Evolution Features of the F2 OODBMS

Lina Al-Jadir Thibault Estier Gilles Falquet Michel Léonard

Centre Universitaire d’Informatique, Université de Genève,
24 rue Général-Dufour, CH-1211 Genève 4, Switzerland
{aljadir, estier, falquet, leonard}@cui.unige.ch

Abstract

Features specially devoted to schema evolution in the F2 object-oriented database system are presented in this paper. They include a meta-circular and reflective model where primitive operations (create, update, delete) on schema objects form a complete taxonomy of schema changes. All the schema objects of a database are updatable and the consequences of these changes on objects are handled either by the trigger mechanism or by the automatic enforcement of the model invariants. The non-traditional physical storage of objects is particularly well suited to efficiently support schema changes.

1 Introduction

It has long been recognized that database management systems (DBMSs) should provide evolution features to accommodate data structures to an ever changing application environment. For instance, changes in system's requirement may occur during the design or implementation phase, unexpected changes in the system's environment may occur when the system is already in operation, etc. A designer needs then at different steps of the application lifecycle, especially at the production phase (i.e. when data is stored), to: (i) make the database structure (schema) *evolve* when the application domain changes, (ii) *complete* and *refine* the schema when it is incomplete, (iii) *correct* the schema when it has mistakes, (iv) *reuse* the schema when it has been used for a prototype.

Early relational DBMSs already provided basic evolution operations such as creating or deleting tables and adding table columns. With the advent of new data models comprising concepts like generalization/specialization, instantiation, referential integrity, or objects behaviour, the schema evolution problem becomes more complex. In particular, the following questions arise:

- which schema changes should the system provide?
- how to define their semantics?
- how to implement them efficiently?

The starting point of the “Rebirth/F2” project was to consider the evolution of schema and objects as an *inherent* and fundamental property of a system rather than a “later-added” functionality. Our experiences in semantic data models [18] and information systems design [17][16] have clearly shown that evolution could not be well integrated into a DBMS by simply adding a set of features on top of it. Quoting [23] about extensibility, we state that evolution “cannot be retrofitted; it must be a fundamental goal and permeate every aspect of the system’s design”. We adopted this idea as a major principle to motivate our design decisions at every level, including the data model, the object manipulation operations and the object storage management. This led us to develop the F2 database system prototype [14] [15] [2].

The F2 model is object-oriented and supports multiple instantiation, automated object classification and integrity constraints (existential dependencies and keys). Objects behaviour is defined by primitive methods (create, update, delete) and triggered methods. Meta-circularity and reflectivity are important features of the model. In F2 there is only one kind of object: a class is an object and a meta-class is a class whose instances are classes [12]. Thus there is no conceptual or technical distinction between classes and meta-classes. Consequently, the same primitive methods are used to handle database objects as well as schema objects and meta-schema objects. This makes F2 an uniform, flexible and easily extensible system.

Schema evolution is inherent to the F2 system and is accomplished by applying primitive methods on schema objects. The trigger mechanism allows to alter the standard behaviour of the primitive methods in order to define a specific semantics for some schema changes. In these cases triggers execute methods which check conditions and perform repercussions on schema and database objects to keep the database in a consistent state.

This research was partially supported by the Swiss National Fund for Scientific Research (FNRS) under grant no. 1603-0.87 and no. 20-34056.92.

Proceedings of the Fourth International Conference on
Database Systems for Advanced Applications (DASFAA’95)
Ed. Tok Wang Ling and Yoshifumi Masunaga
Singapore, April 10-13, 1995

To support schema changes with as few storage reorganization as possible we choose to implement a transposed storage structure in F2. This physical structure possesses interesting properties with respect to evolution, for instance adding or deleting an attribute is performed in constant time whatever the size of the database.

The paper is organized as follows. Section 2 presents the structural and behavioural aspects of the F2 data model. Section 3 describes the F2 kernel and its manipulation. Section 4 explains how schema evolution is supported in F2 thanks to the kernel, the primitive methods and the triggers. Section 5 discusses the physical storage used in F2. Section 6 presents briefly other DBMSs supporting schema evolution. Section 7 summarizes the paper and gives some future directions.

2 F2 model

The F2 model is based on the following concepts: object, class, attribute and specialization. It supports multiple instantiation and automated classification of objects. It includes integrity constraints like existential dependencies and keys. Objects behaviour is defined by primitive methods and triggered methods.

2.1 Structural part of the model

Object, class, attribute

Each *object* is an instance of a class. An object has a system-defined object identifier *oid* and a *value*. The *extent* of a class is the set of all its instances. There are two kinds of classes: atomic and tuple classes. *Atomic classes*, like Integer, Real, String, etc..., contain atomic objects which are self-identifying, i.e. their oid is equal to their value. They cannot be created, updated or deleted (these are called “abstract objects” in [1]). An interval is associated with an atomic class to constrain its object values. *Tuple classes* have *attributes* that correspond to functions defined between an *origin class* and a *domain class*. The value of an attribute for an object is either the *unknown value* or a set of objects belonging to the attribute’s domain class (the unknown value is different from the empty set). The size of this set is constrained by the *minimal* and *maximal cardinality* of the attribute.

Example 1. The following F2 DDL [15] expressions define an atomic class Boolean whose integer values belong to the interval [0,1] and a tuple class Student with several attributes.

```
class Boolean: intBase (0..1);
class Student
     (reg_number: Integer,
     lastname: String,
     firstname: String * 1..4
           /* min. card.= 1, max. card.= 4 */,
     nationality: String,
     scholarship: Boolean,
     in_dept: Department)
  key (reg_number); /* key explained later */
```

The value of an object *o* in a tuple class *C* is a tuple formed by the oids of objects linked to *o* through the attributes of *C*. For instance, an object *s* of class Student may have the following value:

```
[reg_number: 98765, lastname: “Dunand”, first-
name: {“Henri”, “Marc”}, nationality: “Swiss”,
scholarship: 1, in_dept: d]
```

where *d* is the oid of a Department instance.

Specialization

The *generalization/specialization* mechanism allows a class, called *subclass*, to be defined as a specialization of some other existing class, called *superclass*. The class hierarchy is a forest, i.e. a collection of specialization trees. The semantics of specialization in F2 corresponds to *specialization inheritance* plus *constraint inheritance* as defined in [6]. A subclass inherits attributes from its superclass and may have additional attributes. *Specialization constraints* may be defined on a subclass as a set of triples <attribute, operator, value>. The extent of a subclass is composed of all the instances of its superclass which satisfy all its specialization constraints.

Multiple instantiation

Subclasses are not exclusive, i.e. an object can belong to several subclasses in the same specialization tree, this is often referred to as *multiple instantiation*.

Example 2. An instance of class Student may simultaneously belong to both subclasses SwissStudent and ScholarStudent defined hereafter. This would mean that this student is swiss *and* he has a scholarship.

```
subclass SwissStudent of Student
when nationality = “Swiss”
  (canton: String);

subclass ScholarStudent of Student
when scholarship = 1
  (organization: String,
  amount: Integer);
```

Exclusive subclasses may of course be specified by defining exclusive specialization constraints on them. Multiple instantiation is an elegant way to solve the problem of combinatorial explosion of sparsely populated classes caused by multiple inheritance. Without multiple instantiation, one must define in the last example, a subclass Swiss&Scholar that inherits from both SwissStudent and ScholarStudent. This “intersection class” adds no new state and may rise name conflicts (same name of attributes in two superclasses). Multiple instantiation is proposed in [26] [31] [30] but is not supported by current commercial OODBMSs.

Automated classification

In order to support the constraint inheritance semantics of specialization, F2 possesses an *automated classification* mechanism. When an object *o* is created in a class *C*, this mechanism finds in the specialization tree of *C* all the classes to which *o* may belong, according to its attribute values, and adds *o* to these classes. Similarly, when some attribute value of *o* is updated, *o* must be re-classified and, as a result, it may migrate between classes in the same specialization tree.

It is interesting to note that automated classification is a way to implement "selection views" as defined in [32]. Automated classification is usually not supported by commercial OODBMSs.

Existential dependency enforcement

Each attribute induces an existential dependency. The dependencies are enforced according to the minimal cardinality of the attribute. Let *att* be an attribute of class *C* with domain class *D* and minimal cardinality *min*. Let *o* be an instance of *C* with $att(o) = \{d_1, d_2, ..., d_s\}$. If one deletes d_1 from *D*, d_1 is removed from $att(o)$ in order to maintain referential integrity. If it lets the cardinality of $att(o)$ below *min*, *o* is automatically deleted from *C*.

An existential dependency can be assigned to any attribute whose domain class is tuple (by setting *min* greater than zero). It is not related to a specific relationship between classes, as in [7] [20] where dependency can be expressed only on a composite attribute (is-part-of relationship).

Keys

A *key* of a tuple class *C* defines a constraint on *C* instances. It is a subset of the attributes of *C* such that no two instances of *C* have the same values on this attributes set. A class may have zero, one, or several keys. A subclass inherits the keys of its superclass and may have additional keys.

Keys provide a useful mechanism to refer to objects in the F2 manipulation language. If $\{att_1, att_2, .., att_k\}$ is a key of class *C*, an expression of the form $C'[att_1\colon v_1, att_2\colon v_2, ..., att_k\colon v_k]$ designates the unique instance *o* of *C* such that $att_1(o) = v_1$, ..., and $att_k(o) = v_k$. If $k = 1$ one can use the simpler form $C'v_1$. For instance, since {reg_number} is a key of class Student, Student'[reg_number: 98765] and Student'98765 both designate an unique student.

2.2 Objects behaviour

Three *primitive methods* are defined in F2 to manipulate objects: create, update and delete. Primitive methods check integrity constraints (existential dependencies, keys) and perform automated classification.

Create

The *Create* method creates an object *o* in a class *C* and returns its oid. With the provided attribute values, the method checks cardinalities, keys, and classifies the object into every subclass whose specialization constraints are satisfied.

Example 3. The following F2 DML [15] expression creates an object and adds it to class Student (root class of the specialization tree), class SwissStudent (constraint <nationality = "Swiss"> satisfied) and class ScholarStudent (constraint <scholarship = 1> satisfied).

```
create SwissStudent'[reg_number: 98765, lastna-
me: "Dunand", firstname: {"Henri", "Marc"}, na-
tionality: "Swiss", scholarship: 1, in_dept:
Department'Math, canton: "Geneva"];
```

Update

The *Update* method updates the value taken by an object *o* on an attribute *att*. As for *Create*, the method checks cardinalities, keys and specialization constraints. It may make the object *o* migrate between classes of the specialization tree.

Example 4. The following expression updates the scholarship value of student number 98765 and consequently removes it from class ScholarStudent.

```
update Student'98765 scholarship:0;
```

Delete

The *Delete* method deletes an object *o* from all the classes it belongs to. In order to maintain existential dependencies, it may cause other recursive deletions.

Example 5. Deleting the student number 98765 deletes it from the classes Student and SwissStudent. Suppose that this student is referenced by an object *e* in class Enrolment. Since the attribute student has minimal cardinality 1, the object *e* is also deleted. We can avoid the recursive deletion by first setting student(e) to the unknown value, and then deleting the student.

```
class Enrolment
    (student: Student,
    course: Course);

delete Student'98765;
```

Triggers

F2 supports a *trigger* mechanism with six types of triggers: pre-create, post-create, pre-update, post-update, pre-delete and post-delete. A trigger associates a class and a trigger type with a set of methods. When a primitive method *M* is invoked on an instance of class *C*, the trigger for class *C* and type pre-*M* executes the associated methods. If one of these fails, the primitive method is rejected; else it is executed and the methods associated with the trigger for class *C* and type post-*M* are executed.

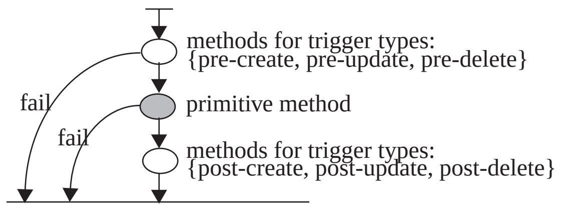
methods for trigger types:
{pre-create, pre-update, pre-delete}
fail
primitive method
fail
methods for trigger types:
{post-create, post-update, post-delete}

Fig. 1 Primitive method execution

If a primitive method is invoked on an object that belongs to several classes (multiple instantiation), the complete set of associated triggers is involved, i.e. all the "pre" methods are executed in an arbitrary order; if they all succeed *M* is executed followed by all the "post" methods.

Example 6. The class Course has a computed attribute nb_students which gives the number of students enrolled in a course. A trigger for class Enrolment and type post-create executes a method which increments the nb_students value of the concerned course.

3 F2 kernel and database schemes

The F2 model is *reflective* and *meta-circular*, i.e. it is self-descriptive and implemented using its own concepts. In F2 every class is an object and every object is an instance of a class (i.e. every class is an instance of a class). The instantiation hierarchy is finite by the fact that every class is an instance of class Class which is an instance of itself.

The F2 *kernel* is a set of objects which is necessary and sufficient to describe the structure and behaviour of any F2 database. In fact, the F2 kernel is a minimal database that describes itself, it corresponds to what is also called the *meta-schema* of the system. It is composed of all the classes and attributes shown in fig. 2 (attributes whose domain class is atomic are not drawn to avoid overloading of the graph).

A database schema is a user-defined set of classes, keys, triggers, methods, etc. that describe the structure and behaviour of database objects. In current commercial OO-DBMSs schema components are not standard objects, i.e. one cannot apply the standard object creation, update, and deletion operations to these objects. Since F2 treats classes as standard objects, schema objects are not different from database objects. Consequently, create, update and delete methods are applicable to database objects as well as schema objects. Thus no special purpose methods had to be defined to handle schema changes. Nevertheless, the manipulation of schema objects must take into account the database objects that the schema describes. Triggers may be added for this purpose, in order to give a specific semantics to schema changes as we will see in §4.3.

Example 7. The following operations create an atomic class String30, a tuple class Department and an attribute of Department with domain String30.

```
s := create AtomClass'[className: "String30",
classKind: "atomic", isSubClass: 0, atomBase:
"string", maxLength: 30];
```

```
d := create TupleClass' [className: "Depart-
ment", classKind: "tuple", isSubClass: 0];
```

```
create Attribute'[attName: "name", origClass:
d, domClass: s, minCard: 1, maxCard: 1];
```

This is exactly how the F2 DDL compiler translates the following class definitions:

```
class String30: stringBase [30];
class Department (name: String30);
```

Changing the domain of attribute name simply amounts to the following update:

```
update Attribute'[attName: "name", origClass:
d] domClass: Class'String55;
```

To make class Department a subclass of class OrganizationalUnit one could do the following:

```
update d isSubClass: 1;
update d superClass: Class'OrganizationalUnit;
```

Since F2 makes no difference between schema and meta-schema, one can create, update and delete meta-classes with the same primitive methods. Thus one can easily extend the F2 data model by adding new classes to the kernel and defining their behaviour with triggered methods. One can also change the dynamic part of the model by (re)defining triggers attached to existing kernel classes.

4 Schema evolution

The schema evolution framework of a DBMS is usually defined by a set of model invariants, a taxonomy of schema changes and their semantics. In order to be usable, the set of schema changes that forms the taxonomy must be "complete" in a certain sense. In [7] a set of schema changes is

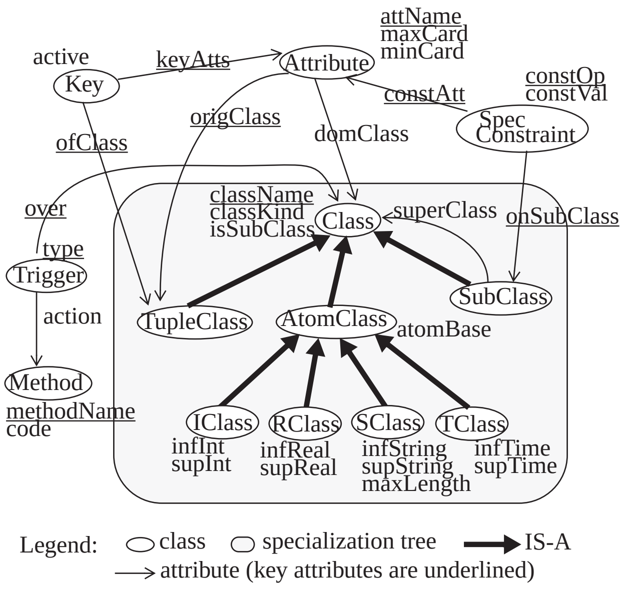

Fig. 2 F2 meta-schema

considered to be complete if given two arbitrary schemes it is possible to transform one into the other by a finite sequence of these schema changes. We propose another definition of completeness: a set of changes is *complete* if it contains all the possible primitive changes on the kernel instances. This definition is stronger than the one given in [7].

4.1 Model invariants

An *invariant* is a property which must be preserved across schema and data changes. Invariants hold at every quiescent state of the schema, that is before and after a schema change operation. F2 model invariants are:

I1: the value of an attribute is a set of objects that belong to the attribute's domain class and satisfies the cardinality constraints.

I2: the instances of a subclass are exactly those of its superclass that satisfy its specialization constraints.

I3: the extent of a class satisfies the class' keys.

I4: the class hierarchy is a forest, i.e. a collection of specialization trees.

I5: attributes in a specialization tree have distinct names.

I6: key attributes of a class are attributes defined on that class (local or inherited).

I7: the attribute used in a specialization constraint on a subclass *C* is an attribute defined on the superclass of *C* (local or inherited).

Other model invariants are consequences of the keys and existential dependencies of the meta-schema objects. For instance: the key {className} of class Class implies that class names are unique; the attributes domClass and origClass of class Attribute induce an existential dependency between an attribute and its origin and domain classes.

4.2 Taxonomy

The F2 schema changes *taxonomy* is a direct consequence of our definition of completeness. It is given by considering the F2 meta-schema (fig. 2) as a database schema and applying all the primitive methods on its instances. This contrasts with approaches that develop a taxonomy of "useful" schema changes [7] [29] [36]. The obtained list of schema changes is shown in figure 3.

4.3 Semantics of schema changes

To preserve the model invariants, we define the semantics of schema changes in F2 with pre-conditions and post-actions. *Pre-conditions* must be satisfied to allow a schema change to occur. *Post-actions* are executed on schema and database objects to guarantee the consistency of the database. F2 propagates schema changes instantly to main memory objects and makes them permanent upon issuing a commit operation.

- *Create a new class*
- *Delete an existing class*
- *Update an existing class*
    - *Change its description:*
        - *Change its name (className), its interval if atomic class (infInt, supInt, etc.), its maximal length if string class (maxLength)*
    - *Modify its position in the class hierarchy:*
        - *Change its superclass (superClass), make it a subclass / non-subclass, i.e. attach / detach it from a specialization tree (isSubClass)*
- *Create a new attribute of a class*
- *Delete an existing attribute*
- *Update an existing attribute:*
    - *Change its name (attName), its maximal cardinality (maxCard), its minimal cardinality (minCard), its domain class (domClass), its origin class (origClass)*
- *Create a new key of a class*
- *Delete an existing key*
- *Update an existing key:*
    - *Change the class on which it is defined (ofClass), its attributes (keyAtts), activate / deactivate it (active)*
- *Create a new specialization constraint of a subclass*
- *Delete an existing specialization constraint*
- *Update an existing specialization constraint:*
    - *Change the subclass on which it is defined (ofSubClass), its attribute (constAtt), its operator (constOp), its value (constVal)*
- *Create a new trigger over a class*
- *Delete an existing trigger*
- *Update an existing trigger:*
    - *Change the class over which it is defined (over), its type (type), the set of methods it triggers (action)*
- *Create a new method*
- *Delete an existing method*
- *Update an existing method:*
    - *Change its name (methodName), its code (code)*

Fig. 3 Schema evolution taxonomy

Changes directly implemented by primitive methods

Some schema changes have neither pre-conditions nor post-actions. They are simply executed by applying a primitive method on the schema. For example: delete an existing key, delete an existing trigger, update the action (set of methods) of a trigger.

Other schema changes have a semantics which is already implemented in the F2 model through keys and existential dependencies. For example, when adding or renaming a class, uniqueness of name is a pre-condition which is automatically verified by the key {className} defined on class Class.

Another interesting example is to consider what happens when a class *C* is deleted: all the attributes which have *C* as origin or domain class are deleted and all the subclasses, specialization constraints, keys and triggers of *C* are deleted too. All these repercussions are automatically executed when deleting a class thanks to existential dependencies induced by the meta-schema attributes origClass, domClass, superClass, onSubClass, ofClass, over, which have minimal cardinality 1.

Changes implemented with triggers

Other schema changes have a specific semantics expressed by pre-conditions and post-actions on schema and database objects. Their semantics is implemented by triggers. We describe an example of such changes: change the superclass of a subclass *C*.

Pre-conditions: the new superclass is in the specialization tree of *C* and is not a successor of *C* (avoid cycles).

Post-actions on schema objects: let *S* be the set of classes containing *C* and its successors. A class in *S* may lose inherited attributes from the old superclass of *C*. So:

- if there is a class *C'* in *S* on which is defined a key containing a lost attribute, then delete this key.
- if there is a class *C'* in *S* on which is defined a specialization constraint containing a lost attribute, then delete this constraint.

Post-actions on database objects:

- for each object *o* in *C*, if *o* is not in the new superclass, then remove *o* from *C* and its successors.
- for each object *o* in the new superclass, if *o* is not in *C* and *o* satisfies the specialization constraints of *C*, then put *o* in *C* and possibly *C*'s successors according to their respective constraints.

Implementation: we define four methods: CheckSuperClass, SuppressInvalidKey, SuppressInvalidConstraint, MigrateObjects. Then we create two triggers in the meta-class Trigger:

```
create Trigger’[type: “pre-update”, over:
Class’SubClass, action: Method’CheckSuper-
Class];
```

```
create Trigger’[type: “post-update”, over:
Class’SubClass, action: {Method’SuppressInva-
lidKey, Method’SuppressInvalidCons-
traint,Method’MigrateObjects}];
```

The first trigger executes the method CheckSuperClass before updating a subclass. If it succeeds, the schema change is done and the second trigger executes the methods SuppressInvalidKey, SuppressInvalidConstraint and MigrateObjects.

5 Transposed physical storage: a non-classical approach

F2 uses the transposed storage approach [9]. At the physical storage level, attributes are seen as functions from origin oids to domain oids. Each attribute is stored into a vector indexed by origin oids and consisting of a set of physical pages not necessarily contiguous. Thus values of a given attribute are grouped into physical areas. This way of storing objects is not wide-spread in DBMSs. Relational DBMSs as Oracle [27] or object-oriented DBMSs as Orion [7] support a classical tuple-oriented storage scheme, i.e. they store one tuple or object into one record.

5.1 Storage format

A tuple class *C* with *n* attributes att_1, att_2, ..., att_n is stored in (*n*+1) vectors, one vector for each attribute plus one vector for a state attribute *state_C* (fig. 4a). The state attribute is managed internally by the DBMS and is not visible to the end users. An object *o* of a class *C* has:

- an oid <*R*, *r*> formed by the identifier *R* of the class *C* (within the meta-class Class) and the identifier *r* of the object *o* within the class *C*.
- a value [att_1: $att_1(r)$, att_2: $att_2(r)$, ..., att_n: $att_n(r)$], where $att_i(r)$ is a set of identifiers of objects within the domain class of the attribute att_i.
- an internal state *state_C(r)* which is: a) negative, if *o* does not belong to *C*; b) equal to zero, if *o* belongs to *C*; c) greater than zero, if *o* belongs to *C* and is referenced by objects (acts like a reference counter).
  If *o* belongs to a subclass *C'* of *C*, it has the oid <*R'*, *r*>, which means that it keeps the same identifier *r* inside all the classes of a specialization tree (fig. 4b).

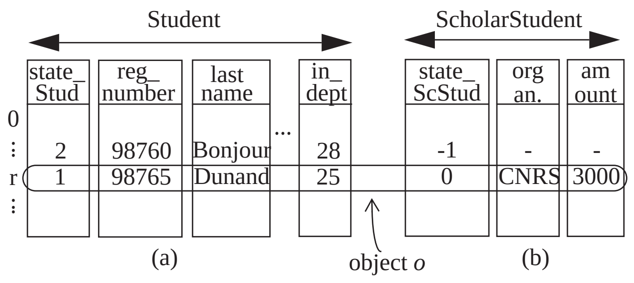

Fig. 4 a) object of a tuple class; b) object in a subclass

5.2 Advantages and disadvantages

Transposed storage is interesting in the context of schema evolution for two reasons:

- it avoids the reorganization of the physical storage. For all the schema changes we mentioned the physical storage is not reorganized. In other words, a schema change does not require to move data, which is often costly. For example, if an attribute *att* of class *C* is deleted, the physical pages of *att* are set free. The physical storage is not affected, while with traditional storage the values of *att* must be removed from records of all existing instances of *C*. Since a class can have many instances, this is potentially a very expensive undertaking. Another example is add a new attribute to class *C*. New physical pages are reserved to store later the attribute values. Here again, the storage is left untouched. With traditional storage, all existing instances of C will have a field added to their record to store the attribute values.

- it reduces considerably the number of input/output operations from disk to main memory and vice-versa. Schema changes do not bring whole objects into main memory as with classical storage. Only the affected attributes are loaded, thus reducing the number of input/output operations. For example, when one updates the domain class of an attribute *att* of class *C*, only the physical pages of *att* are loaded into main memory (to change *att* values). Their number is much smaller than the number of physical pages of *C* instances, which are loaded with traditional storage.

Transposed storage can be disadvantageous when retrieving all the attribute values of some instances of a class *C*. However transposed storage becomes advantageous when computing an aggregate function (sum, average, etc.) on a single attribute.

6 Related work

Orion [7] is an OODBMS which supports schema evolution. A set of invariants and rules (when an invariant can be preserved by more than one way) is defined to guarantee the consistency of the database schema. Schema changes are categorized as: (1) changes to class definitions, including changes to instance variables and methods, (2) changes to the class hierarchy. The Orion approach, called screening, defers the propagation of schema changes to persistent objects. These changes are propagated to the affected objects only when they are retrieved.
The framework of schema evolution in **Goose** [25] is based on that of Orion. But Goose propagates schema changes instantly to main memory objects and makes them permanent upon issuing a save operation.

Gemstone [29] is another OODBMS which extends Smalltalk-80 capabilities and supports simple inheritance. A set of invariants is defined and a limited set of schema changes is supported (changes to inheritance links and changes to methods are not provided). The semantics of these changes differs from that given in Orion. In the Gemstone approach, called conversion, schema changes are made instantly.

Otgen [22], an object-oriented system, supports more complex schema changes (not local to class definitions) for the purpose of reorganizing the database. A transformation table is used to describe the relationships among objects before and after the schema change. The table has an entry for each class. It indicates how instances of that class should be transformed. Schema changes are collected and released as a unit. They are propagated to objects when these are accessed.

Lispo$_2$ [8] is a language extending Lisp with the O$_2$ object-oriented data model and orthogonal persistence. The Lispo$_2$ programming environment supports schema evolution operations and addresses their impact at the class, instance and method levels. The method compiler catches inconsistencies in methods with regard to the schema and may mark a method as invalid or recompile it. Lispo$_2$ uses a semi-lazy policy: for instances in main memory schema changes are immediately propagated while for instances on disk changes are performed when the instances are loaded in main memory.

All these systems take the approach that applies a schema change to a single schema and updates the affected instances. A second approach is to create a new version of the modified object and keep the old one. Versions of different granularities are proposed in the literature: schema versions in **Orion** [19], class versions in **Encore** [34] and in [24], object versions in **Charly** [28]. A third approach is to handle schema evolution with an external schema as views in [11] and context versions in [5].

7 Conclusion

We showed in this paper that the F2 OODBMS is evolution oriented thanks to:

- a powerful data model that supports multiple instantiation, automated classification and integrity constraints as keys and existential dependencies.
- a self-descriptive model allowing to treat database objects, schema objects and meta-schema objects similarly, thus providing uniformity and extensibility of the data model.
- a trigger mechanism allowing primitive methods to trigger other methods. It is used to implement the semantics of schema changes.
- a transposed storage that avoids database reorganization and greatly reduces the execution time of schema changes.

The F2 DBMS is composed of three layers. The lowest layer implements persistent storage. It encapsulates the paging and block buffering mechanism, and a transaction mechanism. The second layer implements the primitive methods. The third layer includes the query and manipulation language interpreter and the Ada application program interface (API).

The whole code is written in Ada, using exclusively standard library packages. The choice of Ada proved to be judicious for at least two reasons: language features for genericity have been extensively used in F2 and eased the packages structure design, and second, the code was found very easily portable on any platform with an Ada compiler.

It currently runs under VAX/VMS, DEC-Alpha/VMS, SunOS and MacOS.

All the schema changes listed in this paper are supported in the current F2 system. In addition, a user-friendly interactive schema editor eases the user's task.

One direction of our actual research activity consists in studying how schema evolution transactions could execute concurrently. Another direction is to integrate the concept of context versions defined in [5] and [3]. Contexts [4] are abstraction that isolate the database user or programmer from the database schema thus they can be used to hide schema changes to users and application programs.

References

[1] Abrial, J.R., "Data Semantics", in Database Management, J.W. Klimbie & L.L. Koffman (eds), North-Holland, 1974.

[2] Al-Jadir, L., "Visite Guidée de Farandole 2, version lac", Technical report, Cahiers du CUI, no 77, Centre Universitaire d'Informatique, Genève 1993.

[3] Al-Jadir, L., Falquet, G. , Léonard, M., "Context Versions in an Object-Oriented Model", Proc. DEXA Conference, 1993.

[4] Al-Jadir, L., Le Grand, A., Léonard, M., Parchet, O., "Contribution to the Evolution of Information Systems", in Methods and Associated Tools for the Information Systems Lifecycle, A.A. Verrijn-Stuart & T.W. Olle (eds), IFIP, Elsevier, 1994.

[5] Andany, J., Léonard, M., Palisser, C., "Management of Evolution in Databases", Proc. 17th VLDB Conference, 1991.

[6] Atkinson, M., Bancilhon, F., DeWitt, D., Dittrich, K., Maier, D., Zdonik, S., "The Object-Oriented Database System Manifesto", Proc. DOOD Conference, 1989.

[7] Banerjee, J., Kim, W., Kim, H-J., Korth, H.F., "Semantics and Implementation of Schema Evolution in Object-Oriented Databases", Proc. ACM SIGMOD Conference, 1987.

[8] Barbedette, G., "Schema Modifications in the LISPO2 Persistent Object-Oriented Language", Proc. ECOOP Conference, Pierre America (ed), Springer-Verlag, 1991.

[9] Batory, D.S., "On Searching Transposed Files", ACM Transactions on Database Systems, vol. 4, no 4, december 1979.

[10] Bellahsene, Z., "An Active Meta-model for Knowledge Evolution in an Object-oriented Database", Proc. CAISE Conference, 1993.

[11] Bertino, E., "A View Mechanism for Object-Oriented Databases", Proc. EDBT Conference, 1992.

[12] Cointe, P., "Metaclasses are First Class : the ObjVlisp Model", Proc. OOPSLA Conference, 1987.

[13] Deux O. et al., "The O2 System", Communications of the ACM, vol. 34, no. 10, oct. 1991.

[14] Estier, T., Falquet, G., "Le modèle Farandole 2 et son implantation", Technical report, Cahiers du CUI, no 53, Centre Universitaire d'Informatique, Genève 1990.

[15] Estier, T., Falquet, G., "Le petit manuel de Farandole 2", Technical report, Cahiers du CUI, no 54, Centre Universitaire d'Informatique, Genève 1992.

[16] Estier, T., Falquet, G., Guyot, J., Léonard, M., "Six Spaces for Global Information Systems Design", in The Object Oriented Approach in Information Systems, F. van Assche & B. Moulin & C. Rolland (eds), IFIP, North-Holland, 1991.

[17] Falquet, G., Guyot, J., Junet, M., Léonard, M., et al., "Concept Integration as an Approach to Information Systems Design", in Computerized Assistance during the Information Systems Life Cycle, T.W. Olle & A.A. Verrijn-Stuart (eds), IFIP, North-Holland, 1988.

[18] Junet, M., Falquet, G., Léonard, M., "ECRINS/86 : An Extended Entity-Relationship Data Base Management System and its Semantic Query Language", Proc. 12th VLDB Conference, 1986.

[19] Kim, W., Chou, H-T., "Versions of Schema for Object-Oriented Databases", Proc. 14th VLDB Conference, 1988.

[20] Kim, W., Bertino, E., Garza, J.F., "Composite Objects Revisited", Proc. ACM SIGMOD Conference, 1989.

[21] Lécluse, C., Richard, P., Valez, F., "O2, an Object-Oriented Data Model", Proc. ACM SIGMOD Conference, 1988.

[22] Lerner, B.S., Habermann, A.N., "Beyond Schema Evolution to Database Reorganization", Proc. OOPSLA Conference, 1990.

[23] Lohman, G.M., Lindsay, B., Pirahesh, H., Schiefer, K.B., "Extensions to Starburst: Objects, Types, Functions and Rules", Communications of the ACM, Vol. 34, No.10, October 1991.

[24] Monk, S.R., Sommerville, I., "A Model for Versioning of Classes in Object-Oriented Databases", Proc. BNCOD Conference, 1992.

[25] Morsi, M.M.A., Navathe, S.B., Kim, H., "A Schema Management and Prototyping Interface for an Object-Oriented Database Environment", in Object Oriented Approach in I.S., F. Van Assche & B. Moulin & C. Rolland (eds), IFIP, North-Holland, 1991.

[26] Nguyen, G.T., Rieu, D., Escamilla, J., "An Object Model for Engineering Design", Proc. ECOOP Conference, 1992.

[27] Oracle, "Oracle7 Server Concepts Manual", Oracle Corporation, 1992.

[28] Palisser, C., "Le Modèle de Versions du Système CHARLY", Actes 6èmes Journées Bases de Données Avançées, INRIA, 1990.

[29] Penney, D.J., Stein, J., "Class Modification in the GemStone Object-Oriented DBMS", Proc. OOPSLA Conference, 1987.

[30] Pernici, B., "Objects with Roles", Proc. IEEE Conf. on Office Information Systems, 1990.

[31] Richardson, J., Schwarz, P., "Aspects: Extending Objects to Support Multiple, Independent Roles", Proc. ACM SIGMOD Conference, 1991.

[32] Scholl, M.H., Laasch, C., Tresch, M., "Updatable Views in Object-Oriented Databases", Proc. DOOD Conference, 1991.

[33] Sciore, E., "Using Annotations to Support Multiple Kinds of Versioning in an Object-Oriented Database System", ACM Transactions on Database Systems, vol. 16, no 3, september 1991.

[34] Skarra, A.H., Zdonik, S.B., "Type Evolution in an Object-Oriented Database", in Research Directions in OO Programming, B. Shriver & P. Wegner (eds), MIT Press, 1987.

[35] Tresch, M., Scholl, M.H., "Schema Transformation without Database Reorganization", SIGMOD RECORD, vol. 22, no 1, 1993.

[36] Zicari, R., "A Framework for Schema Updates in an OO Database System", Rapporto interno no 90-025, Dipartimento di Elettronica, Politecnico di Milano, 1990.