Answering Queries in Temporal Object-Oriented Databases
Mohamed E. El-Sharkawi*
Department of Computer Science and Communication Engineering
Kyushu University
Fukuoka 812, Higashi, Hakozaki, Japan

Abstract
In this paper, we study problems in processing queries against temporal object-oriented databases (TOODB's). Solutions are given for the following problems: (1) A problem due to object migration. In object-oriented databases an update to some instance variables of an object may enforce the object to "migrate" to a new class. Due to this migration an object's history in a TOODB is distributed among several classes. Given a temporal query, we need to determine which classes in the schema should be searched to answer the query. (2) Problems due to the rich semantics of the model, specifically problems in creating versions of shared and default-valued instance variables and problems associated with complex instance variables. (3) Problems due to schema evolution. When the schema is changed, we have to keep the old schema in order to answer temporal queries. (4) Problems due to updating the history. In historical and temporal databases, it is possible to correct the history information when some errors have been discovered. Updating the history of an object may affect its current position in the class lattice. We present an approach to determine whether a history update of an object will affect its current position.

1. Introduction
Advanced database applications like computer-aided design (CAD) and office information systems (OIS) have several requirements that are not met by existing record-oriented data models. These applications require more semantically rich data models. The object-oriented data model has promising features that make it suitable for these new applications. It has the following important features:

*Author's mailing address from May, 1991
170 Nile Street, Agoza, Giza, Egypt

DATABASE SYSTEMS FOR ADVANCED APPLICATIONS '91
Ed. A. Makinouchi
©World Scientific Publishing Co.

(1) Its rich semantics provide a more natural way to model the real-world. Entities in the real-world are objects in the model. Objects are grouped into classes, classes are organized in a hierarchy (lattice) representing IS-A (specialization) relationship between classes. A class inherits all properties of its superclass(es).
(2) Dynamic aspects of entities can be stored in the database through associating methods to classes.
(3) It supports a uniform language to write applications as well as data definition and manipulation statements.

Another important requirement necessary to support new applications is adding the time dimension into databases. A database should store the real-world history as well as its current status. Users may access history data as well as current data. Extending the relational model with the time dimension has received a lot of research efforts [Mck]. It seems, however, that the issue is not studied extensively in the context of the object-oriented data model.

The focus of our paper is studying and solving problems in answering queries in temporal object-oriented databases. Due to the rich semantics of the model, approaches used in extending the relational model with time are not applicable.

Solutions are given for the following problems:
(1) The effect of object migration on TOODB's. In an object-oriented database an update to some instance variables of an object may enforce the object to migrate to a new class. Due to this migration the object's history is distributed among several classes. Given a query, we need to find an efficient way to determine the class of the queried object.
(2) Problems due to the rich semantics of the model, specifically problems in creating versions of shared and default-valued instance variables and problems associated with complex instance variables. Creating versions of objects affected by updating a shared value or default-valued instance variable is not efficient. Instead, we postpone creating versions till query time.
(3) Problems due to schema evolution. When the schema changes as described in [BKKK], we have to create versions of all affected objects and keep the old
schema in order to answer temporal queries. Schema changes are classified into two groups. In the first group of changes, we have to create versions of objects affected by schema change. In the second group, however, we modify the database catalog to reflect schema change and versions of objects are created at query time.

(4) Problems due to updating the history. In historical and temporal databases, it is possible to correct the history information when some errors have been discovered. Updating the history of an object may affect the current version (and/or some of the historical versions) of the object in the class lattice.

2- Basic Concepts

2-1 The Object-Oriented Data Model

We assume that there are the following sets:

(1) A finite set of classes called Basic-Classes. This set includes primitive, system-defined classes INTEGER, REAL, STRING, and BOOLEAN.

(2) A countably infinite set of classes C.

(3) A countably infinite set of names IV called instance variables.

(4) A countably infinite set of names M called methods.

(5) A countably infinite set of object identifiers O.

A database schema is defined as follows:

\[
\text{DBS} = (C, IV, DOM, PC, M, IS-A),
\]

where, C is a finite subset of C. IV is a function that assigns to each class in C a finite subset of instance variables from IV. DOM is a function that assigns for each instance variable a domain. This domain is either one of the Basic-Classes or one of the user-defined classes. If each instance variable of a class has its domain as one of the Basic-Classes, the class is called a simple class, otherwise is called a complex class. An object which is an instance of a simple class is called a simple object. On the other hand, if it is an instance of a complex class, it is called a complex object. PC is a function that assigns for each class Ci in C a predicate PCi. M is a function that assigns to each class in C a finite subset of M. IS-A is the hierarchy in which classes are organized in the schema and is defined as follows:

The IS-A relationship between classes is depicted as a rooted direct acyclic graph (DAG). The root is a system-defined class called OBJECT. Other nodes in the graph represent user-defined classes in the schema. An edge e(Ci, Cj), from class Ci to class Cj means that Ci IS-A Cj. Class Cj (Cj) is called to be a subclass (superclass) of Cj (Ci), and written Ci \( \ll Cj \) (Cj \( \gg Ci \)). The IS-A relationship is transitive. That is, if Ci \( \ll Cj \) and Cj \( \ll Ck \), then Ci \( \ll Ck \). Therefore, we need to distinguish between immediate and non-immediate super (sub) classes of a class.

The set of immediate superclasses of a class C is defined as:

\[
\{Cj | Cj \gg C \wedge \exists Ci, C1 \ll Cj \wedge C1 \gg C\}.
\]

Similarly, the set of immediate subclasses of a class C is defined as:

\[
\{Cm | Cm \ll C \wedge \exists Cn, Cn \gg Cm \wedge Cm \ll C\}.
\]

An instance of a schema is an assignment of objects to classes. An object O belongs to a class Cj iff PCj(O) is true. We define a subset of instances of a class as its direct instances. This set is defined as: Direct instances of class C =

\[
\{O | PC(O) = \text{true} \}
\]

We also put the following two restrictions on objects in the database instance:

(1) An object can never be a direct instance of class OBJECT, and (2) Each object is a direct instance of only one class [K]. This restriction is put in many object-oriented systems (e.g. ORION).

An instance of the schema also assigns for each method in a class a code that implements the method.

Fig.1 shows a schema for a university database.

2-2 Temporal Databases

For some applications, it is necessary to store the current status as well as the history of the world. In temporal databases, users may access history data by using temporal queries. The query contains, in addition to conditions should be satisfied by the output, the time at which the output is valid. For example, find the Salary of Tom from March 1989 to Feb. 1990. Several
efforts have been done to extend the relational data model with the time dimension.

Temporal databases are classified in [SA] into three types: rollback, historical, and temporal databases. This classification is based on two criteria, the type of the time supported by the database and whether it is permitted to update the history. There are two types of time: transaction time and valid time. Transaction time is the time at which information was stored in the database. Valid time is the time at which the real-world was changed. Rollback databases support transaction time, historical databases support valid time, and temporal databases support both transaction and valid times. In rollback databases, however, it is not permitted to update the history as known of now.

In our discussion it is immaterial which time is supported. On the other hand, however, permitting updating the history has several problems. These problems will be discussed in Section 4-4. In context of the relational data model either tuples or attributes are time stamped, here, we assign time to objects (i.e., tuples). Tables 1 and 2 show examples of temporal relations with time assigned to attributes and tuples, respectively.

### Table 1 Time associated with tuples

<table>
<thead>
<tr>
<th>Emp#</th>
<th>Salary</th>
<th>Manager</th>
<th>FROM</th>
<th>TO</th>
</tr>
</thead>
<tbody>
<tr>
<td>52</td>
<td>18K</td>
<td>Smith</td>
<td>5</td>
<td>9</td>
</tr>
<tr>
<td>52</td>
<td>20K</td>
<td>Smith</td>
<td>10</td>
<td>20</td>
</tr>
<tr>
<td>52</td>
<td>20K</td>
<td>Jones</td>
<td>30</td>
<td>38</td>
</tr>
<tr>
<td>52</td>
<td>31K</td>
<td>Jones</td>
<td>39</td>
<td>Now</td>
</tr>
</tbody>
</table>

### Table 2 Time associated with attributes

<table>
<thead>
<tr>
<th>Emp#</th>
<th>Salary</th>
<th>Manager</th>
</tr>
</thead>
<tbody>
<tr>
<td>52</td>
<td>18K(5,9)</td>
<td>Smith(5,29)</td>
</tr>
<tr>
<td>52</td>
<td>20K(10,38)</td>
<td>Jones(30,Now)</td>
</tr>
<tr>
<td>52</td>
<td>31K(39,Now)</td>
<td></td>
</tr>
</tbody>
</table>

3- Updates and Object Migration in Object-Oriented Databases

In this section, we, informally, discuss updates and the notion of object migration in object-oriented databases. A detailed discussion of these issues is given in [EK]. We consider three types of updates on objects: (1) Adding instance variables to an object, (2) Dropping instance variables from an object, and (3) Modifying instance variables of an object.

The first two types of updates (adding and dropping instance variables of an individual object) are not permitted in conventional databases. In the relational model, we cannot add some additional attributes to a tuple in a relation.

Objects belong to classes. An update to an object may cause object migration. That is, after the update the object is no longer an instance of its class and it has to migrate into a new class. For example, instances of class GOOD-GRAD students are defined to be graduate students with total marks exceeding certain value. An update that modifies this value may cause an object to migrate from class GRAD to class GOOD-GRAD. Such object migration has some side effects that should be handled by the system. If class GOOD-GRAD has some instance variables not defined in class GRAD, values of these instance variables have to be provided by a user, or considered null (which sometimes may not be possible). An object may move in three directions. It may move to one of its superclasses, subclasses, or to a class which is neither its super nor its subclass. In [EK], we have studied these types of object migration along with actions that should be taken by the system when an object migrates from one class to another.

We need the following definitions which are related to the coming discussions.

**Definition 1.** Class $C_j$ is the stabilization point of class $C_i$, if an object in $C_i$ migrates (via one or more updates) into class $C_j$ and then any update does not affect its position in $C_j$.

Stabilization point(s) of a class is defined by the database administrator. It can also be changed according to changes in the real-world.

There are three types of object migrations: Generalization, Specialization, and Migration. Generalization is done when an object moves to one of its superclasses. Specialization is done when an object moves to one of its subclasses. Migration is done when an object migrates to class which is neither its super nor subclass.

The notion of migration graph is introduced in order to trace how an object may migrate. It is defined as follows.

**Definition 2.** A migration graph $(V, E, L)$ is a directed labeled graph. $V$ is the set of nodes. Each node corresponds to a user-defined class in the schema and rooted with a system-defined class called OBJECT. $E$ is the set of edges in the graph. Each edge has a label, and $L$ is the set of labels. There is an edge from class $C_i$ to class $C_j$, if some update on object in $C_i$ will cause object migration to class $C_j$. The set $E$ consists of three types of edges, generalization edges (GE), specialization edges (SE), and modification edges (ME). A generalization edge corresponds to migration to a superclass. A specialization edge corresponds to migration to a
subclass. A modification edge corresponds to migration to a class neither super nor subclass. An edge is labeled with predicates that when satisfied by some updated instance variables an object will be enforced to migrate (e.g. the label of the edge between GRAD and GOOD-GRAD is TotalMarks > 100).

The migration graph of the example schema is shown in Fig. 2 (for clarity labels are omitted).

4- Temporal Object-Oriented Databases

Applications like medical and office information systems require adding the time dimension to the database in order to access the history of the real-world. Several research efforts have been devoted to study adding time to relational databases (see [M] for a bibliography). It seems that adding the time dimension to object-oriented databases has not been extensively studied [UD]. Most of the work on extending semantic databases with time concentrates on defining a temporal model and a temporal query language [AQ, LW]. In temporal databases, objects’ histories have to be stored. All versions of an object have the same object identity. Enhancing the performance of query processing is an important issue to implement temporal databases.

In this section we use the term temporal database to refer to a database that is augmented with the time dimension and an update on an object creates a new version of the object. Associated with each version two time-related instance variables FROM and TO. These two instance variables determine the period this version is valid. The type of time supported by the database is immaterial, however, we consider transaction time.

Before we discuss these problems, a comparison between temporal database issues and version management and control issues [CK86, CK88] is given:

1. In temporal databases there is a total order between versions of an object. That is, the notion of existence of several “current versions” of an object is not applicable.
2. The number of objects in a temporal database is very large. Hence, the problem of efficiently answering queries is critical for the system performance. In CAD databases, there are many of transient and working versions of an object, however, the number of released versions may not be so large. Thus, the problem of managing and controlling versions is important in CAD systems.
3. The problem of object migration is not considered in the literature on CAD systems.
4. The notions of static and dynamic binding of an object with its components is not applicable in databases with the time dimension. Binding in these systems is based on matching the life time of an object version with the versions of the objects it refers. That is, if a version of an object O is created at time $t_1$ and updated at time $t_2$, its life time spans $t_1$ to $t_2$. Binding this object with the objects it references is determined as all versions of the referenced object with life spans intersect $t_1$ to $t_2$.

In this section, we study and give solutions to four problems in answering temporal queries.

4-1 A Problem Due to Object Migration

From the previous analysis, we notice that classes of the versions in an object's history can be specified by the set of classes that consists of the class of the initial version, all its stabilization points, and the classes between the initial class and each of its stabilization points. For example, consider the schema shown in Fig. 3, an object O was updated several times creating a group of versions. The versions are not instances of one class. At the beginning, O joined the university as an undergraduate student and currently it is an assistant professor.

Given the query GET(O, July 1985.) The very intuitive approach to answer such a query is to search the set of all instances in the initial class of the queried object, if it is not found search one of the next contiguous classes. This process continues until the required object is found. If the object cannot be found after visiting all the stabilization points of its initial class, a message is sent...
to the user that the object cannot be found. This approach is very much time consuming. If the object has been updated several times, it is necessary to search instances of several classes and in some cases the search is unnecessarily performed. For example, class TA should be searched since it is contiguous to class GRAD. Object 0, however, was not at any time an instance of class TA.

We propose two solutions to this problem. In the first solution, a new system-defined class is introduced. This class is used to keep track of updates in the system. In the other solution, each user defined class is extended by adding some system-defined instance variables to the class.

In the first solution, we introduce a new system-defined class called UPDATES. Instances of this class are updates introduced to the system. There is a tradeoff between storing all updates and only storing updates that enforce objects to migrate. In the first case, methods that answer queries on the real-world evolution can be associated with the class UPDATES. Examples of these methods are, find all updates at a specific time period, find all updates done on a certain object at a certain time, and so on. However, the number of instances in this class will be very large. In the second case, the number of instances is drastically reduced, however, we may not be able to easily answer queries that investigate the real-world evolution.

Class UPDATES has the following instance variables: OBJECT, OBJECT-CLASS, TIME, and DEL-INDICATOR. OBJECT is the updated object, OBJECT-CLASS is its class after the update. TIME is the time at which the update has occurred. DEL-INDICATOR is used to mark updates that are deletion of objects. When an update occurs, the object migration mechanism will create an instance of UPDATES. To speed up accessing instances in class UPDATES, all updates over an object may be grouped together and sorted chronologically.

Answering a query will proceed in two steps: first, the class of the queried version is determined by consulting the class UPDATES. Second, when the class has been specified, a query will be invoked to this class on behalf of the user. We assume that a temporal query has the form: GET (O, Time-Specification), where O is the object to be accessed and Time-Specification is either a specific point of time (as zero, now, or t) or a period of time. When the system receives this message, it sends to class UPDATES the message FindClass(O, Time-Specification). When receiving this message,
UPDATES finds the classes to which object 0 belongs at this Time-Specification. Then, it sends the message GET (0, Time-Specification) to each class Ci of these classes. Now, we give an outline of the procedure used to implement the message FindClass(O, Time-Specification).

1. Check that there is an update over O in UPDATES, if there is no such an update, O belongs to its initial class. Send the message GET (O) to class Ci.

2. Check that object O has not been deleted before time-specification T.

3. According to the time specification in the query, class UPDATES determines classes that contain the needed versions of the object O. For example, if Time-Specification is the time point “zero”, UPDATES sends the message GET(O, zero) to the initial class of O. If Time-Specification is the time point “now”, UPDATES finds the most recent update on O, determines the class of O, and sends it the message GET(O, now). If Time-Specification is the point “= tqs”, find an update which is done on object O at time ti such that t ≤ tqs, and there is no update on O done at time tj, tj < tqs. Get the CLASS instance variable of this update. Send the message GET (O, tqs) to this class. In these cases, UPDATES sends a message to only one class. When Time-Specification is a period of time, UPDATES may send messages to different classes. For example, if Time-Specification is a period of time “tqs to tqe”, UPDATES will find an update which is done on object O at time ti such that t ≤ tqs, and there is no update on O done at time tj, tj < tqs. Get the CLASS instance variable of this update. Send the message GET (O, tqs to tqe). Time-Specification may also be a period from “zero” to “now”, or “zero” to “now”.

Next, we give an outline of the second approach to solve the problem of finding the class of a specific version. The approach is as follows.

(1) Associate with each class a set n (n is the number of classes in the path from this class to each stabilization point) of system-defined instance variables called CLASS-HISTORY.

(2) Each CLASS-HISTORY is an ordered pair (Class-Name, Start, End), where Class-Name is the name of a class in n, Start is the time at which a version of an object has been created in this class, and End is the time at which the version in Class-Name is updated to create a version in another class in n.

(3) At the beginning, an object has no values for CLASS-HISTORY instance variables except for the initial class. When an update that causes migration occurs on the object, it modifies CLASS-HISTORY in its initial class. This is done by adding the new class name, and initiating its Start to be the update time. End of the previous class is set to be the update time.

(4) To answer a query Q(O, Tq), the initial class of O is visited and values of CLASS-HISTORY are checked to find the period in which Tq lies. When this period is found, the version’s class of O is identified.

Note that: (1) Due to cycles in the migration graph, some classes in the CLASS-HISTORY instance variables of a class may have more than one START and END pairs. (2) If we use this approach, the initial class of an object should be visited each time the object migrates. This can be done as an action which is automatically carried out as a result of object migration. The migration procedure presented in [EK] can be modified to implement this action.

4-2 Problems Due to the Semantics of the Model

There are two problems due to the rich semantics of the model:

(1) Updating a shared-value instance variable in a class requires creating new versions of all instances of the class. Also, when a default valued instance variable is updated, versions of all instances that consider this default value have to be created.

It is not efficient to create versions of all objects affected by updating the shared-value and default-valued instance variables. Instead, we associate with the class two arrays that store updates on these instance variables and the time of each update. Versions of objects are only created when a query is submitted. For example, to answer the query GET(O, t), the array of the shared-value instance variable is searched to find the version of this instance variable at time point ti. Here, there are two alternatives. Actually create and store a version of the object or only create the version to answer the query and then destroy it. Selecting one of the alternatives depends on the tradeoff between storage space available and time required to generate a version of the object.

A synchronization process is necessary to determine the temporally matching default-valued or shared-value versions. For example, suppose that in class STAFF there is a shared-value instance variable named UNIVERSITY and its value is “Kyushu Institute”. This value is propagated to the instances of all subclasses of STAFF. The UNIVERSITY value is updated twice, at time t1 the value was changed to be “Kyushu Imperial University” and at time t2 it was changed to be “Kyushu University”. A query to get the...
to the user that the object cannot be found. This approach is very much time consuming. If the object has been updated several times, it is necessary to search instances of several classes and in some cases the search is unnecessarily performed. For example, class TA should be searched since it is contiguous to class GRAD. Object O, however, was not at any time an instance of class TA.

We propose two solutions to this problem. In the first solution, a new system-defined class is introduced. This class is used to keep track of updates in the system. In the other solution, each user defined class is extended by adding some system-defined instance variables to the class.

In the first solution, we introduce a new system-defined class called UPDATES. Instances of this class are updates introduced to the system. There is a tradeoff between storing all updates and only storing updates that enforce objects to migrate. In the first case, methods that answer queries on the real-world evolution can be associated with the class UPDATES. Examples of these methods are, find all updates at a specific time period, find all updates done on a certain object at a certain time, and so on. However, the number of instances in this class will be very large. In the second case, the number of instances is drastically reduced, however, we may not be able to easily answer queries that investigate the real-world evolution.

Class UPDATES has the following instance variables: OBJECT, OBJECT-CLASS, TIME, and DEL-INDICATOR. OBJECT is the updated object, OBJECT-CLASS is its class after the update. TIME is the time at which the update has occurred. DEL-INDICATOR is used to mark updates that are deletion of objects. When an update occurs, the object migration mechanism will create an instance of UPDATES. To speed up accessing instances in class UPDATES, all updates over an object may be grouped together and sorted chronologically.

Answering a query will proceed in two steps: first, the class of the queried version is determined by consulting the class UPDATES. Second, when the class has been specified, a query will be invoked to this class on behalf of the user. We assume that a temporal query has the form: GET (O, Time-Specification), where O is the object to be accessed and Time-Specification is either a specific point of time (as zero, now, or t) or a period of time. When the system receives this message, it sends to class UPDATES the message FindClass(O, Time-Specification). When receiving this message,
UPDATES finds the classes to which object O belongs at this Time-Specification. Then, it sends the message GET(O, Time-Specification) to each class Ci of these classes. Now, we give an outline of the procedure used to implement the message FindClass(O, Time-Specification).

1. Check that there is an update over O in UPDATES, if there is no such an update, O belongs to its initial class. Send the message GET(O) to class Ci.
2. Check that object O has not been deleted before time specification T.
3. According to the time specification in the query, class UPDATES determines classes that contain the needed versions of the object O. For example, if Time-Specification is the time point “zero”, UPDATES sends the message GET(O, zero) to the initial class of O. If Time-Specification is the time point “now”, UPDATES finds the most recent update on O, determines the class of O, and sends it the message GET(O, now). If Time-Specification is the point “= tq”, find an update which is done on object O at time t; such that t; ≤ tq, and there is no update on O done at time t; such that t; < t; ≤ tq. Get the CLASS instance variable of this update. Send the message GET(O, t; to tq) to this class. In these cases, UPDATES sends a message to only one class. When Time-Specification is a period of time, UPDATES may send messages to different classes. For example, if Time-Specification is a period of time “tq to tqe”, UPDATES will find an update which is done on object O at time t; such that ti ≤ tq, and there is no update on O done at time tj, ti < tj ≤ t;. Get the CLASS instance variable of this update. Send the message GET(O, t; to t;+1) to this class. In these cases, UPDATES sends a message to only one class. When Time-Specification is a period of time, UPDATES may send messages to different classes. For example, if Time-Specification is a period of time “tq to tqe”, UPDATES will find an update which is done on object O at time t; such that ti ≤ tq, and there is no update on O done at time tj, ti < tj ≤ t;.

Next, we give an outline of the second approach to solve the problem of finding the class of a specific version. The approach is as follows.

1. Associate with each class a set n (n is the number of classes in the path from this class to each stabilization point) of system-defined instance variables called CLASS-HISTORY.
2. Each CLASS-HISTORY is an ordered pair (Class-Name, Start, End), where Class-Name is the name of a class in n, Start is the time at which a version of an object has been created in this class, and End is the time at which the version in Class-Name is updated to create a version in another class in n.
3. At the beginning, an object has no values for CLASS-HISTORY instance variables except for the initial class. When an update that causes migration occurs on the object, it modifies CLASS-HISTORY in its initial class. This is done by adding the new class name, and initiating its Start to be the update time. End of the previous class is set to be the update time.
4. To answer a query Q(O, tq), the initial class of O is visited and values of CLASS-HISTORY are checked to find the period in which Tq lies. When this period is found, the version’s class of O is identified.

4-2 Problems Due to the Semantics of the Model

There are two problems due to the rich semantics of the model:

1. Updating a shared-value instance variable in a class requires creating new versions of all instances of the class. Also, when a default-valued instance variable is updated, versions of all instances that consider this default value have to be created.

It is not efficient to create versions of all objects affected by updating the shared-value and default-valued instance variables. Instead, we associate with the class two arrays that store updates on these instance variables and the time of each update. Versions of objects are only created when a query is submitted. For example, to answer the query GET(O, t;), the array of the shared-value instance variable is searched to find the version of this instance variable at time point t;.

Next, we give an outline of the second approach to solve the problem of finding the class of a specific version. The approach is as follows.

1. Associate with each class a set n (n is the number of classes in the path from this class to each stabilization point) of system-defined instance variables called CLASS-HISTORY.
2. Each CLASS-HISTORY is an ordered pair (Class-Name, Start, End), where Class-Name is the name of a class in n, Start is the time at which a version of an object has been created in this class, and End is the time at which the version in Class-Name is updated to create a version in another class in n.
3. At the beginning, an object has no values for CLASS-HISTORY instance variables except for the initial class. When an update that causes migration occurs on the object, it modifies CLASS-HISTORY in its initial class. This is done by adding the new class name, and initiating its Start to be the update time. End of the previous class is set to be the update time.
4. To answer a query Q(O, tq), the initial class of O is visited and values of CLASS-HISTORY are checked to find the period in which Tq lies. When this period is found, the version’s class of O is identified.

4-2 Problems Due to the Semantics of the Model

There are two problems due to the rich semantics of the model:

1. Updating a shared-value instance variable in a class requires creating new versions of all instances of the class. Also, when a default-valued instance variable is updated, versions of all instances that consider this default value have to be created.

It is not efficient to create versions of all objects affected by updating the shared-value and default-valued instance variables. Instead, we associate with the class two arrays that store updates on these instance variables and the time of each update. Versions of objects are only created when a query is submitted. For example, to answer the query GET(O, t;), the array of the shared-value instance variable is searched to find the version of this instance variable at time point t;.

Next, we give an outline of the second approach to solve the problem of finding the class of a specific version. The approach is as follows.

1. Associate with each class a set n (n is the number of classes in the path from this class to each stabilization point) of system-defined instance variables called CLASS-HISTORY.
2. Each CLASS-HISTORY is an ordered pair (Class-Name, Start, End), where Class-Name is the name of a class in n, Start is the time at which a version of an object has been created in this class, and End is the time at which the version in Class-Name is updated to create a version in another class in n.
3. At the beginning, an object has no values for CLASS-HISTORY instance variables except for the initial class. When an update that causes migration occurs on the object, it modifies CLASS-HISTORY in its initial class. This is done by adding the new class name, and initiating its Start to be the update time. End of the previous class is set to be the update time.
4. To answer a query Q(O, tq), the initial class of O is visited and values of CLASS-HISTORY are checked to find the period in which Tq lies. When this period is found, the version’s class of O is identified.

4-2 Problems Due to the Semantics of the Model

There are two problems due to the rich semantics of the model:

1. Updating a shared-value instance variable in a class requires creating new versions of all instances of the class. Also, when a default-valued instance variable is updated, versions of all instances that consider this default value have to be created.

It is not efficient to create versions of all objects affected by updating the shared-value and default-valued instance variables. Instead, we associate with the class two arrays that store updates on these instance variables and the time of each update. Versions of objects are only created when a query is submitted. For example, to answer the query GET(O, t;), the array of the shared-value instance variable is searched to find the version of this instance variable at time point t;.
UNIVERSITY value of Prof. Yamada at time \( t_i \) is submitted. The answer is given by synchronizing \( t_i \) with one of the periods \( 0-t_l, t_l-t_2, \) and \( t_2-\text{now} \) to determine the appropriate value of UNIVERSITY.

(2) An update on an object \( O' \) may create two versions, one of \( O' \), and the other of an object \( O \) that has \( O' \) as one of its (complex) instance variables. The graph representing complex objects rooted at a certain object may be big. For example, in Fig. 4, creating a version of object \( O_4 \), requires creating versions of objects \( O_3, O_2, \) and \( O_1 \). The approach of immediately creating a version of a complex object as soon as one of the objects it refers is updated, may make the number of versions very big. Instead of this approach, we follow the approach of not to create versions until the object is required. Note that, if updating one the complex object instance variables (which is an object) will enforce the object to migrate, we need to create a new version.

Suppose there is a query that accesses a version of object \( O \) at time \( t_q \). \( O \) has object \( O' \) as one of its instance variables. This query is split into two queries. One is accessing the version of object \( O \) at time \( t_q \) and the other is accessing the version of object \( O' \) at the same time. If the query is accessing versions of \( O \) at some period of time \( t_{q1} \) to \( t_{q2} \) or the whole history of \( O \), time synchronization of versions of \( O \) with versions of \( O' \) is necessary. Note that, if \( O' \) has some complex instance variable the query has to be split into three queries. In general, a query on an object \( O \) (the query is accessing the whole object) should be split into \( n \) queries, where \( n \) is the number of nodes in the complex object graph rooted at \( O \).

4-3 Handling Schema Evolution

Advanced applications require modifying the schema without full interruption of system operations. [BKKK] presents the approach taken in ORION to support schema modifications. In this section, we study how to handle schema changes in temporal object-oriented databases. We follow the approach taken in [BKKK], we list schema changes and see how to handle them in temporal object-oriented databases. We are not considering schema versioning as in [KC]. In CAD applications, the schema may frequently modified in order to experiment several design alternatives. We assume that the scheme is rather stable and schema changes are not frequent. When the schema is updated, we create a new schema version and also a new schema instance. That is, we create a new version for each object affected by the schema change. For example, when, at time \( t_i \), a new instance variable is added to the definition of class \( C_i \) the system has to create a new version of the class along with new versions of all its instances. All queries that access objects in \( C_i \) before time \( t_i \) will not see the value of the new added instance variable. Following this approach, i.e. creating for each schema update versions of affected objects, may make the database size grow unmanageably. Schema changes can be classified into two groups. In the first group, we have to create a version of the schema along with versions of all objects affected by the schema change. In the other group, the system has only to store the kind and time of the schema change in order to decide which versions will be accessed as a query answer. That is, we do not immediately create versions of objects affected by the change, instead at the time of answering a query.

The first group of changes consists of:
1. Add a new instance variable \( V \) to class \( C \),
2. Drop an instance variable \( V \) from class \( C \),
3. Make a class \( S \) a superclass of a class \( C \),
4. Remove a class \( S \) from the superclass list of a class \( C \), and
5. Make a shared variable \( V \) of class \( C \) unshared.

These changes are handled by creating a new version of the affected class and versions of all its objects. These changes are handled by creating a new version of the affected class and versions of all its objects. Instances of the new version of the class are created by creating versions from the most recent versions of the objects in the old class. The time of a schema change should be stored in the database catalog.

The second group of changes consists of:
6. Change the name of an instance variable \( V \) of a class \( C \),
7. Change the name of a method of a class \( C \),
8. Add a new method to a class \( C \), and
9. Change the name of a
class C, and (10) Change the code of a method in class C.

All these changes can be handled by extending the database catalog. The catalog consists of three schema classes:

Classes (ID, Name, InstanceVariables, SuperClasses, Methods), instance variables InstanceVariables, SuperClasses, SubClasses, and Methods are complex instance variables. We add the time related instance variables FROM and TO. A version of a class is created when an instance variable, a superclass, subclass, or a method is added or dropped from its definition.

InstanceVariables (ID, Name, Class, Domain), and Methods (ID, Name, Class, Code).

For these two classes, we also add the instance variables FROM and TO. A version is created for an instance of class InstanceVariables, when its Class or Domain values are updated. Similarly, class Methods is handled.

For example, suppose we have a method CalculateSalary in class Employee, and at Aug. 1989, the code implementing this method has changed. Instead of creating a new version of class Employee along with versions of all its instances, the object that corresponds to this method in class Methods is modified. A new code value of method CalculateSalary will be added with FROM Aug. 1989 and the TO time value of the old code will be set to July 1989.

4-4 History Update in TOODB's

Databases that support the time dimension are classified into three types rollback, historical, and temporal [SA]. One of the features of historical and temporal databases is the possibility of updating the history as known of now. In this section, we study history update in TOODB's. Modifying the history in TOODB's may have some side effects. To show that consider the schema shown in Fig. 5. At t1 update U1 is done on object O to increase Total-Markes of O to 87. Accordingly, O will migrate to class GOOD-GRAD. At t2, O is updated again to move into class TA. Later, at t3, it is discovered that the Total-Markes of O should be corrected to be 78. If this history update is done, O will not satisfy conditions to be an instance of class TA. The system should consult the user before correcting the history.

There are two possibilities when the history is updated:

1. The correction of an update done at ti does not affect any update done at tj, tj > ti.

In this case, we apply the correction. It happens when:

(a) The corrected instance variable is not one of those causing objects in the class to migrate, for example correcting an update that modified the address of object O when it was an instance of class GRAD will not affect its position in class TA.

(b) The corrected instance variable is one of those causing objects in the class to migrate, however the correction will not affect the object's current position or any of its historical versions. For example, in Fig. 5, correcting the Total-Markes from 87 to 78 will not invalidate the existence of O in class TA.

The corrected update may affect other updates in the following cases:

(a) The correction will make one of the migration conditions satisfiable. For example, an object in class C may migrate into class C' iff: A ≥ 50, B < 80, and C = 20, where A, B, and C are instance variables. Suppose at time t1 A is updated to be 45, at time t2 B is updated to 70 and C to 20. The update at time t2 does not cause any migration. Later, at time t3, it is discovered that the first update was not correct, A should be 54 instead of 45. If this mistake is corrected, the update at time t2 should be completed, since, now, all conditions of migration are satisfied.

(b) The corrected instance variable does not cause any migration to objects in the class, however, it may cause migration of some versions in some other class. For example, consider an object O* that has two versions one in class C and the other in class C'. The version in class C was generated from the version in class C. The condition for O* to migrate from C to some other class C3 is that A > 45. Assume that a history update is applied on the version in class C to make its A equal to 60. Applying this history update will enforce the object to migrate from class C to class C3.

The system has to detect whether correcting an update has side effects. When the correction does not have any effect, it can be executed. Otherwise, the user has to be informed and the decision of performing the correction should be taken by the user. The system may support the user with information that help him to take the decision. These information may include the instance variables that causes the side effects, the new position of the object when the history is corrected, and etc.

Before we give an outline of the procedure that checks history corrections, we need the following definitions.
Definition 3. Associated with each class $C_i$ a set of migration conditions denoted by $MC(C_i)$. A migration condition has the form:
Instance variable $OP$ Value, where $OP$ belongs to $\{=, \neq, >, \geq, <, \leq\}$ and Value is obtained from the domain of the instance variable.

Definition 4. The class of the most recent version of an object $O$ is called the current class of $O$ and denoted $CC(O)$. The class of the version that will be updated is called the history update class and denoted $HUC(O)$. The history path of object $O$, denoted $HP(O)$, is the union of $HUC(O)$, $CC(O)$, and each class $C_j$, in the path between $HUC(O)$ and $CC(O)$, such that $O$ has a version belongs to $C_j$.

Note that the history update class (HUC) and the current class (CC) of an object can be obtained by consulting class UPDATES.

Procedure History-Update-Validation
Input: A correction $U(O, IV_i, NEW-VALUE)$
OUTPUT: The correction is done, or, the user is informed of the side effects through a fiction modification.
METHOD: for every class $C_k$ in $HP(O)$ do
if correcting the history makes one of $MC(C_k)$ true then inform the user; execute a fiction history update; exit;
end do;
perform the correction;
exit;
END HISTORY UPDATE

5- Conclusion
In this paper, we gave some basic problems in querying temporal object oriented databases. Our study is in its first steps. The notion of object migration is introduced and its effects on answering temporal queries were shown. We need to consider some other aspects, like using indexes and removing the assumption that an object is an instance of only one class.

Acknowledgments
The author is grateful to his supervisor Prof. Y. Kambayashi for valuable discussions and comments on this paper.
References


