form of a set of time points. This set can used as a domain for time point variables, such as the variable t above. This approach of using time points is more declarative than ours because it hides the implementation detail of using a set of pairs of time intervals and values to implement the partial function from time points to values. It is not clear, however, whether this higher level of abstraction leaves more opportunities for optimization or whether it is more user-friendly. We believe that their approach is not well-suited to version control, since version control inherently requires to associate one time interval with a version.

Another temporal OODB model is proposed by Cheng and Gadia [3]. Their temporal OODB language, called TempSQL, is based on temporal expressions, which map boolean predicates to temporal domains. For example, the temporal expression [[e.salary>15K]] returns the time period (lifespan) in which the salary of the employee e was greater than 15K. Such temporal expressions have also been used in the temporal query language GORDAS proposed by Elmasri and Wuu for the extended Entity-Relationship model [6].

5 Conclusion

We have presented an extension to the ODMG-93 model that incorporates a time dimension to data. In contrast to other models, time dimension in our model is an orthogonal property of data types. We have also presented an extension to OQL to accommodate time information. These language extensions are minimal, uniform, and easy to understand.

As a future work, we are planning to extend our model with parameterized types for representing transaction time and bi-temporal queries, and for representing time-series data and calendars. In addition, we are planning to design an optimization framework to evaluate the TOQL language constructs efficiently using currently proposed temporal evaluation techniques. We are also planning to work on the specification of the temporal integrity constraints and their use on semantic query optimization.

Acknowledgements: This work is partially supported by NSF Cooperative Agreement DDM 9320949 and by NSF under grant IRI-9509955.

References

[1] M. Atkinson, F. Bancilhon, D. DeWitt, K. Dittrich, D. Maier, and S. Zdonik. The Object-Oriented Database System Manifesto. In *Proceedings of the First International Conference on Deductive and Object-Oriented Databases, Kyoto, Japan*, pp 223–240, December 1989.

[2] R. Cattell. *The Object Database Standard: ODMG-93*. Morgan Kaufmann, 1994.

[3] T. S. Cheng and S. K. Gadia. An Object-Oriented Model for Temporal Databases. In R. T. Snodgrass, editor, *Proceedings of the International Workshop on an Infrastructure for Temporal Databases*, Arlington, TX, June 1993.

[4] U. Dayal and G. T. J. Wuu. Extending Existing DBMSs to Manage Temporal Data: An Object-Oriented Approach. In R. T. Snodgrass, editor, *Proceedings of the International Workshop on an Infrastructure for Temporal Databases*, Arlington, TX, June 1993.

[5] R. Elmasri, V. Kouramajian, and S. Fernando. Temporal Database Modeling: An Object-Oriented Approach. In *Proceedings of the 2nd International Conference on Information and Knowledge Management (CIKM)*, pp 574–585, Washington, DC, November 1993.

[6] R. Elmasri and G. Wuu. A Temporal Model and Query Language for ER Databases. In *Proceedings of the 6th International Conference on Data Engineering*, pp 76–83, February 1990.

[7] H. Gunadhi and A. Segev. A Framework for Query Optimization in Temporal Databases. In *Fifth International Conference on Statistical and Scientific Database Management*, pp 131–147, Charlotte, NC, April 1990. LNCS 420.

[8] C. S. Jensen, J. Clifford, S. K. Gadia, A. Segev, and R. T. Snodgrass. A Glossary of Temporal Database Concepts. *SIGMOD Record*, 21(3):35–43, September 1992.

[9] W. Kafer and H. Schoning. Realizing a Temporal Complex-Object Data Model. In *Proceedings of the ACM-SIGMOD International Conference on Management of Data, San Diego, California*, pp 266–275, June 1992.

[10] G. Ozsoyoglu and T. Snodgrass. Temporal and Real-Time Databases: A Survey. *IEEE Transactions on Knowledge and Data Engineering*, 7(4):513–532, August 1995.

[11] N. Pissinou, K. Makki, and Y. Yesha. On Temporal Modeling in the Context of Object Databases. *SIGMOD Record*, 22(3):8–15, September 1993.

[12] R. Snodgrass. Temporal Object-Oriented Databases: A Critical Comparison. In W. Kim, editor, *Modern Database Systems: The Object Model, Interoperability, and Beyond*, pp 386–408. Addison-Wesley, 1995.

[13] R. Snodgrass. *The TSQL2 Temporal Query Language*. Kluwer Academic Publishers, 1995.

[14] G. Wuu and U. Dayal. A Uniform Model for Temporal Object-Oriented Databases. In *Proceedings of the International Conference on Data Engineering*, pp 584–593, Tempe, Arizona, February 1992.

```
interface Time_Interval
{    attribute Time start;
     attribute Time end;
     Integer duration ();
     Time_Interval intersect ( in Time_Interval te );
     Boolean equal ( in Time_Interval te );
     Boolean overlaps ( in Time_Interval te );
     Boolean before ( in Time_Interval te );
     Boolean contains ( in Time_Interval te ); };
```

```
parameterized type Temporal ⟨ T ⟩
{    Structure { Time_Interval valid_time, Integer version, T value }
                Temporal_value;
     attribute List⟨ Temporal_value ⟩ history;
     attribute Integer size;
     attribute Temporal_value latest_version;
     Temporal_value version_projection ( in Short version )
                raises (no_such_version);
     Temporal_value time_projection ( in Time time )
                raises (not_defined_at_that_time);
     Temporal⟨ T ⟩ interval_projection ( in Time_Interval i );
     new_version ( in T value, in Time event );
     close_version ( in Time event ); };
```

The TOQL syntactic constructs are translated into the following OQL expressions:

$[time]$	$\Rightarrow$	**struct**(start: $time$, end: $time$)
$[time_1, time_2]$	$\Rightarrow$	**struct**(start: $time_1$, end: $time_2$)
$e@version$	$\Rightarrow$	e.version_projection($version$).value
$e : time$	$\Rightarrow$	e.time_projection($time$).value
$e : interval$	$\Rightarrow$	e.interval_projection($interval$).history
$?v$	$\Rightarrow$	e.valid_time
$\#v$	$\Rightarrow$	e.version

If a temporal variable v is mentioned in a query (in a term other than $?v$ or $\#v$), it is translated into v.value. A projection $e.A$, where expression e is of type Temporal⟨ T ⟩ is translated first into e : **now**.A.

For example, Queries 1 and 2 are translated as follows:

```
select i.name, i.rank.latest_version.value
from i in Instructors
     c in i.teaches.latest_version.value
where c.offered_by.latest_version.value.name = 'CSE'

select i.name, r.value
from i in Instructors,
     d in i.dept.interval_projection(
          struct(start:1/1/88,end:1/1/90)).history,
     r in i.rank.interval_projection(d.valid_time)
          .history
where d.value.name = 'CSE'
```

Updating an element e of type Temporal⟨ T ⟩ with a new value v of type T is calling e.new_version(v, t), where t is the time when this update will take in effect. This operation will close the latest version of e (i.e. it will replace the value of e.latest_version.end from **now** to t) and it will insert a new version with valid time $[t,$ **now**$]$ at the top of the list (updating the latest_version attribute accordingly). Creating a new object of type Temporal⟨ T ⟩ with initial state v of type T at time t is also achieved by calling e.new_version(v, t). Logically deleting e at time t is closing the latest version of e using e.close_version(t).

4 Comparison with Other Ongoing Research

There are several proposals for temporal object-oriented databases with similar goals. A survey of the issues involved in temporal object-oriented database modeling can be found in [12]. Here we present only the approaches that are most relevant to our work.

Our work is motivated by the temporal OODB model proposed by Elmasri, Koramajian, and Fernado [5]. Their work analyzed two ways of assigning a temporal dimension to an OODB: using either object or attribute versioning. They analyzed the storage cost of each method and concluded that attribute versioning requires less storage space than object versioning. They also identified the temporal integrity constraints needed to be enforced to objects and relationships for each of these two methods. More importantly, they proposed a method for translating temporal schemas into regular OODB schemas. In particular, the translation template they used for compiling attribute versioning is very similar to the implementation of the class Temporal⟨ T ⟩ in TOQL. The only difference is that they inlined this implementation (i.e., the list that represents the history of a type) inside the type definition, instead of using a parametric type similar to Temporal⟨ T ⟩. Our work extends their work by providing a query language for these schemas and a default query translation.

Our temporal extensions to OQL resemble the language extensions proposed by Dayal and Wuu for SQL3 [4] and OODAPLEX [14]. In both proposals, they use a syntax for temporal projections that looks very similar to ours. For example, they use name(manager(dept(e)(t))(t)) to denote the TOQL time projection e.dept:t.manager:t.name. They introduce a function lifespan, which, when applied to a temporal value, it retrieves the lifespan of this value in a

Variable d, called a *temporal variable*, ranges over this history. The value of d is called a *temporal element* and can be viewed as a triple of a Time_Interval, a version number, and a Department. When d is mentioned in the query, it refers to the actual department; when ?d is mentioned, it refers to the time interval of d; when #d is mentioned, it refers to the version number of d. The interval projection i.rank:?d uses the time interval of d to retrieve the rank of i at this time period. This is a way to explicitly synchronize events in TOQL. The following example uses the current time **now** to retrieve the history during a time window:

Query 3) List the salary history of all instructors who were employed sometime during the last 4 years.

```
select i.name, s
from i in Instructors,
     s in i.salary:[now-4years,now]
```

Time constraints can be assigned even in complex navigations through the database objects:

Query 4) Give me the head of the department in which Smith worked on 1/1/88.

```
select d.head:1/1/88
from i in Instructors,
     d in i.dept:[1/1/88]
where i.name = 'Smith'
```

Note that it would not be a good choice to write i.dept:1/1/88.head:1/1/88 instead of d.head:1/1/88, because it would raise an exception if Smith did not work at any department at that time.

Temporal aggregation in TOQL is achieved using the standard OQL aggregation operators:

Query 5) What was the average salary of the new assistant professors before they joined CSE?

```
avg(select i.salary:(?d.start-1)
    from i in Instructors,
         r in i.rank:[start,now],
         d in i.dept:?r
    where r = 'Assistant Professor'
      and d.name = 'CSE')
```

the value of ?d.start is the starting time of the time interval of d. Thus, i.salary:(?d.start-1) retrieves the salary of i at the time point immediately before ?d.start. The following query uses the method duration to retrieve duration of a time interval before an aggregation:

Query 6) For how long has Smith taught CSE5330?

```
sum( select ?i.end-?i.start
     from c in Courses,
          i in c.taught_by:[start,now]
     where c.code = 'CSE5330'
       and i.name = 'Smith')
```

Retrieving time periods is also straightforward:

Query 7) Retrieve the time period when Smith was an associate professor.

```
select ?r
from i in Instructors,
     r in i.rank:[start,now]
where i.name = 'Smith'
  and r = 'Associate Professor'
```

Queries that retrieve old object versions can be specified using version projection:

Query 8) List the names and salaries of all instructors who started with a first salary of at least 30K.

```
select i.name, i.salary
from i in Instructors
where i.salary@0 ≥ 30K
```

Recall that i.salary in the query projection is equivalent to i.salary:**now**. The following example uses the #i operation, which retrieves the version of the temporal variable i, to retrieve versions in the same neighborhood:

Query 9) Who was the instructor of CSE5330 after Smith was the instructor and when did this change take effect?

```
select c.taught_by@(#i+1).name, ?i.end
from c in Courses,
     i in c.taught_by:[start,now]
where c.code = 'CSE5330'
  and i.name = 'Smith'
```

3 The Temporal Data Type

Figure 2 defines the classes for Time_Interval and Temporal⟨ T ⟩. The history of an object of type T is represented by a list of triples of type Temporal_value. The head of the list is the latest version while the last element of the list is the initial version (version number 0). The latest version can be either open (if its valid time has end=**now**) or closed, which indicates that this object does not exist at the current time. The attribute size is the list size. The attribute latest_version is a pointer to the latest version if the latest version is open, otherwise it is null. It is used for faster access to the object at the current time.

```
interface Department
(    extent Departments
     keys dno, name ): persistent
{    attribute Short dno;
     attribute String name;
     attribute Temporal⟨ Instructor ⟩ head;
     temporal relationship Set⟨ Instructor ⟩ instructors
       inverse  Instructor::dept;
     temporal relationship Set⟨ Course ⟩ courses_offered
       inverse  Course::offered_by; };

interface Instructor
(    extent Instructors
     key ssn ): persistent
{    attribute Short ssn;
     attribute String name;
     attribute Temporal⟨ Integer ⟩ salary;
     attribute Temporal⟨ String ⟩ rank;
     temporal relationship Department dept
       inverse  Department::instructors;
     temporal relationship Set⟨ Course ⟩ teaches
       inverse  Course::taught_by; };
```

```
interface Course
(    extent Courses
     keys code, name ): persistent
{    attribute String code;
     attribute String name;
     temporal relationship Department offered_by
       inverse  Department::courses_offered;
     temporal relationship Instructor taught_by
       inverse  Instructor::teaches;
     temporal relationship Set⟨ Course ⟩ is_prerequisite_for
       inverse  has_prerequisites;
     temporal relationship Set⟨ Course ⟩ has_prerequisites
       inverse  is_prerequisite_for; };
```

Operation	*Type*	*Explanation*
$e@version$	T	Version projection
$e : time$	T	Time projection
$e : interval$	Temporal⟨ T ⟩	Interval projection
$?v$	Time_Interval	The time interval of v
$\#v$	Integer	The version of v

where $time$ is an expression of type Time, e is an expression of type Temporal⟨ T ⟩, $version$ is an expression of type Integer that may depend on the parameter **last** (which indicates the latest version of an object), $interval$ is an expression of type Time_interval, and v is a temporal variable.

For example, let Smith be the object that represents the instructor with name Smith. The version projection Smith.dept@n returns the n*th* version of dept, i.e., the n*th* department where Smith worked, where n=0 is the first version and n=**last** is the latest. If there is no version n, then this expression will raise an exception at run time. The time projection Smith.dept:1/1/90 returns a value of type Department, which represents the department where Smith worked on 1/1/90. If Smith did not work in any department at that time, then this expression will raise an exception at run time. Similarly, Smith.dept:**now** returns the current department of Smith. This is the default, if no time constraint is specified. That is, Smith.dept is equivalent to Smith.dept:**now**. Notice that Smith.dept:**now** is not necessarily the same as Smith.dept@**last**. If Smith does not work at any department at the current time, then the first expression would raise an exception, while the second would return the last department.

To avoid raising exceptions, one may use interval projections instead of version or time projections. For example, the interval projection Smith.dept:[1/1/88,1/1/90] returns a value of type Temporal⟨ Department ⟩, which contains the history of the departments where Smith worked between 1/1/88 and 1/1/90. This history may be empty. Similarly, the interval projection Smith.dept:[1/1/90] returns a value of type Temporal⟨ Department ⟩, which is either an empty history of departments, if Smith did not work in any department at that time, or a history with one element, namely Smith's department on 1/1/90. To retrieve the entire department history for Smith, one can use Smith.dept:[**start**,**now**], where **start** is the beginning of time.

The mechanism for traversing a history of type Temporal⟨ Department ⟩ is similar to the way of traversing any collection data type in OQL. For example, consider the following temporal projection query:

> *Query 2)* Give me the name and the rank of all persons who were CSE instructors between 1/1/88 and 1/1/90.

```
select i.name, r
from i in Instructors,
     d in i.dept:[1/1/88,1/1/90],
     r in i.rank:?d
where d.name = 'CSE'
```

The interval projection i.dept:[1/1/88,1/1/90] retrieves the history of the departments where the instructor i worked.

types. More specifically, we define a parameterized ODL class, Temporal⟨ T ⟩, that lifts any type T to a temporal type. Our temporal query language, TOQL, recognizes the syntax of OQL but it also provides additional syntactic constructs to manipulate historical information. If these additional syntactic constructs were not used, then the semantics of a query in TOQL would become the semantics of this query in OQL. TOQL treats a Temporal⟨ T ⟩ type like any ODL collection type. That is, one can scan the whole history of a value or do an associative access over instances of this type using the time or the data dimension alone or both. This approach of treating historical data as ordinary collection types, provides a seamless integration of time information with non-historical data.

In the general definitions of temporal databases [8], two main time dimensions are supported: the valid time, when changes occur in the real world, and the transaction time, when changes are recorded in the database. The version of TOQL presented in this paper handles valid time only. This is similar to supporting object versions, except that the validity time of a version is also included. A future extension of this work will include the treatment of transaction time.

This paper is organized as follows. The TOQL query language is described in Section 2. The class definition of Temporal⟨ T ⟩ and the default implementation of the TOQL queries are described in Section 3. Finally, Section 4 summarizes related work.

2 Our Data Model and Query Language

The ODMG'93 standard provides a data description language, ODL, for defining object classes, and a data manipulation language, OQL, for defining OODB queries. Our data description language, TODL, is ODL enhanced with a number of class definitions and syntactic constructs that add a time dimension to types. Our query language, TOQL, is an extension of OQL.

Figure 1 defines a University database schema in TODL. The parameterized type Temporal⟨ T ⟩ adds a temporal dimension to a type T. For example, the type of the attribute head in the Department class is Temporal⟨ Instructor ⟩, which means that we keep the history of all department heads. The **relationship** type in ODL is used to define binary one-to-one, one-to-many, and many-to-many relationships between object types. For example, the relationship instructors in the class Department is the set of all instructors who work in a department. The inverse relationship is dept, which is specified in the Instructor class. That is, this is a one-to-many relationship between instructors and departments. Relationships between objects can also be classified into temporal and non-temporal. A special keyword temporal is used to tag a temporal relationship (both participant classes in a relationship must be tagged in order for this relationship to be temporal). Any two objects that participate in a temporal relationship are temporal entities. The schema in Figure 1 does not include any non-temporal relationships. One example of such relationship is between parent and children, which never changes. One example of temporal relationship is between instructors and courses, which makes the participated entities temporal, i.e. the temporal relationship between taught_by: Instructor and teaches: Set⟨ Course ⟩ becomes a simple relationship between taught_by: Temporal⟨ Instructor ⟩ and teaches: Temporal⟨ Set⟨ Course ⟩ ⟩.

The query language TOQL is a superset of OQL: it recognizes the syntax of OQL but it also provides additional syntactic constructs to manipulate historical information. If these additional syntactic constructs were not used, then the semantics of a query in TOQL would become the semantics of this query in OQL. For example, the following query:

> *Query 1)* Give me the name and the rank of all instructors who teach at least one CSE course.

```
select i.name, i.rank
from i in Instructors,
     c in i.teaches
where c.offered_by.name = 'CSE'
```

does not consider the historical information of objects; instead, it retrieves the latest version of each referred object (at the current time).

We define the following TODL classes for specifying historical information:

Time_Interval	The class of time intervals
Temporal⟨ T ⟩	The class that represents the history of T

Class Time_Interval has two public attributes, start and end, both of the TODL type Time, which indicate the beginning and the end of the time interval. Time intervals can be constructed as follows:

Operation	*Type*
$[time]$	Time_Interval
$[time_1, time_2]$	Time_Interval

where the type of $time$, $time_1$, and $time_2$ is Time. The time interval, $[time]$, contains just one time point: $time$, while the time interval, $[time_1, time_2]$, contains all the time points between and including $time_1$ and $time_2$. TOQL supports a number of syntactic constructs to compare two time intervals a and b, such as a **before** b which is equivalent to a.end $<$ b.start.

The Temporal⟨ T ⟩ parameterized class adds a temporal dimension to the type T. There are four syntactic constructs in TOQL that are used to manipulate Temporal⟨ T ⟩ objects:

join (or TE-join [7]) in a temporal relational query than in a non-temporal query that manipulates time information in an ad-hoc way. We believe that the lack of a higher-level associative language in the old OODBs resulted in a lack of temporal OODB language support in the past. But this is not true any more since most recent OODB languages provide an SQL-like interface, which integrates the best features of both worlds: the expressiveness of the OODB type models and the declarativeness of the traditional relational languages. In addition, until very recently, there was no standard for object-oriented databases that is acceptable by most researchers and database vendors. This too is not true any more. There is now a proposal for an OODB standard, called ODMG-93 Release 1.2 [2], which supports a declarative higher-level object query language, called OQL. Essentially all OODB companies have committed to supporting an OQL interface to their systems in the near term. When restricted to flat (1NF) relations, OQL closely resembles SQL. The temporal language, TOQL, described in this paper, is an extension of OQL. TOQL extends OQL with a number of new syntactic constructs to manipulate historical information. A query without these constructs has the semantics of a regular OQL query. That is, it manipulates the database state at the current time.

Our goal is not to invent new temporal query language constructs, since there are many proposals for such constructs [10]. Rather, we are trying to incorporate temporal constructs to OQL with minimal changes to the existing OQL structure, so that the nature of OQL is not affected. Instead of designing yet another temporal query language, our temporal language extensions are syntactic sugar on top of OQL, which can be expressed in terms of the basic OQL constructs. This modeling is only possible if a language supports nested collections, since historical data hold multiple versions of the same object, and, thus, collections of historical data are nested collections of objects in that respect. Even though temporal queries in such modeling are expressed in terms of regular queries in the underlying language, this does not necessarily imply that these queries cannot be optimized to make use the currently proposed temporal evaluation methods. In fact, the way our proposed temporal features are specified in ODL is using a small number of carefully defined ODL classes. Like any abstract data type, the actual implementation of these classes can be separated from the specification, and we may have numerous different implementations, thus resulting to different evaluation algorithms for the optimizer to select from. We leave it for future work to demonstrate this claim.

When extending a language with temporal features, there are many issues to consider [11]. The most important one is to decide which objects need to be historical. We have two options: one is that every object or value has a time dimension; the other is that the user chooses which objects or values are historical. The first option seems attractive for systems that support write-once-read-many (WORM) devices, such as optical disks, since no object version is physically removed from the database. The second option, though, models better the real world because there are some objects, such as a person's date of birth, that never change. This option requires a syntactic construct in the data definition language to declare an object or a value as temporal. If an object is defined as non-temporal, there are two choices to consider for updates: whether the language allows updates to this object but only remembers the last update or whether it allows this object to be initialized but forbids any update, in which case the non-temporal object becomes immutable. The first choice is not very attractive because it makes the system unable to rollback to an old state, since some object versions may be lost during updates.

Another issue to consider when designing a temporal OODB language is the object level on which we add a temporal dimension. We have three choices to consider [5]: object-versioning, where we associate historical information to objects, attribute-versioning, where we associate historical information to object attributes, and type-versioning, where we associate historical information to any typed object or value. The latter choice can easily capture the first two choices. Type-versioning is useful for a temporal OODB that supports attributes with complex values. For example, if an attribute is a set of tuples, then it might be a better choice to assign versions to the tuples instead of the whole set. In addition, type-versioning gives a better handle for implementation, since the temporal annotation that assigns a history to a typed object can always be defined as a parameterized type (a type constructor), such as Temporal$\langle$ T $\rangle$, that promotes a type T into a temporal type that contains the history of all versions of T. That way, the implementation of a temporal type is actually the implementation of the OQL parameterized object class Temporal$\langle$ T $\rangle$.

The query language of a temporal OODB must support many different types of temporal queries, including temporal projection and coincidence queries [9]. The most difficult task in expressing temporal queries is event synchronization, mostly found in coincidence queries. For example, inquiring about the salary of the head of a department at some point in time, is requesting to synchronize the temporal attribute, head, of the department with the temporal attribute, salary, of the head. It would be most convenient if a temporal OODB language assumes such a synchronization of events along the object traversal paths (such as the path from the department object to the head object) as a default, only to be changed when a temporal constraint is explicitly attached to a path, e.g. on the salary attribute.

Our data model is based on type-versioning. That is, time dimension in our model is an orthogonal property of data

A Temporal Object Query Language

Leonidas Fegaras Ramez Elmasri

Department of Computer Science and Engineering
The University of Texas at Arlington
416 Yates Street, P.O. Box 19015
Arlington, TX 76019-19015
email: {*fegaras,elmasri*}*@cse.uta.edu*

Abstract

One of the main reasons that current commercial DBMSs provide only minimal support for temporal concepts is the size and complexity involved. Object-oriented databases, on the other hand, were developed to deal with complex database applications. Because temporal concepts require complex type support and advanced modeling concepts, object-oriented databases are excellent candidates for realization of temporal databases without requiring fundamental extensions to the basic data model.

The ODMG (Object Data Management Group) has proposed a standard for object-oriented databases, including a standard object model, an object query language (OQL), and an object definition language (ODL). These do not include temporal support except at the data type level, as in SQL2. In this paper, we present a language extension to OQL to accommodate time information. Our goal is not to propose a new temporal query language, but to incorporate temporal features into the existing OQL framework.

Areas: temporal databases, temporal languages and architectures.

1 Introduction

Temporal databases provide a complete history of all changes to a database and include the times when changes occurred. This permits users to query the current state of the database, as well as past states, and even future states that are planned to occur. Many applications require the existence of temporal data. For example, medical information systems store information on patient histories and how each patient responds to certain treatments over time. Financial databases use histories of stock prices to make investment decisions. Databases for planning applications store information about future events. Many other applications require storing the histories of changes over time. Although current commercial DBMSs do not provide explicit temporal support, users have implemented temporal applications by programming the temporal semantics themselves in an ad-hoc way. The long-term goal of temporal database research is to include temporal concepts in commercial DBMS products, thus making it easier for users to develop temporal applications.

One of the main reasons that current commercial DBMSs provide only minimal support for temporal concepts is the size and complexity involved. Obviously, large amounts of data will need to be stored and managed in most temporal applications. Query languages and data models that are user friendly in supporting temporal data and queries need to be developed, as well as processing and optimization techniques for efficiently executing temporal queries.

Object-oriented databases (OODBs) were developed to deal with complex database applications [1]. Because temporal concepts require complex type support and advanced modeling concepts, OODBs are excellent candidates for realization of temporal databases without requiring fundamental extensions to the basic data model.

Even though there is already a sizable body of proposals for temporal relational databases (extensive surveys can be found in [10] and in Chapter 10 of [13]), there is little work reported on temporal object-oriented databases [12]. The main reason is that early OODB systems lacked a declarative language for associative access of data. Instead, they used simple pointer chasing to perform object traversals. Even though it is always possible to extend any data model (including the OO model) to capture time, accessing historical data becomes cumbersome if the temporal operations are not directly supported in the language by means of special language constructs. Furthermore, the task of the optimizer becomes easier if the temporal features are supported directly in the language. For example, it is easier for an optimizer to identify an opportunity of using a temporal equi-