ResearchGate

See discussions, stats, and author profiles for this publication at: https://www.researchgate.net/publication/220711892

Language Extensions for the Automation of Database Schema Evolution.

Conference Paper · January 2008

Source: DBLP

CITATIONS
16

READS
57

6 authors, including:

George Papastefanatos
Athena-Research and Innovation Center in Information, Communication and Kno…
67 PUBLICATIONS **472** CITATIONS
SEE PROFILE

Panos Vassiliadis
University of Ioannina
192 PUBLICATIONS **4,286** CITATIONS
SEE PROFILE

Yannis Vassiliou
National Technical University of Athens
126 PUBLICATIONS **2,132** CITATIONS
SEE PROFILE

Some of the authors of this publication are also working on these related projects:

Project PROMAP - Mapping old and new forms of protest in Greece. View project

Project Schema evolution View project

All content following this page was uploaded by Panos Vassiliadis on 29 May 2014.

The user has requested enhancement of the downloaded file.

LANGUAGE EXTENSIONS FOR THE AUTOMATION OF DATABASE SCHEMA EVOLUTION

George Papastefanatos[1], Panos Vassiliadis[2], Alkis Simitsis[3],
Konstantinos Aggistalis[2], Fotini Pechlivani[2], Yannis Vassiliou[1]

[1]Department of Electr. and Comp. Eng., National Technical University of Athens, Athens, Greece
{gpapas, yv}@dbnet.ece.ntua.gr
[2]Department of Computer Science, University of Ioannina, Ioannina, Greece
{pvassil, fpechliv, kostazz}@cs.uoi.gr
[3]IBM Almaden Research Center, San Jose, California, USA
asimits@us.ibm.com

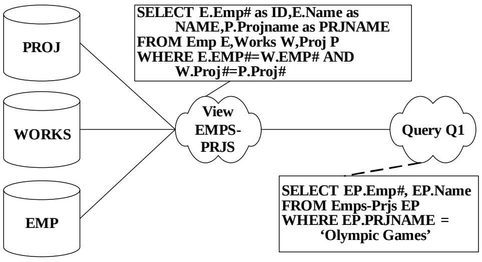
Keywords: Database schema evolution, SQL extension.

Abstract: The administrators and designers of modern Information Systems face the problem of maintaining their systems in the presence of frequently occurring changes in any counterpart of it. In other words, when a change occurs in any point of the system –e.g., source, schema, view, software construct– they should propagate the change in all the involved parts of the system. Hence, it is imperative that the whole process should be done correctly, i.e., the change should be propagated to all the appropriate points of the system, with a limited overhead imposed on both the system and the humans, who design and maintain it. In this paper, we are dealing with the problem of evolution in the context of databases. First, we present a coherent, graph-based framework for capturing the effect of potential changes in the database software of an Information System. Next, we describe a generic annotation policy for database evolution and we propose a feasible and powerful extension to the SQL language specifically tailored for the management of evolution. Finally, we demonstrate the efficiency and feasibility of our approach through a case study based on a real-world situation occurred in the Greek public sector.

1 INTRODUCTION

In typical organizational Information Systems, the designer/administrator is frequently faced with the necessity to predict the impact of a small change or a more sophisticated reorganization in the overall database configuration. For instance, consider the simplest configuration of a company's database which holds information about the employees and the projects they work for, shown in Figure 1. A view selects the employees who work for a project along with the project name. A query accesses this view, selecting all employees who work for the Olympic Games. Suppose the administrator decides that employees' name should be split into last and first name, by adding two new attributes in the underlying relation and deleting the existing attribute name. Should these changes be also reflected to the view and the query, then these constructs must be rewritten. Even a small change like this, usually impacts a large variety of applications and data stores related to the system: queries and data entry forms can be invalidated, application programs might crash (resulting in the overall failure of more complex workflows), and several pages in the corporate Web server may become invisible; i.e., they cannot be generated any more. It is imperative that such changes should be resolved and propagated to the involved counterparts of the system.

Syntactic as well as semantic adaptation of workload –mainly queries and views– to changes occurring in the database schema is a time-consuming task, treated in most situations manually by the administrators or the application developers. Current DBMS languages do not incorporate evolution semantics, so that administrators / developers could prescribe the behavior of the system when database schema evolution changes occur. On the contrary, to deal with the problems occurred by the evolution in databases, several practical techniques are usually used for this reason, like the use of variable names as



```
SELECT  E.Emp# as ID,E.Name as
        NAME,P.Projname as PRJNAME
FROM Emp E,Works W,Proj P
WHERE E.EMP#=W.EMP# AND
        W.Proj#=P.Proj#
```

```
SELECT  EP.Emp#, EP.Name
FROM Emps-Prjs EP
WHERE EP.PRJNAME  =
        'Olympic Games'
```

Figure 1: A simple configuration of a query over a view, defined over three relations.

placeholders for the real names of constructs like attributes and tables. For example, Oracle's `PL/SQL` uses the `%TYPE` and `%ROWTYPE` constructs to define variables as they are defined within the database. If the datatype or precision of a column changes, the program automatically picks up the new definition from the database without having to make any code changes. Hence, the appropriate enrichment of the procedural code with such constructs provides data independence, reduces maintenance costs, and allows programs to adapt as the database changes to meet novel business requirements. However, such techniques partially confront the problem, as they are dealing with the simplest cases of evolution.

Database evolution is a more complicated issue; we mention here an experience described by (Sjoberg, 1993). In his report, a quantification of the database schema evolution problem in large long-lived application systems is presented. Over a period of 18 months, which included both the development and the operational phase of the examined system, they recorded 140% increase in the number of relations and over 200% increase in the number of attributes, as well as several evolution changes in all existing relations of the system.

Due to its great significance and practical importance, the database evolution has already gained research attention. Several works have identified this problem as a great challenge for database researchers (Roddick, 2000) and various efforts have been proposed so far (Bellahsene, 2002), (Gupta, 2001), (Nica, 1998), (Velegrakis, 2004). In our work, we extend previous results (Nica, 1998) by incorporating the addition of attributes and by appropriately treating conditions. Also, we allow the restructuring of the database, which is considered as a graph in our framework, towards either the retention of the original query semantics – a similar but quite restrictive approach has been proposed by (Velegrakis, 2004) – or their appropriate readjustment to the new semantics. In addition, we complement our approach by proposing an elegant extension to the SQL language for the management of database evolution. Compared with previous efforts (Roddick, 1992), our work does not require either schema versioning or integration of time within database schema evolution. In fact, we provide rules for the transformation and adaptation of queries and views to the last valid database schema without the assumption that the transformed queries retain the same semantics.

Contributions. Briefly, the main contributions of this work are as follows.

– We describe a graph-based model that uniformly covers database constructs, such as relational tables, views, database constraints and SQL queries, as first class citizens (Section 2.)
– We present a mechanism for the annotation of the graph's constructs with elements that facilitate what-if analysis and predetermine the reaction to evolution events occurring in the database schema (Section 3.)
– We propose feasible and powerful SQL extensions that enable the implementation of our approach for evolution management (Section 4.)
– We demonstrate the efficiency and feasibility of our approach through a real-world case study occurred in the Greek public sector (Section 5.)

2 GRAPH-BASED MODELING OF DATABASE SCHEMA EVOLUTION

This section proposes a graph modeling technique that uniformly covers relational tables, views, database constraints, and SQL queries as first class citizens. Our technique provides an overall picture not only for the actual source database schema but also for views and queries accessing the database, since these constructs are incorporated in the model.

Formally, an evolving database schema along with its workload (i.e., queries and views) is represented as a directed graph $\mathbf{G}=(\mathbf{V},\mathbf{E})$. The nodes of the graph represent the entities of our model, where the edges represent the relationships among these entities. Moreover, we distinguish the following essential components, which are included in our model: *relations*, *conditions* (covering database constraints and query conditions), *queries* and *views*.

Relations, $\mathbf{R}$. Each relation $R(\Omega_1,\Omega_2,\ldots,\Omega_n)$ in the database schema, is represented as a directed graph, which comprises: (a) a *relation node*, R, representing the relation schema; (b) n *attribute nodes*, $\Omega_i \in \mathbf{\Omega}$, $i=1..n$, one for each of the attributes; and (c) n *schema relationships*, $\mathbf{E_S}$, directing from the relation node towards the attribute nodes, indicating that the attribute belongs to the relation.

Conditions, C. Conditions refer both to *selection conditions*, of queries and views and *constraints*, of the database schema. We consider three classes of atomic conditions that are composed through the appropriate usage of an operator op belonging to the set **Op**, containing the usual binary operators, (e.g., <, >, =, ≤, ≥, !=, IN, EXISTS, ANY): (a) Ω op constant; (b) Ω op Ω′; and (c) Ω op Q. (Ω, Ω′ are attributes of the underlying relations and Q is a query). A *condition node* is used for the representation of the condition. The node is tagged with the respective operator and it is connected to the *operand nodes* of the conjunct clause through the respective *operand relationships*, **O**. Composite conditions are easily constructed by tagging the condition node with a Boolean operator (e.g., AND or OR) and the respective edges, to the conditions composing the composite condition.

Queries, Q. The graph representation of a Select - Project - Join - Group By (SPJG) query involves a new node representing the query, namely *query node*, and *attribute nodes* corresponding to the schema of the query. Thus, the query graph is a directed graph connecting the query node with all its schema attributes, through *schema relationships*. In order to represent the relationship between the query graph and the underlying relations, the query is resolved into its essential parts: SELECT, FROM, WHERE, GROUP BY, HAVING, and ORDER BY, each of which is eventually mapped to a subgraph.

Select part. Each query is assumed to own a schema that comprises the attributes, either with their original or alias names, appearing in the SELECT clause. In this context, the SELECT part of the query maps the respective attributes of the involved relations to the attributes of the query schema through *map-select relationships*, $\mathbf{E_M}$, directing from the query attributes towards the relation attributes.

From part. The FROM clause of a query is considered as the relationship between the query and the relations involved in this query. Thus, the relations included in the FROM part are combined with the query node through *from relationships*, $\mathbf{E_F}$, directing from the query node towards the relation nodes.

Where and Having parts. We assume the WHERE and/or HAVING clauses of a query in conjunctive normal form. Thus, we introduce two directed edges, namely *where relationships*, $\mathbf{E_W}$, and *having relationships*, $\mathbf{E_H}$, both starting from a query node towards an operator node corresponding to the conjunction of the highest level.

Group and Order By part. For the representation of aggregate queries, two special purpose nodes are employed: (a) a new node denoted as GB∈**GB**, to capture the set of attributes acting as the aggregators; and (b) one node per aggregate function labeled with the name of the employed aggregate function; e.g., COUNT, SUM, MIN. For the aggregators, we use edges directing from the query node towards the GB node that are labeled <group-by>, indicating *group-by relationships*, $\mathbf{E_G}$. The GB node is connected with each of the aggregators through an edge tagged also as <group-by>, directing from the GB node towards the respective attributes. These edges are additionally tagged according to the order of the aggregators; we use an identifier i to represent the i-th aggregator. Moreover, for every aggregated attribute in the query schema, there exists an edge directing from this attribute towards the aggregate function node as well as an edge from the function node towards the respective relation attribute. Both edges are labeled <map-select> and belong to $\mathbf{E_M}$, as these relationships indicate the mapping of the query attribute to the corresponding relation attribute through the aggregate function node. The representation of the ORDER BY clause is performed similarly, whereas nested queries and functions used in queries are also incorporated in our model.

Views, V. Views are considered either as queries or relations (materialized views), thus, **V** ⊆ **R**∪**Q**.

3 CONSTRUCTS ANNOTATION WITH EVOLUTION POLICIES

Evolution changes may affect the software around the database (mainly views and queries) in two ways: (a) *syntactically*, a change may evoke a compilation or execution failure during the execution of a piece of code; and (b) *semantically*, a change may have an effect on the semantics of the software used. In the context of the proposed graph, changes in the database schema are events, which transform specific parts of the graph (e.g., a relation graph sustaining a change) and eventually affect other dependent graph constructs (e.g., a view graph depending on the specific relation). The latter may raise, in turn, new evolution changes, which have impact on other graph constructs (such as a query graph depending on the specific view.)

To handle schema evolution, the constructs of the graph are annotated with elements that facilitate what-if analysis and predetermine the reaction to evolution events that may occur. Each construct is enriched with policies that allow the designer to specify the behavior of the annotated construct whenever events occur. The combination of an event with a policy determined by the designer triggers the execution of the appropriate action that either *blocks* the event or *reshapes* the graph to adapt to the proposed change. The annotated graph is stored in a metadata repository and it is accessed from a what-if analysis module. This module notifies the designer on the effect of a potential change and the extent to which the modification to the existing code can be fully automated for adapting to the change.

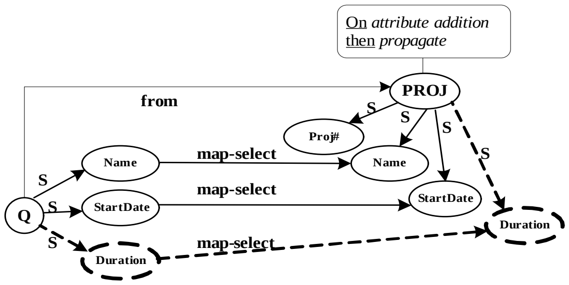


Figure 2: Annotating relation PROJ.

The space of potential events comprises the Cartesian product of two subspaces. The space of hypothetical actions (addition, deletion, and modification) over graph constructs sustaining evolution changes (e.g., relations, views, attributes, and conditions). For each of the above events, the administrator annotates graph constructs affected by the event with policies that dictate the way they will regulate the change. Three kinds of policies exist, as follows.

(a) *Propagate* the change, meaning that the graph must be reshaped to adjust to the new semantics incurred by the event.

(b) *Block* the change, meaning that we want to retain the old semantics of the graph and the hypothetical event must be blocked or, at least, constrained, through some rewriting that preserves the old semantics (Nica, 1998), (Velegrakis, 2004). In this case, the specific method that may be used is orthogonal to our approach.

(c) *Prompt* the administrator to interactively decide what will eventually happen.

Consider the graph of the query `SELECT Name, StartDate FROM PROJ` (Figure 2), which projects the name and the start date of all projects stored in the database. The annotation of relation `PROJ` with policy for *propagating addition* indicates that the addition of a new attribute, namely `DURATION`, to the `PROJ` relation will be propagated to the query and will be included in the `SELECT` clause of the query.

The graph annotation of the database ecosystem with policies for events occurs in a principled fashion:

1. First, we prescribe the default policies for all kinds of constructs, in a database-wide context.
2. Next, we prescribe defaults policies for specific relations, queries and views of the system, with respect to any combination of the following: the deletion of the construct per se, as well as the addition, deletion or modification of a construct's descendants. The descendants can be appropriately specified by their type, as applicable (i.e., attributes, constraints or conditions).
3. Lastly, we annotate specific low-granularity constructs, i.e., attributes, constraints or conditions, with policies for their deletion or modification.

The above arrangement is order-dependent and exploits the fact that there is a partial order of policy overriding. The order is straightforward: defaults are overridden by specific annotations and high-level construct annotations concerning their descendants are overridden by any annotation of such descendant:

$$Default \leq \{relation, query, view\}$$
$$\leq \{attribute, condition, constraint\}$$

4 LANGUAGE EXTENSIONS

In this section, we present SQL extensions that enable the implementation of the previous techniques for the management of evolution. For extending a system catalog with extra information regarding evolution purposes, we provide extensions to SQL regarding both top level construct definitions, like tables, views, and queries, as well as fine grain constructs such as attributes, conditions of views/ queries, and database constraints. All extensions outlined are given in BNF and throughout the section we refer to the configuration shown in Figure 1.

4.1 Database-wide Default Values

Regarding the definition of database default policies, we consider each assertion as a tuple (*event*, *policy*). Syntactically, this is expressed as follows:

```
ON <event> THEN <policy>
```

An event refers to evolution events in the database schema comprising an event type, such as `Delete`, `Add`, `Modify`, `Rename` and a construct type, which takes any of the following values in the partial order presented:

i. `NODE`
ii. `RELATION, QUERY, VIEW`
iii. `ATTRIBUTE, CONDITION, PK, FK, NNC, UC`

Note that we annotate nodes with default values only for changes applied to themselves and not to any of their ancestors or descendants. For example, we can have the following annotations:

```
ON DELETE NODE THEN PROPAGATE
ON DELETE ATTRIBUTE THEN PROMPT
```

The definitions of the default policies are expressed in SQL as follows.

- **SQL Syntax**

```
db-spec::=CREATE DATABASE <db-name> [policy-list]
policy-list::= policy-clause [,policy-clause]
policy-clause::= ON event THEN policy
event::= event-type construct-type
event-type::= Add | Delete | Modify | Rename
construct-type::=  NODE | RELATION | QUERY |
VIEW | ATTRIBUTE | CONDITION | PK | FK | NNC | UC
policy::= propagate | block | prompt
```

- **Example**

```
CREATE DATABASE company
ON DELETE ATTRIBUTE THEN PROMPT
```

4.2 Top Level Constructs

We extend SQL syntax to include evolution-based semantics both in DDL statements as well as in SQL queries. The general syntax is:

```
   ON <event> TO <construct> THEN <policy>
```

where event again refers to evolution events in the database schema, construct refers to the specific database part suffering the event and policy can take the values {*propagate, block, prompt*}.

4.2.1 Relations

Definition of policies on relations regarding their behaviour on evolution changes is primarily enforced upon creation, and thus, we extend CREATE TABLE syntax with certain policy clauses. Policies imposed in a relation-wide scope can be applied both to the relation itself as well as to all schema attributes and constraints. In that way, the administrator has the ability to annotate with a single clause the entire relation schema instead of annotating each constituent attribute or constraint separately.

- **SQL Syntax**

```
table-spec::= CREATE TABLE <table-name>
   (table-element-list [, policy-list])
policy-list::= policy-clause [,policy-clause]
policy-clause::=ON event TO construct THEN policy
event::=Add Attribute | Delete Attribute | Rename
Attribute | Delete Relation | Rename Relation | Add
Condition | Delete Condition | Modify Condition
policy::= propagate | block | prompt
construct::= <table-name>
```

- **Example**

```
CREATE TABLE works
  (EMP# NUMBER(3), PROJ# NUMBER(3), HOURS NUMBER(5),
   ON Add Attribute TO works THEN propagate)
```

The above syntax corresponds to the annotation of the respective relation node (i.e., works) with the policy that allows the addition of attributes and propagates this addition to all queries and views accessing this relation. Similarly, policy clauses can extend ALTER TABLE statements, enabling the administrator to define policies on existing relations.

4.2.2 Views

Views are inherent constructs of the database schema that constitute queries over the database schema –w.r.t. to views' definition– and relations to other queries –w.r.t. views' functionality. Therefore, views invoke evolution events when (a) their definition is altered, affecting all queries defined over them and (b) the relations over which they are defined are affected by schema changes. We enrich existing SQL syntax for views creation to capture potential events on their definitions as follows.

- **SQL Syntax**

```
view-spec::= CREATE VIEW <view-name> AS
   query-expression [policy-list]
policy-list::= policy-clause [,policy-clause]
policy-clause::=ON event TO construct THEN policy
event::= Add Attribute | Delete Attribute |
Rename Attribute | Delete View | Rename View |
Delete Relation | Rename Relation | Add Condi-
tion | Delete Condition | Modify Condition
policy::= propagate | block | prompt
construct::= <view-name> | <table-name>
```

The policies capture events occurring at the source tables of views' definition (i.e., the construct is a table-name) or events occurring at the view definition itself (i.e., the construct is a view-name).

- **Example**

```
CREATE VIEW emps-prjs AS
  SELECT E.Emp#, E.Name, P.Projname
  FROM Emp E,Works W,Proj P
  WHERE E.EMP#=W.EMP# AND W.Proj#=P.Proj#
  ON Modify Condition TO emps-prjs THEN block
```

Such syntax corresponds to the annotation of the view node emps-prjs with a policy, which blocks changes in the WHERE clause of the view definition.

4.2.3 Queries

Queries are considered as top-level constructs in our framework and they are the primary consumers of evolution changes occurring at the database level. Policies' clauses enrich query syntax with evolution semantics regarding the reaction of the query to such changes and have a query-wide scope, i.e., prescribe the behavior of the query itself and the query constituents (query attributes, query conditions). In such way, the developer may define a query-wide reaction to an evolution change instead of assigning explicit policies to each query attribute and condition.

- **SQL Syntax**

```
query-expression::=
SELECT [ALL|DISTINCT] scalar-expression-list
   FROM table-expression
   [WHERE search-condition]
   [GROUP BY grouping-column-list]
   [HAVING group-condition]
   [ORDER BY sort-specification-list]
   [policy-list]
policy-list::= policy-clause [,policy-clause]
policy-clause::=ON event TO construct THEN policy
event::= Add Attribute | Delete Attribute |
Rename Attribute | Delete View | Rename View |
Delete Relation | Rename Relation | Add Condi-
tion | Delete Condition | Modify Condition
policy::= propagate | block | prompt
construct::= <view-name> | <table-name>
```

- **Example**

```
Q: SELECT EP.Emp#, EP.Name
   FROM emps-prjs EP
   WHERE EP.PRJNAME = 'Olympic Games'
   ON Add Attribute TO emps-prjs THEN block
```

The above syntax corresponds to the annotation of the query node `Q` with a policy, which blocks the inclusion of added attributes in the underlying view `emps-prjs` in the select clause of the query syntax.

4.3 Fine Grain Constructs

Policy annotation can be further specialized to fine grain constructs such as attributes, database constraints and conditions of views/queries. Such annotations enable the administrator to define specific policies on these constructs, which override policies defined on their top-level containers.

4.3.1 Attributes

Policies are defined for relation attributes in table definition and for view or query attributes in view or query definitions, respectively. Policies' clauses refer to attribute constructs, which may be affected by an evolution change, prescribing in that way the specific behavior of that attribute.

- **SQL Syntax**

```
policy-clause::=ON event TO construct THEN policy
event::= Delete Attribute | Rename Attribute |
         Modify Domain
policy::= propagate | block | prompt
construct::=  [<table-name>  |  <view-name>.]
              <attribute-name>
```

- **Example**

```
CREATE TABLE emp
  (EMP# NUMBER(3),
   Name Varchar2(150),
    ... ,
   ON Delete Attribute TO Name THEN block)
```

Such syntax corresponds to the annotation of the attribute node `Name` with the explicit policy that blocks the node deletion from the container relation.

```
Q: SELECT E.Emp#, E.Name, P.Projname
   FROM Emp E,Works W,Proj P
   WHERE E.EMP#=W.EMP# AND W.Proj#=P.Proj#
  ON Delete Attribute TO Name THEN propagate
```

Such syntax corresponds to the annotation of the projected attribute node `Name` of the query `Q` with the explicit policy for allowing the node deletion from the select clause of the query (i.e., the respective attribute is removed from the underlying database.)

4.3.2 Constraints

Similarly, policies are defined on database constraints to override potential defined policies on their top-level containers (i.e., relation) and thus to prescribe their specific behavior to evolution changes.

- **SQL Syntax**

```
policy-clause::=ON event TO construct THEN policy
event::= Delete Constraint|Modify Constraint
policy::= propagate | block | prompt
construct::=[<table-name>.]<constraint-name>
```

- **Example**

```
CREATE TABLE emp
(EMP# NUMBER(3),
 Name Varchar2(150),
 Constraint EMP.PK PRIMARY KEY (EMP#),
 ON Modify Constraint TO EMP.PK THEN propagate)
```

The above syntax corresponds to the annotation of the constraint node `Emp.PK` with the explicit policy for allowing the modification of itself and propagating this change to all dependent constructs.

4.3.3 Conditions

Policies are defined on condition clauses of queries and views for prescribing their behavior to evolution events too. The modification or deletion of a view or a query condition semantically impacts dependents parts of the system. Thus, policies imposed on conditions override query- or view-wide policies and handle semantic changes invoked by such events.

- **SQL Syntax**

```
policy-clause::=ON event TO construct THEN policy
event::= Delete Condition|Modify Condition
policy::= propagate | block | prompt
construct::=[<view-name>.]<condition-name>
```

Moreover, we provide a facility for the management of conditions as first class citizens. We employ a specific name for each condition as follows.

```
CREATE CONDITION <condition> AS <expression>
```

For instance, we might have the following statements, expressing (a) a simple condition employed in a query, (b) a foreign key constraint, and (c) a join condition, respectively.

```
CREATE CONDITION Emp_Age_Cond AS AGE>50
CREATE CONDITION Works_Emp_FK AS WORKS.EMP#
       IN EMP.EMP#
CREATE CONDITION Works_Emp_J AS
   WORKS.EMP#=EMP.EMP#
```

Traditional statements for the definition of foreign keys or assertions for attribute domains are easily refined to the above "normal form", without necessarily obliging the database designer or administrator to abide by the above syntax.

Conditions may be employed in the WHERE clause. For example, a query SELECT * FROM EMP WHERE AGE_COND would simply use the condition as a macro. Parametric conditions, to allow referring to aliases in SQL queries are straightforward. One can also deal with the problem of existing code in a straightforward manner, since automatic condition names can be assigned to all the queries.

5 EVALUATION

We evaluated the proposed framework and capabilities of the approach presented via the reverse engineering of a real-world evolution scenario extracted from an application of the Greek public sector. Our goal was to minimize the human effort required for defining and setting the evolution metadata on the system by using the proposed language extensions.

We extracted queries and views from applications and stored procedures, and we monitored the events occurred on the database schema and the way affected constructs had been manually adjusted by the designers (e.g., through some rewriting) to each evolution event. In doing so, we resolved the appropriate policies per event for all affected constructs. Next, we used our approach for mapping constructs to graphs and annotate them with policies. Our GUI, namely HECATAEUS, allows for the representation of the database graph and its annotation with policies regarding evolution semantics and enables the user to explicitly define policies on graph constructs and perform what-if analysis for several evolution cases.

The configuration used comprises a total set of 52 queries over 18 relations. The evolution events occurred in the database schema include renaming of relations and attributes, modification of attribute domain, deletion of attributes, and modification of primary key constraints. Per event, we employed the appropriate *propagate* or *block* policy on the relations, queries or attributes affected by the event.

In the context of our graph model, our configuration comprised approximately 2500 nodes manually annotated with policies for each event that were affected by. This was a rather time-consuming task, as queries, query attributes, and relations had to be explicitly annotated. Appropriate policies were defined over different kinds of nodes (Table 1.)

Per query and relation, we counted the number of nodes manually annotated with policies *propagate* or *block* per event and the results are summarized in Table 2. Each node may have been, annotated with more than one policy when such annotations address different events; e.g., an attribute node may permit its renaming, whereas block its deletion.

Additionally, we employed the proposed SQL extensions to impose the same policies on the graph. We measured the number of the policy clauses, which must enrich existing SQL and DDL commands in order to annotate the same policies on the graph as opposed to the number of manual annotations on nodes. Hence, we evaluated 3 different cases: a) use of a default *propagate* policy for a *specific* query and for the events Delete, Rename and Modify Domains of attributes (query scope) instead of manually annotating each query attribute, b) use of default policies for all relations (relation scope) for propagating the aforementioned events, instead of annotating each query and c) use of default *propagate* policy for database (database scope) to allow the renaming of relations and the addition of attributes instead of annotating each relation. The results are shown in Table 3.

With the usage of the proposed SQL extensions, the human effort for explicitly annotating these nodes is minimized. Specifically, in the case study previously described, the whole process of manually identifying and adapting the changes lasted for 6 man-months, whereas by using our approach and appropriately annotating the database constructs and applying the respective policies, the same process lasted for less than half a man-month.

Event	Annotated nodes
Rename relation	Relation nodes
Add attributes	Relation/Query nodes
Delete Attributes	Attribute nodes
Rename Attributes	Attribute nodes
Domain Modification	Attribute nodes
Condition Modification	Condition nodes

Table 1: Kind of nodes annotated per event

Event	# of nodes	
	Propagate	Block
Rename relations	18	0
Add attributes	64	13
Delete Attributes	1608	92
Rename Attributes	1615	85
Domain Modification	1690	10
Condition Modification	0	21
Total Annotations	4995	221

Table 2: Distribution of annotated nodes per kind of policies and events

Scope	# of operations	
	Annotations	Policy Clauses
Query scope	486	9
Relation Scope	5180	293
Database Scope	36	2

Table 3: Operations with and without SQL extensions

6 RELATED WORK

SQL Extensions. SQL/SE is a query language extension for databases supporting schema evolution (Roddick, 1992). SQL/SE provides extensions for querying evolvable database schemas in the context of schema versioning and temporal databases. Our proposed set of extensions does not require the existence of schema versioning or the integration of time within database schema evolution. We provide rules for the transformation and adaptation of queries and views to the last valid database schema without the assumption that the transformed queries retain the same semantics. Another extension to SQL, namely SchemaSQL supports multi database querying (Lakshmanan, 2001). SchemaSQL focuses on the problem of interoperability between different schemas and their respective instances, enabling the user to express queries over different schemas.

Evolution. A number of research works are related to the problems of database schema evolution. Roddick surveys schema versioning and evolution (Roddick, 1995) and presents a categorization of the overall issues regarding evolution and change in data management (Roddick, 2000). The problem of view adaptation after redefinition is mainly investigated in (Bellahsene, 2002; Gupta, 2001), where changes in views definition are invoked by the user and rewriting is used to keep the view consistent with the database schema. Bellahsene (2002) deals also with warehouse adaptation, but only for SPJ views. Nica (1998) deals with the view synchronization problem, where the views become invalid after schema changes in the underlying base relations. We extend that work to incorporate attribute additions and the treatment of conditions. In (Fan, 2004), AutoMed, a framework for managing schema evolution in data warehouse environments is presented. They introduce a schema transformation-based approach to handle evolution of the source and the warehouse schema. Also, in (Velegrakis, 2004) they propose a framework for the management of evolution, but their model is more restrictive as it is retains the original semantics of the queries. Our work is a larger framework that allows the restructuring of the database graph (i.e., model) either towards keeping the original semantics or towards its readjustment to the new semantics.

7 CONCLUSIONS

In this paper, we have dealt with the problem of database evolution. We have provided a coherent framework for propagating potential changes of the database software to all the affected points of the system, with a limited overhead imposed on both the system and the humans, who design and maintain it. We have proposed a feasible and powerful extension to the SQL language specifically tailored for the management of evolution. The applicability and efficiency of our approach has been tested in a real-world scenario occurred in the Greek public sector.

Regarding future work, we plan to pursue our research toward the identification of patterns of evolution sequences.

ACKNOWLEDGEMENTS

Information dissemination of this work was supported by the European Union in the framework of the project “Support of Computer Science Studies in the University of Ioannina” of the “Operational Program for Education and Initial Vocational Training” of the 3rd Community Support Framework of the Hellenic Ministry of Education, funded by national sources and by the European Social Fund (ESF).

REFERENCES

Bellahsene, Z., 2002. Schema evolution in data warehouses. In *Knowledge and Information Systems* 4(2).

Fan, H., Poulovassilis, A., 2004. Schema Evolution in Data Warehousing Environments - A Schema Transformation-Based Approach. In *ER‘04*.

Gupta, A., et al., 2001. Adapting materialized views after redefinitions: Techniques and a performance study. In *Information Systems* (26).

Lakshmanan, L.V.S., Sadri, F., Subramanian, I.N., 2001. SchemaSQL – An Extension to SQL for multi database interoperability. In TODS, 26(4): 476-519.

Nica, A., Lee, A.J., Rundensteiner, E.A., 1998. The CSV algorithm for view synchronization in evolvable large-scale information systems. In *EDBT*.

Roddick, J.F., 1992. SQL/SE: A query language extension for databases supporting schema evolution. In *ACM SIGMOD Record*, 21(3): 10-16.

Roddick, J.F., 1995. A survey of schema versioning issues for db systems. In *Information Software Techn.* 37(7).

Roddick, J.F., et al, 2000. Evolution and change in data management. In *SIGMOD Record* 29(1).

Sjoberg, D., 1993. Quantifying Schema Evolution. In *Information and Software Technology*, 35(1), 35-44.

Velegrakis, Y., et al., 2004. Preserving mapping consistency under schema changes. In *VLDB Journal,* 13(3).

View publication stats