Policy-Regulated Management of ETL Evolution

George Papastefanatos[1], Panos Vassiliadis[2], Alkis Simitsis[3], and Yannis Vassiliou[1]

[1] National Technical University of Athens, Greece
{gpapas,yv}@dbnet.ece.ntua.gr
[2] University of Ioannina, Greece
pvassil@cs.uoi.gr
[3] Stanford University, USA
alkis@db.stanford.edu

Abstract. In this paper, we discuss the problem of performing impact prediction for changes that occur in the schema/structure of the data warehouse sources. We abstract Extract-Transform-Load (ETL) activities as queries and sequences of views. ETL activities and its sources are uniformly modeled as a graph that is annotated with policies for the management of evolution events. Given a change at an element of the graph, our method detects the parts of the graph that are affected by this change and highlights the way they are tuned to respond to it. For many cases of ETL source evolution, we present rules so that both syntactical and semantic correctness of activities are retained. Finally, we experiment with the evaluation of our approach over real-world ETL workflows used in the Greek public sector.

Keywords: Data Warehouses, ETL, Evolution, Impact of changes.

1 Introduction

Data warehouses are complicated software environments that are used in large organizations for decision support based on OLAP-style (On-Line Analytical Processing) analysis of their operational data. Currently, the data warehouse market is of increasing importance; e.g., a recent report from *the OLAP Report* (http://www.olapreport.com) mentions that this market grew from $1 Billion in 1996 to $5.7 Billion in 2006 and showed an estimated growth of 16.4 percent in 2006. In a high level description of a data warehouse general architecture, data stemming from operational sources are extracted, transformed, cleansed, and eventually stored in fact or dimension tables in the data warehouse. Once this task has been successfully completed, further aggregations of the loaded data are also computed and subsequently stored in data marts, reports, spreadsheets, and several other formats that can simply be thought of as materialized views. The task of designing and populating a data warehouse can be described as a workflow, also known as Extract–Transform–Load (ETL) workflow, which comprises a synthesis of software modules representing extraction, cleansing, transformation, and loading routines. The whole environment is a very complicated architecture, where each module depends upon its data providers to fulfill its task. This strong flavor of inter-module dependency makes the problem of *evolution* very important in data warehouses, and especially, for their back stage ETL processes.

S. Spaccapietra et al. (Eds.): Journal on Data Semantics XIII, LNCS 5530, pp. 146–176, 2009.

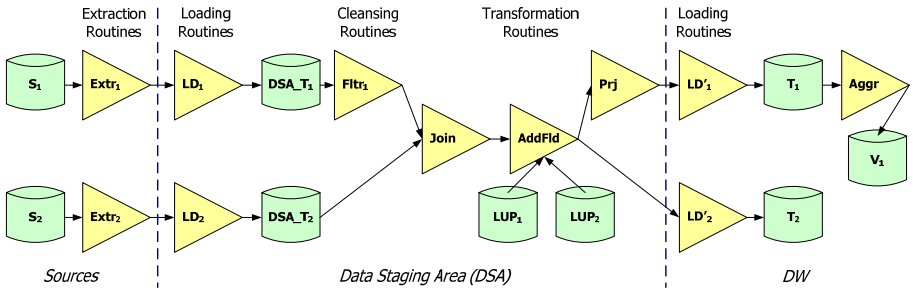

Fig. 1. An example ETL workflow

Figure 1 depicts an example ETL workflow. Data are extracted from two sources, S_1 and S_2, and they are transferred to the Data Staging Area (DSA), where their contents and structure are modified; example transformations include filters, joins, projection of attributes, addition of new attributes based on lookup tables and produced via functions, aggregations, and so on. Finally, the results are stored in the data warehouse (DW) either in fact or dimension tables and materialized views. During the lifecycle of the warehouse it is possible that several counterparts of the ETL process may evolve. For instance, assume that a source relation's attribute is deleted or renamed. Such a change affects the entire workflow, possibly, all the way to the warehouse (tables T_1 and T_2), along with any reports over the warehouse tables (abstracted as queries over a view V_1.) Similarly, assume that the warehouse designer wishes to add an attribute to the source relation S_2. 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 WHERE clause of a view is modified. Assume that the view definition is modified by incorporating an extra selection condition. Can we still use the view in order to answer existing queries (e.g., reports) that were already defined over the previous version of the view? 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, [23]. In other words, whenever a query is defined over a view, there exist two possible ways to interpret its semantics: (a) the query is defined with respect to the semantics of the view at the time of the query definition; if the view's definition changes in the future, the query's semantics are affected and the view should probably be re-adjusted, (b) the query's author uses the view as an API ignoring the semantics of the view; if these semantics change in the future, the query should not be affected. 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.

Research has extensively dealt with the problem of schema evolution, in object-oriented databases [1, 18, 25], ER diagrams [11], data warehouses [5, 6, 9, 10] and materialized views [2, 6, 10, 13]. Although the study of evolution has had a big impact in the above areas, it is only just beginning to be taken seriously in data

warehouse settings. A recent effort has provided a general mechanism for performing impact prediction for potential changes of data source configurations [15]. In this paper, we build on [15] (see related work for a comparison) and present an extended treatment of the management of evolution events for ETL environments.

Our method is fundamentally based on a graph model that uniformly models relations, queries, views, ETL activities, and their significant properties (e.g., conditions). This graph representation has several roles, apart from the simple task of capturing the semantics of a database system, and one of them is the facilitation of impact prediction for a hypothetical change over the system. In this paper, we present in detail the mechanism for performing impact prediction for the adaptation of workflows to evolution events occurring at their sources. The ETL graph is annotated with policies that regulate the impact of evolution events on the system. According to these policies, rules that dictate the proper actions, when additions, deletions or updates are performed to relations, attributes and conditions (all treated as first-class citizens of the model) are provided, enabling the automatic readjustment of the graph. Affected constructs are assigned with statuses (e.g., *to-delete*) designating the transformations that must be performed on the graph. Moreover, we introduce two mechanisms for resolving contradictory or absent policies defined on the graph, either during the runtime of the impact analysis algorithm (*on-demand*), or before the impact analysis algorithm executes (*a-priori*). Finally, we present the basic architecture of the proposed framework and we experimentally assess our approach with respect to its effectiveness and efficiency over real-world ETL workflows.

Outline. Section 2 presents a graph-based modelling for ETL processes. Section 3 formulates the problem of evolving ETL processes and proposes an automated way to respond to potential changes expressed by the *Propagate Changes* algorithm. Section 4 discusses the tuning of the *Propagate Changes* algorithm. Section 5 presents the system architecture of our prototype. Section 6 discusses our experimental findings and Section 7 concludes our work.

2 Graph Based Modeling for ETL Processes

In this section, we propose a graph modeling technique that uniformly covers relational tables, views, ETL activities, database constraints and SQL queries as first class citizens. The proposed technique provides an overall picture not only for the actual source database schema but also for the ETL workflow, since queries that represent the functionality of the ETL activities are incorporated in the model.

The proposed modeling technique represents all the aforementioned database parts as a directed graph **G(V,E)**. The nodes of the graph represent the entities of our model, where the edges represent the relationships among these entities. Preliminary versions of this model appear in [14, 15, 16]. The elements of our graph are listed in Table 1.

The constructs that we consider are classified as *elementary*, including relations, conditions, queries and views and *composite*, including ETL activities and ETL processes. Composite elements are combinations of elementary ones.

Table 1. Elements of our graph model

Nodes		Edges	
Relations	**R**	Schema relationships	$\mathbf{E_S}$
Attributes	**A**	Operand relationships	$\mathbf{E_O}$
Conditions	**C**	Map-select relationships	$\mathbf{E_M}$
Queries	**Q**	From relationships	$\mathbf{E_F}$
Views	**V**	Where relationships	$\mathbf{E_W}$
Group-By	**GB**	Having relationships	$\mathbf{E_H}$
Order-By	**OB**	Group-By relationships	$\mathbf{E_{GB}}$
Parameter	**P**	Order-By relationships	$\mathbf{E_{OB}}$
Function	**F**		
ETL activities	**A**		
ETL summary	**S**		

Relations, **R.** Each relation $R(A_1,A_2,\ldots,A_n)$ in the database schema, either a table or a file (it can be considered as an external table), is represented as a directed graph, which comprises: (a) a *relation node*, R, representing the relation schema; (b) n *attribute nodes*, $A_i \in \mathbf{A}$, 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 two classes of atomic conditions that are composed through the appropriate usage of an operator op belonging to the set of classic binary operators, **Op** (e.g., <, >, =, ≤, ≥, !=): (a) A op constant; (b) A op A' (A, A' are attributes of the underlying relations). Also, we consider the classes of A IN Q, and EXISTS Q, with Q being a subquery.

A *condition node* is used for the representation of the condition. Graphically, 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*, $\mathbf{E_O}$. These edges are indexed according to the precedence of each operand (i.e., op_1 for the left-side operand and op_2 for the right-side) in the condition clause. Composite conditions are easily constructed by tagging the condition node with a Boolean operator (e.g., AND or OR) and connecting the respective edges to the conditions composing the composite condition.

Well-known constraints of database relations – i.e., primary/foreign key, unique, not null, and check constraints – are easily captured by this modeling technique with use of a separate condition node. Foreign key constraints are subset conditions between the source and the target attributes of the foreign key. Check constraints are simple value-based conditions. Primary keys, not null and unique constraints, which are unique-value constraints, are explicitly represented through a dedicated node tagged by their names connected with operand edges with the respective attribute nodes.

Queries, Q. 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 relationships*. In order to represent the relationship between the query graph and the underlying relations, we resolve the query 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 can be regarded 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 that the WHERE and/or the HAVING clauses of a query involve composite conditions. 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 condition of the highest level.

Nested Queries. 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.

Group and Order By part. For the representation of aggregate queries, we employ two special purpose nodes: (a) a new node denoted as GB$\in$**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*, $\mathbf{E_{GB}}$. 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 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 labelled <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 of the query is performed similarly.

Self-Join Queries. For capturing the set of self-join queries, we stress that each reference via an alias to a relation in the FROM clause of the query is semantically equivalent with an inline view projecting all attributes of the referenced relation (i.e., SELECT *) and named with the respective alias. Self Join query subgraph is connected with the corresponding views' subgraphs.

Functions, F. Functions used in queries are integrated in our model through a special purpose node $F_i \in \mathbf{F}$, denoted with the name of the function. Each function has an input parameter list comprising attributes, constants, expressions, and nested functions, and one (or more) output parameter(s). The function node is connected with each input

parameter graph construct, nodes for attributes and constants or sub-graph for expressions and nested functions, through an operand relationship directing from the function node towards the parameter graph construct. This edge is additionally tagged with an appropriate identifier i that represents the position of the parameter in the input parameter list. An output parameter node is connected with the function node through a directed edge $E \in \mathbf{E_O} \cup \mathbf{E_M} \cup \mathbf{E_{GB}} \cup \mathbf{E_{OB}}$ from the output parameter towards the function node. This edge is tagged based on the context, in which the function participates. For instance, a map-select relationship is used when the function participates in the SELECT clause, and an operand relationship for the case of the WHERE clause.

Views, V. Views are considered either as queries or relations (materialized views). Thus, in the rest of the paper, whatever refers to a relation R it refers to a view too (R/V), and respectively, whatever refers to a query Q, it also refers to a view (Q/V). Thus, $\mathbf{V} \subseteq \mathbf{R} \cup \mathbf{Q}$.

ETL activities, A. An ETL activity is modeled as a sequence of SQL views. An ETL activity necessarily comprises: (a) one (or more) *input view(s)*, populating the input of the activity with data coming from another activity or a relation; (b) an *output view*, over which the following activity will be defined; and (c) a *sequence of views* defined over the input and/or previous, internal activity views.

ETL summary, S. An ETL summary is a directed acyclic graph $\mathbf{G_s}=(\mathbf{V_s},\mathbf{E_s})$ which corresponds to an ETL process of the data warehouse [22]. $\mathbf{V_s}$ comprises activities, relations and views that participate in an ETL process. $\mathbf{E_s}$ comprises the edges that connect the providers and consumers. Conversely to the overall graph where edges denote dependency, edges in the ETL summary denote data provision. The graph of the ETL summary can be topologically sorted and therefore, execution priorities can be assigned to activities. ETL summaries act as zoomed-out descriptions of the detailed ETL processes, and comprise only relations and activities without their internals; this also allows the visualization of the ETL process without overloading the screen with too many details (see for example, figure 9).

Modules. A module is a sub-graph of the graph in one of the following patterns: (a) a relation with its attributes and all its constraints, (b) a view with its attributes, functions and operands (c) a query with all its attributes, functions and operands. Modules are disjoint and they are connected through edges concerning foreign keys, map-select, where, and so on. Within a module, we distinguish *top-level* nodes comprising the query, relation or the view nodes, and *low-level* nodes comprising the remaining subgraph nodes. Additionally, edges are classified into *provider* and *part-of* relationships. Provider edges are intermodule relationships (e.g., $\mathbf{E_M}$, $\mathbf{E_F}$), whereas part-of edges are intramodule relationships (e.g., $\mathbf{E_S}$, $\mathbf{E_W}$).

Fig. 2 depicts the proposed graph representation for the following aggregate query:

```
Q:  SELECT      EMP.Emp# as Emp#, Sum(WORKS.Hours) as T_Hours
    FROM        EMP, WORKS
    WHERE       EMP.Emp# = WORKS.Emp#
                AND  EMP.STD_SAL >5000
    GROUP BY    EMP.Emp#
```

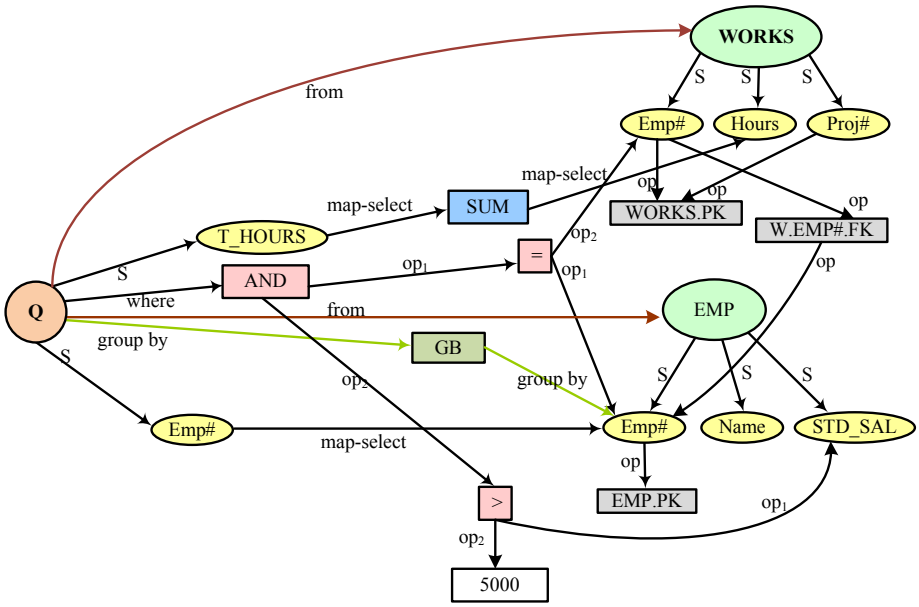

Fig. 2. Graph representation of aggregate query

DML Statements. As far as DML statements are concerned, there is a straightforward way to incorporate them in the graph, since their behavior with respect to adaptation to changes in the database schema can be captured by a graph representation that follows the one of SELECT queries and captures the intended semantics of the DML statement. In our discussions, we will use the term *graph equivalence* to refer to the fact that evolution changes (e.g., attribute addition) can be handled in the same way we handle the equivalent SELECT query, either these changes occur in the underlying relation of the INSERT statement or the sources of the provider subquery Q.

(a)The general syntax of INSERT statements can be expressed as:

INSERT INTO *table_name* (*attribute_set*)
[VALUES (*value_set*)] | [*Q*]

where Q is the provider subquery for the values to be inserted.

The graph equivalent SELECT query, which corresponds to the INSERT statement, comprises a SELECT and a FROM clause, projecting the same attribute set with the attribute set of the INSERT statement, and a WHERE clause for correlating the attribute set with the inserted values - value set or the projected attribute set of the subquery, i.e.,:

SELECT (*attribute_set*) FROM *table_name*
WHERE [(*attribute_set*) IN (*value_set*)] | [(*attribute_set*) IN *Q*]

(b) Similarly, 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*)] | [(*attribute_set*) = *Q*]
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*
AND [(*attribute_set*) IN (*value_set*)] | [(*attribute_set*) IN *Q*]

3 Evolution of ETL Workflows

In this section, we formulate a set of rules, which allow the identification of the impact of evolution changes to an ETL workflow and propose an automated way to respond to these changes. The impact of the changes affects the software used in an ETL workflow – mainly queries, stored procedures, triggers, etc. – 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 section 3.1, we detail how the graph representing the ETL workflow is annotated with actions that should be taken when a change event occurs. The combination of events and annotations determines the policy to be followed for the handling of a potential change. The annotated graph is stored in a metadata repository and it is accessed from an impact prediction module. This module notifies the designer or the 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. The algorithm presented in subsection 3.2 explains the internals of this impact prediction module.

3.1 The General Framework for Handling Schema Evolution

The main mechanism towards handling schema evolution is the annotation of the constructs of the graph (i.e., nodes and edges) with elements that facilitate impact prediction. 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 event with a policy determined by the designer/administrator 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 comprises the Cartesian product of two subspaces; specifically, (a) the space of hypothetical actions (addition/ deletion/modification), and, (b) the space of the graph constructs sustaining evolution changes (relations, 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 are defined: (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 [13, 22, 7]; and (c) *prompt* the administrator to interactively decide what will eventually happen. For the case of blocking, the specific method that can be used is orthogonal to our approach, which can be performed using any available method [13, 22, 7].

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 2. We indicate the parts of the system that can be affected by each kind of event. For instance, for the case of an attribute addition, affected parts of the system comprise the relation or view on which the new attribute was added as well as any view or query defined on this relation / view.

Table 2. Parts of the system affected by each event and annotation of graph constructs with policies for each event

		parts of the system affected						**nodes annotated with policies**			
event on database schema		**R/V**	**R/V Attr.**	**R/V Cond.**	**Q/V**	**Q/V Attr.**	**Q/V Cond.**	**R**	**A**	**V/Q**	**C/F/GB/ OB/P**
Add	A	√			√			√		√	
	C	√	√		√			√	√	√	
	R/V										
Delete	A	√	√		√	√	√	√	√	√	√
	C	√	√	√	√			√	√	√	√
	R/V	√			√			√		√	
Modify/ Rename	A	√	√	√	√	√	√	√	√	√	√
	C	√	√	√	√			√	√	√	√
	R/V	√			√			√		√	

A = Attribute, C = Constraint, R= Relation, V=View, Q=Query, F= Function, GB = GroupBy, OB=OrderBy, P=Parameter

The definition of policies on each part of the system involves the annotation of the respective construct (i.e., node) in our graph framework. Table 2 presents the allowed annotations of graph constructs for each kind of event.

Example. Consider the simple example query SELECT * FROM EMP as part of an ETL activity. Assume that the provider relation EMP is extended with a new attribute PHONE. There are two possibilities:

– The * notation signifies the request for any attribute present in the schema of relation EMP. In this case, the * shortcut can be treated as "return all the attributes that EMP has, independently of which these attributes are". Then, the query must also retrieve the new attribute PHONE.
– The * notation acts as a macro for the particular attributes that the relation EMP originally had. In this case, the addition to relation EMP should not be further propagated to the query.

A naïve solution to a modification of the sources; e.g., the addition of an attribute, would be that an impact prediction system must trace all queries and views that are potentially affected and ask the designer to decide upon which of them must be modified to incorporate the extra attribute. We can do better by extending the current modeling. For each element affected by the addition, we annotate its respective graph construct with the policies mentioned before. According to the policy defined on each construct the respective action is taken to correct the query.

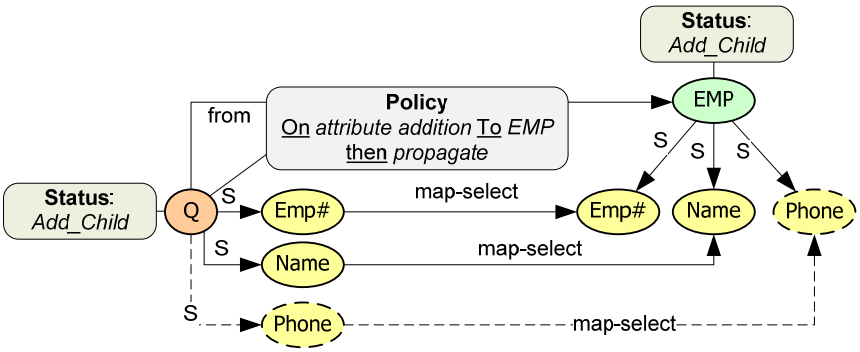

Fig. 3. Propagating addition of attribute PHONE

Therefore, for the example event of an attribute addition, the policies defined on the query and the actions taken according to each policy are:

– *Propagate attribute addition*. When an attribute is added to a relation appearing in the FROM clause of the query, this addition should be reflected to the SELECT clause of the query.
– *Block attribute addition*. The query is immune to the change: an addition to the relation is ignored. In our example, the second case is assumed, i.e., the SELECT * clause must be rewritten to SELECT A1,…,An without the newly added attribute.
– *Prompt*. In this case, the designer or the administrator must handle the impact of the change manually; similarly to the way that currently happens in database systems.

The graph of the query SELECT * FROM EMP is shown in Figure 3. The annotation of the Q node with *propagating addition* indicates that the addition of PHONE node to EMP relation will be propagated to the query and the new attribute is included in the SELECT clause of the query.

3.2 Adapting ETL Workflows to Evolution of Sources

The mechanism determining the reaction to a change is formally described in Figure 4 by the algorithm *Propagate Changes*. Given a graph **G** annotated with policies and an event e, *Propagate Changes* assigns a status to each affected node of the graph, dictating the action that must be performed on the node to handle the event.

Specifically, given an *event e* over a *node* n_0 altering the source database schema, *Propagate Changes* determines those nodes that are directly connected to the node altered and an appropriate *message* is constructed for each of them, which is added into the queue. For each processed node n_R, its prevailing *policy* p_R for the processed event *e* is determined. According to the prevailing policy, the *status* of each construct is set (see more on statuses in section 4.2). Subsequently, both the initial changes, along with the readjustment caused by the respective actions, are recursively propagated as *new events* to the consumers of the activity graph. In Figure 3, the statuses assigned to the affected nodes by the addition of an attribute to EMP relation are depicted. First, the algorithm sends a message to EMP relation for the addition of attribute PHONE to its schema, with a default *propagate* policy. It assigns the status ADD CHILD to relation EMP and propagates the event sending a new message to the query. Since an appropriate policy capturing this event exists on the query, the query is also assigned an ADD CHILD status. In the following sections, we discuss in more details the main components of the proposed algorithm.

Algorithm *Propagate Changes*

Input: (a) a session id `SID`
(b) a graph `G(V,E)`
(c) an event `e` over a node n_0
(d) a set of policies `P` defined over nodes of `G`
(e) an optional default policy p_0 defined by the user for the event `e`
Output: a graph `G(V,E)` with a `Status` value for each $n \in V' \subseteq V$
Parameters: (a) a global queue of messages E_{msg}
(b) each message `m` is of the form `m = [SID,` n_S`,` n_R`, e,` p_S`]`, where
`SID` : The unique identifier of the session regarding the evolution event `e`
n_S : The node that sends the message
n_R : The node that receives the message
`e` : The event that occurs on n_S
p_S : Policy of n_S for the event `e` {*Propagate, Block, Prompt*}

Begin
1. E_{msg}.enqueue([SID,user,n_0, e, p_0])
2. while (E_{msg} != ∅){
3. m = E_{msg}.dequeue();
4. p_R = determinePolicy(m);
5. n_R.Status=set_status(m,p_R);
6. decide_next_to_signal(m,E_{msg},G);} *//enqueue m*

End

Fig. 4. *Propagate Changes* Algorithm

4 Tuning the Propagation of Changes

In this section, we detail the internals of the algorithm Propagate Changes. Given an event arriving at a node of the graph, the algorithm involves three cases, specifically, (a) the determination of the appropriate policy for each node, (b) the determination of the node's status (on the basis of this policy) and (c) the further propagation of the event to the rest of the graph. The two first issues are detailed in sections 4.1 and 4.2 respectively. The third issue is straightforward, since the processing order of affected graph elements is determined by a BFS traversal on the graph. Therefore, after the status determination at each node, a message is inserted into the queue for all adjacent nodes connected with incoming edges towards this node.

4.1 Determining the Prevailing Policy

It is possible that the policies defined over the different elements of the graph do not always align towards the same goal. Two problems might exist: (a) over-specification refers to the existence of more than one policies that are specified for a node of the graph for the same event, and, (b) under-specification refers to the absence of any policy directly assigned to a node.

Consider for example the case of Figure 5, where a simplified subset of the graph for a certain environment is depicted. A relation R with one attribute A populates a view V, also with an attribute A. A query Q, again with an attribute A is defined over V. Here, for reasons of simplicity, we omit all the parts of the graph that are irrelevant to the discussion of policy determination. As one can see, there are only two policies defined in this graph, both concerning the deletion of attributes of view V. The first policy is defined on view V and says: 'Block all deletions for attributes of view V', whereas the second policy is defined specifically for attribute V.A and says 'If V.A must be deleted, then allow it'.

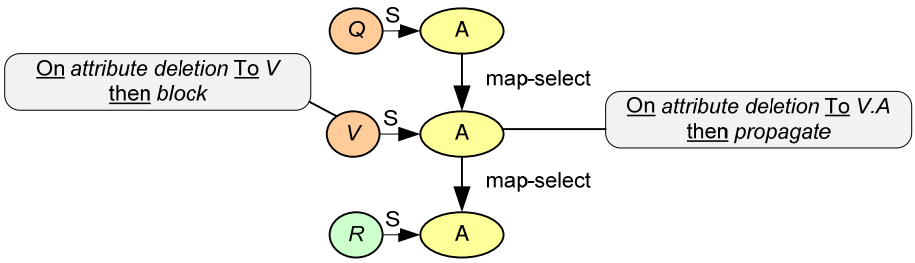

Fig. 5. Example of over-specification and under-specification of policies

The first problem one can easily see is the over-specification for the treatment of the deletion of attribute V.A. In this case, one of the two policies must *override* the other. A second problem has to do with the fact that neither R.A, nor Q.A, have a policy for handling the possibility of a deletion. In the case that the designer initiates such an event, how will this under-specified graph react? To give you a preview, under-specification can be either offline prevented by specifying default policies for all attributes or online compensated by following the policy of surrounding nodes. In the rest of this section, we will refer to any such problems as *policy misspecifications*.

We provide two ways for resolving policy misspecifications on a graph construct: on-demand and a-priori policy misspecification resolution. Whenever a node is not explicitly annotated with a policy for a certain event, on-demand resolution determines the prevailing policy during the algorithm execution based on policies defined on other constructs. A-priori resolution prescribes the prevailing policy for each construct potentially affected by an event with use of default policies. Both a-priori and on-demand resolution can be equivalently used for determining the prevailing policy of an affected node. A-priori annotation requires the investment of effort for the determination of policies before hypothetical events are tested over the warehouse. The policy overriding is tuned in such a way, though, that general annotations for nodes and edges need to be further specialized only wherever this is necessary. Our experiments, later, demonstrate that a-priori annotation can provide significant earnings in effort for the warehouse administrator. On the other hand, one can completely avoid the default policy specification and annotate only specific nodes. This is the basic idea behind the on-demand policy and this way less effort is required at the expense of runtime delays whenever a hypothetical event is posed on the system.

4.1.1 On-Demand Resolution

The algorithm for handling policy misspecifications on demand is shown in Figure 6. Intuitively, the main idea is that if a node has a policy defined specifically for it, it will know how to respond to an event. If an appropriate policy is not present, the node looks for a policy (a) at its container top-level node, or (b) at its providers.

Algorithm *Determine Policy*

Input: a message m of the form m=[SID,n_S,n_R,e,p_S]
Output: a prevailing policy p_R

```
Begin
 1.  if (edge(n_s,n_r) isPartOf)          // if m came from partof edge
 2.     return p_S;                       // child node policy prevails
 3.  else                                 // m came from provider
 4.     if exists policy(n_R,e)           // check if n_R has policy for this event
 5.        return policy(n_R);            // return this policy
 6.     else if exists policy(n_R.parent,e)
 7.           return policy(n_R.parent);  // return n_R parent's policy
 8.        else return p_s;               // else return providers policy
End
```

Fig. 6. Determine Policy Algorithm

Algorithm *Determine Policy* implements the following basic principles for the management of an incoming even to a node:

- If the policy is over-specified, then the higher and left a module is at the hierarchy of Figure 7, the stronger its policy is.
- If the policy is under-specified, then the adopted policy is the one coming from lower and right.

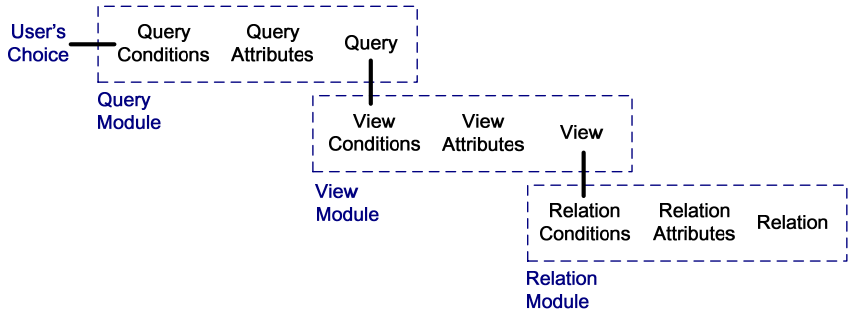

Fig. 7. On Demand Policy Resolution

The algorithm assumes that a message is sent from a sender node n_s to a receiver node n_r. Due to its complexity, we present the actual decisions taken in a different order than the one of the code:

- *Check 1* (lines 6-7): this concerns child nodes: if they do not have a policy of their own, they inherit their parent's policy. If they do have a policy, this is covered by lines 4-5.
- *Check 2* (lines 1-5): if the event arrives at a parent node (e.g., a relation), and it concerns a child node (e.g., an attribute) the algorithm assigns the policy of the parent (lines 4-5), unless the child has a policy of its own that overrides the parent's policy (lines 1-2). A subtle point here is that if the child did not have a policy, it has already obtained one by its parent in lines 6-7.
- Check 3 (line 7): Similarly, if an event arrives from a provider to a consumer node via a map-select edge, the receiver will make all the above tests, and if they all fail, it will simply adopt the provider's policy. For example, in the example of figure 5, Q.A will adopt the policy of V.A if all else fails.

4.1.2 A-Priori Resolution

A-priori resolution of policy misspecifications enables the annotation of all nodes of the graph with policies before the execution of the algorithm. A-priori resolution guarantees that every node is annotated with a policy for handling an occurred event and thus no further resolution effort is required at runtime. That is, the receiver node of a message will always have a policy handling the event of the message. A-priori resolution is accomplished by defining default policies at 3 different scopes [18].

System-wide scope. First, we prescribe the default policies for all kinds of constructs, in a system-wide context. For instance, we impose a default policy on all nodes of the graph that blocks the deletion of the constructs per se.

Top-level scope. Next, we prescribe defaults policies for top-level nodes, namely 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).

Low-level scope. 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:

$$System\text{-}wide\ Scope \leq Top\text{-}Level\ Scope \leq Low\text{-}Level\ Scope$$

Furthermore, certain nodes or modules that violate the above default behaviors and must obey to an opposite reaction for a potential event are explicitly annotated. For example, if a specific attribute of an activity must always block the deletion of itself, whereas the default activity policy is to propagate the attribute deletions, then this attribute node is explicitly annotated with block policy, overriding the default behavior.

4.1.3 Completeness

The completeness problem refers to the possibility of a node that is unable to determine its policy for a given event. It is easy to see that it is sufficient to annotate all the source relations for the on-demand policy, in order to guarantee that all nodes can determine an appropriate policy. For the case of a-priori annotation, it is also easy to see that a top-level, system-wide annotation at the level of nodes is sufficient to provide a policy for all nodes. In both cases, it is obvious that more annotations with extra semantics for specific nodes, or classes of nodes, that override the abovementioned (default) policies, are gracefully incorporated in the policy determination mechanisms.

4.2 Determination of a Node's Status

In the context of our framework, the action applied on an affected graph construct is expressed as a status that is assigned on this construct. The status of each graph construct visited by Propagate Changes algorithm is determined *locally* by the prevailing policy defined on this construct and the event transmitted by the adjacent nodes. The status of a construct with respect to an event designates the way this construct is affected and reacts to this event, i.e., the kind of evolution action that will be applied to the construct.

A visited node is initially assigned with a *null status*. If the prevailing policy is *block* or *prompts* then the status of the node is *block* and *prompt* respectively, independently of the occurred event. Recall that blocking the propagation of an event implies that the affected node is annotated for retaining the old semantics despite of change occurred at its sources. The same holds for prompt policy with the difference that the user, e.g., the administrator, the developer, etc. must decide upon the status of the node.

For determining the status of a node when a *propagate* policy prevails, we take into account the event action (e.g., attribute addition, relation deletion, etc.) transmitted to the node, the type of node accepting the event and lastly the scope of the event action. An event raises actions that may affect the node itself, its ancestors within a module or its adjacent dependent nodes. Thus, we classify the *scope of* evolution impacts with respect to an event that arrives at a node as:

- *SELF*: The impact of the event concerns the node itself, e.g., a 'delete attribute' event occurs on an attribute node.
- *CHILD*: The impact of the event concerns a descending node belonging to the same module, e.g., a view is notified with a 'delete attribute' event for the deletion of one of its attributes.

– *PROVIDER*: The impact of the event concerns a node belonging to a provider module, e.g., a view is notified for the addition of an attribute at the schema of one of its source relations (and, in return, it must notify any other views or queries that are defined over it).

In that manner, combinations of the event type and the event scope provide a non finite set of statuses, such as: DELETE SELF, DELETE CHILD, ADD CHILD, RENAME SELF, MODIFY PROVIDER and so on. It is easy to see that that the above mechanism is extensible both with respect to event types and statuses. Lastly, the status assignment to nodes induces new events on the graph which are further propagated by *Propagate Changes* algorithm to all adjacent constructs. In the Appendix of this paper, the statuses assigned to visited nodes for combinations of events and types of nodes are shown, when *propagate* policy prevails on the visited node. For each status, the new event induced by the assignment of a node with status, which is further propagated to the graph, is also shown.

5 System Architecture

For the representation of the database graph and its annotation with policies regarding evolution semantics, we have implemented a tool, namely HECATAEUS [16, 17]. The system architecture is shown in Figure 8.

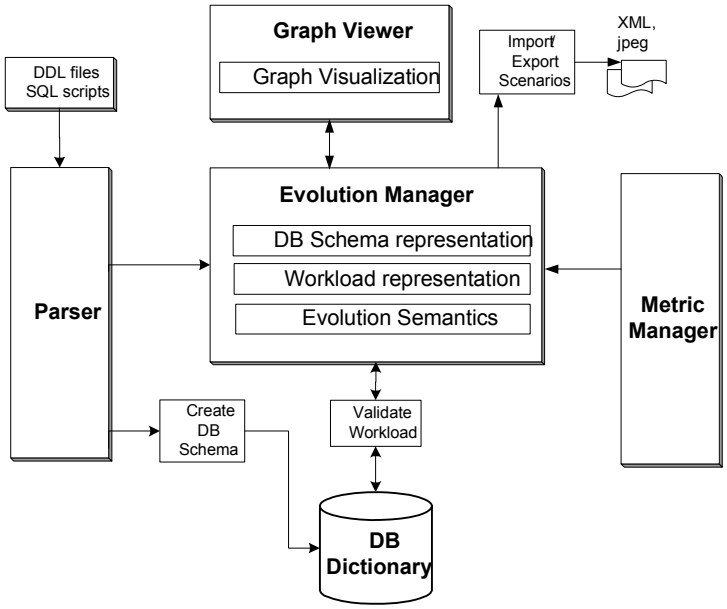

Fig. 8. System Architecture of HECATAEUS

HECATAEUS enables the user to transform SQL source code to database graphs, explicitly define policies and evolution events on the graph and determine affected and adjusted graph constructs according to the proposed algorithm. As mentioned in

the introduction, the graph modeling of the environment has versatile utilizations: apart from the impact prediction, we can also assess several graph-theoretic metrics of the graph that highlight sensible regions of the graph (e.g., a large node degree denotes strong coupling with the rest of the graph). This metrics management is not part of this paper's investigations; still, we find it worth mentioning.

The tool architecture (see Figure 8) consists of the coordination of HECATAEUS' five main components: the *Parser*, the *Evolution Manager*, the *Graph Viewer,* the *Metric Manager* and the *Dictionary*.

The *Parser* is responsible for parsing the input files (i.e., DDL and workload definitions) and for sending each command to the database Catalog and then to the Evolution Manager.

The functionality of the *Dictionary* is to maintain the schema of the relations as well as to validate the syntax of the workload parsed (i.e., activity definitions, queries, views), before they are modeled by the Evolution Manager.

The *Evolution Manager* is the component responsible for representing the underlying database schema and the parsed queries in the proposed graph model. The Evolution Manager holds all the semantics of nodes and edges of the aforementioned graph model, assigning nodes and edges to their respective classes. It communicates with the catalog and the parser and constructs the node and edge objects for each class of nodes and edges (i.e., relation nodes, query nodes, etc.). It retains all evolution semantics for each graph construct (i.e., events, policies) and methods for performing evolution scenarios and executing *Propagate Changes* algorithm. It contains methods for transforming the database graph from/to an XML format.

The *Metric Manager* is responsible for the application of graph metrics on the graph. It measures and provides results regarding several properties of the graph, such as nodes' degrees, graph size, etc.

Finally, the *Graph Viewer* is responsible for the visualization of the graph and the interaction with the user. It communicates with the Evolution Manager, which holds all evolution semantics and methods. Graph Viewer offers distinct colorization for each set of nodes, edges according to their types and the way they are affected by evolution events, editing of the graph, such as addition, deletion and modification of nodes, edges and policies. It enables the user to raise evolution events, to detect affected nodes by each event and highlight appropriate transformations of the graph.

6 Experiments

We have evaluated the proposed framework and capabilities of the approach presented via the reverse engineering of seven real-world ETL scenarios extracted from an application of the Greek public sector. The data warehouse examined maintains information regarding farming and agricultural statistics. Our goal was to evaluate the framework with respect to its *effectiveness* for adapting ETL workflows to evolution changes occurring at ETL sources and its *efficiency* for minimizing the human effort required for defining and setting the evolution metadata on the system.

The aforementioned ETL scenarios extract information out of a set of 7 source tables, namely S_1 to S_7 and 3 lookup tables, namely L_1 to L_3, and load it to 9 tables, namely T_1 to T_9, stored in the data warehouse. The 7 scenarios comprise a total

number of 59 activities. Our approach has been built on top of the Oracle DBMS. All ETL scenarios were source coded as PL\SQL stored procedures in the data warehouse.

First, we extracted embedded SQL code (e.g., cursor definitions, DML statements, SQL queries) from activity stored procedures. Table definitions (i.e., DDL statements) were extracted from the source and data warehouse dictionaries. Each activity was represented in our graph model as a view defined over the previous activities, and table definitions were represented as relation graphs. In Figure 9, we depict the graph representation of the first ETL scenario as modeled by our framework. For simplicity reasons, only top level nodes are shown. Activities are depicted as triangles; source, lookup and target relations as dark colored circles.

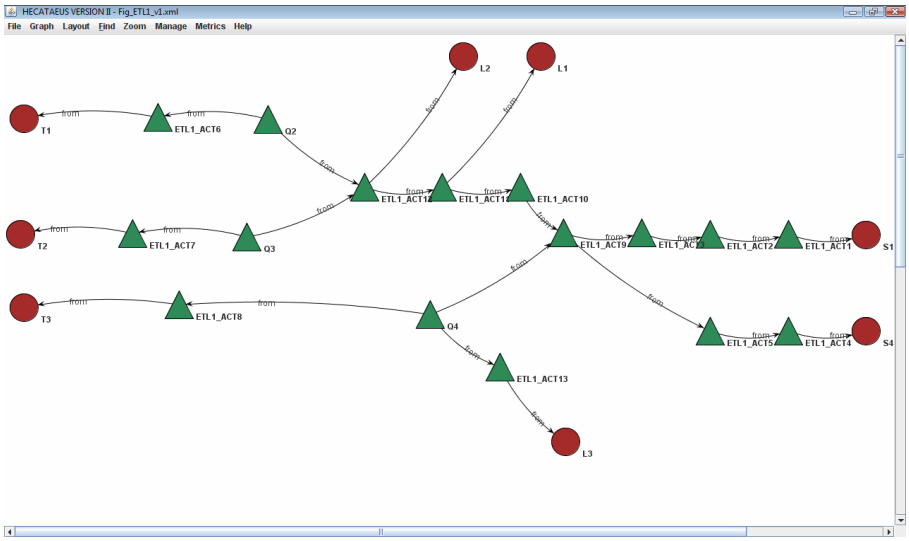

Fig. 9. Graph representation for the first ETL scenario

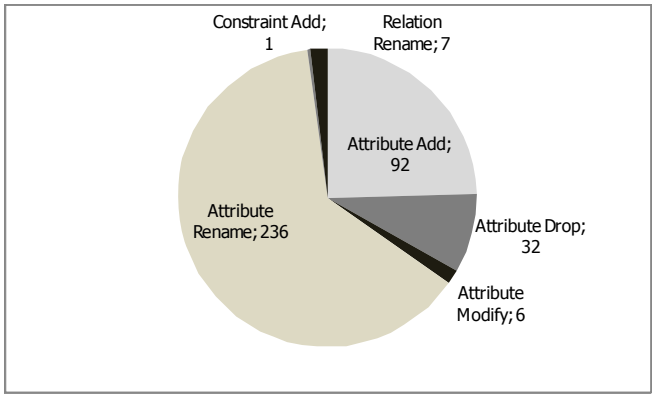

Fig. 10. Distribution of occurrence per kind of evolution events

Afterward, we monitored the schema changes occurred at the source tables due to changes of requirements over a period of 6 months. The set of evolution events occurred in the source schema included renaming of relations and attributes, deletion of attributes, modification of their domain, and lastly addition of primary key constraints. We counted a total number of 374 evolution events and the distribution of occurrence per kind of event is shown in Figure 10.

In Table 3 we provide the basic properties of each examined ETL scenario and specifically, its size in terms of number of activities and number of nodes comprising its respective graph, its evolved source tables and lookup tables and lastly the number of occurred events on these tables.

Table 3. Characteristics of the ETL scenarios

Scenario	# Activ.	# Nodes	Sources	# Events
1	16	1428	S1, S4 ,L1, L2, L3	142
2	6	830	S2, L1	143
3	6	513	S3, L1	83
4	16	939	S4, L1	115
5	5	242	S5	3
6	5	187	S6	1
7	5	173	S7	6

The intent of the experiments is to present the impact of these changes to the ETL flows and specifically to evaluate our proposed framework with respect to its effectiveness and efficiency.

6.1 Effectiveness of Workflow Adaptation to Evolution Changes

For evaluating the extent to which affected activities are effectively adapted to source events, we imposed policies on them for each separate occurred event. Our first goal was to examine whether our algorithm determines the correct status of activities in accordance to the expected transformations, i.e., transformations that the administrators/developers would have manually enforced on the ETL activities to handle schema changes at the sources, by inspecting and rewriting every activity source code.

Hypothesis H1. Algorithm "Propagate Changes" effectively determines the correct status of activities for various kinds of evolution events.

Methodology:

1. We first examined each event and its impact on the graph, by finding all affected activities.
2. Since all evolution events and their impact on activities were a-priori known, each activity was annotated with an appropriate policy for each event. An appropriate policy for an event is the policy (either propagate or block), which adjusts the activity according to the desired manual transformation, when this event occurs on the activity source.
3. In that manner, each event at the source schema of the ETL workflows was separately processed, by imposing *a different policy set* on the activities. We

employed both propagate and block policies for all views and queries subgraphs comprising ETL activities. Policies were defined both at query and attribute level, i.e., query, view and attribute nodes were annotated.

4. We invoked each event and examined the extent to which the automated readjustment of the affected activities (i.e., the STATUS assigned to each activity) adheres to the desired transformation. We, finally, evaluated the effectiveness of our framework by measuring the number of affected activities by each event, i.e., these that obtained a STATUS, with respect to the number of successfully readjusted activities, (or, in other words, those activities whose STATUS was consistent with the desired transformation).

In Table 4, we summarize out results for different kinds of events. First, we note that most of the activities were affected by attribute additions and renaming, since these kinds of events were the most common in our scenarios. Most important, we can conclude that our framework can effectively adapt activities to the examined kinds of events. Exceptions regarding attribute and constraint additions are due to the fact that specific events induced ad hoc changes in the functionality of the affected activities, which *prompts* the user to decide upon the proper readjustments. These exceptions are mainly owed to events occurred on the lookup tables of the scenarios. Additions of attributes at these tables incurred (especially when these attributes were involved in primary key constraints) rewriting of the WHERE clause of the queries contained in the affected activities.

Finally, whereas the above concern the precision of the method (i.e., the percentage of correct status determination for affected activities), we should also report on the recall of our method. Our experimental findings demonstrate that the number of those activities that were not affected by the event propagation, although they should have been affected, is zero.

Table 4. Affected and adjusted activities per event kind

	Activities	
Event Type	**with Status**	**with Correct Status**
Attribute Add	1094	1090
Attribute Delete	426	426
Attribute Modify	59	59
Attribute Rename	1255	1255
Constraint Add	13	5
Table Rename	8	8
Total	**2855**	**2843**

6.2 Effectiveness of Workflow Annotation

Our second goal was to examine the extent to which different annotations of the graph with policies affect the effectiveness of our framework. This addresses the real case when the administrator/developer does not know the number and the kind of potential events that occur on the sources and consequently cannot decide a priori upon a specific policy set for the graph.

Hypothesis H2. Different annotations affect the effectiveness of the algorithm.

Methodology:

1. We first imposed a policy set on the graph.
2. We then invoked each event in sequence, retaining the same policy set on the graph.
3. We again examined the extent to which the automated readjustment of the affected activities (i.e., their obtained status) adheres to the desired transformation and evaluated the effectiveness of our framework for several annotation plans.

We experimented with 3 different policy sets.

- **Mixture annotation.** A mixture annotation plan for a given set of events comprises the set of policies imposed on the graph that maximizes the number of successfully adjusted activities. For finding the appropriate policy for each activity of the ETL scenarios, we examined its most common reaction to each different kind of event. For instance, the appropriate policy of an activity for attribute addition will be *propagate* if this activity propagates the 70% of the new attributes added at its source and blocks the rest 30%. In mixture annotation, *propagate* policies were applied on most activities for all kinds of events whereas *block* policies were applied on some activities regarding only attribute addition events.
- **Worst-Case annotation.** As opposed to the mixture annotation plan, the worst case scenario comprises the set of policies imposed on the graph that minimizes the number of successfully adjusted activities. The less common reaction to an event type was used for determining the prevailing policy of each activity.
- **Optimistic annotation.** Lastly, an optimistic annotation plan implies that all activities are annotated with a propagate policy for all potential events occurred at their sources.

Again, we measured the number of affected activities that obtained a specific status with respect to the number of correctly adapted activities. In Figures 11, 12, 13 we present the results for the different kinds of events and annotations.

As stated in the hypothesis, different annotations on the graph have a different impact on the overall effectiveness of our framework, as they vary both the number of the affected activities (i.e., candidates for readjustment) and the number of the adjusted activities (i.e., successfully readjusted) on the graph. The mixture annotation manages most effectively to detect these activities that should be affected by an event and adjust them properly. In mixture annotation, the policies, imposed on the graph, manage to propagate event messages towards activities that should be readjusted, whereas block messages from activities that should retain their old functionality. On the contrary, the worst case annotation, fails to detect all affected activities on the graph as well as to adjust them properly, as it blocks event messages from the early activities of each ETL workflow. Since events are blocked in the beginning of the workflow, further activities cannot be notified for handling these events. Lastly, optimistic annotation provides both good and bad results. On the good side of things, the optimistic annotation is close to the mixture annotation in several categories. On the other hand, the optimistic annotation propagates event messages even towards activities, which should retain their old

semantics. In that manner, optimistic annotations increases the number of affected activities (i.e., actually all the activities of the workflow are affected) without however handling properly their status determination.

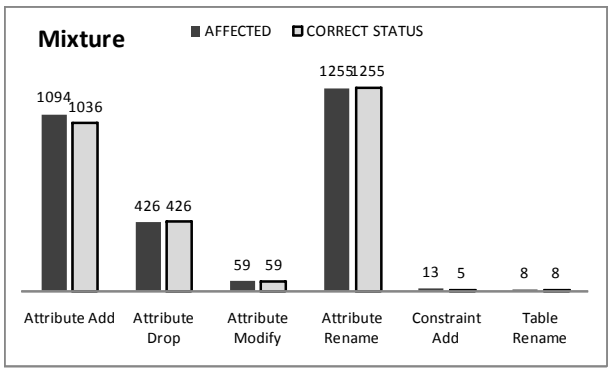

Fig. 11. Mixture Annotation

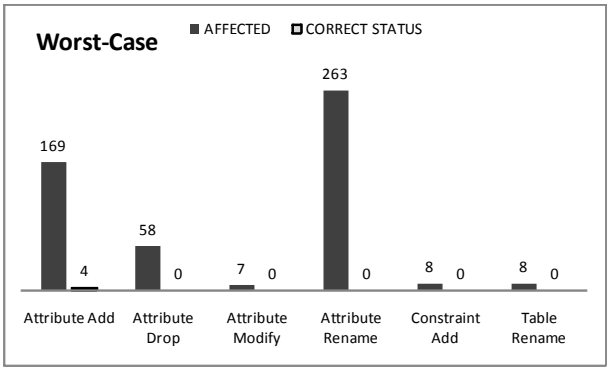

Fig. 12. Worst Case Annotation

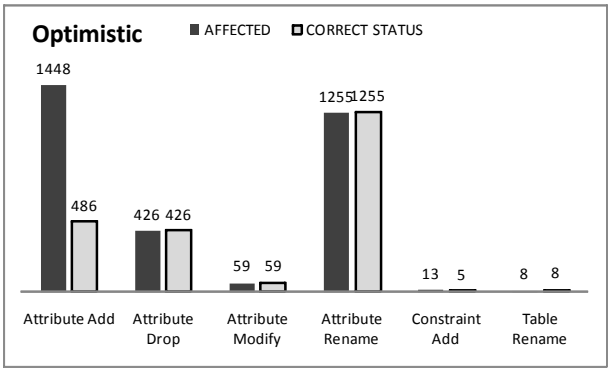

Fig. 13. Optimistic Annotation

Overall, a reasonable tactic for the administrator would be to either choose a mixture method, in case there is some a-priori knowledge on the desired behavior of constructs in an environment, or, progressively refine an originally assigned optimistic annotation whenever nodes that should remain immune to changes are unnecessarily affected.

6.3 Efficiently Adapting ETL Workflows to Evolution Changes

For measuring the efficiency of our framework, we examined the cost of manual adaptation of the ETL activities by the administrator / developer with respect to the cost of setting the evolution metadata on the graph (i.e., annotation with policies) and transforming properly the graph with use of our framework.

Developers' effort comprises the detection, inspection and where necessary the rewriting of affected activities by an event. For instance, given an attribute addition in a source relation of an ETL workflow, the developer must detect all activities affected by the addition, decide how and whether this addition must be propagated or not to each SQL statement of the activity and lastly rewrite, if necessary, properly the source code. The effort required for the above operations depends highly on the developers' experience but on the ETL workflow characteristics as well (e.g., the complexity of the activity source code, the workflow size, etc.). Therefore, the cost in terms of human effort for manual handling of source evolution, *MC*, can be quantified as the sum of (a) the number of SQL statements per activity, which are affected by an event and must be manually detected, *AS*, plus (b) the number of SQL statements, which must be manually rewritten for adapting to the event, *RS*. Thus human effort for manual adaptation of an activity, *a,* to an event, *e*, can be expressed as:

$$MC_a^e = (AS_a^e + RS_a^e) \tag{1}$$

For a given set of evolution events E, and a set of manually adapted activities A in an ETL workflow, the overall cost, *OMC,* is expressed as:

$$OMC = \sum_{e \in E} \sum_{a \in A} MC_a^e \tag{2}$$

For calculating OMC, we recorded affected and rewritten statements for all activities and events.

If HECATAEUS had been used, instead of manually adapting all the activities, the human effort can be quantified as the sum of two factors: (a) the number of annotations (i.e., policy per event) imposed on the graph, *AG*, and (b) the cost of manually discovering and adjusting activities A_R that escape the automatic status annotation of the tool, e.g., no annotations have been set on these activities or a prompt policy is assumed for these activities. The latter cost is expressed as:

$$RMC = \sum_{e \in E} \sum_{a \in A_R} MC_a^e \tag{3}$$

Therefore, overall cost for automated adaptation, *OAC,* is expressed as:

$$OAC = AG + RMC \tag{4}$$

Hypothesis H3. The cost of the semi-automatic adaptation, *OAC*, is equal or less than the cost of manually handling evolution, *OMC*.

For calculating *OAC*, we followed the mixture plan for annotating each attribute and query node potentially affected by an event occurred at the source schema and measured the number of explicit annotations, *AG*. We then applied our algorithm and measured the cost of manual adaptation for activities which were not properly adjusted. Figure 14 compares the *OMC* with *OAC* for 7 evolving ETL scenarios.

Figure 14 shows that the cost of manual adaptation is much higher than the cost of semi automating the evolution process. The divergence becomes higher especially for large scenarios such as scenario 1 and 4 or scenarios with many events such as scenario 2, in which the administrator must manually detect a large number of affected activities or handle a large number of events.

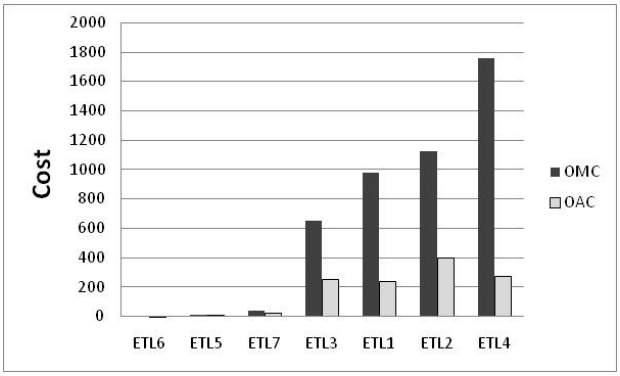

Fig. 14. Manual (OMC) and Semi-automatic (OAC) Adaptation Cost per ETL Scenario

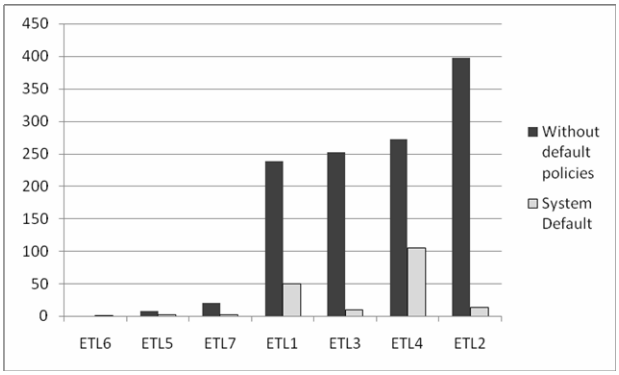

Fig. 15. Cost of Adaptation with and without use of Default Policies

Furthermore, to decrease the annotation cost, *AG*, we applied system wide default policies on the graph. With use of default policies, the annotation cost, *AG*, decreases to the number of explicit annotations of nodes that violates the default behavior. We, again, measured the number of explicit annotations as well as the remaining *RMC*. As shown in figure 15, the cost of adaptation with use of our framework is further

decreased, when default policies are used. With use of default policies, the overall adaptation cost is dependent neither to the scenario size (e.g., number of nodes) nor to the number of evolution events, but rather to the number of policies, deviating from the default behavior, that are imposed on the graph. Scenarios 1 and 4 comprised more cases for which the administrator should override the default system policies and thus, the overall cost is relatively high. On the contrary, in scenarios 2 and 3 the adaptation is achieved better by a default policy annotation, since the majority of the affected activities react in a uniform way (i.e., default) to evolution events.

7 Related Work

Evolution. A number of research works are related to the problems of database schema evolution. In [21] a survey on schema versioning and evolution is presented, whereas a categorization of the overall issues regarding evolution and change in data management is presented in [20]. The problem of view adaptation after redefinition is mainly investigated in [2, 8, 12] where changes in views definition are invoked by the user and rewriting is used to keep the view consistent with the data sources. In [9] the authors discuss versioning of star schemata, where histories of the schema are retained and queries are chronologically adjusted to ask the correct schema. [2] deals also with warehouse adaptation, but only for SPJ views. [13] deals with the view synchronization problem, which considers that views become invalid after schema changes in their definition. 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. In this context, our work can be compared with that of [13] in the sense that policies act as regulators for the propagation of schema evolution on the graph similarly to the evolution parameters introduced in [13]. We furthermore extend this approach to incorporate attribute additions and the treatment of conditions. Note that all algorithms for rewriting views when the schema of their source data change (e.g., [2,8]), are orthogonal to our approach. This is due to the fact that our algorithm stops at status determination and does not perform any rewritings. A designer can apply any rewriting algorithm, provided that he pays the annotation effort that each of the methods of the literature requires (e.g., LAV/GAV/GLAV or any other kind of metadata expressions) For example, such an expression could be stating that two select-project fragments of two relations are semantically equivalent. Due to this generality, our approach can be extended in the presence of new results on such algorithms.

A short, first version of this paper appears in [15] where (a) the graph model is presented and (b) the general framework is informally presented. [15] sketches the basic concepts of a framework for annotating the database graph with policies concerning the behaviour of nodes in the presence of hypothetical changes. In this paper, we extend the above work in the following ways. First, we elaborate an enriched version of the graph model, by incorporating DML statements. ETL activities utilize DML statements for storing temporary or filtering out redundant data; thus, the incorporation of such statement complements the representation of ETL workflows. Second, we present the mechanism for impact prediction in much more detail, both in terms of the algorithmic internals and in terms of system architecture. In this context, we also give a more elaborate version

of the algorithm for the propagation of changes. Third, the discussions on the management of incomplete or overriding policies are novel in this paper. Finally, we present a detailed experimental study for the above that is not present in [15].

Model mappings. Model management [3, 4], provides a generic framework for managing model relationships, comprising three fundamental operators: match, diff and merge. 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. Velegrakis at al., have proposed a similar framework, namely ToMas, for the management of evolution. Still, the model of [24] is more restrictive, in the sense that it is intended towards retaining the original semantics of the queries by preserving mappings consistent when changes occur. 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. Lastly, in [7], 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. Complex evolution events are expressed as simple transformations comprising addition, deletion, renaming, expansion and contraction of a schema construct. They also deal with the evolution of materialized data with use of IQL, a functional query language supporting several primitive operators for manipulating lists. Both [24] and [7] can be used orthogonally to our approach for the case that affected constructs must preserve the old semantics (i.e., block policy in our framework) or for the case that complex evolution events must be decomposed into a set of elementary transformations, respectively.

8 Conclusions

In this paper, we have dealt with the problem of impact prediction of schema changes in data warehouse environments. The strong flavor of inter-module dependency in the back stage of a data warehouse makes the problem of *evolution* very important under these settings. We have presented a graph model that suitably models the different constructs of an ETL workflow and captures DML statements. We have provided a formal method for performing impact prediction for the adaptation of workflows to evolution events occurring at their sources. Appropriate policies allow the automatic readjustment of the graph in the presence of potential changes. Finally, we have presented our prototype based on the proposed framework and we have experimentally assessed our approach with respect to its effectiveness and efficiency over real-world ETL workflows.

Acknowledgments

We would like to thank the anonymous reviewers for their suggestions on the structure and presentation of the paper, which have improved the paper a lot.

References

1. Banerjee, J., Kim, W., Kim, H.J., Korth, H.F.: Semantics and implementation of schema evolution in object-oriented databases. In: Proc. ACM Special Interest Group on Management of Data, pp. 311–322 (1987)

2. Bellahsene, Z.: Schema evolution in data warehouses. Knowledge and Information Systems 4(3), 283–304 (2002)
3. Bernstein, P., Levy, A., Pottinger, R.: A Vision for Management of Complex Models. SIGMOD Record 29(4), 55–63 (2000)
4. Bernstein, P., Rahm, E.: Data warehouse scenarios for model management. In: Laender, A.H.F., Liddle, S.W., Storey, V.C. (eds.) ER 2000. LNCS, vol. 1920, pp. 1–15. Springer, Heidelberg (2000)
5. Blaschka, M., Sapia, C., Höfling, G.: On Schema Evolution in Multidimensional Databases. In: Proc. First International Conference on Data Warehousing and Knowledge Discovery (DAWAK 1999), pp. 153–164 (1999)
6. Bouzeghoub, M., Kedad, Z.: A logical model for data warehouse design and evolution. In: Kambayashi, Y., Mohania, M., Tjoa, A.M. (eds.) DaWaK 2000. LNCS, vol. 1874, pp. 178–188. Springer, Heidelberg (2000)
7. Fan, H., Poulovassilis, A.: Schema Evolution in Data Warehousing Environments – A Schema Transformation-Based Approach. In: Atzeni, P., Chu, W., Lu, H., Zhou, S., Ling, T.-W. (eds.) ER 2004. LNCS, vol. 3288, pp. 639–653. Springer, Heidelberg (2004)
8. Gupta, A., Mumick, I.S., Rao, J., Ross, K.A.: Adapting materialized views after redefinitions: Techniques and a performance study. Information Systems J. 26(5), 323–362 (2001)
9. Golfarelli, M., Lechtenbörger, J., Rizzi, S., Vossen, G.: Schema Versioning in Data Warehouses. In: Atzeni, P., Chu, W., Lu, H., Zhou, S., Ling, T.-W. (eds.) ER 2004. LNCS, vol. 3288, pp. 415–428. Springer, Heidelberg (2004)
10. Kaas, C., Pedersen, T.B., Rasmussen, B.: Schema Evolution for Stars and Snowflakes. In: Sixth Int'l. Conference on Enterprise Information Systems (ICEIS 2004), pp. 425–433 (2004)
11. Liu, C.T., Chrysanthis, P.K., Chang, S.K.: Database schema evolution through the specification and maintenance of changes on entities and relationships. In: Loucopoulos, P. (ed.) ER 1994. LNCS, vol. 881, pp. 132–151. Springer, Heidelberg (1994)
12. Mohania, M., Dong, D.: Algorithms for adapting materialized views in data warehouses. In: Proc. International Symposium on Cooperative Database Systems for Advanced Applications (CODAS 1996), pp. 309–316 (1996)
13. Nica, A., Lee, A.J., Rundensteiner, E.A.: The CSV algorithm for view synchronization in evolvable large-scale information systems. In: Schek, H.-J., Saltor, F., Ramos, I., Alonso, G. (eds.) EDBT 1998. LNCS, vol. 1377, pp. 359–373. Springer, Heidelberg (1998)
14. Papastefanatos, G., Vassiliadis, P., Vassiliou, Y.: Adaptive Query Formulation to Handle Database Evolution. In: Proc. Forum of the Eighteenth Conference on Advanced Information Systems Engineering (CAISE 2006) (2006)
15. Papastefanatos, G., Vassiliadis, P., Simitsis, A., Vassiliou, Y.: What-if analysis for data warehouse evolution. In: Song, I.-Y., Eder, J., Nguyen, T.M. (eds.) DaWaK 2007. LNCS, vol. 4654, pp. 23–33. Springer, Heidelberg (2007)
16. Papastefanatos, G., Kyzirakos, K., Vassiliadis, P., Vassiliou, Y.: Hecataeus: A Framework for Representing SQL Constructs as Graphs. In: Proc. Tenth International Workshop on Exploring Modeling Methods in Systems Analysis and Design (held with CAISE) (2005)
17. Papastefanatos, G., Anagnostou, F., Vassiliadis, P., Vassiliou, Y.: Hecataeus: A What-If Analysis Tool for Database Schema Evolution. In: Proc. Twelfth European Conference on Software Maintenance and Reengineering (CSMR 2008) (2008)
18. Papastefanatos, G., Vassiliadis, P., Simitsis, A., Aggistalis, K., Pechlivani, F., Vassiliou, Y.: Language Extensions for the Automation of Database Schema Evolution. In: 10th International Conference on Enterprise Information Systems (ICEIS 2008) (2008)

19. Ra, Y.G., Rundensteiner, E.A.: A transparent object-oriented schema change approach using view evolution. In: Proc. Eleventh International Conference on Data Engineering (ICDE 1995), pp. 165–172 (1995)
20. Roddick, J.F., et al.: Evolution and Change in Data Management - Issues and Directions. SIGMOD Record 29(1), 21–25 (2000)
21. Roddick, J.F.: A survey of schema versioning Issues for database systems. Information Software Technology J. 37(7) (1995)
22. Simitsis, A., Vassiliadis, P., Terrovitis, M., Skiadopoulos, S.: Graph-based modeling of ETL activities with multi-level transformations and updates. In: Tjoa, A.M., Trujillo, J. (eds.) DaWaK 2005. LNCS, vol. 3589, pp. 43–52. Springer, Heidelberg (2005)
23. Tsichritzis, D., Klug, A.C.: The ANSI/X3/SPARC DBMS Framework Report of the Study Group on Database Management Systems. Information Systems 3(3), 173–191 (1978)
24. Velegrakis, Y., Miller, R.J., Popa, L.: Preserving mapping consistency under schema changes. VLDB J. 13(3), 274–293 (2004)
25. Zicari, R.: A framework for schema update in an object-oriented database system. In: Proc. Seventh International Conference on Data Engineering (ICDE 1991), pp. 2–13 (1991)

Appendix

In Table A.1, the statuses, i.e., the actions dictated at the detailed level of nodes, assigned to visited nodes by *Propagate Changes* Algorithm for combinations of events and types of nodes are shown, when *propagate* policy prevails on the visited node. For each status the new event induced by the assignment of a node with status, which is further propagated to the graph, is also shown.

Table A.1. Statuses assigned to nodes when propagate policy prevails

Event on the graph		**On node**	**Scope**[1]	**Status**	**Raised Event**
Add	R/V/Q	*None affected*	*N/A*	*N/A*	*N/A*
	A	R	S	Add Child	Add Attribute
		V	S	Add Child	Add Attribute
		Q	S	Add Child	Add Attribute
	C	R	C	Add Child	Add Condition
		V	S, P	Add Child, Modify Provider	Add Condition, Modify Condition
		Q	S, P	Add Child, Modify Provider	Add Condition, Modify Condition
		A	S	Add Child	Add Condition
	GB	V	S, P	Add Child, Modify Provider	Add GB, Modify GB
		Q	S, P	Add Child, Modify Provider	Add GB, Modify GB
	OB	V	S, P	Add Child, Modify Provider	Add OB, Modify OB
		Q	S, P	Add Child, Modify Provider	Add OB, Modify OB

Delete	R	R	S	Delete Self	Delete Relation[2]
	V	V	S	Delete Self	Delete View[2]
	Q	Q	S	Delete Self	Delete Query[2]
	A	R	C	Delete Child	*None*
		V	C	Delete Child	*None*
		Q	C	Delete Child	*None*
		A	S	Delete Self	Delete Attribute
		C	P	Delete Self	Delete Condition
		F	P	Delete Self	Delete Function
		GB	P	Delete Self, Modify Self[3]	Delete GB, Modify GB
		OB	P	Delete Self, Modify Self[3]	Delete OB, Modify OB
	C	R	C	Delete Child	Delete Condition
		V	C, P	Delete Child, Modify Provider	Delete Condition, Modify Condition
		Q	C, P	Delete Child, Modify Provider	Delete Condition, Modify Condition
		A	C	Delete Child	Delete Condition
		C	S, C	Delete Self, Delete Child	Delete Condition, Modify Condition
	F	A	C	Delete Self	Delete Attribute
		C	C	Delete Self	Delete Condition
		F	C	Delete Self	Delete Function
		GB	C	Delete Self, Modify Self[3]	Delete GB, Modify GB
		OB	C	Delete Self, Modify Self[3]	Delete OB, Modify GB
	GB	V	C, P	Delete Child, Modify Provider	Delete GB, Modify GB
		Q	C, P	Delete Child, Modify Provider	Delete GB, Modify GB
		GB	S	Delete Self	Delete GB
	OB	V	C, P	Delete Child, Modify Provider	Delete OB, Modify OB
		Q	C, P	Delete Child, Modify Provider	Delete OB, Modify OB
		OB	S	Delete Self	Delete OB

Rename	R	R	S	Rename Self	Rename Relation
		V	P	Rename Provider	None
		Q	P	Rename Provider	None
	V	V	S	Rename Self	Rename View
		Q	P	Rename Provider	None
	A	R	C	Rename Child	None
		V	C	Rename Child	None
		Q	C	Rename Child	None
		A	S	Rename Self	Rename Attribute
		C	P	Rename Provider	None
		F	P	Rename Provider	None
		GB	P	Rename Provider	None
		OB	P	Rename Provider	None
Modify Domain	A	R	C	Modify Child	*None*
		V	C	Modify Child	*None*
		Q	C	Modify Child	*None*
		A	S	Modify Self	Modify Attribute
		C	P	Modify Provider	Modify Condition
		F	P	Modify Provider	Modify Function
		GB	P	Modify Provider	Modify GB
		OB	P	Modify Provider	Modify OB

Modify	C	R	C	Modify Child	Modify Condition
		V	C, D	Modify Child, Modify Provider	Modify Condition
		Q	C, D	Modify Child, Modify Provider	Modify Condition
		A	C	Modify Child	Modify Condition
		C	S, C	Modify Self, Modify Child	Modify Condition
	F	A	C	Modify Self	Modify Attribute
		C	C	Modify Self	Modify Condition
		GB	C	Modify Self	Modify GB
		OB	C	Modify Self	Modify OB
	GB	V	C, D	Modify Child, Modify Provider	Modify GB
		Q	C, D	Modify Child, Modify Provider	Modify GB
	OB	V	C, D	Modify Child, Modify Provider	Modify OB
		Q	C, D	Modify Child, Modify Provider	Modify OB
	P	P	S	Modify Self	Modify Parameter
		C	C	Modify Self	Modify Condition

[1]**Scope**: S (SELF), C(CHILD), P(PROVIDER)
[2]All attributes in the schema are first deleted before Delete Relation, Delete View and Delete Query events occur.
[3]The value for the status depends on whether GB / OB node have other children. If no other children exist then Delete GB/OB is assigned, Modify GB/OB otherwise.