```
create collection Clients type client lifespan { [1980 - inf) };
create collection Tenants
   subcollection of Clients type tenant lifespan{ [1980-inf) };
```

TOMS stores meta-data on collections in three persistent data structures. A first persistent data structure *collection* contains the temporal object identifier and the name and member type of the collection. TOMS assumes that a collection always has the same member type during its existence. However, instead of storing a single member type for a collection, the history of member types could be stored. This way, schema evolution would be supported.

A collection can be a subcollection of another collection. This hierarchy is stored in a second persistent data structure *subcollection*. The collection hierarchy is mapped to pairs of subcollection-supercollection relationships. In a third persistent data structure *collection_extent*, the extent of the collections is stored. It contains the collection name and the object identifiers and membership times of the objects contained in the collection.

6.2.3.3 Temporal Associations

Associations are stored in a similar way to collections. As seen before, an association relates a source and a target collection with each other, and cardinality constraints specify how often an object of the source or target collection may take part in a relationship at a specific time instant. TOMS stores this meta-data in a data structure *association*.

The extent of an association contains binary objects of the form

$$< (\texttt{OID}_1,\ \texttt{OID}_2);\ \texttt{ls} >$$

A binary object ($\texttt{OID}_1$, $\texttt{OID}_2$) consists of two object identifiers $\texttt{OID}_1$ and $\texttt{OID}_2$, denoting which two objects – one of the source and one of the target collection – are related with each other. The lifespan specifies the time period, during which the two objects are related with each other. In TOMS, the extent of an association is stored separately in a persistent data structure *association_extent*.

In both data models OM and TOM, it is possible to have hierarchies of associations in a similar fashion to collections. These sub-association relationships are stored in data structure *subassociations* where pairs of subassociation-superassociation relationships are stored.

6.2.4 Implementing the Temporal Constraints

Chapter 5 introduced the different forms of constraints supported in the OM data model and its temporal generalisation TOM. There are *model inherent constraints* such as (temporal) subcollection relationships, (temporal) association constraints, (temporal) referential integrity, and *user-specified constraints* such as (temporal) partition, cover, intersection and cardinality constraints.

In the following two subsections, these different constraints are briefly discussed, showing how they are implemented in TOMS.

6.2.4.1 Model Inherent Constraints

The *(temporal) subcollection relationship* demands that a subcollection may only exist when the corresponding supercollection exists. Additionally, an object may only be member of a subcollection at those time instants when it is also a member of the supercollection. After creating a new subcollection, TOMS checks whether or not the lifespan of the subcollection is contained in the lifespan of its supercollection. This is done by subtracting the lifespan of the supercollection

from the lifespan of the subcollection. A non-empty result denotes the violation of the constraint. Objects added to a subcollection are tested whether they are also member of the corresponding supercollection during the specified time period. If not, TOMS automatically propagates them.

The *(temporal) referential integrity constraint* checks the validity of references to objects. Such references may appear in several places, for example, a relationship stored in an association actually consists of references to objects, and an association itself references a source and a target collection. In the non-temporal case, it simply needs to be checked whether or not the referenced object exists. In the temporal case, it not only needs to be checked if a referenced object does exist at some point in time, but whether this object exists during the *whole* time period it is referenced. In TOMS, temporal referential integrity is tested using the temporal difference operation for sets of intervals. Assume, for example, an association `A` having a lifespan $\texttt{ls}_A$, relating objects in a source collection $\texttt{C}_S$ with objects in a target collection $\texttt{C}_T$. The temporal subtraction of source collection $\texttt{C}_S$ from the domain of association `A`, $\texttt{domain(A)} -^t \texttt{C}_S$, should return an empty collection having an empty lifespan. Otherwise the association either exists during a time period in which the source collection does not exist, or it references objects in the source collection during time periods they are not visible in the source collection. The same is also done with respect to the target collection.

A model inherent constraint of TOM which does not have a counterpart in the non-temporal data model OM is *visibility*. Recall that visibility is the time period an object is visible in a collection, and that this time period depends on the collection's lifespan, the object's lifespan and the user-specified time of the object's membership in the collection. The user-specified membership time is stored explicitly in the database, and the visibility of an object is calculated on demand, for example, when the object is displayed. At commit time, newly inserted objects are tested to determine whether or not they are visible in the collection. If not, the user is notified and the object removed from the collection.

Note that while TOMS does neither support the timestamping of types nor the changing of member types of collections, it would be possible to consider types to be objects as well. This means that in this case a type also has a lifespan. As mentioned earlier, it would be further possible to store the *member type history of collections*. In this case, the DBMS must check additionally for the *temporal member type constraint* which demands that a collection must always be associated with an existing type.

6.2.4.2 User-Specified Constraints

User-specified constraints such as the partition, cover, intersection and cardinality constraints and their temporal counterparts are dynamic in the sense that the *user* specifies the exact criteria which need to be checked. This is done using the constraint specification language which allows the definition of (temporal) partition, cover, intersection and cardinality constraints. The dynamic aspects of these constraints, for example, the collections involved, are stored as meta-data. At commit time, this meta-data is accessed by the constraint checker and corresponding tests are done. For example, a temporal disjoint constraint for two subcollections $\texttt{C}_1$ and $\texttt{C}_2$ calculates the temporal intersection of them, $\texttt{C}_1 \cap^t \texttt{C}_2$. In the case that a non-empty collection extent results, the constraint is violated. The other constraints are tested accordingly.

Generally it is possible for constraints to be rewritten as queries, as it was the case for TimeDB. TOMS also uses this approach for constraint evaluation. Hence the implementation of the (temporal) algebra operations can be reused for constraint checking.

TOMS does not verify all of these constraints every time a database is committed. It registers the modifications done to the database which might lead to an inconsistent state, for example, modifying the extent of a collection. Then, only the constraints related with the modified collections have to be checked.

6.2.5 Implementing the Temporal Collection Algebra

Most of the algebra operations in TOMS are actually simply manipulating object identifiers and lifespans. This provides for an easy implementation of the (temporal) algebra operations. Types

and attribute values of objects are needed, for example, when selecting objects with respect to some attribute value, when mapping objects to new ones or when displaying objects.

Important for the temporal algebra are three operations defined on sets of intervals. These basic operations on lifespans and visibilities are used in several places. In TOMS, they are the basis for the implementation of the temporal collection algebra. First, a short description of these basic operations is given and then the implementation of the temporal algebra operations is described.

6.2.5.1 Operations on Lifespans

Section 2.1.4 introduced the three set-theoretic operations union, intersection and difference for sets of intervals. In TOMS, these operations are used for calculating time periods, for example, during query evaluation or constraint checking. Since they are used wherever temporal data is handled in the system, we consider them to be essential in building the TOMS DBMS. Each operation has two arguments which are sets of intervals and returns a set of intervals:

```
intersect(S1 : set(interval), S2 : set(interval))  : set(interval)
union(S1 : set(interval), S2 : set(interval))      : set(interval)
difference(S1 : set(interval), S2 : set(interval)) : set(interval)
```

Each operation returns maximal time periods. For example, the union of two sets

```
S1 = { [1980-1990), [1992-1997) }
S2 = { [1975-1982), [1988-1993) }
```

returns a set containing a single interval :

```
union(S1, S2) = { [1975-1997) }
```

As mentioned already in chapter 2, these operations are closed with respect to sets of intervals.

6.2.5.2 Temporal Collection Algebra Operations

This section describes how the temporal collection algebra was implemented in TOMS. As shown in chapter 5, the collection algebra consists of an extensive set of operations. There are operations such as the set operations union, intersection and difference of collections, the selection operation and the cross product of two collections. Additionally, there are operations on *binary collections* such as the compose, inverse, domain and range operations.

Most of these operations do not access attribute values during their calculations. They can be implemented by simply manipulating object identifiers and time periods. Each temporal operation returns the lifespan of the resulting collection, the extent of this result collection plus a description of its member type. When the result objects are displayed, the type description is analysed and the corresponding attribute value histories are displayed.

In the following, the implementation of some of these temporal algebra operations is described using pseudo code. First, the implementation of the temporal set intersection operation is presented. As defined in section 5.2, the *resulting type of an intersection operation* is the *greatest common subtype* (GCS). In TOMS, the valid-time set intersection is implemented the following way:

```
valid_intersect(IN  Lifespan1-Extent1-Type1,
                IN  Lifespan2-Extent2-Type2,
                OUT Lifespan-ResultExtent-GCS(Type1, Type2));
BEGIN
  Lifespan := intersect(Lifespan1, Lifespan2);
  ResultExtent := {};

  FOR EACH <OID; V1> IN Extent1 DO
    IF <OID; V2> IN Extent2 THEN
```

```
      V := intersect(intersect(V1, V2), Lifespan);
      IF not_empty(V) THEN add(<OID; V>, ResultExtent) END IF;
    END IF;
  END FOR;
END valid_intersect;
```

The valid-time intersection has two input and one output parameter. The two input parameters are the two collections which shall be intersected, and the output parameter returns the resulting collection. Each collection is described as a triple `Lifespan-Extent-Type` where `Lifespan` is the lifespan of the collection, `Extent` is the set of objects contained in the collection and `Type` is the member type of the collection.

The *lifespan of the resulting collection* is the intersection of the two lifespans of the collections to be intersected. The *extent of the resulting collection* then is determined by taking an object from the extent of the first argument collection and testing whether this object is also a member of the second collection. If yes, the object's visibility in the first collection (`V1`) is intersected with the visibility of the object in the second collection (`V2`). Additionally, the resulting visibility has to be restricted by the lifespan of the resulting collection. In the case that the resulting visibility `V` is not empty, the OID of the object together with this new membership time period is inserted in the extent of the result collection. The resulting type is the greatest common subtype of the two collection types, `GCS(Type1, Type2)`.

The valid-time cross product of two unary collections is calculated in TOMS in the following way :

```
valid_cross(IN  Lifespan1-Extent1-Type1,
            IN  Lifespan2-Extent2-Type2,
            OUT Lifespan-ResultExtent-(Type1, Type2));
BEGIN
  Lifespan := intersect(Lifespan1, Lifespan2);
  ResultExtent := {};

  FOR EACH <OID1; V1> IN Extent1 DO
    FOR EACH <OID2; V2> IN Extent2 DO
      V := intersect(intersect(V1, V2), Lifespan);
      IF not_empty(V) THEN add(<(OID1, OID2); V>, ResultExtent) END IF;
    END FOR;
  END FOR;
END valid_cross;
```

There are again two input and one output parameter. The input parameters contain the lifespans, extents and types of the two collections. The lifespan of the resulting collection is once more the intersection of the two lifespans of the input collections. The resulting collection contains binary objects in its extent, thus, the resulting type is the binary type `(Type1, Type2)`. The extent is determined by combining each object of the first collection with each object in the second collection. The visibilities of the resulting binary objects are the intersections of the visibility of the object in the first collection (`V1`) with the visibility of the object in the second collection (`V2`), restricted to the lifespan of the resulting collection. Each binary object having a non-empty visibility `V` is then added to the resulting extent.

As seen before, neither the temporal intersection nor the temporal cross product operation accesses any type information or attribute values. The temporal selection operation, however, usually selects objects with respect to attribute values. In TOMS, the valid-time selection operation has been implemented according to the following pseudo code:

```
valid_select(IN  Lifespan-Extent-Type,
             IN  Predicate,
             OUT Lifespan-ResultExtent-Type);
BEGIN
```

```
    ResultExtent := {};

    FOR EACH <OID; V1> IN Extent DO
      V2 := true_time(Predicate, OID, V1, Type);
      IF not_empty(V2) THEN add(<OID; V2>, ResultExtent) END IF;
    END FOR;
  END valid_select;
```

The resulting collection of a valid-time selection has the same lifespan and type as the given collection. A predicate is used to determine which objects in the extent of the argument collection shall be selected. Each object in the extent is tested whether it fulfills the predicate during a non-empty time period. This is done in procedure `true_time`, which accesses the attribute values of the objects when needed. Its result is a time period, which, in the case that it is non-empty, is the new visibility of the object in the resulting collection.

Next, the implementation of the algebra operations on binary collections is sketched. The example operation described is the valid-time domain operation. The following pseudo code describes the valid-time domain operation as implemented in TOMS:

```
valid_domain(IN  Lifespan-Extent-(Type1, Type2),
             OUT Lifespan-ResultExtent-Type1);
BEGIN
  ResultExtent := {};

  FOR EACH <(OID1, OID2); V1> IN Extent DO
    IF <OID1; V2> IN ResultExtent THEN
      V := union(V1, V2);
      remove(<OID1; V2>, ResultExtent);
    ELSE
      V := V1
    END IF;
    add(<OID1; V>, ResultExtent);
  END FOR;
END valid_domain;
```

The input is a binary collection, whereas the output collection is unary. The lifespan of the resulting collection is the same as the lifespan of the argument collection. The member type of the resulting collection is `Type1`. The resulting extent contains all OID which appear on the left hand side of the binary objects contained in the extent of the given collection.

The corresponding *temporal range* operation is implemented the same way. It returns a unary collection of type `Type2` and contains the set of OID on the right hand side of the binary objects in the given extent.

Furthermore, the temporal inverse operation simply changes each binary object

```
<(OID1, OID2); V>
```

in the given extent to

```
<(OID2, OID1); V>
```

and returns a binary collection of type `(Type2, Type1)`.

All other temporal algebra operations can be implemented in a similar way. It is obvious that the implementation of the temporal collection algebra is not very complicated. In most operations, only temporal object identifiers have to be accessed and manipulated. The temporal object identifier carries enough information needed when calculating the results of the temporal algebra operations. No attribute values have to be accessed. This efficient way to implement the temporal algebra is another advantage of the temporal object data model TOM. It is possible due to the separation of typing and classification introduced in the underlying OM model.

The prototype system TOMS was implemented in SICStus Prolog [Swe93]. As mentioned before, the data is stored in files for persistence. The performance of the algebra operations, as described above, can additionally be improved using well-known optimisation techniques, for example, algebraic query optimisation and indexes.

6.3 Implementing the Temporal Object Data Model TOM using O_2

There is an ongoing debate as to whether object-oriented data models should be extended or generalised after all, since their inherent extensibility can be used to support, for example, temporal applications [Sno95a]. This section shows how it is possible to use an ADT for time to support temporal database applications. Specifically, the temporal object data model TOM is mapped to the type system of the object-oriented DBMS O_2.

6.3.1 Using O_2 to manage Temporal Data

The system architecture of the object-oriented DBMS O_2 [O2] is divided into several layers. As depicted in figure 6.2, the base of O_2 is the O_2Engine which provides all the features of a DBMS and all the features of an object-oriented system. Several programming interfaces are built on top of the O_2Engine.

To implement TOM on top of O_2, the O_2C and OQL interfaces are used to support temporal functionality as specified in the TOM model. O_2C is a fourth generation language based on the programming language C. OQL is an SQL-like query language.

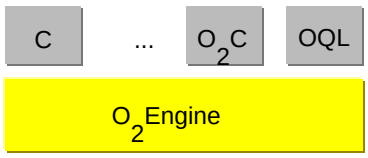

Figure 6.2: Part of the system architecture of O_2

The approach chosen to extend O_2 with time is based on the idea of a root class supporting time attributes and special methods operating on them. These methods, used together with the non-temporal query language OQL, allow temporal queries to be written. Another approach would be to use the C interface of O_2 and supply temporal functionality and maybe even a new *temporal* query language by adding a library written in C. This would effectively produce a new data model. However, as mentioned before, we consider here only the *general application programming level* of such a system and not extensions at lower levels which are more the task of the database engineer.

The following subsections show how we implemented the ADT for time in O_2. The structural part of the root class `TempObject` is described and examples are given showing how it can be used to implement valid-time collections and associations. Finally, the implementation of the special methods operating on timestamps together with a few examples of temporal queries written in OQL are discussed.

6.3.2 The O_2 Data Model

Prior to the description of how the temporal object data model TOM was mapped to O_2, a short overview on the data model of O_2 is given.

The O_2 data model distinguishes between types and classes. A type defines the structure of *values*, whereas a class describes the data structure and the signatures of the methods of *objects*. The identity of a value is the value itself, whereas an object has a unique object identifier.

An object encapsulates both data and behaviour. The state of the object is specified in the type specification of a class. The signatures of the methods of the object are specified in the method part of the class definition. The method body is implemented separately from the class definition.

Objects and values do not automatically remain in the database after the program that created them has ended. O_2 uses the concept of *persistence by reachability*. An object or value is persistent if it is named, or it is attached to a named object or value, or is attached to another persistent object or value. In other words, it is persistent if it is reachable from another persistent object or value. A collection as we know it from the OM model can be implemented in O_2 as a *named* set, a bag or a list of objects.

Example 6.1 *We would like to model a collection of client objects in O_2. A client object is an instance of a class* `client`*. The class* `client` *consists of a data structure which is a tuple containing attributes such as the name and the address of a client and a method which can be used to initialise the data structure with values. Values in attribute* `address` *are of type* `address_type` *which is a tuple containing attributes* `street` *and* `city`*. This type is defined separately from the class definition. A collection which persistently stores instances of class* `client` *is defined as a named set of client objects.*

```
type address_type : tuple(street : string, city : string);

class client public inherit Object
  type                                          /* Specification of the state */
    tuple(name: string, address : address_type)
  method                                        /* Specification of the behaviour */
    public init(...)
end;

name Clients : set(client);                     /*  A named value */
```

Example 6.1 shows the definition of a class of client objects and a persistent collection `Clients`. A named (persistent) value consists of a name and a type specification, for example, `set(client)`. The definition of a named object would consist of a name and a class.

O_2 is compatible with ODMG [Cat93]. The ODMG standard defines a common object data model and object definition and object query language for object-oriented DBMS.

6.3.3 Lifespans in O_2

O_2 supports the specification of types and classes. A class consists of a state and behaviour. The state is specified by a data structure. The behaviour of the class consists of different methods. A *valid-time* object now shall consist of at least a timestamp containing the lifespan and methods which can be used to do timestamp calculations.

A lifespan in TOM consists of a set of non-overlapping time periods. A time period corresponds to a time interval. These time intervals are the basic time units for lifespans, which we define in O_2 as follows:

```
type Interval : tuple(VTS : Date, VTE : Date);
```

In the examples, time intervals which are closed at the lower and open at the upper bound, consist of a starting (`VTS`, Valid Time Start) and an ending date (`VTE`, Valid Time End). A lifespan then can be modeled as a *set of intervals*.

As already mentioned in the previous chapter, TOM supports temporal object role modeling. An important feature which must be supported for role modeling is that an object can be dressed with different types at the same time. The object-oriented DBMS O_2 does not allow an object to have several types at the same time, however. In O_2, an object which is a member of two different sets has the same attribute values in both sets. As with the temporal model TOM, it should be

possible to model the fact that an object's roles vary over time and, further, it may have many roles at the same time. To do this, a way must be found to be able to represent the logical entity and its different temporal roles in O_2.

The problem is to determine which objects denote the same real world entity. A set valued attribute could be added, for example, to each object which contains references to other objects which actually stand for the different roles of the real world entity. Or objects denoting a role could be pointing to a root object, an approach which is similar to what has been proposed for views in O_2 [dS95]. A third approach is to add a key value to objects. This is depicted in figure 6.3. A logical entity `Andreas` is stored as several instances in the database, where each instance represents a specific role of the logical entity. The key value is unique for one real world entity. Objects in the DBMS with the same key value refer to the same real world entity.

The first and second approaches lead to quite a lot of pointer chasing and it is hard to keep the references consistent. For simplicity, the third approach is used in the following.

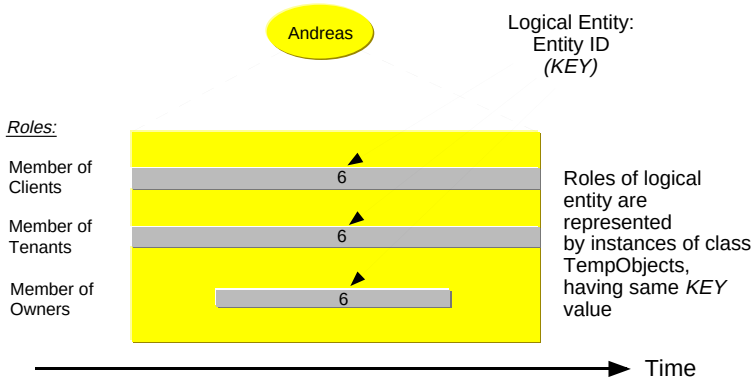

Figure 6.3: Roles represented in O_2

These ideas lead to the following definition of the structural part of a root class `TempObject` for temporal objects:

```
class TempObject inherit Object
  public type
    tuple(VALID : set(Interval),
          KEY   : integer)
end;
```

Attribute `KEY` shall be some form of system-generated entity identifier. Each object to be timestamped now can be derived from class `TempObject`.

6.3.4 Temporal Data Structures

In the previous section, the notion of a temporal object `TempObject` for O_2 has been specified. Now, this class is used to model a temporal database application. Any class whose instances shall be timestamped is derived from class `TempObject`. As an example, the database for a property leasing company is implemented as it was introduced in the previous chapter. The schema of the corresponding database was given in figure 5.1.

In O_2, the timestamps are part of the type specification. Additionally, the user has to manage the attribute value histories himself. These are important differences to the approach described in section 6.2 where timestamps are part of the object identifier and attribute value histories are handled directly by the system. In O_2, we model attribute value histories as *sets of temporal objects*. Each temporal object in such a set contains a specific value the attribute had during a certain time period. In the following example, only attribute `price` in type `property` is assumed to be time-varying.

Example 6.2 *We implement the database schema for the property leasing company as given in figure 5.1. First, we define classes* `client`, `tenant`, `owner` *and* `property` *which correspond to the type specifications for TOMS given in example 5.2. Class* `property` *contains the time-varying attribute* `prices` *which is modeled as a set of temporal objects of class* `price`*. Class* `price` *is derived from* `TempObject`*.*

```
class client inherit TempObject
  public type tuple(name : string)
end;

class tenant inherit client
  public type tuple(profession : string)
end;

class owner inherit client
  public type tuple(bank_account : string)
end;

class price inherit TempObject
  public type tuple(price : integer)
end;

class property inherit TempObject
  public type
    tuple(prices : set(price),
          street : string,
          city   : string)
end;
```

So far, the types of the collections have been defined. The next step is to implement the collections as given in the schema depicted figure 5.1. In O_2, a collection of objects is implemented as a *named value* of type `set`. As mentioned earlier, objects which belong to a named value are persistent.

In TOM, collections are objects and, in the case that they are temporal, have a lifespan. This can be modeled in O_2 by deriving a temporal collection class from class `TempObject`.

```
class client_collection inherit TempObject
  public type
    tuple(extent : set(client))
end;
```

Now the temporal collection `Clients` can be defined to be a named value of type `client_collection`:

```
name Clients : client_collection;
```

In this case, collection `Clients` is timestamped similar to the temporal collections in the TOM model. Members of collection `Clients` are added to attribute `extent`. In order to keep the following examples of queries as simple as possible, non-timestamped collections will be used, though.

Example 6.3 *In O_2, we implement the collections depicted in figure 5.1 as named values which contain sets of temporal objects:*

```
name Clients : set(client);
name Tenants : set(tenant);
name Owners  : set(owner);
```

```
name Properties : set(property);
name Rented     : set(property);
name Available  : set(property);
name Renovating : set(property);
name Residences : set(property);
name Offices    : set(property);
```

With these named values, it is possible to store timestamped objects. The next section shows how associations can be implemented.

6.3.5 Temporal Associations

Temporal associations can be implemented in O_2 similarly to temporal collections. Classes – derived from class `TempObject` – are specified which contain reference attributes to source- and target-objects. Then, associations are defined as named values.

Example 6.4 *In the example schema depicted in figure 5.1, two associations are defined. Association* `Owns` *relates objects in source collection* `Owners` *with objects in target collection* `Properties`*, expressing which client classified as an owner owns which property. The relationship specifying who is renting which property is modeled with an association* `Rents`*, relating objects in source collection* `Tenants` *with objects in target collection* `Rented`*. Each relationship contained in one of these associations then is an instance of a class* `own` *or* `rent`*, respectively.*

```
class own inherit TempObject
  public type
    tuple(source : owner,
          target : property)
end;

class rent inherit TempObject
  public type
    tuple(source : tenant,
          target : property)
end;
```

Now we create named values containing the relationships between owners and properties and tenants and properties:

```
name Owns  : set(own);
name Rents : set(rent);
```

In the TOM model, associations are objects and thus timestamped. Similar to the approach for collections presented in section 6.3.4, timestamped associations could be created.

6.3.6 Temporal Constraints

There are several kinds of constraints which are defined in the TOM data model and supported in the DBMS TOMS. *Model inherent constraints* such as the (temporal) subcollection relationship, the temporal association constraint, (temporal) referential integrity, the temporal member type constraint and visibility have been described, and on the other hand *user-specified constraints* such as (temporal) partition, cover, intersection and cardinality constraints. As mentioned in section 6.2.4, the TOMS DBMS supports these constraints directly and checks them at commit time.

O_2, however, does not support any kind of constraints. Constraints have to be provided by the application programmer, for example, in form of functions and methods. During an update of the database, these methods then are used to check the consistency of the modified database. This means that in O_2, all the constraints defined in TOM have to be implemented by the application programmer.

6.3.7 Operations on Lifespans

So far, the data structures have been described to store time-varying data in O_2. The next step is to come up with functions which refer to the object timestamps and perform temporal calculations on them. As mentioned before, they are implemented as methods of class `TempObject`.

Basically, the methods `T_INTERSECT`, `T_MINUS` and `T_UNION` are supported to write queries equivalent to those that can be expressed using the temporal algebra operations introduced in section 6.2.5.1. Additionally, temporal comparison predicates such as *meets* and *overlaps* as proposed by [All83] are supported. The signatures of these methods which are part of class `TempObject` are the following:

```
class TempObject inherit Object
  public type
    tuple(VALID : set(Interval),
          KEY   : integer)
  method
    public T_INTERSECT(T : set(Interval)) : set(Interval),
    public T_MINUS(T : set(Interval))     : set(Interval),
    public T_UNION(T : set(Interval))     : set(Interval),

    public T_MEETS(I1 : Interval, I2 : Interval)    : boolean,
    public T_OVERLAPS(I1 : Interval, I2 : Interval) : boolean,
    ...
end;
```

Operations `T_INTERSECT`, `T_MINUS` and `T_UNION` have a single argument – a set of intervals. The second *implicit* argument to these operations is attribute `VALID` of the object itself. The comparison predicates are implemented as functions on two *intervals*, returning a Boolean value.

With these methods, it is possible write *temporal queries* in *O_2 OQL*. In the following, the queries discussed in examples 5.8 and 5.9 are expressed in O_2 OQL using the methods specified in class `TempObject`.

Example 6.5 *What is the history of tenants renting one of Herbert's properties? An OQL query calculating the corresponding result looks like*

```
select t
from   t in
   (select tenant(VALID      : r.T_INTERSECT(o.VALID),
                  KEY        : r.source.KEY,
                  name       : r.source.name,
                  profession : r.source.profession)
    from   o in Owns, r in Rents
    where  o.source.name = "Herbert"
      and  o.target = r.target)
where t.VALID != set();
```

This OQL query, for example, returns the two objects shown in figure 6.4.

The OQL query in example 6.5 explicitly calculates the resulting valid-time periods. First, those relationship instances in association `Owns` are selected whose source objects refer to Herbert. This

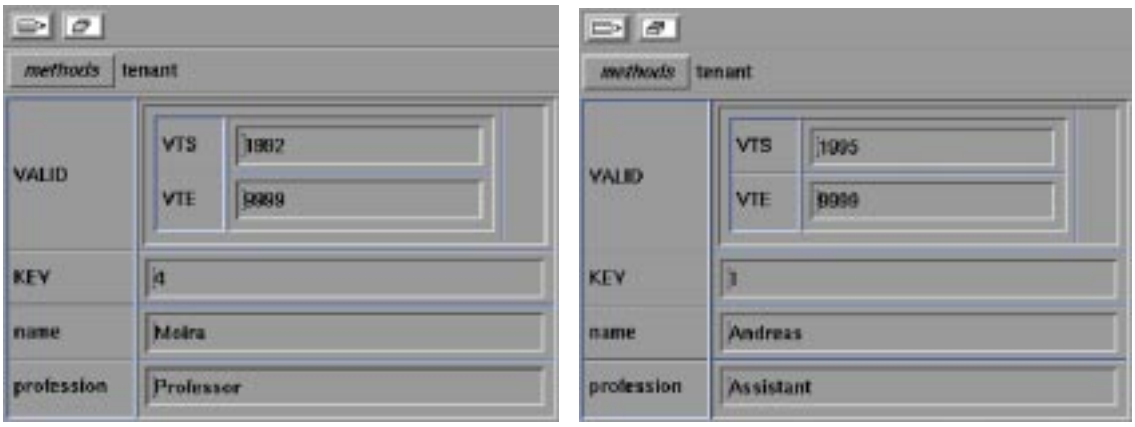

Figure 6.4: Resulting tenant objects of the temporal query given in example 6.5

way the properties owned by Herbert (`o.target`) are found. These properties then are used to find the relationship instances in association `Rents` referring to one of Herbert's properties (`o.target = r.target`). New resulting tenant objects are created and initialised with the attribute values of the source objects selected from association `Rents` (`r.source`). The valid-time periods of the resulting tenant objects are calculated by intersecting the valid-time period of the owner relationship with the valid-time period of the rent relationship (`r.T_INTERSECT(o.VALID)`). In the case that an empty valid-time period results, the tenant is renting the property during a time period it was not owned by Herbert. Thus, those objects from the resulting tenant objects are selected which have a non-empty valid-time period.

The next query uses temporal negation. Additionally, it shows how attribute value histories have to be treated to get correct query results. Resulting objects are shown in figure 6.5.

Example 6.6 *We would like to find those residences and the corresponding time period during which no higher priced offices exist. An OQL query calculating the corresponding result is*

```
select property(VALID  : r.VALID,
                KEY    : r.KEY,
                prices : select p1
                         from   p1 in (select price(VALID : p2.T_INTERSECT(r.VALID),
                                                    KEY   : p2.KEY,
                                                    price : p2.price)
                                       from   p2 in r.prices)
                         where  p1.VALID != set(),
                street : r.street,
                city   : r.city)
from   r in
  (select tuple(VALID  : r.T_MINUS
                         (flatten
                           (select r.T_INTERSECT(p1.T_INTERSECT(p2.VALID))
                            from   o in Offices, p1 in r.prices, p2 in o.prices
                            where  p1.price < p2.price)),
                KEY    : r.KEY,
                prices : r.prices,
                street : r.street,
                city   : r.city)
   from r in Residences)
where r.VALID != set();
```

Figure 6.5: Resulting property objects of the temporal query given in example 6.6

The query in example 6.6 turns out to be rather complicated. In the `from`-clause of the outermost query, those `property` objects from collection `Residences` are gathered for which no higher priced office exists, together with the corresponding time period. This is done the following way: First, a residence object `r` from collection `Residences` is picked. In the `select`-clause of this subquery those price objects of offices are selected which are higher than a price of the residence object `r`. Intersecting their valid-time periods returns the time period during which a higher priced office exists with respect to residence `r`. This is done for all objects `o` in collection `Offices`. The result of this is a set of sets of intervals. This result is flattened into a set of intervals, containing those time periods during which higher priced offices exist with respect to residence `r`. Subtracting these time periods from the valid-time period of `r` results in those time periods during which *no* higher priced office for residence `r` exist.

Thus, the subquery in the `from`-clause returns for each residence object in collection `Residences` a tuple containing the corresponding attribute values plus the resulting valid-time period. Due to the subtraction of valid-time periods, the result of this selection contains residence objects with valid-time periods which are *subsets* of their visibilities in collection `Residences` – these objects are seen only during certain time periods of their actual visibility in the collection. This means that the attribute value histories of these objects also have to be limited to the restricted visibility. This is done in the subquery of the outermost `select`-clause. There, new property objects are created having the same attribute values as the resulting tuples from the outermost `from`-clause. The attribute value history of attribute `price`, however, is temporally restricted to the time period of the resulting tuples of the outermost `from`-clause.

6.4 Summary

In this chapter, two different approaches to implementing the temporal object data TOM have been described. *First*, it was shown how the data model can be implemented directly. Everything is implemented from scratch, and a new query, data definition and modification language has to be supported. This is the work of a DBMS engineer. The single-user DBMS TOMS is such a direct implementation of the TOM data model. It uses files for persistence and supports transactions. *Second*, the approach of using the data model of an existing DBMS directly to implement the temporal object data model TOM has been investigated. It is possible to extend the existing object-oriented DBMS O_2 to store temporal data and calculate complex temporal queries. Thus, it is not necessary to extend a temporal object-oriented query language syntactically to express temporal algebra operations. Temporal algebra operations can be expressed using *non-temporal* algebra operations together with methods. This means the data model of the DBMS is used as-is. This approach resembles the implementation of an application which is based on the DBMS – in

this case the object-oriented DBMS O_2. These two approaches obviously differ in how much the DBMS supports the implementation of an application handling time-varying data and how much is left to the application programmer.

There are several obvious drawbacks when implementing an ADT for time, however. The examples 6.5 and 6.6 show that it is possible but quite difficult to write temporal queries in *O_2 OQL*. The application programmer has to know exactly how the temporal queries have to be formulated using the methods of class `TempObject`. The non-temporal queries he usually writes look totally different from their temporal counterparts.

It is much easier just to specify that all algebraic operations should be evaluated *temporally* by writing a special keyword in front of a *legal non-temporal query*, the approach used for ATSQL2 and also for the query language of TOM. This is not only less error prone and easier for a programmer to understand, but also helps in migrating non-temporal to temporal queries.

Constraints are used to define which states of a database are legal. By adding a time dimension to data, constraint checking is also influenced. Using the DBMS O_2, the user or application programmer has to write methods or functions which check the constraints when attribute values are updated. It would be useful, however, if general constraints were supported by the DBMS directly.

Temporal DBMS have to be able to store and manage huge amounts of data since data is not deleted physically anymore. Besides enhancing the expressive power, temporal operations also increase the complexity of query processing. This means that accessing and querying temporal data need to be optimised in order to provide reasonable answer times. Additionally, efficient constraint checking should be supported. Object-oriented DBMS, extended with time functions and temporal comparison predicates, cannot make use of optimisation techniques for temporal data. They simply do not know about the temporal semantics of the data and functions and how they could be exploited for optimisation. As was shown in the case of temporal relational databases (for example, in [EWK93b, Kol93, LM93, Seg93]), adding temporal data structures and operations to the system allows them to be used in optimisation strategies.

Thus, the drawbacks of the ADT approach are the complexity of temporal queries, the specification of temporal constraints which is left to the user, and the lack of using the special semantics of time for query optimisation.

We argue that the form of extensibility provided by current object-oriented DBMS is not sufficient enough to support temporal databases. What would be good to have additionally are, for example, extensible object identifiers, extensible query languages and the possibility to overwrite algebra operations. These features, however, have to be implemented at lower levels, for example, within the O_2Engine. Concepts like these would help to implement temporal databases with non-temporal object-oriented database management systems, but they would not be restricted to them. Applications using spatial databases or versioning could also be supported.

Chapter 7

Comparing the Different Timestamping Approaches

So far, several approaches for temporal data models, from relational to object-oriented ones, have been discussed. Chapter 3 introduced different proposals found in the temporal database literature. Chapter 4 then described an approach which focused on generalising the query, data modification and constraint specification language of SQL. The data structures, however, were *extended* with special timestamp attributes. Additionally, it was shown how this language was implemented. In chapter 5, a new temporal object data model was proposed which generalises the underlying non-temporal object data model in all aspects – the data structures, the algebra operations and the constraints. The resulting temporal data model is more general than the ones seen before, since it not only supports timestamping attribute values but anything which is an object – for example, collections, constraints, and even types, methods and databases.

In this chapter, these different approaches are compared with each other with respect to their data structures, and it will be shown that the various proposals of temporal data models are actually subsumed in the temporal object data model TOM. First, an evolutionary path of temporal data models from relations to nested relations, complex object and object-oriented data models is given, discussing the different forms of timestamping in a general way. Next, the proposals for temporal data models introduced in this thesis are looked at again with respect to how they timestamp data. Then the reasons why TOM is a more general and orthogonal temporal data model than the other ones are identified. Additionally, it is motivated why timestamping constructs such as collections and constraints with both valid time and transaction time is a desirable feature which should also be considered in temporal data models. It will be obvious then that besides the previously discussed orthogonality of valid time and transaction time in query languages, there is also a notion of *orthogonal application of timestamping data*. Last, proposals of how this can be achieved in data models using the type extension approach for timestamping data will be discussed.

7.1 Temporal Data Model Evolution

In this section, the paradigms of different temporal data models are described according to the evolutionary path of non-temporal data models given in [SS91]. [SS91] shows a unified view of the evolution from relations through nested relations and complex objects to object-oriented data models. This view is used in the following to describe in a uniform and general way the different possible approaches of timestamping data. The evolutionary path then is extended with an additional step from the type based data models to object based data models where objects are allowed having several types simultaneously. It will be shown that besides TOM, all proposed temporal data models actually extend existing non-temporal data models on the *type level* by adding special timestamp attributes.

7.1.1 Data Types and Type Constructors

In the following subsections, a given set T of *atomic* domains or data types is assumed:

$$T = \{\texttt{string}, \texttt{integer}, \texttt{float}, \texttt{real}, \texttt{date}, ...\}.$$

Additional to these basic data types, data models support *data type constructors* such as **set**, **bag**, **list** and **tuple**. The **set** constructor allows the construction of a data type consisting of a set of objects of a specific type without duplicates. Hence, an instance of the data structure **set(***Type***)** contains objects of type *Type*. Type constructor **bag** also constructs a collection of objects of a specific type, but, in contrast to the **set** constructor, allows duplicates. With the **list** constructor, an ordered list of objects can be defined. These data type constructors thus are used to build more complex data structures out of either atomic types or other complex types.

In the following, the different temporal data models will be discussed with respect to valid time. Their extension with transaction time is straightforward.

7.1.2 Temporal First Normal Form Relations

First normal form relations are *sets of tuples*. This means that only the strict sequence of a **set** constructor applied to the **tuple** constructor is allowed. The relational model does not offer the **set** and **tuple** constructors as separate constructors. The domains of attributes must be atomic, and the relational type constructor **set (tuple (..))** can be applied only once per constructed type. While the relational data model [Cod70] specifies a relation to be a *set of tuples*, a relational DBMS usually also supports *bags of tuples*.

As we have seen, timestamping data in first normal form relations is *only possible on tuple level*. A tuple can be extended to store time intervals, for example, a valid-time interval. A time interval is modeled as two attributes, for example, `VTS` and `VTE`, containing the starting and the ending time of a valid-time interval. The query language operations and constraint checker then refer to these special time attributes. Thus, the data model relies on assumptions on the type system.

Chapters 2 and 3 mentioned the problems of *vertical and horizontal temporal anomaly*. Recall that the forced splitting of a logical unit of information into more than one tuple is called the vertical temporal anomaly. The horizontal temporal anomaly addresses the problem that attributes in a tuple change their values asynchronously which, if tuple timestamping is applied, either leads to data redundancy or the relation has to be decomposed. Due to the vertical temporal anomaly, the special operation *coalescing* was introduced to calculate maximal time intervals for split information.

Example 7.1 *A type (or schema)* `Persons` *of a 1NF relation, timestamping tuples with valid-time intervals, may be defined as*

```
type Persons = set ( tuple ( Name       : string,
                             Birthdate  : date,
                             ...,
                             Address    : string,
                             VTS        : date,
                             VTE        : date ))
```

Each attribute is valid during the same time period. Attribute `Birthdate` *contains user-defined time values.*

7.1.3 Temporal Nested Relations

Tuple components in first normal form relations are of an atomic type, for example, of type `integer` or `string`. Nested relations still support only the strict sequence of a **set** (or a **bag**) constructor applied to a **tuple** constructor, as 1NF relations do. However, they allow the use of non-atomic

data types for tuple components. Non-atomic data types can be constructed using again the sequence **set (tuple (..))**, where the tuple components either are atomic or non-atomic. This means that the type constructor sequence **set (tuple (..))** may be applied any number of times in a constructed type.

Nested relations allow, for example, the modeling of attribute histories as time sequences, where each attribute history is a relation containing the attribute together with a timestamp. The set of attribute values together with their timestamps represents the attribute history. This *resolves the vertical temporal anomaly* since it is now possible to store time-varying data about a real world entity in a single tuple. The *horizontal temporal anomaly can also be avoided*, since each attribute can be timestamped independently from other ones. Additionally, relations themselves may be timestamped by nesting a relation into another one having a timestamp expressing when the nested relation itself was valid.

Timestamps may be *time intervals* which are modeled as two attributes, for example, `VTS` and `VTE` for valid time, or it is now possible to store *sets of time intervals* which are stored in a nested relation.

With attribute timestamping, the notion of *homogeneity* becomes important. Recall that homogeneity demands that all tuples of a relation are homogeneous. A tuple is homogeneous, if all attribute value histories in the tuple cover the same time period. Inhomogeneity corresponds to the presence of null values, since time periods are allowed during which an attribute does not have a specific value.

Example 7.2 *The type (or schema)* `Persons` *of a nested relation, using attribute timestamping with valid time, can be defined as*

```
type Persons = set ( tuple ( Name_History    : set ( tuple ( Name : string,
                                                             VTS  : date,
                                                             VTE  : date )),
                             ...,
                             Address_History : set ( tuple ( Street : string,
                                                             City   : string,
                                                             ...,
                                                             VTS    : date,
                                                             VTE    : date ))))
```

Note that it is now also possible to store a more complex form of addresses than in example 7.1. Further, instead of time intervals, the timestamps can also be modeled as sets of time intervals, for example

```
type Persons = set (
                 tuple ( Name_History : set (
                                          tuple ( Name  : string,
                                                  Valid : set (
                                                            tuple ( VTS : date,
                                                                    VTE : date ))),
                         ... ))
```

The move to nested relations with timestamps has major consequences also for a temporal query and modification language and temporal constraints. As pointed out earlier, in temporal data models using type extension, the query and data modification language and the constraints refer to the special time attributes during their evaluation. Since nested relations contain nested timestamps, these parts of the data model have to be adapted accordingly.

Besides attribute and tuple timestamping, there is another approach which does not appear in temporal database literature in this form. It is the combination of tuple and attribute timestamping. For example, the temporal object data model TOM does not assume homogeneity. The notion of visibility is used which – with respect to a nested relation – means that each tuple represents a

real world entity and has a time of existence attached to it. Additionally, each attribute is timestamped allowing the storage of the different attribute value histories of the entity. An attribute then is only visible during the existence of the entity. The following example uses a combination of tuple and attribute timestamping.

Example 7.3 *The type (or schema)* `Persons` *of a nested relation, using a combination of tuple and attribute timestamping with valid-time temporal elements, can be defined as*

```
type Persons = set (
                tuple ( Name_History    : set (
                                          tuple ( Name  : string,
                                                  Valid : set (
                                                            tuple ( VTS : date,
                                                                    VTE : date ))))),
                        ...,
                        Address_History : set (
                                          tuple ( Street : string,
                                                  City   : string,
                                                  ...,
                                                  Valid  : set (
                                                             tuple ( VTS : date,
                                                                     VTE : date ))))),
                        Valid          : set ( tuple ( VTS : date,
                                                       VTE : date ))))
```

The valid-time intervals in attributes `Name_History` *and* `Address_History` *are used to store the history of the names and addresses for a person. The outer valid-time interval can be interpreted as the lifespan of the person. Each attribute history then shall be restricted to the outer valid-time timestamp of the tuple.*

7.1.4 Temporal Complex Objects

Complex objects are built using *any combination* of type constructors. The strict sequence of a **set** constructor applied to a **tuple** constructor is given up. Complex objects thus are more flexible than nested relations with respect to what kind of data structures can be defined.

With respect to time-varying data, it is now possible to also timestamp constant valued attributes. A timestamped constant valued attribute shall be an attribute which contains exactly one value together with a time period, which is in contrast to attributes which store the *history of values* in a set.

Since nested relations can be viewed as a restricted form of complex objects, complex objects also resolve both vertical and horizontal temporal anomaly, and there is again a choice to either use time intervals or sets of time intervals for timestamps.

Example 7.4 *First, we define a type* `Timestamp` *which is a set of intervals. Then, the type* `Persons` *of example 7.3 can be extended to the following complex object timestamped with valid time:*

```
type Interval = tuple ( Start : date, End : date );
type Timestamp = set ( Interval );

type Person = tuple ( Name_History    : set ( tuple ( Name  : string,
                                                      Valid : Timestamp ),
                      ...,
                      Address_History : set ( tuple ( Street : string,
                                                      City   : string,
                                                      ...,
```

```
                                                Valid  : Timestamp ),
                        Phone          : tuple ( Area_Code : integer,
                                                 Number    : integer,
                                                 Valid     : Timestamp ),
                        Valid          : Timestamp )

    type Persons = set ( Person );
```

Besides the timestamps we have already seen in example 7.3, attribute `Phone` *which contains a single phone number may also be timestamped. In the case that the phone number has a valid time shorter than the one of the whole tuple, null values are assumed for the remaining time period. Additionally, the type definition is split into two parts: the definition of types* `Person` *and* `Persons`*. An object of type* `Persons` *contains a set of objects of type* `Person`*.*

TOM supports the timestamping of collections (which are, for example, *sets* of objects). This is also possible in a complex object data model. In this case, however, the type of the set-valued object has to be changed.

Example 7.5 *According to the lifespans of collections supported in TOM, the set* `Persons` *may also be timestamped. Using the type extension approach, we have to turn the set into a tuple :*

```
type Persons = tuple ( extent : set ( Person ),
                       Valid  : Timestamp );
```

7.1.5 Temporal Object-Oriented Data Models

Nested relations and complex objects cannot directly represent non-hierarchical or recursive relationships [SS91]. The solution is to break up the recursion in type definitions by introducing *names* for type instances and *functions* in addition to attributes. Naming instances means to introduce unique object identifiers. Functions or object identifiers are used, for example, to reference the extracted recursive part. These are essential features of object-oriented data models. Thus, the problem of data containment is resolved by the use of references.

In an object-oriented data model, data can be timestamped on the same levels as complex objects. Thus, temporal object-oriented data model also resolve both vertical and horizontal temporal anomaly. Additionally, names (or object identifiers) and functions (or methods) may be timestamped.

Example 7.6 *The following type* `Department` *which is recursively defined on attribute* `Members`

```
type DeptRelation = set ( Department );

type Department = tuple ( DNo     : integer,
                          Name    : string,
                          Members : set ( tuple ( Name   : string,
                                                  ...
                                                  Salary : integer
                                                  Dept   : tuple ( DNo     : integer,
                                                                   Name    : string,
                                                                   Members : ...,
                                                                   Valid   : Timestamp ),
                                                  Valid  : Timestamp )),
                          Valid   : Timestamp );
```

can be broken up into

```
type DeptRelation = set ( Department );

type Department  = tuple ( DNo     : integer,
                           Name    : string,
                           Members : Staff,
                           Valid   : Timestamp);


type Staff = set ( Employee );

type Employee = tuple ( Name   : string,
                        ...
                        Salary : integer,
                        Dept   : Department,
                        Valid  : Timestamp );
```

where `Members` *and* `Dept` *contain references to* named *objects of type* `Staff` *or* `Department`*, given the department or the employee, respectively. Note that for ease of reading, only tuple timestamping is applied. Attribute timestamping could be done as shown in the previous examples.*

An object-oriented data model also must support the concept of *specialisation*. Specialisation allows the reuse of a type definition and the creation of a new *subtype* which is more special than its supertype. The subtype inherits all components of its supertype, maybe overwriting some of the supertype's functions, and it may contain additional components.

Example 7.7 *The type definition of* `Employee` *in example 7.6*

```
type Employee = tuple ( Name   : string,
                        ...
                        Salary : integer,
                        Dept   : Department ,
                        Valid  : Timestamp )
```

was meant to contain all the attributes defined for a tuple in type `Persons` *(as given in example 7.1), extended by attributes* `Salary` *and* `Dept`*. Instead of repeating attributes common to both types, we define them the following way:*

```
type Person = tuple ( Name      : string,
                      ...,
                      Valid     : Timestamp );

type Employee subtype Person = tuple ( Salary : integer,
                                       Dept   : Department )
```

The rigid application of type constructors is gradually relaxed in the evolution from relational to object-oriented data models. Usually, object-oriented data models and systems restrict objects to be an instance of only a single type at a specific time point. To be able to support role modeling, a data model should allow objects to be dressed with several types simultaneously, however. We consider giving up this restriction and using a different form of timestamping as a next step in our evolutionary path, since it adds additional flexibility to the data model.

7.1.6 A Temporal Collection Model

Most object-oriented data models assume an object to have a single type. For role modeling, however, a data model must allow objects to be dressed with several types simultaneously. This

is possible in the non-temporal collection model OM and its temporal generalisation TOM. In the temporal collection model, each role of an entity then has its own history.

In general, an object-oriented data model supports the definition of object types. A class then is the extent of a type, containing all objects of the same type. Each object is member of exactly one class which it is assigned to at creation time.

The collection model OM and its temporal generalisation TOM distinguish between collections and classes. A collection is a semantic grouping of objects. An object may be a member of several collections. The members of each collection have to be dressed with the corresponding member type of the collection. Additionally, the object is a member of the extents of the types with which the object is dressed. This distinction thus separates between the *representation* and the *semantic grouping* of objects.

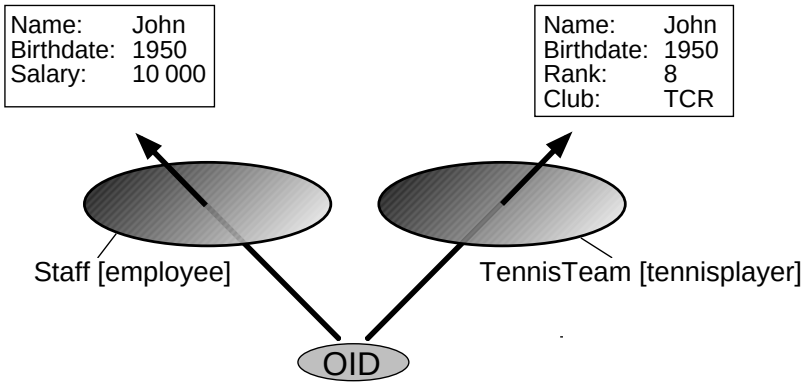

Figure 7.1: An object viewed through different roles

In OM, an object is viewed through a collection, thereby displaying the corresponding attribute values as specified by the member type of the collection. The object is referred to by its unique object identifier. In other words, we take an object identifier and look at it through different roles, and, in the case that the object takes part in a role, it shows the corresponding attribute values. This is depicted in figure 7.1, where an object is viewed either as a member of a staff or as a member of a tennis team.

In temporal data models using the type extension approach, the temporal algebra operations and the constraint checker refer to special timestamp attributes. These operations thus are directly depending on specific types and the presence of specific attributes in the objects in discourse. The approach used in the temporal object data model TOM does not extended the data structures but focuses on the object identifier. The natural extension of an object identifier to support time is the *temporal object identifier*. As shown in figure 7.2, the temporal object identifier is separated from the state and behaviour of an object. It references a specific object with its overall lifespan which then can be viewed in different roles. Thus, TOM does not timestamp the values of an object and then has to calculate the union of all timestamps of values of this object in order to find out during which time the object existed, but it timestamps the object itself with its overall time of existence and keeps track of the history of its values separately.

The temporal object identifier naturally extends the timestamping of data to other constructs of the model. As we have seen, it automatically supports the timestamping of collections, constraints or even types and databases. With temporal object identifiers, it is still possible to timestamp the same data structures as in the models discussed in the previous sections.

The approach of timestamping the existence of an object rather than the validity of its property values is more natural. The distinction between lifespans of objects and time periods during which property values of these objects are valid actually represents the way we deal with time in the real world. When we think of persons and their lifespans, we usually relate the date of birth and the date of death to the lifespan of a person and not the union of all valid time periods of his different

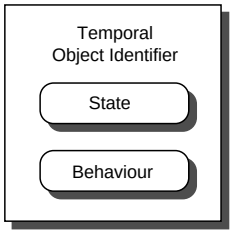

Figure 7.2: Components of a Temporal Object

addresses during his life.

In TOM, objects are created separately from type specifications. An object may be dressed with different types simultaneously. In the following example, an explicit `dress` operation is used which may be applied several times with different types to the same object.

Example 7.8 *After creating the necessary types and collections, we can create an object John, dress it with two different types, for example, type* `employee` *and type* `tennisplayer`*, and initialise it with corresponding attribute values. Object John then is inserted into both collections* `Staff` *and* `TennisTeam`*.*

```
create type person( Name      : string,
                    Birthdate : date );

create type employee( Salary : integer,
                      Dept   : department ) subtype of person;

create type tennisplayer( Rank : integer,
                          Club : string ) subtype of person;

create collection Staff type employee lifespan { [1980 - inf) };
create collection TennisTeam type tennisplayer lifespan { [1991 - 1995) };

create object John lifespan { [1950 - inf) };
dress object John with type employee;
...
dress object John with type tennisplayer;
...
insert object John to Staff during { [1987 - inf) };
insert object John to TennisTeam during { [1992 - 1995) };
```

It is possible to support the timestamping of sets, bags, lists and other complex objects also in complex object and object-oriented data models. However, there are several advantages to the approach of using temporal object identifiers. First, the orthogonality with which different constructs in the data model such as collections, constraints and so on are timestamped follows automatically. Second, the algebra operations simply rely on the temporal object identifier and not on the data structure of the state of the object. As it was stated for temporal nested relations, the possibilities of timestamping data in data models supporting complex types also have major consequences in terms of changes to the query language and constraint evaluation, for example. Further, if a set is turned into a temporal object in TOM, it is timestamped without turning it first into a tuple with additional timestamp attributes. Third, necessary constraints are automatically supported and tested. For example, a set of temporal objects should contain only objects during the set's own lifespan. In TOM, is called the visibility. An extension approach which allows the timestamping of sets, bags and lists should also test such constraints.

Paradigm	Model	Timestamping
1NF	[JMS79]	Tuple
	[Cli82, CW83]	Tuple
	[Sno84, Sno87, Sno93]	Tuple
	[Ari86]	Tuple
	[NA88, NA89, NA93]	Tuple
	[Sar90b, Sar93]	Tuple
	[SBJS96b, SBJS96a]	Tuple
NFNF	[CT85, Tan86]	Attribute
	[GV85, Gad86, Gad88, GN93]	Attribute
	[TG89, Tan93]	Attribute
	[CT85, CC87, CC93]	Tuple/Attribute
Complex Object, OO	[KRS90]	Tuple
	[Wuu91]	Tuple
	[SC91]	Tuple
	[KS92b]	Tuple
	[EW90, EWK93a]	Attribute
	[RS91, RS93]	Attribute
	[GÖ93]	Attribute
	[BFG96]	Attribute
	[DW92, WD93]	Tuple/Attribute
OO	[SN97a, SN97b, SN97c]	Object Identifier

Table 7.1: List of the different temporal data models

7.2 Different Forms of Timestamping Data

The temporal database literature only considers two forms of timestamping data, namely attribute and tuple timestamping. Object timestamping, as it usually appears in the literature, corresponds to tuple timestamping. This is more in line with complex object rather than object-oriented data models.

In this section, the different approaches to timestamping data are compared and the models presented in chapters 3, 4 and 5 are discussed with respect to this issue.

Table 7.1 lists the temporal data models discussed in this thesis with respect to the paradigm they are based on and what kind of timestamping is used. There are two models which support *both tuple and attribute timestamping*. As we have seen in chapter 3, [CT85, CC87, CC93] timestamp attributes within the relation schemas, expressing when they were part of the schema. The tuples then are timestamped with a time period specifying when the attribute values were valid. [DW92, WD93] support either tuple or attribute timestamping, however independently from each other. Both of the approaches do not discuss a possible combination of both tuple and attribute timestamping as it was used in example 7.3, however.

Except TOM [SN97a, SN97b, SN97c], all other proposals extend the underlying type system by either applying attribute or tuple timestamping. All first normal form (1NF) relational data models use tuple timestamping, which is no surprise. On the other hand, the non first normal form (NFNF) relational data models use mostly attribute timestamping. Complex object and object-oriented (OO) temporal data models use either attribute or tuple timestamping.

The following table summarises *all possible kinds of timestamping data* for the different kinds of data models, as they were discussed in section 7.1:

Model	Type Constructors	Timestampable Units
Relational Model	relation	Tuples
Nested Relations	relation	Tuples, Attributes, Relations
Complex Object Model, Object-Oriented Data Model	set, bag, list, tuple	Tuples, Attributes, Sets, Bags, Lists, Functions, Object Identifiers

The relational data model supports a single data structure – the relation. As we have seen, the only construct to timestamp in this model is the tuple. Nested relations are more flexible, they allow timestamping not only tuples, but also attributes and relations. Attribute histories are actually relations, containing the timestamp and the attribute. A set or a bag of tuples – a relation – may be timestamped by nesting it into another relation and adding a timestamp attribute. Complex object and object-oriented data models support timestamping of any data structure composed of the type constructors and base types. To do this, however, a data structure has to be embedded into a tuple.

For a *general* temporal object-oriented data model, we propose that it should allow that any part of a type definition may be timestamped, that not only data but also meta-data is treated as time-varying data, and further that any operation of the model may refer to this time information and use it for calculations. This is an *orthogonal approach to timestamping data*. None of the proposed temporal data models in chapters 3 and 4 can be considered orthogonal in this sense. The temporal data model TOM, however, supports all kinds of timestamping listed. Additionally, due to the temporal object identifier, it also *automatically* allows the timestamping of meta-data and thus supports timestamped collections, types and so on, and it features a temporally complete query language which treats valid time and transaction time orthogonally.

The generality and orthogonality of TOM is based on two things. First, the underlying non-temporal data model OM and its implementation OMS [NW97] specify concepts such as collections, types, methods and so on to be objects. In general, the focus is on the objects as an entity and not on their properties. Second, the approach to using temporal object identifiers still supports this distinction. The object – represented by the temporal object identifier – is concentrated on and not its properties. In contrast to other object-oriented temporal data models, the concept of identity is still the main one when the step from non-temporal to temporal objects is made. This concept does not apply for relational data models, where tuples are distinguished only by the values they contain. Due to these aspects, the temporal object data model TOM has turned out to be more general and orthogonal than other proposals, and we can say that TOM is a general and orthogonal form of the other temporal data model proposals.

7.3 Timestamping Meta-Data

So far, the different possibilities of timestamping data have been discussed, and it was shown what has been done in different proposals of temporal data models. We have seen that the temporal object data model TOM uses an orthogonal approach to timestamp data. For example, collections and constraints are also objects and thus may be timestamped with valid time and transaction time. The introduction of a new general and orthogonal form of timestamping has given us the insight that meta-data in general could also be timestamped with both time lines. This aspect has received little attention in temporal database literature so far. Work with respect to timestamping meta-data has focused on schema evolution, mainly with respect to transaction time [DT87, Rod92a, Rod92b, CGS95, RS95].

In the following subsections, we discuss why timestamping meta-data with both valid time and transaction time should also be supported in temporal data models and motivate the timestamping

of constructs such as relations, views, constraints and types. A main reason why we argue that both time lines should be supported is that if only transaction time is used to timestamp meta-data, updates can be effective only from the time point they are executed. Updating earlier or future database states is not possible. Additionally, the timestamping of relations, collections and views with valid time allow the specification of *when* the concepts existed or were relevant in the application domain. It is sometimes necessary, however, to update meta-data retroactively or proactively. This is only possible with respect to valid time.

7.3.1 Timestamping Relations and Collections

Research in the temporal database area concentrated mainly on timestamping tuples or attributes. Timestamping relations has received little to no attention. Usually, only a tuple or its attribute values are timestamped. The relation of which it is member, however, does not have a timestamp attached. This can be interpreted in several ways. Usually, it is assumed that the relation is *always* valid (or stored, respectively).

Relations, however, are created and dropped just as tuples are inserted and deleted. Dropping a relation thus can be considered to be handled in a similar way to deleting a tuple. For example, if the user drops a relation, this means he does not want it to appear in the future database states anymore. However, if he goes back to an earlier database state, it should still be visible. In the case that the relation was deleted physically, this is not possible anymore. Thus, the relation itself needs to be timestamped with transaction time. When creating a relation, the transaction time of the relation is set to the time point the create statement is executed. In the case that the relation is dropped later, the upper bound of the transaction-time interval is set to the time instant the corresponding drop statement is executed. This way, the time period during which the relation was stored in the database is kept track of, thereby still allowing access to the data stored in it. Tuples in a dropped relation then may not be updated anymore.

What can be modeled when relations are timestamped with valid time? As we have seen in the temporal object data model TOM, timestamped collections model the history and validity of *semantic groupings of entities*. This is not possible in proposed temporal relational data models. For example, the lifespan of collection `Departments` determines when the company was organised in departments. Maybe at some point in time, the company is totally restructured and the departments are substituted by institutes. So, the valid-time interval of collection `Departments` expresses when the company was organised in departments.

The discussion about timestamping relations of tuples or collections of objects with valid time and transaction time shows that two different forms of deleting them have to be distinguished. One form of deleting a collection is with respect to dropping it from the database, the other one expresses when it stopped being valid in the real world. The time instant when a created collection is committed is the lower bound of the transaction time interval of this collection, whereas the time instant of the commit of a drop statement corresponds to the upper bound. The lifespan of the collection is the valid time of the collection, which can be modified.

As we have seen in chapter 5, timestamping collections (and thus relations) also influences the query language. It has to be taken into account that collections or relations may not exist during the same time periods.

7.3.2 Timestamping Views

Views can be considered as *virtual collections or relations*, in contrast to base collections or base tables whose objects are actually stored in a database. Views derive their data from other collections or tables according to a specified query.

Views are created and dropped just as base collections or relations are. With respect to valid time and transaction time, this means that views should be treated the same way base collections (or relations, respectively) are treated.

7.3.3 Timestamping Constraints

In chapter 5, the idea of timestamping constraints with valid time has already been discussed. Business rules change over time. To be able to model the evolution of these business rules, to allow the modification of them with respect to past and future database states, and to support different rules for the same data during different time periods, it should be possible to timestamp constraints with valid time.

Since constraints are created and deleted, they should be timestamped with transaction time as well. Constraints may be erroneous just as data might be. Since constraints may be used to model business rules, it should be possible to find out how and when constraints were corrected.

7.3.4 Timestamping Types

As mentioned previously, most work on timestamping meta-data was done with respect to schema evolution, for example, in [DT87, Rod92a, Rod92b, RS95, CGS95]. [RS95] discuss the integration of schema evolution into TSQL2. They propose a schema `R` to contain the union of *all* attributes which have been defined during the lifespan of `R`. A view function then maps `R` to a subset of the attributes in a schema S_t which is active at `t`. They argue that supporting valid time for schemas is not necessary, since *schema evolution defines how reality is modeled by the database.* TSQL2 thus only supports timestamping schemas with transaction time.

[CGS95], on the other hand, argue that storing valid time for schemas is necessary for applications requiring retroactive and proactive schema changes, for example, when new encoding rules for social security numbers are stated today, but they are effective from January, 1, 1995, or in two months. We support their view. Obviously, there are cases which make it convenient to timestamp types with valid time. Thus, a temporal DBMS should be general and also support valid time for types.

7.4 Proposed Changes for Extension Approaches

In the previous section, it was motivated why it is useful to timestamp constructs such as collections (or relations, respectively), views, constraints and types with *both* valid and transaction time. This section discusses how this can be achieved in temporal data models which are based on the schema extension approach.

Meta-data is usually stored, queried and updated using the data structures and language supported by the DBMS. The mini-database storing the meta-data is called the *system catalog* [EN94, HS95]. For example, in a relational DBMS, the system catalog is stored as relations and allows DBMS routines and users to access them using SQL.

Thus, we propose that the system catalog shall store *bitemporal meta-data.* Additionally, constraints must be supplied which check that objects may only be inserted into a relation (or a collection) during the valid time of the relation. A relation may only be modified when its transaction time overlaps time instant *now*, otherwise it can be accessed just for querying. With respect to types, the system has to make sure that a relation or a collection always is dressed with a type. The query language has to be adapted if relations have a lifespan, as was shown, for example, in chapter 5.

Temporal DBMS should, however, not only support temporal objects but also provide for non-temporal ones. Since we propose to store all meta-data in bitemporal tables, we have to find a way to deal with storing, for example, the meta-data of a non-temporal relation in a temporal relation of the system catalog. In chapter 5, the handling of non-temporal objects as members in a temporal collection has been discussed. The idea is to assume a non-temporal object to be valid and stored in a database only at time instant *now*, where *now* moves along the time lines as time passes. Thus, a non-temporal relation created in a bitemporal relational DBMS would be stored in the system catalog as having a valid- and transaction-time interval $[now \Leftrightarrow now]$.

Example 7.9 *A simple example for a meta-data relation storing the 1NF relation schemas is given below. It stores the relation name and the names of its attributes together with their types. We timestamp the meta-data with both a valid-time interval* $[VTS \Leftrightarrow VTE)$ *and transaction-time interval* $[TTS \Leftrightarrow TTE)$*.*

Relation_Name	Attribute_Name	Attribute_Type	VTS	VTE	TTS	TTE
Employee	Name	string	1990	∞	1994	∞
Employee	SSN	integer	1990	1993	1994	1995
Employee	SSN	string	1993	∞	1995	∞
Employee	Salary	integer	1990	∞	1994	∞
Employee	DepNo	integer	1990	∞	1994	∞
Department	Name	string	1988	∞	1994	∞
...	...	...	...	...	...	...

Relation `Employee` *was created in 1994, and data about employees is stored since 1990. In 1995, the schema of relation* `Employee` *was modified. The type of the social security number stored in attribute* `SSN` *was changed* retroactively *from* `integer` *to* `string`*.*

7.5 Summary

This chapter has discussed in a general way the different possibilities of timestamping data with respect to the relational data model, nested relations, complex object and object-oriented data models, independent of any proposed temporal data model. The difference of the approach used in TOM to the extension approaches was then shown. Next, the general timestamping approaches to the temporal data models introduced in this thesis were compared. It was mentioned that – with the exception of the approach used in TOM introducing temporal object identifiers – attribute and tuple timestamping are the only two timestamping approaches appearing in temporal database literature. This chapter also showed that TOM actually subsumes the different proposals. Additionally, the proposal of temporal object identifiers leads to an *orthogonal approach of timestamping data*. One consequence of this is that TOM supports timestamping of collections, constraints and so on. It then has been motivated why it makes sense to have meta-data timestamped. Last, it was argued that timestamping meta-data should also be considered for temporal data models using the extension approach, and a short description of how this can be achieved was given.

The main idea discussed in this chapter is that not only an orthogonal treatment of valid time and transaction time in query languages is desired, but also an orthogonal approach to timestamping data (including meta-data) using both valid time and transaction time. This idea has received little attention so far in temporal database literature, and it is not explicitly supported in any temporal data model using type extension in this generality. Supporting temporal relations is not the only requirement to store temporal meta-data. Additionally, constraints have to be supplied, which check, for example, that no tuple is inserted into a relation during a time period in which the relation does not exist. In the case that relations are timestamped, the query language has to take into account the lifespans of the relations in some way. The way this is dealt with in TOM was described in chapter 5. Last, storing meta-data in bitemporal relations has to provide for properly managing non-temporal relations, since they should still be supported in a temporal DBMS. This means that non-temporal entities have to be stored in a temporal system catalog.

Chapter 8

Conclusions

In this last chapter, the most important results of this thesis are summarised and the topics worth investigating further are discussed.

Although a strong need exists for DBMS which are capable of storing, modifying and querying time-varying data, there are no commercial systems available which support users and application programmers in these tasks. Although there are quite some research results available with respect to this topic, their impact to commercial DBMS is negligible. Why is it like that, and in what direction will DBMS go in the future? The main question is whether temporal support in a DBMS is seen as supporting a special class of applications and thus would mean that additional extensions or a special form of DBMS should be implemented for such support, or if it is accepted that the class of temporal applications actually is the main class and thus DBMS should be enhanced in general to support time-varying data. Other issues might be the complexity of the proposed models and approaches or their lack of generality. Further, maybe a standard is needed to motivate companies to enhance their DBMS to support time-varying data.

This thesis is a contribution towards general and orthogonal concepts for supporting time-varying data in DBMS. Further, while – with respect to commercial temporal DBMS – major changes cannot be expected during the next months and maybe even years, other ways have to be found to support applications dealing with temporal data. Thus, another contribution of this thesis is the investigation and discussion of different approaches to achieving such support using existing database technology. Two extensions were discussed – the layered approach in TimeDB and the specification of an ADT – which can be seen as solutions for the time being.

8.1 Summary

Different levels of support for time-varying data in DBMS can be identified. The question is what kind of support users and application programmers would like to have to manage time-varying data. Do they want to implement most of the temporal aspects in their applications on their own, thereby having to accept that the DBMS may not make use of the special semantics time has, for example, for optimisation? Do they want additional functionality supported by a non-temporal DBMS, such as a time data type plus the essential functions for time calculations? Or do they want a specialist DBMS which supports features such as a temporal query and modification language and temporal constraints?

With respect to this question, four possibilities to implement temporal database applications were identified in chapter 1. These are

- Use a type date and implement all temporal semantics in the application program
- Specify an ADT for time which is the basis in temporal applications to timestamp and query the temporal data

- Extend a non-temporal data model to support time-varying data
- Generalise a non-temporal data model into a temporal one

The advantage of the first two approaches is that they do not need any changes to be made to the data model and system used. While these approaches in some way support the data structures and functionality needed to manage time-varying data, they cannot, however, exploit the advantage of the special semantics time has, for example, for optimisation.

The last two approaches can only be achieved by modifying both the data model and corresponding systems. With respect to these approaches, we argue that if changes are done to both the data model and corresponding systems, they should be general and orthogonal, supporting temporal data structures, temporal operations and temporal constraints without any unnatural restrictions. In other words, we support the generalisation approach since it considers all constructs and concepts of a data model.

In this thesis, the second, third and fourth approach listed above have been investigated. With respect to the second approach, it was shown in chapter 7 how an ADT can be used to implement temporal database applications. The ADT was implemented using the object-oriented DBMS O_2. Additionally, this ADT was used together with the non-temporal query language OQL to specify complex temporal queries. With respect to the third approach, chapter 4 described the implementation of the temporally complete language ATSQL2 which is based on extending relation schemas to store time-varying data. The resulting bitemporal DBMS TimeDB supports a temporal query language, a temporal data definition and modification and a temporal constraint specification language. Since the source code of a relational DBMS is not available, TimeDB was implemented as a front-end to the commercial DBMS Oracle. With respect to the fourth approach, a temporal data model based on the generalisation approach has been described in chapter 5. The non-temporal object data model OM was generalised into its temporal counterpart TOM. Both OM and TOM support object role modeling which requires that objects may be dressed with several types simultaneously. Additionally, the prototype system TOMS was implemented to verify the ideas and the design of the temporal object data model TOM.

When extending or generalising a non-temporal data model, the most important decision to make is what kind of timestamping shall be used. Is tuple or attribute timestamping or a combination of it used, or is there another approach possible? The approach of timestamping used in a temporal data model is essential, since it has major consequences on both the operational part and the constraints supported in a data model and system. The problem of the extension approach which uses special time attributes to timestamp data is that both the temporal operations and temporal constraints refer to these attributes. If the timestamps are nested, for example, as it is possible in temporal nested relations, temporal complex object or temporal object-oriented data models, either both the operations and constraints have to be flexible enough to be able to access these nested timestamps, or timestamping has to be restricted to a single level only.

In this thesis, a new approach to timestamping data – the temporal object identifiers – has been introduced which separates the timestamp from the data structure. With this approach, a simple way has been found to specify where timestamps may appear, thereby not restricting the level of nesting. In the temporal data model TOM, both temporal algebra operations and temporal constraints refer to the object identifiers.

Chapter 7 has shown that TOM actually can be seen as a general and orthogonal form of the temporal data models using the type extension approach. By using temporal object identifiers, not only data but also meta-data may be timestamped. The advantages of timestamping meta-data were discussed and it was argued that all temporal data models should extend their timestamping approach to meta-data with both valid and transaction time.

In chapter 1, the goal of designing a *general*, *generic* and *orthogonal* temporal data model was stated. The model should be generic in the sense that it is independent of any specific type system. General means that data structures, operations and constraints of the non-temporal data model OM are generalised into *temporal* data structures, *temporal* operations and *temporal* constraints, using the notion of *snapshot reducibility* to define their semantics. Further, the model should be

orthogonal in two senses. First, anything which is an object (entity, collection, constraint, type or even database) may be timestamped. Second, valid time and transaction time should be treated as orthogonal time lines, having the *same set of operations* defined on them. The temporal object data model TOM fulfills all of these requirements. It is defined independent of any type system and thus is generic, and it generalises the data structures of the underlying data model OM, its algebra operations and constraints into temporal ones. Additionally, as it was discussed in chapter 7, the use of temporal object identifiers leads to orthogonal timestamping. Further, the algebra operations defined in chapter 5 for valid time can be used with the same semantics for transaction time.

8.2 Future Work

There are several aspects of this thesis which we believe are worth further exploration. First, temporal data models using the extension approach could be *generalised*. For example, the TimeDB system could be enhanced with temporal meta-data relations as described in section 7.4 and the relation schemas, constraints and relations could be timestamped with both valid time and transaction time.

Second, the ideas developed in the prototype system TOMS could be incorporated into the object-oriented DBMS OMS. Changes to the notion of an object identifier, to the algebra and the constraints would be necessary. Additionally, it could be extended with transaction time to support bitemporal objects and a bitemporal query, modification and data definition language.

With respect to the incorporation of the ideas of TOM into the object-oriented DBMS OMS, an important issue for further investigations concerns time and methods. In OMS, methods are objects and thus may be timestamped as well. A general question is whether or not methods should also be timestamped and what that would mean. Methods simply returning values, for example, calculating the age of a person from his birthdate, can be seen as dynamically calculated attribute values. This means that such methods could be treated like attribute values. However, timestamping methods with valid time and transaction time introduces problems if the methods have side effects. A method calling a program, for example, a WWW browser referring to a specific homepage, should only be allowed to be used if its valid time and/or transaction time overlaps time instant *now*, because this expresses that the method is both valid and stored in the database at the time instant it is called. Otherwise, the method (meaning the access to the specific homepage) is either not valid anymore or it was deleted and thus should not be used.

Third, this thesis focuses only on managing time-varying data. The data models introduced and proposed are specialist models since they are restricted to temporal data. However, instead of supporting specialist DBMS handling, for example, temporal or spatial database applications, an approach could be chosen which allows a system to be modified or extended such that special semantics of the application domain can be used for efficient data processing.

A future direction to go into thus would be to extract the essential concepts and ideas of temporal data models and support them in a system such that they can also be used for other application domains. Object-oriented data models and systems are extensible by allowing subtyping and overwriting methods and this way support the reusability of both data structures and code. With respect to this form of extensibility, an idea to investigate further would be to support the concepts used in TOM – the temporal object identifier, the temporal algebra operations and temporal constraints – in a more general way, as depicted in figure 8.1. For example, if the DBMS supported the possibility to extend the notion of an object identifier and to overwrite algebra operations and constraint checking algorithms, application domains such as spatial data and versioning problems could be supported besides temporal data within the same DBMS. Another advantage of such an approach would be that overriding these operations with different semantics would in most cases still allow the use of the storage management, the query processing with the implemented query optimisation algorithm, crash recovery, concurrency control and transaction processing supported in the core system.

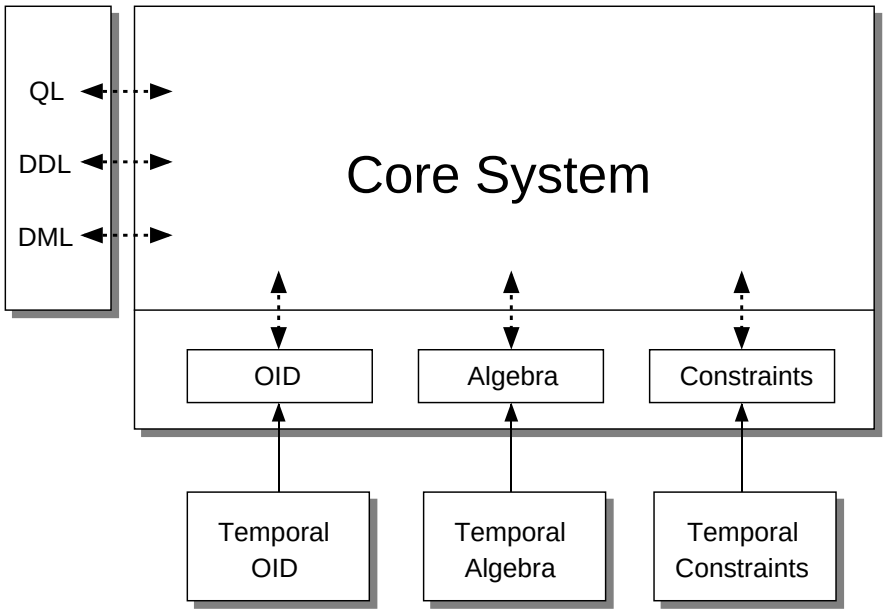

Figure 8.1: An object-oriented DBMS allowing the overwriting of the object identifier, algebra and constraints, e. g. with subclasses which support temporal object identifiers, temporal algebra operations and temporal constraints

Appendix A

Glossary

attribute timestamping The time interval or temporal element denoting the validity or storage time is assigned to attributes.

bitemporal database A bitemporal database is a combination of a historical and a rollback database.

chronon A chronon is a non-decomposable time interval of some fixed, minimal duration, e. g. a second.

class The term *class*, used in object-oriented languages and data models, appears with different meanings. A class can be the type definition of objects, a collection of objects or the set of all instances of a type, also called the extent of a type. In the data models OM and TOM, the term *class* denotes the extent of a type.

coalescing Coalescing is the operation which calculates maximal time intervals for value-equivalent tuples.

collection A collection is a *semantic* grouping of objects of the same type.

derived table In SQL, a derived table is a query expression in the `FROM`-clause used instead of a table reference.

event An event is an instantaneous fact, i. e. something occurring at an instant in time. An event is said to occur at a chronon `t`, if it occurs at any time instant during `t`.

event table An event table contains data timestamped with time instants.

extension approach Using the extension approach to define a temporal data model means that parts of the model, for example, some of the data structures or the algebra operations, are extended with special attributes or special operations, respectively. A different approach is the generalisation approach.

generalisation approach Using the generalisation approach to define a temporal data model means that *all* concepts and constructs of the underlying non-temporal data model are turned into temporal equivalents. A different approach is the extension approach.

historical database A historical database records the history of data with respect to the real world.

homogeneity A relation is called homogeneous if all of its tuples are homogeneous. A tuple is homogeneous if all timestamps of its attribute values cover the same time period.

horizontal temporal anomaly In a temporal relational data model using tuple timestamping, the horizontal temporal anomaly forces the splitting of the horizontal format of a relation due to the asynchronous change of some of its attribute values.

inhomogeneity A relation or tuple is inhomogeneous if it is not homogeneous.

lifespan In the temporal object data model TOM, the lifespan covers the time period an object exists in the real world.

meta-data Meta-data is *data describing data*, i. e. data about the structure of data.

non-sequenced semantics A query has non-sequenced semantics if it is evaluated not interpreting timestamp attributes, using the non-temporal algebra operations.

object identifier Object identifiers are unique system-generated identifiers for objects.

OM OM (Object Model) is a (non-temporal) object data model supporting role modeling. The model contains an algebra based on collections of objects and constraints [Nor92, Nor93].

rollback database A rollback database records all changes done to the database itself and can be viewed as an append-only database.

sequenced semantics A query has sequenced semantics if it is evaluated interpreting timestamp attributes, using temporal algebra operations with snapshot reducible semantics.

snapshot database A database recording only a single state of the real world is a snapshot database.

snapshot reducibility Snapshot reducibility is a reduction proof defining temporal semantics, for example, of algebra operations, by reducing them on single time instants and applying the corresponding non-temporal operations on each time instant. Thus, the temporal semantics are equivalent to the application of non-temporal semantics for each time instant.

temporal completeness In the relational model, a language is temporally complete if

1. it is temporally semi-complete,
2. it is possible to override snapshot reducibility,
3. it is possible to substitute a valid-time relation with a valid-time query,
4. Allen's set of comparison predicates can be used, and
5. it is possible to retrieve and constrain maximal valid-time periods and valid times as specified by the user.

temporal element A temporal element is a finite union of time intervals.

temporal object identifier A temporal object identifier is composed of an object identifier and a lifespan denoting, for example, when the object existed in the real world.

temporal semi-completeness In the relational model, a language is temporally semi-complete if for every non-temporal relation, there exists a temporally equivalent valid-time relation, and if for every non-temporal query a snapshot reducible, syntactically similar valid-time query exists.

temporal upward compatibility With respect to SQL, a language is temporally upward compatible if any legal SQL statement can still be executed on a *temporal* database yielding the same result as if executed on a corresponding non-temporal *snapshot* database.

TimeDB A temporally complete bitemporal DBMS implemented as a front-end to a non-temporal relational DBMS.

time instant A time instant is a time point on an underlying time axis.

time interval A time interval denotes a period of time on an underlying time axis and consists of a starting time instant and an ending time instant.

Time Normal Form A relation is in Time Normal Form if in each tuple all time-varying attributes change their values simultaneously. This can be achieved by decomposing a relation.

TOM TOM (Temporal Object Model) is a temporal object data model based on generalising the data structures, the algebra and the constraints of the underlying data model OM into temporal ones.

TOMS TOMS (Temporal Object Model System) is a temporal object management system based on the data model TOM.

transaction time The transaction time of a fact denotes the time interval during which the fact is stored in a database.

transaction-time database Synonym for rollback database.

tuple timestamping The time interval or temporal element denoting the validity or storage time is assigned to tuples.

upward compatibility With respect to SQL, a language is upward compatible if it contains SQL as a subset.

user-defined time User-defined time is an uninterpreted attribute domain of type time.

valid time The valid time of a fact denotes the time interval during which the fact is true with respect to the real world.

valid-time database Synonym for historical database.

value-equivalent tuple Value-equivalent tuples are tuples which have identical non-timestamp attribute values.

vertical temporal anomaly In a temporal relational data model, the vertical temporal anomaly forces the splitting of a logical unit of data into more than one tuple.

Bibliography

[All83] J. F. Allen. Maintaining Knowledge about Temporal Intervals. *Communications of the ACM*, 1983, 16(11), pages 832–843.

[Ari86] G. Ariav. A Temporally Oriented Data Model. *ACM Transactions on Database Systems*, 1986, 11(4), pages 499–527.

[BFG96] E. Bertino, E. Ferrari, and G. Guerrini. A Formal Temporal Object-Oriented Data Model. In P. Apers, M. Bouzeghoub, and G. Gardarin, editors, *Advances in Database Technology*. Springer, 1996, pages 342–356.

[BJS95] M. Böhlen, C. Jensen, and R. Snodgrass. Evaluating the completeness of TSQL2. In J. Clifford and A. Tuzhilin, editors, *Recent Advances in Temporal Databases*, 1995, pages 153–172.

[BM88] D. Beech and B. Mahbod. Generalized Version Control in an Object-Oriented Database. In *Proceedings of the International Conference on Data Engineering*, 1988, pages 14–22.

[BM92] M. Böhlen and R. Marti. A Temporal Extension of the Deductive Database System ProQuel. Technical report, Departement Informatik, ETH Zürich, 1992.

[BM94] M. Böhlen and R. Marti. On the Completeness of Temporal Database Query Languages. In *Proceedings of the First International Conference on Temporal Logic*, July 1994, pages 283–300.

[Böh94] M. Böhlen. *Managing Temporal Knowledge in Deductive Databases.* PhD thesis, Departement Informatik, ETH Zürich, 1994.

[Böh95] M. Böhlen. Temporal Database System Implementations. *SIGMOD RECORD*, 1995, 24(4), pages 53–60.

[Bur92] J. Burse. ProQuel: Using Prolog to Implement a Deductive Database System. Technical report, Departement Informatik, ETH Zürich, 1992.

[Cat93] R. G. G. Cattell. *The Object Database Standard: ODMG-93.* Morgan Kaufmann Publishers, San Mateo, California, 1993.

[Cat94] R. G. G. Cattell. *Object Data Management.* Addison Wesley, 1994.

[CC87] J. Clifford and A. Croker. The Historical Relational Data Model (HRDM) and Algebra Based on Lifespans. In *Proceedings of the International Conference on Data Engineering.* IEEE Computer Society Press, 1987, pages 528–537.

[CC93] J. Clifford and A. Croker. The Historical Relational Data Model (HRDM) Revisited. In A. Tansel, J. Clifford, S. Gadia, S. Jajodia, A. Segev, and R. Snodgrass, editors, *Temporal Databases: Theory, Design, and Implementation*, Benjamin/Cummings Publishing Company, 1993, pages 6–27.

[CG92] T. S. Cheng and S. K. Gadia. A Seamless Object-Oriented Model for Spatio-Temporal Databases. Technical Report 92-41, Computer Science Department, Iowa State University, 1992.

[CGS95] C. De Castro, F. Grandi, and M. R. Scalas. On Schema Versioning in Temporal Databases. In J. Clifford and A. Tuzhilin, editors, *Recent Advances in Temporal Databases*, 1995, pages 272–291.

[Che76] P. P. Chen. The Entity-Relationship Model: Toward a Unified View of Data. *ACM Transactions on Database Systems*, 1976, 1(1), pages 9–36.

[Cho94] J. Chomicki. Temporal Query Languages: A Survey. In D. M. Gabbay and H. J. Ohlbach, editors, *Proceedings of the First International Conference on Temporal Logic*, 1994, pages 506–534.

[Cli82] J. Clifford. A Model for Historical Databases. In *Proceedings of Workshop on Logical Bases for Data Bases*, 1982.

[Cod70] E. F. Codd. A Relational Model of Data for Large Shared Data Banks. *Communications of the ACM*, 1970, 13(6), pages 377–387.

[CS88] M. Caruso and E. Sciore. Meta-Functions and Contexts in an Object-Oriented Database Language. In *Procedings of the ACM-SIGMOD International Conference on Management of Data*, 1988, pages 56–65.

[CT85] J. Clifford and A. U. Tansel. On an Algebra for Historical Relational Databases: Two Views. In S. Navathe, editor, *SIGMOD RECORD*, 1985, pages 247–265.

[CW83] J. Clifford and D. S. Warren. Formal Semantics for Time in Databases. *ACM Transactions on Database Systems*, 1983, 8(2), pages 214–254.

[Dat89] C. J. Date. *A guide to THE SQL STANDARD*. Addison-Wesley Publishing Company, 1989.

[Day89] U. Dayal. Queries and Views in an Object-Oriented Data Model. In *Proceedings of the Second Workshop on Database Programming Languages*, 1989, pages 80–102.

[DD93] C. J. Date and H. Darwen. *A guide to THE SQL STANDARD*. Addison-Wesley Publishing Company, 1993.

[DLHC95] C. Davies, B. Lazell, M. Hughes, and L. Cooper. Time is just another Attribute – or at least, just another Dimension. In J. Clifford and A. Tuzhilin, editors, *Recent Advances in Temporal Databases*, 1995, pages 153–172.

[dS95] C. S. dos Santos. Design and Implementation of Object-Oriented Views. In *Proceedings of the Database and Expert Systems Applications (DEXA) Conference*, 1995, pages 91–102.

[DT87] P. Dadam and J. Teuhola. Managing Schema Versions in a Time-Versioned N1NF Relational Database. In *BTW*, 1987, pages 161–179.

[DW92] U. Dayal and G. Wuu. A Uniform Approach to Processing Temporal Data. In *Proceedings of the International Conference on Very Large Databases (VLDB)*, 1992, pages 407–418.

[EdOP93] N. Edelweiss, M. de Oliveira, and B. Pernici. An Object-Oriented Temporal Model. In *Proceedings of the Conference on Advanced Information Systems Engineering (CAiSE)*, 1993, pages 397–415.

[EEAK90] R. Elmasri, I. El-Assal, and V. Kouramajian. Semantics of Temporal Data in an Extended ER Model. In *Proceedings of the Conference on the Entity-Relationship Approach*, 1990, pages 249–264.

[EN94] R. Elmasri and S. Navathe. *Fundamentals of Database Systems*. Benjamin/Cummings Publishing Company, 1994.

[EW90] R. Elmasri and G. T. J. Wuu. A Temporal Model and Query Language for ER Databases. In *International Conference on Data Engineering*, 1990, pages 76–83.

[EWK93a] R. Elmasri, G.T.J. Wuu, and V. Kouramajian. A Temporal Model and Query Language for EER Databases. In A. Tansel, J. Clifford, S. Gadia, S. Jajodia, A. Segev, and R. Snodgrass, editors, *Temporal Databases: Theory, Design, and Implementation*, Benjamin/Cummings Publishing Company, 1993, chapter 9, pages 212–229.

[EWK93b] R. Elmasri, G.T.J. Wuu, and V. Kouramajian. The Time Index and the Monotonic B^+-tree. In A. Tansel, J. Clifford, S. Gadia, S. Jajodia, A. Segev, and R. Snodgrass, editors, *Temporal Databases: Theory, Design, and Implementation*, Benjamin/Cummings Publishing Company, 1993, chapter 18, pages 433–456.

[Gad86] S. K. Gadia. Toward a Multihomogeneous Model for a Temporal Database. In *Proceedings of the International Conference on Data Engineering*, 1986, pages 390–397.

[Gad88] S. K. Gadia. A Homogeneous Relational Model and Query Languages for Temporal Databases. *ACM Transactions on Database Systems*, 1988, 13(4), pages 418–448.

[GBB94] G. Guerrini, E. Bertino, and R. Bal. A Formal Definition of the Chimera Object-Oriented Data Model. Technical Report IDEA.DE.2P.011.01, ESPRIT Project 6333, 1994.

[GN93] S. K. Gadia and S. S. Nair. Temporal Databases: A Prelude to Parametric Data. In A. Tansel, J. Clifford, S. Gadia, S. Jajodia, A. Segev, and R. Snodgrass, editors, *Temporal Databases: Theory, Design, and Implementation*, Benjamin/Cummings Publishing Company, 1993, chapter 2, pages 28–66.

[GÖ93] I. A. Goralwalla and M. T. Özsu. Temporal Extensions to a Uniform Behavioral Object Model. In *Proceedings of the 10th International Conference on the ER Approach*, 1993, pages 110–121.

[Gro96] R. Gross. *Implementation of Constraint Database Systems Using a Compile-Time Rewrite Approach*. PhD thesis, ETH Zürich, 1996.

[GV85] S. K. Gadia and J. H. Vaishnav. A Query Language for a Homogeneous Temporal Database. In *Proceedings of the International Conference on Principles of Database Systems*, 1985, pages 51–56.

[GY88] S. K. Gadia and C. Yeung. A Generalized Model for a Relational Temporal Database. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, 1988, pages 251–259.

[GY91] S. K. Gadia and C. Yeung. Inadequacy of Interval Timestamps in Temporal Databases. *Information Systems*, 1991, 54, pages 1–22.

[Haw88] S. W. Hawking. *A Brief History of Time*. Bantam Books, New York, 1988.

[HS95] A. Heuer and G. Saake. *Datenbanken: Konzepte und Sprachen*. International Thomson Publishing, 1995.

[HSW75] G. D. Held, M. Stonebraker, and E. Wong. INGRES – A Relational Database Management System. In *Proceedings of the AFIPS National Computer Conference*, 1975, pages 409–416.

[Jea93] C. Jensen and et. al. A Consensus Glossary of Temporal Database Concepts. Technical Report R 93-2035, Aalborg University, November 1993.

[JMR91] C. S. Jensen, L. Mark, and N. Roussopoulos. Incremental Implementation Model for Relational Databases with Transaction Time. *IEEE Transactions on Knowledge and Data Engineering*, 1991, 3(4), pages 461–473.

[JMS79] S. Jones, P. Mason, and R. Stamper. LEGOL 2.0: A Relational Specification Language for Complex Rules. *Information Systems*, 1979, 4(4), pages 293–305.

[KGBW90] W. Kim, J. F. Garza, N. Ballou, and D. Woelk. Architecture of the ORION Next-Generation Database System. *IEEE Transactions on Knowledge and Data Engineering*, 1990, 2(1), pages 109–124.

[Kol93] C.P. Kolovson. Indexing Techniques for Historical Databases. In A. Tansel, J. Clifford, S. Gadia, S. Jajodia, A. Segev, and R. Snodgrass, editors, *Temporal Databases: Theory, Design, and Implementation*, Benjamin/Cummings Publishing Company, 1993, chapter 17, pages 418–432.

[KRS90] W. Käfer, N. Ritter, and H. Schöning. Support for Temporal Data by Complex Objects. In *Proceedings of the International Conference on Very Large Databases (VLDB)*, 1990, pages 24–35.

[KS92a] W. Käfer and H. Schöning. Mapping a Version Model to a Complex-Object Data Model. In *Proceedings of the International Conference on Data Enginering*, 1992, pages 348–357.

[KS92b] W. Käfer and H. Schöning. Realizing a Temporal Complex-Object Data Model. In *SIGMOD Conference 1992*, 1992, pages 266–275.

[LJ88] N. Lorentzos and R. G. Johnson. TRA: A Model for a Temporal Relational Algebra. In *Proceedings of the Conference on Temporal Aspects in Information Systems*, 1988, pages 99–112.

[LM93] T.Y.C. Leung and R.R. Muntz. Stream Processing: Temporal Query Processing and Optimization. In A. Tansel, J. Clifford, S. Gadia, S. Jajodia, A. Segev, and R. Snodgrass, editors, *Temporal Databases: Theory, Design, and Implementation*, Benjamin/Cummings Publishing Company, 1993, chapter 14, pages 329–355.

[Lor93] N. Lorentzos. The Interval-extended Relational Model and its Application to Valid-time Databases. In A. Tansel, J. Clifford, S. Gadia, S. Jajodia, A. Segev, and R. Snodgrass, editors, *Temporal Databases: Theory, Design, and Implementation*, Benjamin/Cummings Publishing Company, 1993, chapter 3, pages 67–91.

[Mit88] B. Mitschang. Towards a Unified View of Design Data and Knowledge Representation. In *Proceedings of the 2^{nd} International Conference on Expert Database Systems*, 1988, pages 33–50.

[Mit89] B. Mitschang. Extending the Relational Algebra to Capture Complex Objects. In *Proceedings of the International Conference on Very Large Databases (VLDB)*, 1989, pages 297–305.

[MS91] L. E. McKenzie and R. T. Snodgrass. Evaluation of Relational Algebras Incorporating the Time Dimension in Databases. *ACM Computing Surveys*, 1991, 23(4), pages 501–543.

[MS93] J. Melton and A. R. Simon. *Understanding the new SQL: A Complete Guide*. Morgan Kaufmann Publishers, 1993.

[NA88] S. B. Navathe and R. Ahmed. TSQL : A Language Interface for History Databases. In M. Leonard C. Rolland, F. Bodart, editor, *Proceedings of the Conference on Temporal Aspects in Information Systems*, 1988, pages 113–128.

[NA89] S. B. Navathe and R. Ahmed. A Temporal Relational Model and Query Language. *Information Sciences*, 1989, 49(2), pages 147–175.

[NA93] S. Navathe and R. Ahmed. Temporal Extensions to the Relational Model and SQL. In A. Tansel, J. Clifford, S. Gadia, S. Jajodia, A. Segev, and R. Snodgrass, editors, *Temporal Databases: Theory, Design, and Implementation*, Benjamin/Cummings Publishing Company, 1993, pages 92–109.

[Nor92] M. C. Norrie. *A Collection Model for Data Management in Object-Oriented Systems*. PhD thesis, University of Glasgow, Department of Computing Science, Glasgow G12 8QQ, Scotland, December 1992.

[Nor93] M. C. Norrie. An Extended Entity-Relationship Approach to Data Management in Object-Oriented Systems. In *Proceedings of the 12th International Conference on the ER Approach*, 1993, pages 390–401.

[NSWW96] M. C. Norrie, A. Steiner, A. Würgler, and M. Wunderli. A Model for Classification Structures with Evolution Control. In *Proceedings of the 15th International Conference on Conceptual Modelling*, 1996, pages 456–471.

[NW97] M. C. Norrie and A. Würgler. OM Framework for Object-Oriented Data Management. *Journal of the Swiss Informaticians Society*, August 1997.

[O2] O2 Technology. *The O_2 System, Release 4.6; Manuals.*

[PÖS92] R. J. Peters, M. T. Öszu, and D. Szafron. TIGUKAT: An Object Model for Query and View Support in Object Database Systems. Technical Report 92-14, University of Alberta, 1992.

[Pul95] D. Pulfer. Optimierung von temporalen Queries. Master's thesis, Institute for Information Systems, ETH Zürich, February 1995.

[Roc75] M. J. Rochkind. The Source Code Control System. *IEEE Transactions on Software Engineering*, 1975, 1(4), pages 364–370.

[Rod92a] J. F. Roddick. Schema Evolution in Database Systems – An Annotated Bibliography. *ACM SIGMOD Record*, 1992, 21(4), pages 35–40.

[Rod92b] J. F. Roddick. SQL/SE – A Query Language Extension for Databases supporting Schema Evolution. *ACM SIGMOD Record*, 1992, 21(3), pages 10–16.

[RS91] E. Rose and A. Segev. TOODM - A Temporal Object-Oriented Data Model with Temporal Constraints. In *Proceedings of the 10th International Conference on the ER Approach*, 1991, pages 205–230.

[RS93] E. Rose and A. Segev. TOOA - A Temporal Object-Oriented Algebra. In O. Nierstrasz, editor, *Proceedings ECOOP '93*, 1993, pages 297–325.

[RS95] J. F. Roddick and R. Snodgrass. Schema Versioning Support. In R. Snodgrass, editor, *The TSQL2 Temporal Query Language*, Kluwer Academic Publishers, 1995, chapter 22, pages 427–449.

[SA85] R. Snodgrass and I. Ahn. A Taxonomy of Time in Databases. In S. Navathe, editor, *Proceedings of the ACM SIGMOD International Conference on Management of Data*, 1985, pages 236–246.

[SA86] R. Snodgrass and I. Ahn. Temporal Databases. *IEEE Computer*, 1986, 19(9), pages 35–42.

[Sar90a] N. Sarda. Algebra and Query Language for a Historical Data Model. *The Computer Journal*, 1990, 33(1), pages 11–18.

[Sar90b] N. Sarda. Extensions to SQL for Historical Databases. *IEEE Transactions on Knowledge and Data Engineering*, 1990, 2(2), pages 220–230.

[Sar93] N. Sarda. HSQL: A Historical Query Language. In A. Tansel, J. Clifford, S. Gadia, S. Jajodia, A. Segev, and R. Snodgrass, editors, *Temporal Databases: Theory, Design, and Implementation*, Benjamin/Cummings Publishing Company, 1993, pages 110–138.

[SBJS96a] R. T. Snodgrass, M. H. Böhlen, C. S. Jensen, and A. Steiner. Adding Transaction Time to SQL/Temporal. *SQL/Temporal Change Proposal, ANSI X3H2-96-502r2, ISO/IEC JTC1/SC21/WG3 DBL MAD-147r2*, November 1996.

[SBJS96b] R. T. Snodgrass, M. H. Böhlen, C. S. Jensen, and A. Steiner. Adding Valid Time to SQL/Temporal. *SQL/Temporal Change Proposal, ANSI X3H2-96-501r2, ISO/IEC JTC1/SC21/WG3 DBL MAD-146r2*, November 1996.

[SC91] S. Y. W. Su and H. M. Chen. A Temporal Knowledge Representation Model OSAM*/T and its Query Language OQL/T. In *Proceedings of the International Conference on Very Large Databases (VLDB)*, 1991, pages 431–442.

[Sci91] E. Sciore. Using Annotations to Support Multiple Kinds of Versioning in an Object-Oriented Database System. *ACM Transactions on Database Systems*, 1991, 16(3), pages 417–438.

[Sci94] E. Sciore. Versioning and Configuration Management in an Object-Oriented Data Model. *Proceedings of the International Conference on Very Large Databases (VLDB)*, 1994, 3(1), pages 77–106.

[SD94] T. Smith and J. Drukman. *Programmer's Guide to the Oracle Call Interface*. Oracle Corporation, Oracle Corporation, Belmont, California, USA, 1994.

[SDJ+93] R. Snodgrass, C.E. Dyreson, C.S. Jensen, N. Kline, M.D. Soo, L. So, and J. Whelan. *The* MultiCal *System, Release 1.0*, October 1993.

[Seg93] A. Segev. Join Processing and Optimization in Temporal Relational Databases. In A. Tansel, J. Clifford, S. Gadia, S. Jajodia, A. Segev, and R. Snodgrass, editors, *Temporal Databases: Theory, Design, and Implementation*, Benjamin/Cummings Publishing Company, 1993, chapter 15, pages 356–387.

[Shi81] D. W. Shipman. The Functional Data Model and Data Language DAPLEX. *ACM Transactions on Database System*, 1981, 6(1), pages 140–173.

[SK86] A. Segev and K. Kawagoe. Temporal Data Management. In *Proceedings of the International Conference on Very Large Databases (VLDB)*, 1986, pages 79–88.

[SL76] D. G. Severance and G. M. Lohman. Differential Files : Their Application to the Maintenance of Large Databases. *ACM Transactions on Database Systems*, 1976, 1(3), pages 256–267.

[SLK89] S. Y. W. Su, H. Lam, and V. Krishnamurthy. An Object-Oriented Semantic Association Model (OSAM*). In S. T. Kumara, A. L. Soyster, and R. L. Kashyap, editors, *Artificial Intelligence: Manufacturing Theory and Practice*, Industrial Engineering and Management Press, Norcross, GA, 1989, chapter 17.

[SN97a] A. Steiner and M. C. Norrie. A Temporal Extension to a Generic Object Data Model. Technical Report 265, Institute for Information Systems, ETH Zürich, May 1997.

[SN97b] A. Steiner and M. C. Norrie. Implementing Temporal Databases in Object-Oriented Systems. In R. W. Topor and K. Tanaka, editors, *Database Systems for Advanced Applications (DASFAA)*, 1997, pages 381–390.

[SN97c] A. Steiner and M. C. Norrie. Temporal Object Role Modelling. In Olive A. and Pastor J. A., editors, *Proceedings of the Conference on Advanced Information Systems Engineering (CAiSE)*, 1997, pages 245–258.

[Sno84] R. Snodgrass. The Temporal Query Language TQuel. In *Proceedings of the International Conference on Principles of Database Systems*, 1984, pages 204–212.

[Sno87] R. Snodgrass. The Temporal Query Language TQuel. *ACM Transactions on Database Systems*, 1987, 12(2), pages 247–298.

[Sno93] R. Snodgrass. An Overview of TQuel. In A. Tansel, J. Clifford, S. Gadia, S. Jajodia, A. Segev, and R. Snodgrass, editors, *Temporal Databases: Theory, Design, and Implementation*, Benjamin/Cummings Publishing Company, 1993, pages 141–182.

[Sno95a] R. Snodgrass. Temporal Object-Oriented Databases: A Critical Comparison. In W. Kim, editor, *Modern Database Systems*, ACM Press, 1995, chapter 19, pages 386–408.

[Sno95b] R. Snodgrass, editor. *The TSQL2 Temporal Query Language*. Kluwer Academic Publishers, 101 Philip Drive, Assinippi Park, Norwell, Massachusetts 02061, USA, 1995.

[SQL93] American National Standards Institute. *ANSI X3H2-93-091/YOK-003, ISO-ANSI (Working Draft) Database Language SQL3*, February 1993.

[SRH90] M. R. Stonebraker, L. Rowe, and M. Hirohama. The Implementation of POSTGRES. *IEEE Transactions on Knowledge and Data Engineering*, 1990, 2(1), pages 125–142.

[SS87] A. Segev and A. Shoshani. Logical Modelling of Temporal Data. In *Proceedings of the ACM SIGMOD Annual Conference on Management of Data*, May 1987, pages 454–466.

[SS91] H. J. Schek and M. H. Scholl. From Relations and Nested Relations to Object Models. In M. S. Jackson and A. E. Robinson, editors, *Aspects of Database Systems: Proceedings of the 9th British National Conference on Databases*. Butterworth-Heinemann, 1991, pages 202–225.

[Ste95] A. Steiner. The TimeDB Temporal Database Prototype. Institute for Information Systems, ETH Zürich. http://www.timeconsult.com, September 1995.

[Swe93] Swedish Institute of Computer Science. *SICStus Prolog User's Manual*, prolog 2.1 #8 edition, 1993.

[Tan86] A. Tansel. Adding Time Dimension to Relational Model and Extending Relational Algebra. *Information Systems*, 1986, 11(4), pages 343–355.

[Tan93] A. Tansel. A Generalized Relational Framework for Modeling Temporal Data. In A. Tansel, J. Clifford, S. Gadia, S. Jajodia, A. Segev, and R. Snodgrass, editors, *Temporal Databases: Theory, Design, and Implementation*, Benjamin/Cummings Publishing Company, 1993, pages 183–201.

[TAO89] A. Tansel, E. Arkun, and G. Ozsoyoglu. Time-By-Example Query Language for Historical Databases. *IEEE Transactions on Software Engineering*, 1989, 15(4), pages 464–478.

[TCG+93] A. Tansel, J. Clifford, S. Gadia, S. Jajodia, A. Segev, and R. Snodgrass. *Temporal Databases: Theory, Design, and Implementation*. Benjamin/Cummings Publishing Company, 1993.

[TG89] A. Tansel and L. Garnett. Nested Historical Relations. *SIGMOD RECORD*, 1989, 18(2), pages 284–293.

[WD93] G.T.J. Wuu and U. Dayal. A Uniform Model for Temporal and Versioned Object-Oriented Databases. In A. Tansel, J. Clifford, S. Gadia, S. Jajodia, A. Segev, and R. Snodgrass, editors, *Temporal Databases: Theory, Design, and Implementation*, Benjamin/Cummings Publishing Company, 1993, chapter 10, pages 230–247.

[Wuu91] G.T.J. Wuu. SERQL: An ER Query Language Supporting Temporal Data Retrieval. In *Proceedings of the 10th International Phoenix Conference on Computers and Communications*, 1991, pages 272–279.

Lebenslauf

Name :	Andreas Steiner
Geburtsdatum :	12. Juni 1964
Geburtsort :	Richterswil
Zivilstand :	ledig

1971 – 1977	Primarschule in Pfäffikon (SZ)
1977 – 1985	Gymnasium Stiftsschule Einsiedeln (SZ)
1982 – 1983	Austauschjahr in Florida, USA
Juni 1985	Matura Typ B
1985 – 1986	Elektrotechnikstudium an der ETH Zürich
1986 – 1991	Informatikstudium an der ETH Zürich
November 1991	Abschluss als Dipl. Informatik-Ing. ETH
1992 – 1993	Stelle als Software-Entwickler in der Privatwirtschaft
1993 – 1995	Assistent am Institut für Informationssysteme der ETH Zürich in der Gruppe für wissensbasierte Systeme (Prof. Dr. Robert Marti)
1995 – 1996	Assistent am Institut für Informationssysteme der ETH Zürich in der Gruppe für Entwicklung und Anwendung – interimistisch (Prof. Dr. Carl A. Zehnder)
1996 – 1997	Assistent/Doktorand am Institut für Informationssysteme der ETH Zürich in der Gruppe für globale Informationssysteme (Prof. Dr. Moira C. Norrie)