Adaptive Query Formulation to Handle Database Evolution

George Papastefanatos¹, Panos Vassiliadis², Yannis Vassiliou¹

¹National Technical University of Athens, Dept. of Electrical and Computer Eng., Athens, Hellas {gpapas,yv}@dbnet.ece.ntua.gr
²University of Ioannina, Dept. of Computer Science, Ioannina, Hellas pvassil@cs.uoi.gr

Abstract. Databases are continuously evolving environments, where design constructs are added, removed or updated quite often. Research has extensively dealt with the problem of database evolution. Nevertheless, problems arise with existing queries, mainly due to the fact that in most cases, their role as integral parts of the environment is not given the proper attention. Furthermore, the queries are not designed to handle database evolution. In this paper, we first introduce a graph-based model that uniformly captures relations, views, constraints and queries. For several cases of database evolution we present rules so that both syntactical and semantic correctness of queries are retained. To this end, we also extend the query formulation capabilities by annotating SQL queries with information concerning the semantically aware adaptation of a query in the presence of changes in the underlying database.

1. Introduction

In a typical organizational Information System, there is a variety of components inherently intertwined with each other. Traditional modelling techniques, like ER or UML diagrams, or even database schema blueprints capture only a small fragment of the overall architecture, mainly concerning its static parts (e.g., entities, classes, relations, or tablespaces) without documenting the dynamic parts (i.e., reports, forms, frequent administration queries) to a significant extent. On top of this situation, in an ever-increasing pace, the designer/administrator of the system is faced with the necessity of changing something in the overall configuration. For example, assume that an attribute has to be deleted from the database. A small change like this might impact the full range of applications and data stores around the system: queries and data entry forms can be invalidated, application programs might crash (resulting in the overall failure of a complex workflow), and several pages in the corporate Web server may become invisible (i.e., they cannot be generated any more).

Research has extensively dealt with the problem of database schema evolution so far. We can classify the different efforts, in several categories [Rodd00]: (a) evolution of object-oriented databases, mainly characterized by generalization hierarchies of classes [RaRu95, Bane87, Zica91], (b) evolution of entity-relationship diagrams, taking the special role of relationships into consideration [LiCC94], (c) schema evolution and schema versioning (a detailed survey can be found in [Rodd95]), and
lastly, (d) evolution of (materialized) views, mainly in the context of data warehouses, characterized by the duality of views, which are both queries as far as their intention is concerned and sets of tuples as far as their extension is concerned. In the latter case, the problem of view adaptation comprises two special aspects. The first concerns the view adaptation in the presence of changes in the view definition, trying to minimize the rematerialization effort [GMRR01, MoDo96], whereas the second deals with view adaptation in the presence of changes in the underlying database schema [Bell02], as well as methods for revalidating out-of-date views [NiLR98, RuLN97]. Although complex problems of evolution have been considered in the related literature, to the best of our knowledge, there is no global framework for the management of evolution in relational databases. We clarify this point with some indicative examples.

Observe the configuration of Figure 1, based on the following self-explanatory relations from [GMRR01], involving employees and the projects they work for, which also depicts a view and a query on this view. Assume the designer wishes to add an attribute to the base relation \( \text{Proj} \), say \( \text{EndDate} \). Should this change be propagated to the view or the query? Although related research can handle the deletion of attributes due to the obvious fact that queries become syntactically incorrect, the addition of information is deferred to a decision of the designer. Similar considerations arise when the \( \text{WHERE} \) clause of the view is modified. Assume that a attribute \( \text{STATUS} \) is added to all projects and the view definition is modified by incorporating the extra selection \( \text{STATUS} = 'Active' \). Can we still use the view in order to answer the query or not? The answer is not obvious, since it depends on whether the query uses the view simply as a macro (in order to avoid the extra coding effort) or, on the other hand, the query is supposed to work on the view, independently of what the view definition is [TsKl78]. The problem lies in the fact that there is no semantic difference in the way one defines the query over the view -- i.e., we define the view in the same manner in both occasions.

To deal with the aforementioned issues, our approach, in this paper, is to provide a general mechanism for performing what-if analysis for potential changes of database configurations. A graph model that uniformly models queries, views, relations and their significant properties (e.g., conditions) is introduced. Apart from the simple task of capturing the semantics of a database system, the graph model allows us to predict the impact of a change over the system. Furthermore, we provide a framework for annotating the database graph with policies concerning the behavior in the presence of hypothetical changes. Finally, rules that dictate the proper actions,
when additions or deletions are performed to relations, attributes and conditions (all treated as first-class citizens of the model) are provided. In other words, assuming that a graph construct is annotated with a policy for a particular event (e.g., a relation node is tuned to deny deletions of its attributes), the proposed framework (a) performs the identification of the affected part of the graph and, (b) if the policy is appropriate, automates the readjustment of the graph to fit the new semantics imposed by the change. To alleviate the designer from the burden of manually annotating all graph constructs, a simple extension of SQL with clauses concerning the evolution of important constructs is proposed.

This paper is organized as follows. In Section 2, we present the graph model for databases. A framework of graph annotations and readjustment automation for database evolution is proposed in Section 3. Implementation issues and SQL extensions are discussed in Section 4. Section 5 sketches related work. Finally, in Section 6 we conclude our results and provide insights for future work.

2. Modeling SQL constructs as graphs

In this section, we propose a graph modeling technique that uniformly covers relational tables, views, database constraints and SQL queries as first class citizens. The proposed technique provides an overall picture not only for the actual database schema but also for the architecture of a database system as a whole, since queries are incorporated in the model. Moreover, we distinguish the following essential components, which are included in our model: relations, conditions (covering database constraints and query conditions), queries and views. The proposed modeling technique represents all the aforementioned database parts as a directed graph. Graphs are employed as a modeling technique because they can address the large size and complexity that characterize a database schema. In the rest of this section we discuss how we model each of these constructs as well as the overall graph of the database.

**Relations.** Each relation \( R(A_1, A_2, ..., A_n) \) in the database schema is represented as a directed graph, which comprises (a) a relation node, representing the relation schema, (b) \( n \) attribute nodes, one for each of the attributes, and (c) \( n \) schema edges directing from the relation node towards the attribute nodes, indicating that the attribute belongs to the relation. We call these relationships, schema relationships.

**Conditions.** Conditions refer both to constraints of the database schema and selection conditions of queries and views. We consider three classes of atomic conditions (a) \( A \ op \ constant \), (b) \( A \ op \ A' \), where \( A, A' \) are attributes of the underlying relations and \( op \) is a binary operator (i.e., \(<, >, =, \leq, \geq, != \), etc.) and lastly (c) \( A \ op \ Q \), where \( A \) is an attribute of the underlying relations, \( Q \) is a query and \( op \) is a binary operator (i.e., \( IN, EXISTS, ANY \), etc.).

In our graph model, 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 two operand nodes of the conjunct clause through the respective operand edges. Composite conditions are easily constructed by tagging the condition node with an
AND or an OR symbol and the respective edges (possibly more than two) to the
conditions composing the composite condition.

Well-known constraints of database relations are easily captured by this modeling
technique. Foreign keys are subset relations of the source and the target attribute,
range constraints are simple value-based conditions. Unique-value constraints (hence,
primary keys) require a different modeling – in the context of this paper we explicitly
represent them through a dedicated node and a single operand node.

Queries and Views. Queries and views are mapped to graphs in a similar manner.
The graph representation of a Select-Project-Join-Group By (SPJG) query involves a
new node representing the query, named query node, and attribute nodes
Corresponding to the schema of the query. The query graph is therefore a directed
graph connecting the query node with all its schema attributes, via schema edges. In
order to represent the relationship between the query graph and the underlying
relations, we make the convention that each query is decomposed into the following
Essential parts: the SELECT, FROM, WHERE, and GROUP BY parts, each of which is
Eventually mapped to a subgraph. Based on the above, the graph representation of a
query is the composition of these four subgraphs, which are defined as follows:
- Each query/view is assumed to own a schema that comprises the named
  attributes 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 edges, directing from the
  query attributes towards the relation attributes.
- The FROM part of a query can be regarded as the relationship between the
  query and the relations involved in this query. Therefore, for each relation
  included in the FROM part, a from edge, directing from the query node
  towards the relation node, is used.
- We assume that the WHERE clause of a query is in conjunctive normal form
  (i.e., it comprises a conjunction of disjunctions of conditions). Having
  explained conditions already, we can construct the graph corresponding to
  the WHERE clause of a query by introducing a directed where edge starting
  from the query node towards the operator node corresponding to the
  conjunction of the highest level.
- Concerning nested queries, we extend the WHERE subgraph of the outer query
  by (a) constructing the respective graph for the subquery, (b) employing a
  separate operator node for the respective nesting operator (e.g., IN operator),
  and (c) employing two operand edges directing from the operator node
  towards the two operand nodes (the attribute of the outer query and the
  respective attribute of the inner query) in the same way that conditions are
  represented in simple SPJ queries.
- For the representation of aggregate queries, we employ two special purpose
  nodes: (a) a new node denoted as GB, to capture the set of attributes acting as
  the aggregators and (b) one node per aggregate function (e.g., COUNT, SUM,
  MIN, etc.) labelled with the name of the employed aggregate function. For
  the aggregators, an edge is used, directing from the query node towards the
  GB node, and labelled <group-by>, indicating group-by relationship. Then,
  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 tagged according to the order of the aggregators (i.e., 1 for the first aggregator, 2 for the second, etc.). 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 labelled `<map-select>`, as this relationship indicates mapping of the query attribute to the corresponding relation attribute through the aggregate function node.

The following example demonstrates the proposed graph representation. In Figure 2, we present the graph representation for the following aggregate query:

```
Q: SELECT EMP.Emp#, Sum(WORKS.Hours) as T_Hours
    FROM EMP, WORKS
    WHERE EMP.Emp# = WORKS.Emp#
    GROUP BY EMP.Emp#
```

![Fig. 2. Graph representation of aggregate query [PKV05]](image)

As far as modification queries are concerned, there is a straightforward way to incorporate them in the graph, too. Still, their behavior with respect to adaptation to changes in the database schema can be captured by SELECT queries:

(a) INSERT statements can be dealt as simple SELECT queries. The general syntax of an INSERT statement can be expressed as:

```
INSERT INTO table_name(attribute_set)
VALUES (value_set)
```

The equivalent SELECT query, which corresponds to the above INSERT statement, comprises only a SELECT and a FROM clause, projecting the same attribute set with the attribute set of the INSERT statement, i.e.,:
SELECT (attribute_set) FROM table_name

Equivalence means that evolution changes (e.g., attribute addition) in the underlying relation of an INSERT statement can be handled in the same way we handle the equivalent SELECT query.

(b) DELETE statements can be treated as SELECT * queries comprising a WHERE clause. The general syntax of a DELETE statement can be expressed as:

```
DELETE FROM table_name
WHERE condition_set
```

Again, the equivalent SELECT query, which corresponds to the above DELETE statement, comprises a SELECT clause, projecting all the attributes (i.e., *) of the table, as well as a WHERE clause, containing the same set of conditions with that of the DELETE statement, i.e.:

```
SELECT * FROM table_name WHERE condition_set
```

(c) Finally, UPDATE statements can be treated as SELECT queries comprising a WHERE clause. The general syntax of an UPDATE statement can be expressed as:

```
UPDATE table_name
SET (attribute_set) = (value_set)
WHERE condition_set
```

The equivalent SELECT query, which corresponds to the above UPDATE statement, comprises a SELECT clause, projecting the attribute set which is included in the SET clause of the UPDATE statement, as well as a WHERE clause, containing the same set of conditions with that of the UPDATE statement, i.e.:

```
SELECT attribute_set FROM table_name WHERE condition_set
```

Still, we stress that this is simply a concession that we do for the purpose of adaptation to database evolution and not a complete modeling approach.

3. Adapting queries and views to database evolution

In this section, we formulate a set of rules that allow the identification of the impact of changes to database relations and attributes and propose an automated way to respond to these changes. The impact of the changes involves the software built around the database, mainly queries, stored procedures, triggers etc., which are affected in two ways: (a) syntactically, meaning that it is possible that the execution of the code will produce a compilation/execution failure and (b) semantically, meaning that a change in the database can affect the semantics of the software built around it. We abstract software modules where SQL is embedded within a host language and treat every such module as a set of SQL queries. The rules that we propose are annotations of the graph that determine the policy to be followed in the case of an event that modifies the graph. The combination of events and annotations determines the policy to be followed for the handling of the potential change.

The annotated graph can be part of a metadata repository that is embedded in, or externally accessed from a what-if analysis module that notifies the
designer/administrator on the effect of a potential change and the extent to which the modification to the existing code can be fully automated, in order to adapt to the change. To the best of our knowledge, no such adaptation takes place in the current DBMS; still, there are several cases where this adaptation can be easily predicted and automated. To this end, we build upon the work of [NiLR98] and discuss annotations of our graph-model that regulate the way the queries and views are adapted to potential changes. These graph annotations are easily mapped to SQL extensions, as we will discuss in the following section.

3.1 The general framework for schema evolution

The main mechanism towards handling schema evolution is the annotation of the constructs of the database graph (i.e., nodes and edges) with operators that handle schema evolution. Since we aim to provide a framework for what-if analysis, each such construct is enriched with policies that allow the designer to specify the behavior of the annotated construct whenever events that alter the database graph occur. The combination of an (hypothetical) event with the policy that has been determined by the designer/administer triggers the execution of the appropriate action that either blocks the event, or reshapes the graph to adapt to the proposed change.

The space of potential events is quite simple and comprises the Cartesian product of two subspaces; specifically the space of hypothetical actions (addition/deletion) over specific graph constructs (relations, attributes and conditions).

For each of the above events, the administrator annotates the appropriate graph constructs with policies that dictate the way they will regulate the change. Two kinds of policies are defined: (a) propagate the change, meaning that the graph must be reshaped to adjust to the new semantics incurred by the event and (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 [NiLR98, VeMP04].

Our framework prescribes the reaction of the parts of the system affected by a hypothetical schema change based on their annotation with policies. The correspondence between the examined schema changes and the parts of the system affected by each change is shown in Table 1.

<table>
<thead>
<tr>
<th>Event on database schema</th>
<th>Relation/View</th>
<th>Relation/View</th>
<th>Relation/View</th>
<th>View/Query</th>
<th>View/Query</th>
</tr>
</thead>
<tbody>
<tr>
<td>Add</td>
<td>Attribute</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Condition</td>
<td>✓</td>
<td>✓</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Delete</td>
<td>Attribute</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td></td>
<td>Condition</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td></td>
<td>Relation/View</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Update</td>
<td>Condition</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
</tbody>
</table>

Table 1: Parts of the system affected by each event.
The definition of policies on each part of the system affected by a specific event involves the annotation of the respective construct (i.e. node or edge) in our graph framework. Table 2 presents the allowed annotations of graph constructs for each kind of event. The annotation is performed as follows. For the constructs belonging to the database schema (i.e. relation, attribute and condition node) we annotate the respective nodes, whereas for the constructs belonging to the queries/views affected by the schema change, we primarily annotate the edges connecting the respective nodes with the database node. Specifically, (a) we annotate the \textit{FROM} edges connecting query nodes with relation nodes for policies defined on views/queries, (b) we annotate the \textit{map-select} or \textit{group by} edges for rules defined on query attribute nodes, and (c) we annotate \textit{operand} edges for rules defined on query condition nodes.

<table>
<thead>
<tr>
<th>Event on database schema</th>
<th>Nodes Annotated with Policies</th>
<th>Edges Annotated with Policies</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Relation</td>
<td>Attribute</td>
</tr>
<tr>
<td>Add</td>
<td>Attribute</td>
<td>√</td>
</tr>
<tr>
<td></td>
<td>Condition</td>
<td>√</td>
</tr>
<tr>
<td>Delete</td>
<td>Attribute</td>
<td>√</td>
</tr>
<tr>
<td></td>
<td>Condition</td>
<td>√</td>
</tr>
<tr>
<td></td>
<td>Relation/View</td>
<td>√</td>
</tr>
<tr>
<td></td>
<td>Condition</td>
<td>√</td>
</tr>
</tbody>
</table>

Table 2: Annotation of graph constructs with policies for each event

The mechanism that determines the reaction to the change can be formulated as:

- Given an \textit{event} altering the database schema, the affected structures are determined (i.e. queries, views, conditions in queries, attributes in queries, etc.).
- The \textit{policy} of each affected structure dictates how the graph will be affected by this event -- that is whether this change will be \textit{propagated to} or \textit{blocked} from this construct. We also allow the administrator to interactively decide what will eventually happen (\textit{prompt}).
- According to the prevailing policy, the corresponding \textit{action} dictated by Table 3, is (automatically, if possible) taken to adjust the affected constructs to the new schema. Related work is orthogonal in the case of query rewriting, which can be performed using any available method [NiLR98, VeMP04].
- The procedure is repeated, taking the possible readjustment as a new event over the database graph. Since the graph is finite and no cycles are found, the procedure eventually terminates. Moreover, this stepwise action resolution exploits the principle of locality, allowing the interactive resolution of actions one at a time, in the case where a \textit{prompt} policy has been determined.

In Table 3 we present an overall picture of the framework. Potential \textit{events} tested by the designer/administrator are depicted in the first column of the Table. The two rightmost columns depict the possible policies that the administrator could have set and the actions dictated by our framework. For each event, the candidate modules for
change are also presented as well as the type of impact (i.e., semantic or syntactical) the change has on them.

<table>
<thead>
<tr>
<th>Event on source schema</th>
<th>Candidate Modules For Change</th>
<th>Impact</th>
<th>Prevailing Condition</th>
<th>Action</th>
</tr>
</thead>
</table>
| **Attribute**          | 1. Queries/views that must include the added attribute in the SELECT clause 2. Queries/views with SELECT * clause that must exclude the added attribute | Semantic | Policy = Propagate Include attribute in SELECT clause  
Policy = Block Rewrite SELECT clause excluding added attribute  
Policy = Prompt One of the above |        |
| **Add**                | Queries/views referring to the relation/view over which the condition is added | Semantic | Policy = Propagate Leave query intact  
Policy = Block Retain old view (without the added condition) and all queries with block policy refer to the old view |        |
| **Condition**          | Queries/views referring to the relation/view from which the condition is removed | Semantic | Policy = Propagate Leave query intact  
Policy = Block Rewrite properly query/view in order to be valid  
Policy = Prompt One of the above |        |
| **Delete**             | Queries/views referring to relation/view | Syntactical, Semantic | Policy = Propagate Remove relation from query/view definition (i.e., FROM clause) along with the attributes and conditions involving this relation (i.e., SELECT, WHERE, GROUP BY clauses)  
Policy = Block Rewrite properly query/view in order to be valid  
Policy = Prompt One of the above |        |
| **Relation / view**    | Queries/views referring to relation/view | Syntactical, Semantic | Policy = Propagate Leave query intact  
Policy = Block Retain old view (including the original condition) and all queries with block policy refer to the old view |        |
| **Update**             | Queries/views referring to relation/view of which the condition is modified | Semantic | Policy = Propagate Leave query intact  
Policy = Block Retain old view (including the original condition) and all queries with block policy refer to the old view |        |

Table 3: Actions determined by combinations of events and policies examples

---

1 For the case of condition addition to relation, rewrite query in order to exclude, if present, the added condition from its WHERE clause

2 For the case of condition deletion from relation, rewrite query in order to include the deleted condition in its WHERE clause
It is worth noting here that our framework is complete with respect to the space created by the dimensions hypothetical actions = \{addition/deletion\}, graph constructs sustaining changes = \{relations, views, attributes and conditions\} and policies = \{propagate, block\}. The only omitted case is the addition of relations, which does not seem to have any immediate impact to the environment of the database per se. Naturally, the space can be extended to cover more complex cases. Still, the framework is extensible, in the sense that any possibly change like for example, the testing of composite series of actions (e.g., the addition of a relation along with the redefinition of a query), or the introduction of new graph constructs (e.g., for semi-structured data), or possibly the extension of policies, requires the population of Table 3 with the appropriate rows that dictate how the combination of events with policies are translated to specific actions to be performed by the system. For example, in the long version of this paper [PaVV05], we already give some results on updates of conditions, which for the moment, seems to be the only interesting case where an update to a construct (e.g., a relation) is not resolved to the addition or deletion of one of its constituents (e.g., an attribute).

An example for each of the examined events follows to demonstrate the proposed framework.

**Add Attribute Event**

In order to give the flavor of our approach, we start with the simplest example of an SPJ query, specifically the query \( \text{SELECT * FROM EMP} \). Assume now that the designer extends the relation EMP with a new attribute PHONE. When an attribute is added to a relation of the underlying schema, we need to identify the queries to which the addition must be reflected and propagated. Both the current database systems and the state of the art in research do not react to this change, but rather, they let the designer/administrator propagate the change to any queries he thinks they should be modified to include the extra attribute. Eventually, he is obliged to rewrite the queries, which are to be modified, by adding appropriately the extra attribute to their syntax. This treatment is mainly due to the fact that (a) the addition of an attribute does not syntactically affect the involved queries (i.e., the existing queries can still be executed without any problem) and (b) up to now, we do not have any mechanism to tell the system that once an attribute is added to a relation, it must also be added to certain queries that access this particular relation. In our case, there are two possibilities:

(i) The \(*\) notation signifies the request for any attribute present in the schema of relation EMP (in other words, the \(*\) shortcut can be treated as "return all the attributes that EMP has, independently of which these attributes are"). In this case, the query must also retrieve the new attribute PHONE.

(ii) The \(*\) notation acts as a macro for the particular attributes that the relation EMP originally had. In this case, the addition to relation EMP must not be further propagated to the query.

Based on these, in the presence of an addition of an attribute, an impact prediction system must trace all queries and views that are potentially affected and ask the designer/administrator to decide upon which of them must be modified to incorporate the extra attribute. Naturally, we can do better than that by extending the current modeling. For each element affected by the addition, we annotate its respective graph construct (i.e., node, edges) with the aforementioned policies. According to the policy
defined on each construct the respective action is taken to adjust the query to the change. Therefore, for the event of attribute addition, the policies defined on the query and actions taken according to each policy are:

- **Propagate attribute addition.** In this case, when an attribute is added to a relation appearing in the FROM clause of the query, this addition must be reflected to the SELECT clause of the query.

- **Block attribute addition.** In this case, the aforementioned case (ii) is assumed, i.e., the addition to the relation must be ignored and the query is immune to the change. The SELECT * clause must be rewritten to SELECT A1,...,An without the newly added attribute.

- **Prompt.** In this case (default, for reasons of backwards compatibility) the designer/administrator must handle the impact of the change manually, like what happens now in database systems.

In terms of our graph modeling, the extension of the proposed graph model simply involves annotating the FROM edge (edge directing from the query node towards the relation node) with an additional indication for propagating or blocking addition in the SELECT clause of the query resulting from additions of attributes in the underlying relation. If a FROM edge is not tagged with this additional information, then the default case is assumed and the designer/administrator is prompted to decide.

For example, in the aforementioned addition of attribute PHONE to relation EMP, the graph of the query SELECT * FROM EMP is shown in Figure 3. The annotation of the from edge as propagating addition indicates that the addition of PHONE node will be propagated to the query and the new attribute is included in the SELECT clause of the query.

![Fig. 3: Propagating addition of attribute PHONE to the schema of the query](image)

With respect to attribute addition event, observe that only FROM edges are annotated with policies for capturing this event, i.e., annotating map-select edges are not allowed by the framework as they can not capture the addition of an attribute to a relation, but only changes affecting existing attributes.

**Add Condition Event**

Conditions are first class citizens of our environment. As already mentioned, we employ conditions in two occasions: (a) to capture constraints of the database, e.g., foreign key or range constraints and (b) to capture selection conditions within queries (practically involving value-based selections and joins). Therefore, it is meaningful to
study the impact of the addition of a constraint to the system in the following occasions:

(a) Addition of an extra constraint to a view definition (affecting all queries employing the view).

(b) Addition of an extra constraint to a relation (affecting all queries and views employing the relation).

Assume, for example, the configuration of Figure 4a involving a view with a simple selection condition, characterizing employees near the age of retirement. A query Q5 returns all employees near the age of retirement whose salary is high (SALARY≥100K). Assume, now, that we modify the selection condition of the view, adding the new selection condition ADDRESS LIKE '%HILL%', to restrict the involved employees only to the set of people living in 'North Hill' or 'South Hill'. Again, the query suffers from lack of semantics: is the query defined over the view (meaning that whatever the view definition is, this is the set of employees that the query refers to) or does the query employ the view definition as a macro? As in the case of attribute addition, we can annotate the graph constructs capturing this event, e.g., the from edge from the query to the view with one of the following three tags:

- **Propagate condition addition.** In this case, the query is defined over the view, independently of what the view definition is and the addition of the condition to the view definition does not result in an impact to the query.

- **Block condition addition.** In this case, the view is employed as a macro for the original expression and the query has to retain its original semantics. In this case, the old view is retained and a new view with the extra condition is created.

- **Prompt.** In this case (default, for reasons of backwards compatibility), the designer/administrator must handle the impact of the change manually, like what happens now in database systems.

The situation is similar when, instead of a simple condition involving a value-based selection, a join condition is defined. In this case, the view definition must be extended to incorporate extra relations in the FROM clause as well as the extra join condition at the WHERE clause. Observe Figure 4b, where the original definition of interesting employees is expanded to restrict the result set only to the employees near retirement who have worked at the ‘East Hill’ location. Then, the new view involves two extra join conditions for the new relations and a new selection condition for the ‘East Hill’ location. In this case, the query might react to the change as in the previous case of the addition of a simple selection condition.

The second possible occasion is when a constraint, i.e., a range condition is added to a relation (but not a view). For example, assume that the condition AGE>=0 is
added to the EMP relation. It is possible that there exist some queries already performing this test that need no longer continue doing so. In this case, we might wish to remove the extra constraint from the respective queries. The following policies can be assumed for views or queries directly defined over the relation.

- **Propagate condition addition.** No change happens to the query/view.
- **Block condition addition.** In this case, the identical condition is removed from the query/view definition. All queries defined over the affected views do not lose semantics, since the policy assumes that the relation employed is already performing the request check.
- **Prompt.** In this case, the designer/administrator must handle the impact of the change manually, like what happens now in database systems.

### Delete Attribute Event

Assume an attribute is dropped from a relation. All queries referring to this attribute (in the SELECT clause, in a WHERE condition or in a GROUP BY expression) become syntactically invalid and require rewriting, either through an automatic mechanism [Bell02, NiLR98], or through an administration activity. The state of the art in research treats the automatic propagation of the removal of attribute in two ways. In the first line of research, a query is properly rewritten in order to remove the deleted attribute from all the SELECT, WHERE and GROUP BY clauses [Bell02, NiLR98], resulting in a semantic as well as a syntactical transformation of the query. In the second line of research, the query is properly rewritten in order to refer to the removed attribute via other information sources (i.e., other relations retaining the same information) [NiLR98], resulting in a syntactical but not necessarily in a semantic transformation of the query. However, in many cases, the transformation of a query is not feasible (e.g., complex queries, not available information sources, etc.) or desirable (i.e., the rewritten query does not semantically fulfill the desired requirements) and therefore the reaction of the administrator/designer is required.

In the context of our framework, the administrator/designer may define different policies to any/all graph constructs possible affected by such change, and annotate them respectively. For the case of attribute deletion, the annotated graph constructs may comprise the attribute node to be deleted, the relation node containing the attribute, the query node accessing this relation, the attribute node of the query that is mapped to this attribute and possibly the condition node of the query that refers to this attribute. Therefore, for the event of attribute deletion, the policies defined on graph constructs and actions taken according to each policy are:

- **Propagate attribute Deletion:** In that case, the removal of attributes is propagated to the SELECT clause and to all WHERE conditions, in which the specified attribute is involved resulting in removal of this attribute from the query.
- **Block attribute Deletion:** In that case, the removal of attributes must not be propagated to the query, and therefore the query must be properly rewritten in order to be valid again.
- **Prompt.** In this case (default, for reasons of backwards compatibility), the designer/administrator must handle the impact of the change manually, like what happens now in database systems.
In Figure 4, we present the graph for the following query, annotated with policies for the event of an attribute deletion. Let the query be:

\[
Q: \quad \text{SELECT Emp#, Salary, \text{Sum(Hours)} as THours} \\
    \text{FROM EMP, WORKS} \\
    \text{WHERE EMP.Emp# = WORKS.Emp#} \\
    \text{GROUP BY Emp#, Salary}
\]

The event of deleting the attribute *Hours* belonging to *Works* relation is captured by a policy defined on the *Hours#* attribute of *Works* relation (annotation of *Hours* node) which propagates the deletion from all queries/view referring to this attribute. The annotation of the *Hours#* node as *propagate deletion* indicates that the deletion of *Hours#* attribute must be propagated to the query, and therefore the query must remove this attribute from its clauses. A second policy, however, is defined on the query, (annotation of the *from* edge connecting query with relation *WORKS*) indicating that the *any* attribute deletion from relation *WORKS* must be blocked, including *Hours#* attribute. The annotation of the *from* edge as *block deletion* indicates that the deletion of *Hours#* attribute must not be propagated to the query, and therefore the query must be properly rewritten to be valid again.

According to Table 2, additional graph constructs, which can be annotated with policies for attribute deletion event, comprise *map-select* edges, the *relation* node containing the attribute, or lastly the *operand* edge connecting the operand node with the attribute to be deleted. In section 3.2 a mechanism for resolving policies conflicts is presented for the case that two policies are in conflict with each other.

![Diagram](image_url)

**Fig. 5:** Blocking deletion of attribute *Hours* from query
Delete Condition Event

The treatment of condition deletion is practically the reverse problem of condition addition. In the case where a condition is removed from a relation, i.e., in the case where a constraint is dropped, then, it possible that some queries need to incorporate the test of the condition in their definition. The same applies for the case of the removal of a selection condition from a view definition: the queries accessing the view must adapt to this change in one of the previously defined ways.

We will employ the examples of the previous subsection, in the reverse order. Assume that the view ENR_V01 of Figure 4b exists in the system and the designer decides to relax its constraints and obtain the view EmpsNearRetirement. Then, the query Q1, needs to adapt to the change. Again, the different policies involve either the immunity to the modification (assuming that the query operates over the view, independently of the view definition), or the rewriting of the query, in order to retain the previous result set. In general, the different policies, involving the annotation of the from edge between the query and the view are:

- **Propagate condition deletion.** In this case, the query is defined over the view, independently of what the view definition is, and the removal of the condition from the view definition does not result in an impact to the query.
- **Block condition deletion.** In this case, the view is employed as a macro for the original expression and the query has to retain its original semantics. In this case, the old view is retained and a new view without the removed condition is created. All queries with ‘propagate’ semantics are redefined over the new view, and all queries with ‘block’ semantics remain defined over the old view.
- **Prompt.** In this case (default, for reasons of backwards compatibility), the designer/administrator must handle the impact of the change manually, like what happens now in database systems.

The situation is similar for the case where the selection condition removal involves the selection of a join operation as well as the removal of the respective relations from the FROM clause of a query or view.

The second possible occasion is when a range condition, signifying a constraint, is removed from a relation. In this case, we might wish to extend any queries originally ignoring the built-in constraint. The following policies can be assumed for views or queries directly defined over the relation.

- **Propagate condition deletion.** No change happens to the query/view.
- **Block condition deletion.** In this case, an identical condition is attached to the query/view definition.
- **Prompt.** In this case, the designer/administrator must handle the impact of the change manually, like what happens now in database systems.

As in previous cases, to handle the treatment of such changes, the graph is extended with extra information. Any of the graph constructs capturing this event, e.g., the from edge from the query to the relation/view are annotated with one of the above policies.
Delete Relation/View Event

Handling deletion of whole relation or views is quite similar to the case of an attribute deletion. A relation is explicitly referenced in the FROM clause of the query as well as in its rest clauses (i.e., WHERE, SELECT, GROUP BY, etc.) implicitly via the attributes of this relation. Therefore, the event deletion of a relation can be handled as a set of attribute deletion events. In general, the different policies, involving the annotation of the respective nodes and edges for the event of relation deletion are:

- Propagate relation Deletion: In that case, the removal of a relation is propagated to the query resulting in the removal of this relation from the FROM clause query. This event in turn triggers the attribute deletion and condition deletion events resulting in the removal of the relation attributes from all query clauses and conditions.
- Block relation Deletion: In that case, the removal of a relation must not be propagated to the query, and therefore the query must be properly rewritten in order to be valid again.
- Prompt. In this case (default, for reasons of backwards compatibility), the designer/administrator must handle the impact of the change manually, like what happens now in database systems.

Update Condition Event

Updates to conditions, are treated in the same way that additions and deletions of conditions have been dealt with. As far as selection conditions of views are concerned, an update at the components of a condition is either ignored by the queries accessing the view, or dealt through rewriting of the queries to retain their previous semantics. This involves both value-based and join-conditions. As far as the updates to constraints attached to relations are concerned, again, the reaction to change is the same with the previous cases: either the queries ignore the change (propagate policy), or they are rewritten in the presence of a new constraint (block policy). The annotation of the respective nodes and edges is similar with the previous cases.

3.2 Conflict resolution in the context of schema evolution

It is possible, as mentioned in the previous section, that the policies defined over the different elements of the graph do not always align towards the same goal. For example, consider the case where a view $V$ is defined over a database relation $R$ and a query $Q$ accesses the view $V$. Assume that $R$ is annotated with the policy:

\[
\text{ON attribute addition THEN propagate (i.e., change to the graph)}
\]

whereas $V$ is annotated with the policy

\[
\text{ON attribute addition (i.e., on underlying relations) THEN block}
\]

Independently of what the relation policy is, the attribute propagation must be blocked at the view level, and not further propagated towards the query. Observe that
there is no inconsistency here: the original database designer has configured the relation \( R \) to propagate its changes. Subsequently, the developers who have constructed view \( V \) and query \( Q \), have set the view \( V \) to retain its semantics, independently of what happens to \( R \). At the same time, other views over \( R \) might be re-adjusted to reflect the novel structure of \( R \).

Fig. 6: Hierarchy for policy conflict resolution

The general guideline for handling policy conflicts is simple and follows the following rule: the higher a module is at the hierarchy of Fig. 4, the stronger its policy is. Specifically, we perform the following steps when different policies for the same event have been assigned to different graph modules:

1. Whenever a change takes place in a (lower level) module:
   - The change is applied to the module according to its policy;
   - All (higher level) modules of the graph that can possibly be affected by the change are determined; we will name these modules as candidates for readjustment.
   - The change is propagated or blocked towards these upper modules that are candidates for readjustment over the new structure of the modified module.

2. If an upper module does not have a policy to handle the propagated change, then it abides by the policy dictated by the lower level. For example, if a view is modified and a query accessing it has no policy, then the query is aligned with the policy of the view.

3. If an upper module has a policy of its own, then it overrides the policy dictated by the lower module. For example, if the addition of an attribute to a relation dictates the propagation of the change and a query accessing the relation has a policy block, then the query remains the same, independently of the relation’s policy.

4. For elements belonging to the same module, the above guidelines also stand. For example, if the policy for an attribute deletion defined on an attribute dictates that the change is blocked and the policy for the same event defined on the relation
dictates that the deletion is propagated then the attribute’s policy overrides the relation’s policy.

5. The procedure is repeated from Step 1, taking the new readjustment as a new event over the database graph. Since the graph is finite and no cycles are found, the procedure eventually terminates.

6. The user’s choice can override all other rules. For the example of rule 3, if the user specifically dictates that the query must be readjusted to reflect the table’s modification, then the query’s policy is overridden. Naturally, the user must have the proper access rights to modify the policy of an element.

Fig. 7: Resolution of conflict of policies on the graph. Dotted lines are for the constructs with policies that determine what happens on attribute deletion.

Assume, for example, the configuration of Figure 5 involving a view, namely EmpsNR, characterizing employees near the age of retirement (e.g., \texttt{SELECT * FROM EMP WHERE AGE>60}). A query \( Q \) is defined over this view and returns all employees near the age of retirement whose salary is high (\texttt{SELECT * FROM EmpsNR WHERE SALARY>100K}). Assume, now, that for the event of an attribute deletion (i.e., the \texttt{Salary} of \texttt{EMP} relation) the following policies have been defined on the graph.

1. On \texttt{Salary} attribute of \texttt{EMP} relation: ON \texttt{attribute deletion} THEN propagate (the node of \texttt{Salary} attribute is annotated).
2. On \texttt{EMP} relation: ON \texttt{attribute deletion} THEN block (the node of \texttt{EMP} relation is annotated).
3. On \texttt{EmpsNR} view: ON \texttt{attribute deletion} THEN propagate (the \texttt{FROM} edge between the view and \texttt{EMP} relation is annotated).
4. On \( Q \) query: ON \texttt{attribute deletion} THEN block (the \texttt{FROM} edge between the query and \texttt{EmpsNR} view is annotated).

The above policies are defined on different graph elements, however, they capture the same event. Following the proposed guidelines and the hierarchy of Fig. 4, the resolution of the policy conflict is performed as follows:

1. There is a conflict between the default policy of relation \texttt{EMP} (block deletions of \texttt{EMP} attributes) and the customized policy for attribute \texttt{Salary} (propagate).

Based on the aforementioned methodology, the customized attribute policy
overpowers the relation policy and propagates the attribute deletion to the relation. Attribute EMP.Salary is removed from the graph.

2. The relation propagates the deletion to the view EmpsNearRetirement that has no conflict with the event (propagate) and also removes the attribute EmpeNearRetirement.Salary from the graph.

3. The query Q is notified on the attribute deletion and adjusts itself according to its policy (block). The deletion is blocked from the subgraph of the query, with the result of attribute Q.SALARY and the condition (SALARY>100K) having dangling edges. Assuming that a rewriting possibility exists [NiLR98, Vemp04] the query can be rewritten; otherwise the designer observes that the planned change results in a syntactical error and an inconsistent graph.

4. SQL Extensions

In this section, we present SQL extensions that enable the implementation of the previous techniques for the management of evolution. For lack of space, we omit the discussion over the necessary extensions at the system catalog that are required in order to keep track of the semantics of the previous section. In order to be able to extend a system catalog with extra information regarding evolution purposes, we need to provide the following extensions to SQL:

(a) a facility for the management of conditions as first class citizens, and,
(b) a facility for annotating the components of the graph with evolution policies.

Conditions. Handling conditions is easy. We employ a simple name for each condition:

\[ \text{CREATE CONDITION} \ <\text{condition}> \ AS \ <\text{expression}> \]

For example, 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.

\[ \text{CREATE CONDITION} \ \text{Emp_Age_Cond} \ AS \ \text{AGE}>50 \]
\[ \text{CREATE CONDITION} \ \text{Works_Emp_FK} \ AS \ \text{WORKS.EMP# IN EMP.EMP#} \]
\[ \text{CREATE CONDITION} \ \text{Works_Emp_J} \ AS \ \text{WORKS.EMP#}=\text{EMP.EMP#} \]

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/administrator to abide by the above syntax.

A query can also be written by employing conditions in the WHERE clause. For example, a query \text{SELECT} * \text{FROM} EMP where \text{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.
**Evolution annotations.** 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 <element> THEN <policy>
```

where event can take the values of \{Add Attribute, Delete Attribute, Delete Relation, Add Condition, Delete Condition, Update Condition\}, element refers to the database part on which the event occurs and policy takes the values \{propagate, block, prompt\}.

We extend DDL statements (i.e., `CREATE`, `ALTER`) with the above syntax for imposing policies on relations, attributes and conditions. We employ the following DDL statement of creating a table, namely `WORKS`, to demonstrate the proposed extension:

```sql
CREATE TABLE WORKS
(EMP# <datatype>,
 PROJECT# <datatype>,
 HOURS <datatype>,
 CONSTRAINT FK_1 FOREIGN KEY EMP# REFERENCES EMP.EMP#,
 CONSTRAINT FK_2 FOREIGN KEY PROJECT# REFERENCES PROJECT.PROJECT#)
```

- **Imposing policy on relations:** The definition of the relation `WORKS` is extended with the policy that propagates the addition of an attribute. Then the DDL statement is extended as:

```sql
CREATE TABLE WORKS
(EMP# <datatype>,
 ...
 ON attribute addition TO EMP THEN propagate)
```

The above syntax corresponds to the annotation of the respective relation node.

- **Policy on attributes:** For the case of defining a policy on an attribute itself (i.e., the case of an attribute deletion), the DDL statement is extended as follows:

```sql
CREATE TABLE WORKS
(EMP# <datatype>,
 ...
 ON attribute deletion TO EMP.EMP THEN block)
```

The above syntax corresponds to the annotation of the respective attribute node.

- **Policy on Conditions:** Lastly, DDL statement is similarly extended to include policies defined on a relation constraint, that is:

```sql
CREATE TABLE WORKS
(EMP# <datatype>,
 ...
 ON condition deletion TO EMP.FK THEN block)
```

The above syntax corresponds to the annotation of the respective condition node.

For the case of imposing policies on queries, we employ the following three examples to demonstrate the above extension in SQL. In the first example, a query is appropriately defined over an arbitrary view `V` allowing an addition of a condition in the view to be ignored by the query. Assume that the syntax of the query is:
SELECT Emp#, NAME, AGE
FROM V
ON condition addition TO V THEN propagate

The above syntax corresponds to the annotation of the from edge connecting the query with the view. In the case of adding a new condition to view V then there will be no impact on the query (neither syntactically nor semantically) and the query will continue referring to the changed view.

The second example demonstrates the case of an attribute deletion from relation EMP and how a query defined over this relation accommodates this change. Assume an attribute, say AGE is deleted from relation EMP and we want the deletion to be propagated to the query. Then, the query syntax is:

SELECT Emp#, NAME, AGE
FROM EMP
WHERE AGE>50
ON attribute deletion TO EMP.AGE: propagate

According to the above syntax, if AGE is deleted from EMP then it will also be deleted both from SELECT and WHERE clause of the query. The above syntax also corresponds to the annotation of the map-select, group by and operand edges of the query graph.

5. Related Work

Evolution. A number of research areas are related to the problems of database schema evolution. [Rodd95] presents a survey on schema versioning and evolution, whereas in [Rodd00] a categorization of the overall issues regarding evolution and change in data management is presented. The problem of view adaptation after redefinition is mainly investigated in [GMRR01, MoDo96, Bell02], where changes in views definition are invoked by the user and rewriting is used to keep the view consistent with the data sources. [NiLR98] deals with the view synchronization problem, which considers that views become invalid after schema changes in the underlying base relations. The authors extend SQL, enabling the user to define evolution parameters characterizing the tolerance of a view towards changes and how these changes will be dealt with during the evolution process. Also, the authors propose an algorithm for rewriting views based on interrelationships between different data sources. With respect to the related work, our work in this paper builds mostly on the results of [NiLR98], by extending it to incorporate attribute additions and the treatment of conditions. The treatment of attribute deletions in [NiLR98] is quite elaborate; we confine to a restricted version to avoid overcomplicating both the size of requested metadata and the language extensions. Still, the [NiLR98] tags for deletions can easily be taken into consideration in our framework. Note that all algorithms for rewriting views when the schema of their source data change (e.g., [GMRR01, Bell02]), are orthogonal to our approach. Due to this generality, our approach can easily be extended in the presence of new results on such algorithms in the future.
Model mappings. Recently, model management [BeLP00, BeRa00], has been of particular interest to the research community. In [BeLP00], the authors propose a framework for managing model relationships, comprising three fundamental operators: match, diff and merge. The framework is intentionally generic, in the sense that no specific semantics are pre-assigned to the mappings between schemata and it is left to the designer to specify semantics for these mappings, depending on the context where the source/data integration takes place. Our proposal assigns semantics to the match operator for the case of model evolution, where the source model of the mapping is the original database graph and the target model is the resulting database graph, after evolution management has taken place. In the same context, Velegrakis et al., have recently proposed a similar framework for the management of evolution. Still, the model of [VeMP04] is more restrictive, in the sense that it is intended towards retaining 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.

6. Conclusions

In this paper, we have introduced a graph-based model that uniformly captures relations, views, constraints and queries. We have also presented an automated mechanism that allows a designer to execute what-if analysis scenarios and determine the impact of a potential change over a database graph. The framework allows the insertion and deletion of relations, attributes and query conditions and equips the designer with the possibility of annotating the graph with policies that either accept, or block a potential event. The impact and the possible reshaping of the graph are automatically determined in the proposed framework, based on a set of rules that we provide. Finally, we extend the query formulation capabilities by annotating SQL queries with information concerning the semantically aware adaptation of a query in the presence of changes in the underlying database.

Research can be pursued in several directions. For example, transactions of events can be thought of as combinations of modifications to the database structure (e.g., combined modifications of primary and foreign keys). Also, visualization techniques can be discussed to further automate the evolution of the database.

References


