What-if Analysis for Data Warehouse Evolution

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

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

Abstract. In this paper, we deal with the problem of performing what-if analysis for changes that occur in the schema/structure of the data warehouse sources. We abstract software modules, queries, reports and views as (sequences of) queries in SQL enriched with functions. Queries and relations 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 indicates the way they are tuned to respond to it.

1. Introduction

Data warehouses are complicated software environments where data stemming from operational sources are extracted, transformed, cleansed and eventually loaded 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 stored in data marts, reports, spreadsheets, and other formats. The whole environment involves 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 a data warehouse environment.

Figure 1 depicts a simplified version of an Extraction-Transformation-Loading (ETL) process. Data are extracted from sources and they are transferred to the Data Staging Area (DSA), where their contents and structure are modified; example transformations include joins, addition of new attributes produced via functions, 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 DW it is possible that several counterparts of the ETL process may be evolved. For instance, assume that an attribute is deleted from the underlying database S1 or it is added to the source relation S2. Such changes affect the entire workflow, possibly all the way to the warehouse (tables T1 and T2), along with any reports over the warehouse tables (abstracted as queries over view V3).

Research has extensively dealt with the problem of schema evolution, in object-oriented databases [1, 11, 15], ER diagrams [22], data warehouses [6, 16, 17, 18] and

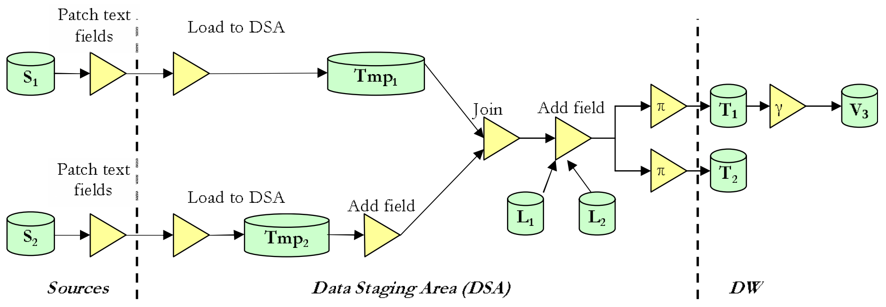


Fig. 1: A simple ETL workflow

materialized views [2, 5, 7, 8]. However, to the best of our knowledge, there is no global framework for the management of evolution in the described setting.

In this paper, we provide a general mechanism for performing what-if analysis [19] for potential changes of data source configurations. We introduce a graph model that uniformly models relations, queries, views, ETL activities, and their significant properties (e.g., conditions). Apart from the simple task of capturing the semantics of a database system, the graph model allows us to predict the impact of a change over the system. We provide a framework for annotating the database graph with policies concerning the behavior of nodes in the presence of hypothetical changes. In addition, we provide a set of rules that dictate the proper actions, when additions, deletions or updates are performed to relations, attributes, and conditions. (All the above concepts are treated as first-class citizens in our model.) Assuming that a graph construct is annotated with a policy for a particular event (e.g., an activity node is tuned to deny deletions of its provider attributes), the proposed framework has the following features: (a) it performs the identification of the affected subgraph; and (b) if the policy is appropriate, it automates the readjustment of the graph to fit the new semantics imposed by the change. Finally, we experimentally assess our proposal.

Outline. Section 2 presents the graph model for databases. Section 3 proposes a framework of graph annotations and readjustment automation for database evolution. Section 4 presents the results of a case study for our framework. Section 5 discusses related work. Finally, Section 6 concludes and provides insights for future work.

2. Graph-based modeling of 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 have been presented in our previous work [9,10].

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

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

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

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

Select part. Each query is assumed to own a schema that comprises the attributes appearing in the SELECT clause, either with their original or alias names. 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 the WHERE and/or HAVING clauses of a query in conjunctive normal form. Thus, we introduce two directed edges, namely *where relationships*, $\mathbf{E_W}$, and *having relationships*, $\mathbf{E_H}$, both starting from a query node towards an operator node corresponding to the conjunction of the highest level.

Group and Order By part. For the representation of aggregate queries, two special purpose nodes are employed: (a) a new node denoted as GB∈**GB**, to capture the set of attributes acting as the aggregators; and (b) one node per aggregate function labeled with the name of the employed aggregate function; e.g., COUNT, SUM, MIN. For the aggregators, we use edges directing from the query node towards the GB node that are labeled <group-by>, indicating *group-by relationships*, $\mathbf{E_G}$. The GB node is connected

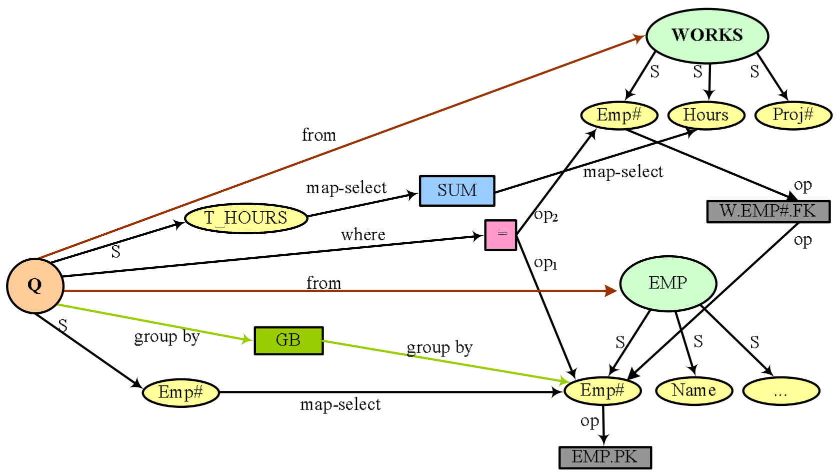


Fig. 2. Graph of an example aggregate query annotated with policies [10]

with each of the aggregators through an edge tagged also as `<group-by>`, directing from the `GB` node towards the respective attributes. These edges are additionally tagged according to the order of the aggregators; we use an identifier `i` to represent the i-th aggregator. Moreover, for every aggregated attribute in the query schema, there exists an edge directing from this attribute towards the aggregate function node as well as an edge from the function node towards the respective relation attribute. Both edges are labeled `<map-select>` and belong to $\mathbf{E_M}$, as these relationships indicate the mapping of the query attribute to the corresponding relation attribute through the aggregate function node. The representation of the `ORDER BY` clause of the query is performed similarly, whereas nested queries and functions used in queries are also incorporated in our model [21].

Views, V. Views are considered either as queries or relations (materialized views), thus, $\mathbf{V} \subseteq \mathbf{R} \cup \mathbf{Q}$.

Figure 2 depicts the proposed graph representation for an example aggregate query.

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

As far as modification queries are concerned, their behavior with respect to adaptation to changes in the database schema can be captured by `SELECT` queries. For lack of space, we simply mention that (a) `INSERT` statements can be dealt as simple `SELECT` queries and (b) `DELETE` and `UPDATE` statements can also be treated as `SELECT` queries, possibly comprising a `WHERE` clause.

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 acts as a zoomed-out variant of the full graph **G** [20]. V_s comprises of activities, relations and views that participate in an ETL process. 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. Figure 1 depicts an ETL summary.

Components. A component 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 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.

3. Adapting ETL workflows for evolution of sources

In this section, we formulate a set of rules that allow the identification of the impact of changes to an ETL workflow and propose an automated way to respond to these changes. Evolution changes may affect the software used in an ETL workflow (like queries, stored procedures, and triggers) 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.

The proposed rules annotate the graph representing the ETL workflow 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 a what-if analysis 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. To alleviate the designer from the burden of manually annotating all graph constructs, a simple extension of SQL with clauses concerning the evolution of important constructs is proposed in the long version of this paper [21].

3.1 The general framework for 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 what-if analysis. Each such construct is enriched with policies that allow the designer to specify the behavior of the annotated construct whenever events that alter the database graph occur. The combination of an 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. The space of hypothetical actions (addition, deletion, and modification) over 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.

Algorithm *Propagate changeS (PS)*

```
Input: an ETL summary S over a graph $G_o=(V_o,E_o)$ and an event e
Output: a graph $G_n=(V_n,E_n)$
Variables: a set of events E, and an affected node A
Begin
  dps(S, $G_o$, $G_n$, {e}, A)
End

dps(S, $G_n$, $G_o$, E, A) {
  I = Ins_by_policy(affected(E))
  D = Del_by_policy(affected(E))
  $G_n = G_o - D \cup I$
  E = E-{e}∪action(affected(E))
  if consumer(A)≠nil
    for each consumer(A)
      dps(S,$G_n$,$G_o$,E,consumer(A))
}
```

Fig. 3. Algorithm Propagate changeS (PS)

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 [8, 14]; 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 [8, 14].

Our framework prescribes the *reaction* of the parts of the system affected by a hypothetical schema change based on their annotation with policies. The mechanism that determines the reaction to a change is formally described by the algorithm *Propagate changeS* (*PS*) in Figure 3. Given an ETL summary `S` over a graph G_o and an event `e`, *PS* produces a new ETL summary G_n, which has absorbed the changes.

Example. Consider the simple example query `SELECT * FROM EMP` as part of an ETL activity. Assume that 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., 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 (i.e., node, edges) 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

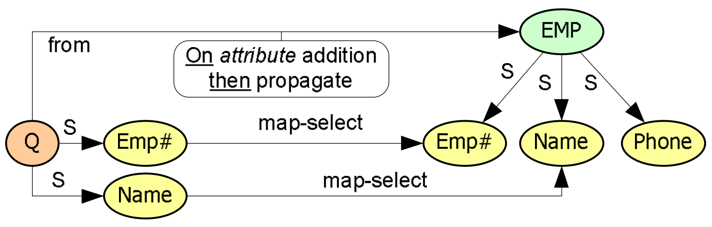


Fig. 4: Propagating addition of attribute PHONE

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 (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 4. The annotation of the `FROM` edge as *propagating addition* indicates that the addition of `PHONE` node will be propagated to the query and the new attribute is included in the `SELECT` clause of the query. If a `FROM` edge is not tagged with this additional information, then a *default case* is assumed and the designer/administrator is prompted to decide.

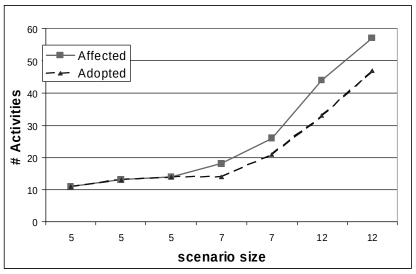
4. Case Study

We have evaluated the effectiveness of our setting via the reverse engineering of real-world ETL processes, extracted from an application of the Greek public sector. We have monitored the changes that took place to the sources of the studied data warehouse. In total, we have studied a set of 7 ETL processes, which operate on the data warehouse. These processes 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 aforementioned scenarios comprise a total number of 53 activities.

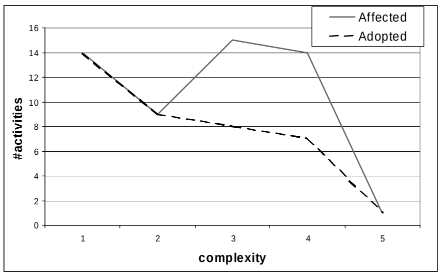
Table 1 illustrates the changes that occurred on the schemata of the source and lookup tables, such as renaming source tables, renaming attributes of source tables, adding and deleting attributes from source tables, modifying the domain of attributes and lastly changing the primary key of lookup tables. After the application of these changes to the sources of the ETL process, each affected activity was properly readjusted (i.e., rewriting of queries belonging to activities) in order to adapt to the changes. For each event, we counted: (a) the number of activities affected both semantically and syntactically, (b) the number of activities, that have automatically been adjusted by our framework (*propagate* or *block* policies) as opposed to those (c) that required administrator’s intervention (i.e., a *prompt* policy).

Source Name	**Event Type**	**Activities (53)**			
		Affected	**Autom. adjusted**	**Prompt**	**%**
S_1	Rename	14	14	0	100%
	Rename Attributes	14	14	0	100%
	Add Attributes	34	31	3	91%
	Delete Attributes	19	18	1	95%
	Modify Attributes	18	18	0	100%
S_4	Rename	4	4	0	100%
	Rename Attributes	4	4	0	100%
	Add Attributes	19	15	4	79%
	Delete Attributes	14	12	2	86%
	Modify Attributes	6	6	0	100%
S_2	Rename	1	1	0	100%
	Rename Attributes	1	1	0	100%
	Add Attributes	4	4	0	100%
	Delete Attributes	4	3	1	75%
S_3	Rename	1	1	0	100%
	Rename Attributes	1	1	0	100%
S_5	Modify Attributes	3	3	0	100%
S_6	*NO_CHANGES*	0	0	0	-
S_7	Rename	1	1	0	100%
	Rename Attributes	1	1	0	100%
L_1	*NO_CHANGES*	0	0	0	-
L_2	Add Attributes	1	0	1	0%
L_3	Add Attributes	9	0	9	0%
	Change PK	9	0	9	0%

Table 1: Analytic results



(a)



(b)

Fig. 5: Evaluation of our method

Figure 5a depicts the correlation between the average number of affected activities versus automatically adapted activities w.r.t. the total number of activities contained in ETL scenarios. For ETL processes comprising a small number of activities most affected activities are successfully adjusted to evolution changes. For longest ETL processes, the number of automatically adjusted activities increases proportionally to the number of affected activities. Furthermore, Table 1 shows that our framework can successfully handle and propagate evolution events to most activities, by annotating the queries included in them with policies. Activities requiring administrator's intervention are mainly activities executing complex joins, e.g., with lookup tables, for which the administrator must decide upon the proper rewriting. Figure 5b presents the average amount of automatically adapted activities w.r.t. the complexity of activities. Complexity refers to the functionality of each activity; e.g., the type of transformation it performs or the types of queries it contains. In general, our framework handles efficiently simple activities. More complex activities, e.g., pivoting activities, are also adequately adjusted by our approach to evolution changes.

5. Related Work

Evolution. Related research work has studied in the past the problem of database schema evolution. A survey on schema versioning and evolution is presented in [13], whereas a categorization of the overall issues regarding evolution and change in data management is presented in [12]. The problem of view adaptation after redefinition is mainly investigated in [2, 5, 7], where changes in views definition are invoked by the user and rewriting is used to keep the view consistent with the data sources. In [6], the authors discuss versioning of star schemata, where histories of the schema are retained and queries are chronologically adjusted to ask the correct schema. The warehouse adaptation for SPJ views is studied in [2]. Also, the view synchronization problem considers that views become invalid after schema changes in the underlying base relations [8]. Our work in this paper builds mostly on the results of [8], by extending it to incorporate attribute additions and the treatment of conditions. The treatment of attribute deletions in [8] is quite elaborate; we confine to a restricted version to avoid overcomplicating both the size of requested metadata and the language extensions. Still, the [8] tags for deletions may be taken into consideration in our method. Finally, the algorithms for rewriting views when the schemas of their source data change (e.g., [2, 5]), are orthogonal to our approach. Thus, our approach can be extended in the presence of new results on such algorithms.

Model mappings. Model management provides a generic framework for managing model relationships, comprising three fundamental operators: match, diff, and merge [3,4]. Our proposal assigns semantics to the match operator for the case of model evolution, where the source and target models of the mapping are the original and resulting database graph, respectively, after evolution management has taken place. A similar framework for the management of evolution has been proposed [14]. Still, the model of [14] is more restrictive, in the sense that it is intended towards retaining the original semantics of the queries. Our work is a larger framework that allows the restructuring of the database graph (i.e., model) either towards keeping the original semantics or towards its readjustment to the new semantics.

6. Conclusions and future work

In this paper, we have discussed the problem of performing what-if analysis for changes that occur in the schema/structure of the data warehouse sources. We have modeled software modules, queries, reports and views as (sequences of) queries in SQL extended with functions. Queries and relations have uniformly been modeled as a graph that is annotated with policies for the management of evolution events. We have presented an algorithm that detects the parts of the graph that are affected by a given change and highlights the way they are tuned to respond to it. Finally, we have evaluated our approach over cases extracted from real world scenarios.

Future work may be directed towards many goals, with patterns of evolution sequences being the most prominent one.

References

1. J. Banerjee et al. Semantics and implementation of schema evolution in object-oriented databases. In SIGMOD, 1987.
2. Z. Bellahsene. Schema evolution in data warehouses. In Knowledge and Information Systems 4(2), 2002.
3. P. Bernstein, A. Levy, R. Pottinger. A Vision for Management of Complex Models. In SIGMOD Record 29(4), 2000.
4. P. Bernstein, E. Rahm. Data Warehouse Scenarios for Model Management. In ER, 2000.
5. A. Gupta, I. S. Mumick, J. Rao, K. A. Ross. Adapting materialized views after redefinitions: Techniques and a performance study. In Information Systems (26), 2001.
6. M. Golfarelli, J. Lechtenbörger, S. Rizzi, G. Vossen, Schema Versioning in Data Warehouses, ECDM 2004, Pages 415 – 428.
7. M. Mohania, D. Dong. Algorithms for adapting materialized views in data warehouses. In CODAS, 1996.
8. A. Nica, A. J. Lee, E. A. Rundensteiner. The CSV algorithm for view synchronization in evolvable large-scale information systems. In EDBT, 1998.
9. G. Papastefanatos, P. Vassiliadis, Y. Vassiliou. Adaptive Query Formulation to Handle Database Evolution. In CAiSE Forum, 2006.
10. G. Papastefanatos, K. Kyzirakos, P. Vassiliadis, Y. Vassiliou. Hecataeus: A Framework for Representing SQL Constructs as Graphs. In EMMSAD, 2005.
11. Y.G. Ra, E.A. Rundensteiner. A transparent object-oriented schema change approach using view evolution. In ICDE, 1995.
12. J.F. Roddick et al. Evolution and Change in Data Management - Issues and Directions. In SIGMOD Record 29(1), 2000.
13. J.F. Roddick. A survey of schema versioning Issues for database systems. In Information Software Technology 37(7), 1995.
14. Y. Velegrakis, R.J. Miller, L. Popa. Preserving mapping consistency under schema changes. In VLDB J. 13(3), 2004.
15. R. Zicari. A framework for schema update in an object-oriented database system. In ICDE, 1991.
16. M. Blaschka, C. Sapia, G. Höfling. On Schema Evolution in Multidimensional Databases. In DaWaK, 1999.
17. C. Kaas, T. B. Pedersen, B. Rasmussen. Schema Evolution for Stars and Snowflakes. In ICEIS, 2004.
18. M. Bouzeghoub, Z. Kedad: A Logical Model for Data Warehouse Design and Evolution. In DaWaK, 2000.
19. M. Golfarelli, S. Rizzi, A. Proli: Designing what-if analysis: towards a methodology. In DOLAP, 2006.
20. A. Simitsis, P. Vassiliadis, M. Terrovitis, S. Skiadopoulos: Graph-Based Modeling of ETL Activities with Multi-level Transformations and Updates. In DaWaK 2005.
21. G. Papastefanatos, P. Vassiliadis, A. Simitsis, Y. Vassiliou: What-if Analysis for Data Warehouse Evolution (Extended Version). Working Draft April 2007, url: www.dbnet.ece.ntua.gr/~gpapas/Publications/ DataWarehouseEvolution-Extended.pdf
22. C.T. Liu, P.K. Chrysanthis, S.K. Chang. Database schema evolution through the specification and maintenance of changes on entities and relationships. In ER, 1994.