A Survey of Schema Evolution in Object-Oriented Databases

Xue Li
School of Information Systems, Technology and Management
The University of New South Wales
Sydney 2052, Australia
xueli@unsw.edu.au

Abstract

Changes in the real world may require both the database population and the database schema to evolve. Particularly, this is the case in CAD/CAM and CASE database systems, in which the design objects constantly evolve in every aspect. On the other hand, the prototyping of a database design may also involve changes to both the structure and behavior of a schema. Unfortunately, most of the current systems offer little support for schema evolution. In this paper, we survey the recent development of the research in the schema evolution of object-oriented databases. The main issues in the current research are identified.

1. Introduction

Future database systems should be able to deal with changes in the real world. Object-oriented database systems are regarded as promising future database systems to support high-level data abstractions. Changes in the real world can be reflected as the evolution of object-oriented databases. Database evolution is about how both schema and data can be changed to capture the nature of the changes in the real world. It mainly concerns two issues: schema evolution and instance evolution.

1.1. Complexity of schema evolution

An object-oriented database schema (hereafter called a schema) is a conceptual description of the real world objects. The real world objects may not physically exist. They may only exist in the designer's mind. The complexity of schema evolution arises because, for an already-existing database, changes in the real world may require both the database population and the database schema to evolve. Particularly, this is the case in CAD/CAM (Computer Aided Design/Computer Aided Manufacturing) and CASE (Computer Aided Software Engineering) database systems, in which the design objects constantly evolve in every aspect. On the other hand, the prototyping of a database design may also involve changes to both the structures and behavior of a schema. Unfortunately, most of the current systems offer little support for schema evolution [15].

There have been a few review and survey articles about schema evolution of databases in general [11, 19, 21, 22]. In addition to a number of research papers, there are also some doctoral dissertations dedicated to various issues related to schema evolution [5, 6, 12, 16, 18, 20, 15]. In this paper, we summarize the issues and categorize the contributions towards the schema evolution in object-oriented databases.

Intuitively, a schema can be regarded as a set of classes with a set of relationships existing between classes. Every class is regarded as a container of object instances that have
the same properties that group them together. Object instances are the images of real world objects. Properties of a class are the attributes (also called instance variables) and methods. Attributes are used to describe the data types of object instances and methods are used to describe the behavior of object instances including arithmetic, search, sort, query, and update operations. Relationships between classes indicate the connections between object instances or the configurations of complex objects. The industry consensus on object-oriented databases has resulted in an object database standard, namely ODMG 2.0 [7].

A schema in an object-oriented database system plays a role to hold the information about structure and behavior of object instances stored in the database. The manipulation of object instances is performed as transactions via a database management system (DBMS) which is also responsible for the enforcement of database integrity and other operational duties.

A schema must be consistent to all database states that are snapshots after database transactions. The consistency is an invariant, inherent in a schema and expressed as a set of integrity constraints to propagate on all database states. However, if we examine the relationship between a schema and the real world, we can see another kind of consistency: that is, the semantics of a schema must capture the semantics of the real world. Therefore, there is a dual consistency: all database states should represent the real world states, and all the real world states must be represented by the database states. Then, the problem is that the already-designed schema may not be able to (or may not be suitable to) describe the changes in the real world and is required to mutate.

The schema evolution is a problem about how a schema can be modified to catch up the changes in the real world. Consequently, the impact of the schema-changes on the object instances of the database as well as on the pre-developed database applications should be considered.

For a particular schema, the main issues of schema evolution may be expressed as the following questions:

1. **Schema representation**: Which aspects of a schema are to be changed?
2. **Schema-change operations**: How is a schema changed?
3. **Schema-change consistency**: How the changes to a schema are qualified and/or propagated?
4. **Database integrity**: How the instances of a database are affected in regarding the schema-changes?
5. **Database availability**: After schema-changes, to what extent should the database be available to its applications?

An evolution-handling mechanism built in a database system should be responsible for handling all of the above questions. The research work in this context is to provide answers to the above questions.

Due to schema-evolution operations, the existing database may expect to have some problems. Firstly, some object instances may no longer be legally retrieved. This problem can be caused by the deletion or update of data structures. Other problems such as some object instances may not be referentially consistent to each other, or the semantics of data may be changed unintentionally. On the other hand, the behavior of object instances may be changed
unintentionally due to the change of either schema structures or methods that are defined to operate on object instances.

2. Requirements in schema evolution

Regarding the questions and problems presented above, issues in the schema-evolution research can be identified as requirements for database semantic integrity, schema evolvability, and application compatibility.

2.1. Semantic integrity

A schema-design methodology is responsible for specifying the object semantic integrity. That is a set of constraint rules for maintaining the consistency between states of a database and states of the real world objects. When a schema is not changed, the semantic integrity is not a problem because the database transaction management is present to enforce the semantic integrity for all objects in a database. When a schema evolves, two requirements should be satisfied with respect to object semantic integrity: referential integrity and constraint consistency.

Referential integrity is a property that holds the parts of an object together to form a complex object. On the other hand, it also constitutionalizes the existence of an object within an object community (i.e., in a database). For example, the relationships of 1:1, 1:m, m:1, m:n between object instances can be specified for the quantification of constituent objects in a complex-object definition. The change to schema structures may invalidate these semantic-integrity constraints. Another case is that a classical multi-inheritance conflict may result from the modification of the inheritance lattice. For example, class Car, as a sub-class of Vehicle may become a sub-class of class Product. In this case, attribute Produced_date may be an attribute in both super-classes. Thus there would be a name conflict in which a datum is duplicated in the both super-classes. This problem is called a syntactic conflict that can be easily solved by a naming convention to select one of them to be inherited. Another example demonstrates semantic conflict. An attribute Registration_date may also be in both super-classes. However the Registration_date in class Product may mean the date when a product is produced, while the Registration_date in class Vehicle may refer the date when a vehicle may be legally used. This problem can be solved if we make an assumption that every attribute name is universally unique, or use a relation operator to prefix the class name. However the syntactic conflict is different from the semantic conflict in that the former is to preserve only one of the conflicting attributes while the latter is to make a distinction among each of them. Therefore whenever a multi-inheritance conflict is detected, we should identify whether it is a syntactic or a semantic conflict before we can resolve it.¹

Constraint consistency is a problem of any semantic-integrity constraints being consistent with each other. For example, if the range of employees’ salary is changed from 300-700 to 100-500, then the range checking for a mechanism to provide transparent access for both the old and new database applications should not become 100-700, because it is not right for either version of ranges.

¹ Perhaps, it would be a good practice not to support multi-inheritance at all - as the case in JAVA.
2.2. Schema evolvability

To answer the questions raised in the beginning of this paper, the evolution of a schema is related to the following requirements:

- an algebra of making changes,
- a data structure of recording changes,
- the checking of schema consistency,
- the propagation of changes to instances, and
- the reflective configuration of database states for the existing database applications.

To fulfill these requirements we need to develop a methodology for building an evolution-handling mechanism into a database system to cope with the real world changes.

In the process of schema evolution, an evolving schema can change in two different directions: the structural evolution and the behavioral evolution. In these two directions, we identify the following problems to be dealt with.

**Structural evolution:** This deals with the modification of the structural part of a schema as well as the enforcement of the consistency of a schema after modification. Changes performed on a schema may be treated in two different ways: versioned or non-versioned. The former results in versions of a schema in order to keep the history of schema changes. The latter creates an up-to-date schema. The consistency of structural evolution is considered as schema-level consistency and instance-level consistency. **Schema-level consistency** is the problem of maintaining the quality of a schema. In either case of versioning or non-versioning, a schema should be consistent in the sense of a well-formed schema (i.e. a schema with some desired properties). **Instance-level consistency** is the problem of propagating schema-changes onto object instances. Depending on versioned or non-versioned approaches, strategies must be given to provide values for the modified schema. For non-versioned changes, a screening (or filtering) [3, 4] or a conversion approach should be defined. For the versioned approach, a variety of approaches [2, 11, 17, 23] are given to decide the values for different versions of a schema.

**Behavioral evolution:** This deals with the modification of methods of a schema. The methods of a schema represent behavior of a schema specified for the manipulation of object instances. Changes of behaviours are performed as operations that delete, insert, and modify methods of a schema. After the method-changing operations, a number of problems can be observed. **Run-time type errors** are problems with method executability that a method is reported to have type-mismatch errors during late binding. This kind of error is typically caused by method migration (i.e., a method is reused by its super/sub-classes) or method versioning. **Side effects** are problems of method functionality that cause “untraceable” changes to a database [27]. **Redundant/incomplete/meaningless methods** are the problems of method validation that can be avoided by a well-defined design methodology. **Unwanted behaviors** are also problems of method functionality that return unspecified results. A detailed explanation on the above problems can be found in [14, 15, 24, 25, 26].

2.3. Application compatibility

A database system should have a policy to maintain the quality of a schema against any changes in order to provide services to database applications. This can be regarded as a
problem of schema-change transparency. The quality of a schema is referred to as its integrity and consistency. However, the concept of “schema-change transparency” needs some further attention and clarification. A fundamental question is that: if a schema changes, do we actually want old database applications still running as if nothing had happened? As we know, the changes of schema can be classified as changes either on the implementation or on the interfaces of a schema. The former means that the old database applications should still be running. In this case, the “schema-change transparency” is granted, and is an asset of a database system. The latter would expect database applications possibly to run into trouble. By looking into the transparency problem, we see two issues: Downward compatibility and Upward compatibility.

Downward compatibility is to guarantee that all old database applications developed prior to the schema-change will have no problems continuously functioning. There are two ways for a system to satisfy the downward compatibility depending on whether the changes should be reflective or non-reflective to database applications. Reflective database changes give up-to-date services to database applications while non-reflective database changes give as-if-before services to database applications. Therefore, the system may have to make a decision on whether to keep the history of changes or make some compatibility adjustment. The former is about database versioning and the latter is about irreversible database mutation.

Upward compatibility is to let database applications developed after the schema-change be still able to run without problems. There are also two issues to be considered: Reflective database changes should allow the database applications developed after the change to evolve accordingly while non-reflective database changes should allow the database applications developed after the change to transparently access the database as if there were no changes. The former requires the system to keep records of changes for the upwardly developed database applications. The latter is to arrange a by-pass for the upwardly developed database applications to overlook the changes.

There are some researches contributed to the issues of application compatibility, mainly related to object-oriented programming. In the database area, application compatibility is a problem about database availability. Depending on whether the versioned or non-versioned approach is used for the evolution handling, the relevant work can be found in [17].

In overview, schema evolvability is a core of the evolution of database systems and much research work still remains [13, 21, 22, 24, 25, 26].

3. A two-dimension evolution paradigm

In this section, we firstly give taxonomy of database-system evolution. A two-dimension evolution paradigm of schema evolution is identified. We regard the major issues of schema evolution being structural evolution and behavioral evolution.

Schema evolution is situated in a more global research area that is database-system evolution. There is a two-dimension evolution paradigm according to how and what a database system is changed. Figure 1, which is modified from [10], illustrates this two-dimensional evolution paradigm.

At “what-dimension”, there is a division of levels based on the abstraction of a database system. There are three levels of database-system abstraction: data level, schema level, and...
data model level. At “how-dimension”, all levels of abstraction are divided into two levels of transformation of evolution: the mutation management and version management. Mutation denotes the non-versioned process in a way that original entities become inaccessible after transformation. In contrast, version management makes the history of changes available for retrospective access [10]. A version is a semantically meaningful snapshot of a design object at a point in time [11].

![Figure 1. A two-dimensional database evolution](image)

According to these two dimensions of classification, we identify six aspects in the database-system evolution paradigm: instance mutation, instance versioning, schema mutation, schema versioning, data model mutation, and data model versioning. In following discussions, we explain these six aspects.

### 3.1. Data-level evolution

At data level, instances mutation and instance versioning are considered. The issues include the mutation of object instances, the management of history, and the configuration of consistent database states.

Instance mutation is also called instance evolution [15] that concerns the problems of structural and/or behavioral modifications on the instances with regard to:

- **Instance migration.** The migration of object instances consists of the generalization and specialization of object instances. In the case of instance generalization, the instances of a sub-class become the instances of a super-class. The problem of consistency is about whether to drop the sub-class and to coerce the values of the generalized instances into the super-class. In the case of instance specialization, the instances of a super-class become the instances of a sub-class. The consistency in this case is about whether to drop the super-class and to deal with missing values for the specialized instances. In either case, a database transaction is allowed to work with migrated instances, and object identifiers (OIDs) are preserved. That means, the instance migration should be transparent to database applications.

- **Dynamic complex objects.** In contrast to instance migration, dynamic complex objects are involved in those OID-related database operations. There are three kinds of OID-related database operations: the OID-preserved, OID-deleted, and OID-created operations. For OID-preserved operations, object instances are retrieved and their values may be modified. For OID-deleted operations, object instances are ceased to exist in the database. For OID-created operations, object instances come to exist. Traditional
Database systems offer this kind of instance mutations. The database is transformed from one consistent state to another by a sequence of read/write operations arranged within transaction [10].

**Instance versioning** is based on the assumption that instances of a database would never be deleted. There are many approaches for the instance versioning.

- **Temporal database** uses some time-stamping mechanism for the record of database changes. However, in certain application domains such as CAD (Computer Aided Design) and CASE (Computer Aided Software Engineering), the history recorded on logical dependency instead of time dependency may be more appropriate for the applications.

- **Object versions** [4] allows for an application-driven recording of versions of a design object. *Configuration* of versioned object instances is a process to cohere the object versions together to reflect a consistent real world state [1]. Regarding the strategy and the granularity considered in the configuration, there are different approaches proposed. In [4], object versions are managed in terms of packets (classes) which group objects instances that have same slices (properties). When a packet is modified, only those packet slices, which have been modified, will be created as a new version. An application is responsible for the configuration of object instances using a set of system-provided operators. In [1], configuration is “user-specified” and is tied to a transaction model. For given $m$ object instances, each with $v$ version, there can be up to $v^m$ possible configurations. The correctness of the configuration is defined on the serialization of the transactions. The serialization of the transactions is the order in that the object versions are created or derived. In other words, the correctness of a configuration is determined by the order of versions that in turn are “syntactically” determined by the order of transactions that create or derive object versions. In [8] a “database-version approach” is proposed. Unlike other approaches that establish explicit links between consistent object versions, the database-version approach actually takes one whole configuration as a database version and links all database versions together as a derivation tree. A state of a database version is defined as the set of values of all the object versions that it contains. The consistency in the database-version approach is maintained by the transactions that create or cancel the database versions. Once created, a new database version can evolve autonomously, according to the non-versioning transactions. So, there are two levels of operations: at the upper level the user creates or cancels a specified database version; at lower level the user retrieves or updates a specified database version.

### 3.2. Schema-level evolution

The schema level evolution can be classified as the schema mutation and schema versioning. The issues include the changes of a schema in both structural and behavioral aspects, the management of the history of changes, the policy of maintaining the instance consistency, and the strategies for the application compatibility.

**Schema mutation** is regarded as a non-versioned approach in that a schema is changed irreversibly. Thereby the database population as well as the application programs of database will be affected. Banerjee and Kim early addressed the semantics of this kind for system ORION in [3]. One of the important characteristics of this approach is that the evolution is under the control of a set of properties called “invariants”. The invariants described in their paper are actually those rules to construct a schema. Invariants such as class lattice invariant, distinct name invariant, and domain compatibility invariant are generic constraints to all
object-oriented database schemas. The impact of schema changes on existing instances are absorbed by introducing default values or screened off values from the memory. Therefore it can be seen that for any schema changes, only one of these three ways is adopted to handle the impact on instances: default values for newly required instances, making inaccessible to those not supposedly existing instances, or keeping original values for consistent changes.

Most current systems support only very limited schema mutation. For examples, GemStone supports modification of leaf classes (classes with no sub-classes) with transparent, lazy, instance migration. O2 supports some basic schema updates but users must manage to convert their instances by hand, using their O2 programs written themselves, and with the aid of the tools that is a set of pre-defined classes (i.e., a meta-schema) plus a save/restore program to load or download database instances. Some research projects for O2 schema evolution can be found in [9, 27]. Ontos provides class modifications in a way that instances do not migrate i.e., there is no support to adopt instances to the new class definition. So schema changes can be done only on classes that do not contain instances and do not have sub-classes that contain instances.

The disadvantages of non-versioned approaches are that:

- Schema changes should be propagated on to database instances immediately after the schema-changing operations. Otherwise, the database will not be available to any application programs in regarding to the schema updates.
- Propagation of changes may be performed in two different ways: screening (or filtering) [3, 4] or conversion of instances. Either of these two ways may result in problems of compatibility with application programs developed prior to the changes.
- The conversion of instances may outage applications.

To deal with these problems, many approaches have been proposed. Roddick gives more information in [21].

Schema versioning is an approach that introduces a version-control mechanism to handle various versions of classes [22]. Commonly, three orthogonal data structures were identified as necessary for schema versioning: (a) the structure for version histories, (b) the structure for version configurations, and (c) the structure for version equivalence. The composition of complex objects is based on using these three orthogonal structures through a layer/context mechanism that provides a way of version ordering.

All approaches surveyed in [11] contribute to these structures to some extent. Monk and Sommerville, in [17], have provided a kind of type change transparency for the application programs. Their approach records versions of classes and provides a conversion mechanism (e.g., update and backdate procedures) that allows a partial mapping between old and current versions of a class. Their approach provides a polymorphic mechanism that a same property in different versions of a class may have different semantics.

3.3. Data-model-level evolution

The concept of data model involves a set of concepts for database schemas and a methodology to produce such schemas. There are many different types of data models such as relational data model, object-oriented data model, and deductive data model.
Data model evolution is a problem about how to migrate a data model from one kind of paradigm to another without loss of semantics or with enriched semantics. To this end, the evolution is “syntactic” – a transformation algorithm can be defined to carry out the migration of data models [10].

However, the interests of data model evolution are beyond that: the data model evolution is to catch up the changes of the real world by changing the abstract representations of a database system. In other words, the evolution has reached a new stage that the database system has to change its configurations of underlying meta structures to be able to satisfy the real world changes. In this context, we identify the issues of data model evolution towards temporal, active, reflection, and federated databases. The main question to be asked is that: “How can an existing database be evolved to one of these kinds of database systems?” The research area on this topic is called database re-engineering and the detailed discussion on this topic is out of the scope of this paper.

4. Conclusions

In recent years, with the development of database technology and the rapid growth of a large number of object-oriented databases in action, schema evolution for object-oriented databases has emerged as a serious research area. With the reference to the standard ODMG 2.0 [7], in this paper we have identified a number of important research issues in the schema evolution. The fundamental point is that the constant changes of real world will involve the evolution of databases in all aspects. With the emphases on the semantic integrity, schema evolvability, and application compatibility, we have addressed the current research directions. Other issues related to the management of schema evolution, such as database transparency, database reflectiveness, and evolution controllability have not been mentioned and more details can be found in [15].

5. References


