Rule-Based Management of Schema Changes at ETL Sources

George Papastefanatos[1], Panos Vassiliadis[2], Alkis Simitsis[3], Timos Sellis[4], and Yannis Vassiliou[1]

[1] National Technical University of Athens
{gpapas,yv}@dblab.ece.ntua.gr
[2] University of Ioannina
pvassil@cs.uoi.gr
[3] HP Labs
alkis@hp.com
[4] Institute for the Management of Information Systems
timos@imis.athena-innovation.gr

Abstract. In this paper, we visit the problem of the management of inconsistencies emerging on ETL processes as results of evolution operations occurring at their 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 rules for the management of evolution events. Given a change at an element of the graph, our framework detects the parts of the graph that are affected by this change and highlights the way they are tuned to respond to it. We then present the system architecture of a tool called Hecataeus that implements the main concepts of the proposed framework.

Keywords: ETL Schema Evolution, Hecataeus.

1 Introduction

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, generally 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.

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

J. Grundspenkis et al. (Eds.): ADBIS 2009 Workshops, LNCS 5968, pp. 55–62, 2010.
© Springer-Verlag Berlin Heidelberg 2010

deleted or renamed. Such a change affects the entire workflow, possibly, all the way to the warehouse, along with any reports over the warehouse tables. Similarly, assume that the warehouse designer wishes to add an attribute to a source relation. Should this change be propagated to ETL activities that depend on this source? Research has extensively dealt with the problem of schema evolution, in data warehouses [1, 2, 3, 8, 11, 12] and materialized views [4, 5, 6]. Although several 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 the described setting.

In this paper, we sketch a framework for detecting and resolving inconsistencies emerging on ETL processes as results of evolution operations. *The goal is to provide a mechanism to the designer for the smooth adaptation of ETL scenarios to evolution changes occurring at their sources as well as for the early detection of vulnerable parts in the overall design.* The proposed framework employs a representation technique that maps all the essential constructs of an ETL configuration to graphs. Thus, its basis is a graph model, called *evolution graph*, which models in a coherent and uniform way internal structural elements of an ETL process such source relations, activities, queries extracted from ETL procedures, etc.

We furthermore provide a suitable technique for handling changes occurring in the ETL source schema, in such way that the human interaction is minimized. The provided technique enriches the evolution graph with semantics, namely evolution events and rules, called policies in our framework, that predetermine the impact of changes on the graph constructs. These rules dictate the actions that are performed, when additions, deletions or modifications occur on the DW sources. Specifically, 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, proposes the readjustment of the graph to fit to the new semantics imposed by the change. All of the above concepts are implemented in a powerful and user friendly tool, called HECATAEUS.

Theoretical aspects concerning the employed graph model, the proposed rule-based framework as well as its experimental evaluation over real case ETL scenarios have been thoroughly presented in [9, 10]. In this paper we provide in details the internals of the system architecture of the proposed tool.

2 Graph-Based Modeling of ETL Processes

We employ a graph theoretic approach to capture the various and complex schema dependencies that exist between software modules comprising an ETL process. The proposed graph modeling uniformly covers relational tables, views, ETL activities, database constraints and SQL queries as first class citizens. All the aforementioned constructs are mapped to a graph, that we call *Evolution Graph*. 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. Originally, the model was introduced in [10] and here, we provide an extended summary.

Each **relation** $R(\Omega_1,\Omega_2,\ldots,\Omega_n)$ 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 *relation node*, R, representing the relation schema; n *attribute nodes*, one for

each of the attributes; and n *schema relationships*, directing from the relation node towards the attribute nodes, indicating that the attribute belongs to the relation. Constraints – i.e., primary/foreign key, unique, not null – are modeled with use of a separate *constraint node* (i.e., PK node, not null node, etc), connected via operand edges with the attribute(s) on which the constraint is applied.

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 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. The edges connecting the query node with its subgraph components (i.e., attributes contained in the SELECT clause, the relation nodes contained in the FROM clause, etc.) are annotated as *map-select*, *from, where, group-by* and *having relationships.* The direction of the edges is from the query subgraph towards its source subgraphs (i.e., the respective relations/views accessed by the query). WHERE and HAVING clauses are modeled via a left-deep tree of logical conditions to represent the selection formulae; the edges involved are annotated as *operand relationships*. Nested queries are also part of this modeling, too. For the representation of aggregate queries, we employ a new node denoted as GB, to capture the set of attributes acting as the aggregators; and one node per aggregate function labeled with the name of the employed aggregate function; e.g., COUNT, SUM, MIN.

Views are considered either as queries or relations (materialized views). They constitute both queries over the database schema as far as their definition is concerned and relations to other queries as far as their functionality and their extension are concerned. Their dual role is captured and represented as intermediate graphs between relations and queries.

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.

Lastly, an ETL **summary** is a directed acyclic graph acting as a zoomed-out variant of the detailed evolution graph**.** The set of nodes comprises all activities, relations and views that participate in an ETL process and the edges connect the providers and consumers.

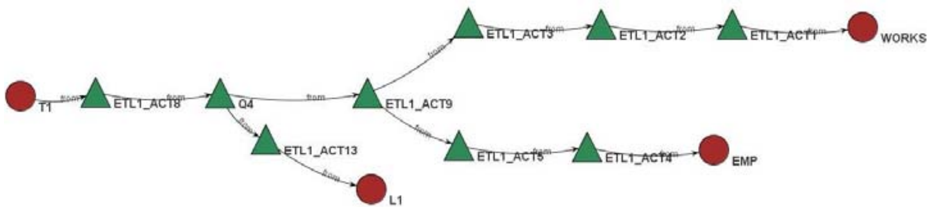

Fig. 1. Zoomed-out view of an ETL scenario

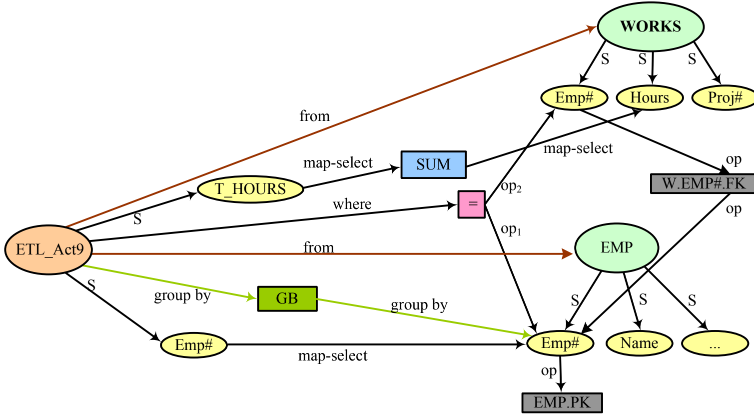

Fig. 2. Detail graph representation of ETL_Act9 activity

Figure 1 shows the summary of a simple ETL workflow involving 9 activities (green triangles) two data sources, (i.e., `EMP`, `WORKS`), one lookup table, (i.e., L_1) and one target table T_1 of the DW.

Fig. 2 depicts the detailed graph representation for a specific activity, namely `ETL_Act9` of the ETL summary, containing the following aggregate query:

```
Act9: SELECT EMP.Emp# as Emp#, Sum(WORKS.Hours) as T_Hours
      FROM   EMP, WORKS
      WHERE  EMP.Emp# = WORKS.Emp#
      GROUP  BY EMP.Emp#
```

3 Regulating Schema Evolution

The basic aspects of our framework involve the detection of the parts of the system, which are affected by an evolution change and the regulation of their reaction to this change. Therefore, we first, exploit the dependencies which are represented as edges in the evolution graph to both detect syntactical and semantic inconsistencies following a schema evolution event. We furthermore regulate the impact of an evolution event towards the nodes of the graph by *annotating the graph with rules, called policies*. The adaptation of a node to an evolution event and furthermore the propagation of the event towards the rest of the graph is dictated by the rule defined on the node. The proposed framework enables the user to proactively identify and regulate the impact of evolution processes. It provides the appropriate semantics to perform hypothetical evolution scenarios and test alternative evolution policies for a given configuration before the evolution process is applied on a production environment.

In such manner, each graph 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 the space of hypothetical actions (addition/ deletion/modification) by the space of graph constructs sustaining evolution changes (e.g., nodes for relations, attributes, conditions, etc.). For each of the above events, the administrator annotates graph constructs with policies that dictate the way they will react to an event when affected. 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 [6, 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 [6, 7].

Specifically, given an *event* altering the source database schema our framework determines those activity graph constructs that are directly connected to the source altered and thus affected by the event. For each affected construct, its prevailing policy is determined. According to the prevailing policy, the status of each construct is set. 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.

Example. Consider the simple example query `SELECT * FROM EMP` as part of the `ETL_ACT4` of Fig 1. Assume that the provider relation EMP is extended with a new attribute `PHONE`. There are two possibilities: First, 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`. Alternatively, 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.

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.

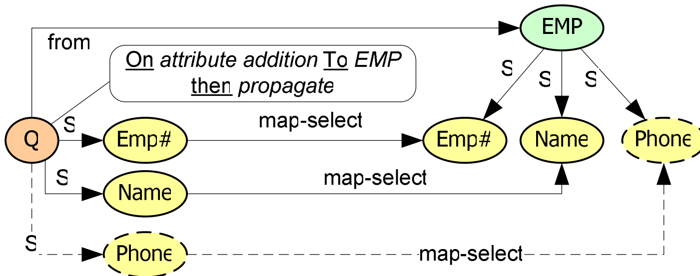

Fig. 3. Propagating addition of attribute PHONE

- *Prompt*. In this case (default, for reasons of backwards compatibility), 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 2. 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.

4 System Architecture

In the context of the proposed framework, we have implemented a tool, called Hecataeus, used for the construction and visualization of the evolution graph, its annotation with policies regarding evolution semantics, and lastly the management of evolution propagation towards the graph. Hecataeus enables the user to transform ETL activities abstracted as SQL source code to evolution graphs, explicitly define policies and evolution events on the graph and determine affected and adjusted graph constructs according to the proposed framework. The graph modeling of the environment has versatile utilizations: apart from the impact prediction and the creation of hypothetical evolution scenarios, the user may also assess several graph-theoretic metrics of the graph that highlight sensible regions of the graph. Hecataeus is a user-friendly visual environment that helps administrators and users to perform hypothetical evolution scenarios on database applications.

The main packages of Hecataeus are shown in the diagram of Fig. 4. The *Parser* is responsible for parsing the input files (i.e., DDL and workload definitions). The functionality of the *Catalog* is to maintain the schema of relations, views, etc., as well as to validate the syntax of the workload processed (i.e., activity definitions, queries, views) by the Parser. The *Evolution Manager* is responsible for representing the underlying schema and the parsed queries abstracted from ETL activities 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 holds all the evolution semantics for each graph construct (i.e., events, policies) and algorithms for performing evolution scenarios. The *Metric Manager* is responsible for maintaining the metrics definition and for their application on the graph. Each metric applied on the evolution graph is implemented as a separate

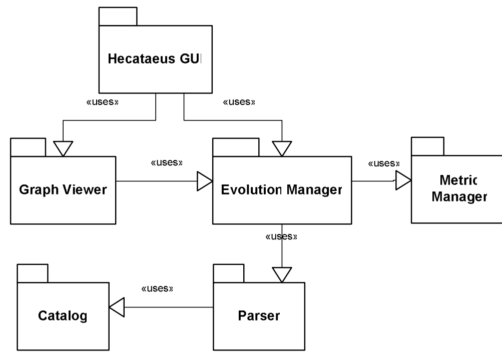

Fig. 4. System Architecture

function in the Metric Manager. The *Graph Viewer* is responsible for the management of the visual properties of the graph. It communicates with the Evolution Manager, which holds all evolution semantics and methods. Graph Viewer offers distinct colorization and shapes for each set of nodes and edges according to their types and the way they are affected by evolution events. It applies layout algorithms on the graph, adjusts the visibility of nodes and visualizes the graph at different levels of abstraction. Lastly, the Hecataeus *GUI* is responsible for the interaction with the user offering a large variety of functions, such as editing of the graph properties, addition, deletion and modification of nodes, edges and policies. The GUI package enables the user to raise evolution events, to detect affected nodes by each event and highlight appropriate transformations of the graph. Lastly, it offers the import or export of evolution scenarios to XML or image formats (i.e., jpeg).

In Fig. 5 the class diagram of the core component of Hecataeus, i.e., Evolution Manager is shown. The `EvolutionGraph` class comprises a collection of *nodes* and *edges*, which belong to a certain *type* (i.e., relation node, from edge, etc.). Each node *is annotated* with a collection of policies; each of them has a *type* (i.e., propagate, block or prompt) for *handling* an event. Additionally, a node *sustains* a collection of

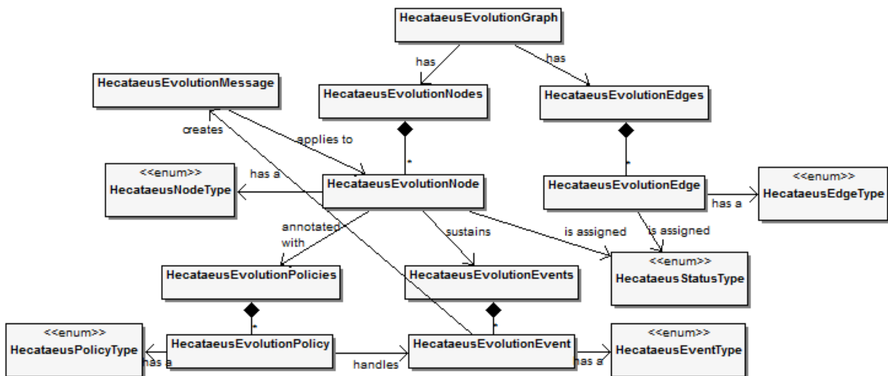

Fig. 5. Evolution Manager Class Diagram

events, which belong to a specific *event type* (i.e., delete attribute, rename relation, etc.) according to the type of node on which they occur. Lastly, a message is *created* for each event occurred on a node of the graph and transmits the impact of the event towards the adjacent nodes. Nodes handle the event and according to the prevailing policy *are assigned with* a status determining the action that is performed on them.

5 Conclusions

In this paper, we have dealt with the internals of a system, Hecataeus that handles the schema evolution in ETL environment. Our goal was to provide a coherent framework for appropriately propagating potential changes occurring at the ETL sources to all affected parts of the system, with a limited overhead imposed on both the system and the humans, who design and maintain it. Toward that aim, we have modeled the internal parts of ETL activities as the constituents of a dependency graph and we annotate parts of this graph with rules that regulate the propagation of evolution changes towards the whole workflow. In this paper we have presented the internal architecture of Hecataeus, which has been specifically designed in an extensible fashion to allow the future incorporation of different kinds of events and policies.

References

1. Golfarelli, M., Lechtenbörger, J., Rizzi, S., Vossen, G.: Schema Versioning in Data Warehouses. In: ECDM 2004, pp. 415–428 (2004)
2. Blaschka, M., Sapia, C., Höfling, G.: On Schema Evolution in Multidimensional Databases. In: Mohania, M., Tjoa, A.M. (eds.) DaWaK 1999. LNCS, vol. 1676, pp. 153–164. Springer, Heidelberg (1999)
3. Kaas, C., Pedersen, T.B., Rasmussen, B.: Schema Evolution for Stars and Snowflakes. In: ICEIS (2004)
4. Bellahsene, Z.: Schema evolution in data warehouses. Knowledge and Information Systems 4(2) (2002)
5. Mohania, M., Dong, D.: Algorithms for adapting materialized views in data warehouses. In: CODAS (1996)
6. 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, p. 359. Springer, Heidelberg (1998)
7. Velegrakis, Y., Miller, R.J., Popa, L.: Preserving mapping consistency under schema changes. VLDB J. 13(3) (2004)
8. 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, p. 178. Springer, Heidelberg (2000)
9. Papastefanatos, G.: Policy Regulated Management of Schema Evolution in Database–centric Environments. PhD Thesis. NTUA (February 2009)
10. 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)
11. Wrembel, R., Bebel, B.: Metadata Management in a Multiversion Data Warehouse. J. Data Semantics (8), 118–157 (2007)
12. Golfarelli, M., Rizzi, S.: A Survey on Temporal Data Warehousing. IJDWM 5(1), 1–17 (2009)