FOUNDATIONS OF COMPUTING AND DECISION SCIENCES
Vol. 38 (2013) No. 2

DOI: 10.2478/fcds-2013-0005 ISSN 0867-6356
e-ISSN 2300-3405

E-ETL: Framework for Managing Evolving ETL Workflows

Artur WOJCIECHOWSKI *

Abstract. Data warehouses integrate external data sources (EDSs), which very often change their data structures (schemas). In many cases, such changes cause an erroneous execution of an already deployed ETL workflow. Structural changes of EDSs are frequent, therefore an automatic reparation of an ETL workflow, after such changes, is of a high importance. This paper presents a framework, called *E-ETL*, for handling the evolution of an ETL layer. Detection of changes in EDSs causes a reparation of the fragment of ETL workflow which interacts with the changed EDSs. The proposed framework was developed as a module external to a standard commercial or open-source ETL engine, accessing the engine by means of API. The innovation of this framework consists in: (1) the algorithms for semi-automatic reparation of an ETL workflow and (2) its ability to interact with various ETL engines that provide API.

Keywords: data warehouse, data integration, ETL, data source evolution, ETL evolution, E-ETL

1 Introduction

A data warehouse (DW) is usually created to integrate multiple heterogeneous, distributed, and autonomous external data sources (EDSs). Such integrated data can be used for analysis, called On-Line Analytical Processing (OLAP). One of the DW elements is an ETL workflow which extracts data from EDSs, transforms data into a common data model, cleans data (removes missing, inconsistent, and redundant values), integrates data, and loads them into a DW. An inherent feature of EDSs is their evolution in time with respect not only to their contents (data) but also to their structures (schemas). Since changes of EDSs structure may cause erroneous execution, after every such a change, an ETL workflow must be redesigned and redeployed.

*Poznań University of Technology, Institute of Computing Science, Poznań, Poland, artur.wojciechowski@cs.put.poznan.pl

Frequent manual modifications of an ETL workflow are complex, prone-to-fail, and time-consuming. Hence, it is of a high importance to develop methods for handling structural changes of EDSs and managing the evolution of the ETL workflow. So far research community has not given much attention to the evolution of the ETL layer and few solutions to this problem have been proposed, e.g., [1, 2].

Paper Contribution. This paper contributes a framework, called *E-ETL*, for: (1) detecting structural changes of EDSs and (2) handling the changes in the ETL layer. This paper extends our previous work [3] with detailed descriptions of the changes detection mechanism and evolution rules. Changes are detected either by means of Event-Condition-Action (triggers) mechanism or by means of comparing two consecutive EDS metadata snapshots. Detection of the EDS schema change causes a reparation of the ETL activities that interact with the changed EDS. The reparation of the ETL activities is guided by several customizable reparation algorithms. The proposed framework was developed as a module external to a standard commercial or open-source ETL engine. Communication between *E-ETL* and the ETL engine is realized by means of the ETL engine API. The framework is customizable and it allows to:

- work with different ETL engines that provide API communication,
- define the set of detected structural changes,
- modify and extend the set of algorithms for managing the changes,
- define rules for the evolution of ETL workflow,
- present to the user the impact analyses of the ETL workflow,
- store versions of the ETL workflow and history of EDS changes.
- framework has a graphical user interface for visualizing the ETL workflow.

Paper Organization. The paper is organized as follows. Section 2 presents the concept of the *E-ETL* framework. Section 3 the *E-ETL* internal metamodel. Section 4 introduces reparation algorithms. Section 5 overviews detected schema changes. Section 6 presents the evolution rules of ETL. Section 7 outlines research related to the topic of this paper. Section 8 summarizes the paper and outlines issues for future development.

2 Concept of the E-ETL Framework

The *E-ETL* project focuses on developing a method and a framework to support the semi-automatic evolution of ETL workflow. In particular, the research and development focus on: (1) the development of a prototype architecture, called *E-ETL* that will be able to co-operate with a leading commercial and open source ETL development environments, (2) a graphical interface for visualizing ETL workflows, (3) tools for detecting structural changes and propagating them into an ETL layer, (4) a language for defining rules for the evolution of ETL workflows, (5) a method for checking the validity of an evolved ETL workflow, (6) a metamodel for storing versions of ETL workflows.

E-ETL is designed to co-operate with ETL development environments (currently the Microsoft SQL Server Integration Services is supported). To this end, *E-ETL* is an external system to an ETL development environment. *E-ETL* connects to a development environment by means of API.

E-ETL analyses the design of an ETL workflow which is defined in an ETL development environment, and on the basis of this project an internal model of the ETL workflow is created. Next, an ETL designer defines a set of rules that specify how the ETL workflow should evolve in response to the detected changes. Then, when *E-ETL* detects structural changes in an EDS, it proposes semi-automatically (in some cases automatically) the modifications of the ETL workflow. After a user's acceptance of the changes, *E-ETL* applies them to the ETL workflow in the ETL development environment.

3 Internal metamodel

Different ETL development environments may use different data models. Therefore, the *E-ETL* framework uses its own internal data model that permits to unify work with external ETL systems. In this model, an ETL workflow is represented as a directed graph. Each activity in the ETL workflow is presented as *SuperNode. SuperNode* consists of *Nodes* that represents input and output parameters of an ETL activity. An input parameter can be a table attribute that the activity reads, a node in XML structure, or a column in a spreadsheet. Dependencies between nodes are determined by edges between nodes. So, if there is a directed edge from node A to node B, then this means that node B depends on node A. Such model permits to do impact analyses. The impact analyses mark the parts of an ETL workflow that has to evolve as the result of structural changes in EDSs. These analyses are done by selecting all nodes succeeding the nodes that have been changed (nodes that describe EDS attributes that have been changed). To make the internal metamodel more readable and organized *SuperNodes* can be grouped into *GroupNodes. GroupNodes* also can be grouped into *GroupNodes.* This mechanism of grouping allows user to work on different levels of details.

Figure 1 presents an example of an internal metamodel. It shows a fragment of an ETL workflow that reads the *People* table and splits read data basing on the *Age* attribute. In the next step *People* tuples are joined with data read from *Addresses* table and *addresses.csv* file. *SuperNodes* that represents ETL activities are depicted in the figure as labeled boxes (e.g., *SQL query (1)*, *Conditional split*). Attributes inside boxes (e.g., *Id*, *Street*, *Name*, *Status*) are *Nodes* and they represent ETL activity parameters (input or output).

Exemplary *SuperNode – SQL query (1)* defines an activity that is described as an SQL query that selects tuples with *Country* equal to 'Poland'. *Id*, *Street* and *City*

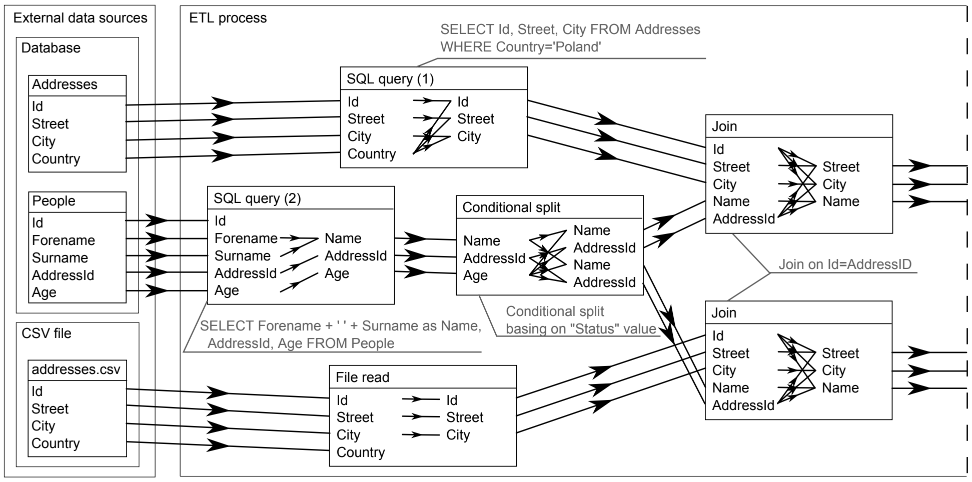

Figure 1: Internal metamodel example

output parameters depend respectively on *Id*, *Street*, and *City* input parameters. The SQL query that defines the exemplary activity contains also *WHERE* clause (Country='Poland'). Therefore modification or removal of *Country* input parameter would influence the result of the query. Therefore all output parameters also depend on *Country* input parameter. The dependencies are shown as directed edges between attributes.

4 Reparation algorithms

The detection of changes in an EDS fires the execution of algorithms that adapt an ETL workflow to the detected changes. These algorithms have been categorized as follows: *Defined rules*, *Replacer*, and *Alternative scenarios*. An ETL designer can specify algorithms that are supposed to be used to modify the ETL workflow, their priorities, and their parameters.

The *Defined rules* algorithm applies evolution rules, defined by a user, to particular elements of an ETL workflow. For each element (*Node*, *SuperNode*, or *GroupNode*) of an ETL workflow, a user can define whether this element is supposed to propagate the changes, to block them, to ask a user, or to fire action specific to a detected change.

The *Replacer* algorithm is based on the solution presented in [2]. For each element of an ETL workflow, a user can define whether this element can be replaced by other element. This replacement can be done when the element has been removed due to changes in EDS.

The *Alternative scenarios* algorithm repairs an ETL workflow basing on the fact that similar EDSs are usually processed in the same way and also the same changes on similar EDSs should be handled in the same way. Therefore, when the structure of one of the EDSs has changed, then the *Alternative scenarios* algorithm tries to find another EDS with a similar structure. After finding a similar EDS, operations related to both EDSs (the changed EDS and the similar one) are analyzed. This analysis provides information about differences between an ETL workflow fragment affected by the detected change and an ETL workflow fragment related to the similar EDS. Basing on this information, the *Alternative scenarios* algorithm proposes modifications of the ETL workflow fragment affected by the detected change. Since the history of the ETL workflow evolution and history of the EDSs changes are stored in the system, the *Alternative scenarios* algorithm can search not only in the current version of an ETL workflow, but also in their previously used versions. Such functionality may be useful when some changes will be undone in EDSs or changes will be done gradually in sequential EDSs.

5 Monitored Structural Changes

As mentioned before, *E-ETL* detects changes in EDSs either by comparing two successive snapshots of an EDS's metadata, or by the mechanism of triggers (if such triggers are supported and allowed to be installed in an EDS). Changes that can be detected are divided into two groups: *Collections* changes and *Collection element* changes. A *Collection* defines a set of tuples. *Collection elements* define elements of the tuple. Database table, spreadsheet, branch in XML are examples of *Collections* and respectively database table column, column in spreadsheet, node in XML are examples of *Collection elements*. Five changes can be distinguished for *Collections*: (1) *Add*, e.g. a new table addition in a database, (2) *Delete*, e.g. a deletion of a spreadsheet in Excel file, (3) *Rename*, e.g. a change of a file name, (4) *Split*, e.g. a partition of a table, (5) *Merge*, e.g. a merger of partitioned tables. Also five changes can be distinguished for *Collection elements*: (1) *Add*, e.g. a new column addition to a database table, (2) *Delete*, e.g. a deletion of a column in a spreadsheet, (3) *Rename*, e.g. a change of a node name in XML, (4) *Type*, e.g. a change of a column type form numeric to string, (5) *Length*, e.g. a change of a column type length from char(4) to char(8).

Regardless of the method used to detect changes in EDS's, *E-ETL* take snapshots of the EDS's structure. These snapshots are used to build representation of data sources in the *E-ETL* internal metamodel. When an EDS is a database, the snapshot can be taken form database metadata (system tables/views that describes user objects). Since every database management system (DBMS) vendor can have its own structure of metadata there are special methods that take snapshots for different DBMSs. The case when an EDS is an Excel or CSV file is more complicated. In such types of files there is no metadata that define the data structure so a snapshot must be build directly from the data. Usually, in spreadsheets and CSV files the first row is a header of the table and it contains names of the columns in that table. By taking and analyzing a sample of data in the column we can discover the type of the column. A snapshot of an XML file is build basing on XSD or DTD description that defines that file.

5.1 Comparing snapshots of an EDS's metadata

The process of comparing two successive snapshots of an EDS's metadata can be divided into the six following steps.

1. The first step is to check which collections from the old snapshot are not present in the new snapshot. Found collections are marked as deleted collections.

2. The second step is to check which collections from the new snapshot are not present in the old snapshot. Found collections are marked as added collections. Those to steps are based only on names of the collections.

3. The third step is to discover changes of collections names. Every deleted collection is compared with every added collection. Since the change of collection name may be connected with changes of its elements, the structure of the renamed collection may be not equal to the structure of the old collection. Therefore, the comparison of two collections is a calculation of a similarity that is based on an edit distance measure. The less changes must be done in one collection to make its structure the same as the structure of the second collection, the more similar these collections are. The following editing changes on collection elements are allowed: adding element, removing element, renaming element, changing the type of the element and changing the length of the type. If the calculated similarity is above a threshold then collections are marked as renamed (they are no longer marked as added or deleted collections). In an analogous way collection splits and merges are detected.

4. In the fourth step, among the deleted collections, sets of collections are detected that can be joined basing on a common key collection element. The detected sets of collections are marked as merge candidates. Similar detection is carried out on added collections and detected sets of collections are marked as split candidates.

5. Next, each of collection marked as deleted is compared with each set of collections marked as split candidates. The similarity is measured between a deleted collection and a collection that would be a result of joining collections form a set marked as the split candidate. If the calculated similarity is above a threshold, then the collections are marked as split. Analogously, each of the added collection is compared with each set of collection marked as merge candidates. If the calculated similarity is above a threshold, then the collections are marked as merged.

6. The next step of comparing two snapshots of an EDS's metadata is to check for changes in all collection. After detecting added, deleted, renamed, split and merged collections, for each collection in a new snapshot it is known its equivalent in the old snapshot. By comparing a collection from a new snapshot

and its equivalent it is possible to identify changes in collection elements. The algorithm for detecting changes in collection elements is similar to detecting changes in a collection. First, deleted elements are marked in a collection from the old snapshot. Second, added elements are marked in a collection from the new snapshot. Next, changes of elements names are detected by comparing elements marked as deleted and elements marked as added. The comparison of the elements is based on elements' data types and data characteristics taken from a data sample. These characteristics differ depending on the type of data. For numeric types it is an average value and a deviation. For string types it is an average length of the string or an average number of words in the string. If collection elements from old and new snapshots have the same type and differences in data characteristics are below a threshold, then the elements are marked as renamed. Finally, all the elements that were not marked as deleted, added, or renamed are checked for a type change or a type length change.

5.2 Detecting changes by the mechanism of triggers

Detecting changes by the mechanism of triggers is based on the solution presented in [4]. This form of changes detection can be used with EDSs that have a management system which controls structural changes of the EDS and support the Event-Condition-Action (triggers) mechanism. An example of such EDS is a fully-functional database. The process of detecting changes by the mechanism of triggers requires to create the *Changes* table. In this table a history of changes will be stored. The event is a modification of the system table that stores information about the database structure (information about tables, columns, views). If that modification indicates modification of user structures then the action is fired. The action is an entry with information about the detected change. This informational entry is stored in the *Changes* table. When *E-ETL* runs, it reads data form the *Changes* table.

6 Evolution rules

All of the mentioned changes are handled by our framework at the level of *SuperNode* (an ETL activity). We adopt a similar solution to the one presented in [1]. On each *GroupNode*, *SuperNode*, or *Node* for every type of change a user can define one of five evolution rules: *Inherit*, *Propagate*, *Block*, *Ask* or *Action*. User can also define default behavior for an element by setting appropriate rule for *Any change*.

The *Inherit* rule means that the rule should be inherited from the *Any change*. If the *Inherit* rule is set on the *Any change* this means that the rule should be inherited from an enclosing element (for *Node* it is *SuperNode*, for *SuperNode* it is *GroupNode*, and for *GroupNode* it is enclosing *GroupNode*). The *Propagate* rule instructs that the detected change should be propagated through the ETL activity (*SuperNode*). Both, input and output attributes of the activity should be modified accordingly to the change and an information about the change should be passed to next activities

(activities that depend on this activity). The *Block* rule ignores the change and does not modify *SuperNode*. The *Ask* rule defines that the system should ask a user to decide what to do at the moment of the change occurrence. The *Action* rule also instructs that the ETL activity (*SuperNode*) should be modified accordingly to the change. Contrary to the *Propagate* rule, only input attributes of the activity should be modified. The evolution of the ETL workflow should stop on this activity and information about the change should not be passed to the next activities.

Table 1 presents all types of EDS structural changes and rules that can be set for them. *Inherit*, *Block*, and *Ask* rules work for every type of change in the same way. Contrary to this, the *Propagate* and *Action* rules are different for every type of change.

Table 1: Monitored structural changes and possible rules to define

Change type		Inherit	Propagate	Block	Ask	Action
Any change		✓	✓	✓	✓	✓
Collection	Add	✓	Join	✓	✓	Ignore
	Delete	✓	Delete	✓	✓	Replace
	Rename	✓	Rename	✓	✓	Map
	Split	✓	Delete	✓	✓	Merge
	Merge	✓	Add	✓	✓	Ignore
Collection element	Add	✓	Add	✓	✓	Ignore
	Delete	✓	Delete	✓	✓	Replace
	Rename	✓	Rename	✓	✓	Map
	Type	✓	Change type	✓	✓	Convert
	Length	✓	Change length	✓	✓	Cast

As mentioned before the *Propagate* rules instruct that the element should be modified accordingly to the change and pass the change to next elements. The intention of these rules is to adjust the *SuperNode* to the change and to propagate the evolution of the ETL workflow to next *SuperNodes*. The *Join* rule that can be set for the *Collection Add* change instructs that the attributes from the new collection should be added to existing attributes. This addition is possible if among the existing attributes there is a foreign key that point to the key attribute in the new collection. The *Delete* rule can be set for *Collection Delete*, *Collection Split*, and *Collection Element Delete* changes. If this rule is set for the *Collection Delete* change then all attributes contained in the deleted collection will be removed from *SuperNode*. Also, all attributes that depend on deleted attributes will be removed from *SuperNode*. In case of a split operation on a collection we get: (1) the primary collection (the collection with the name equal to the name of the not split collection or the collection with the largest number of attributes) and (2) the secondary collection or collections. If the *Delete* rule is set for the *Collection Split* change then attributes that are present in the primary collection will be unchanged and other attributes will be removed, similarly as in the case of the *Collection Delete* change. If the *Delete* rule is set for the *Collection* *element Delete* change then the deleted attribute and attributes that depend on it will be removed from *SuperNode*. The *Rename* rule can be set for the *Collection Rename* and *Collection element Rename* changes. In both cases if this rule is set then the name of the element (*SuperNode* and *Node* accordingly) will be changed. All elements that depend on the renamed elements will be renamed as well. The *Add* rule can be set for the *Collection Merge* and *Collection element Add* changes. If the *Add* rule is set for the *Collection element Add* change then the new attribute will be added to input elements of *SuperNode*. Depending on the activity that is represented by the modified *SuperNode* also new attributes can be added to output elements of that *SuperNode*. If the *Add* rule is set for the *Collection Merge* change then every new attribute (attributes from merged collections) will be added similarly as in the case of the *Collection element Add* change. The *Change type* and *Change length* rules can be set accordingly for the *Collection element Type* and *Collection element Length* changes. Both rules instruct that a type or a length of the type of the attribute should be changed, respectively.

On the contrary to the *Propagate* rules, the intention of the *Action* rules is to try to compensate the change and stop the evolution of the ETL workflow. The *Ignore* rule can be set for the *Collection Add*, *Collection element Add*, and *Collection Merge* changes. If this rule is set, then all new elements will be ignored and the ETL workflow will not change. The *Replace* rule can be set for the *Collection Delete* and *Collection element Delete* changes. The concept of this rule is based on the solution presented in [2]. If this rule is set then in case of some element absence *E-ETL* will try to replace it with a different element (with the same structure). Additionally, for this rule a user can define parameters of replacing element. A user can define if the set of data contained in the replacing element should be equal to the set of data contained in the replaced element, should be the subset, the superset, or can be any set of data with the same structure. The *Map* rule can be set for the *Collection Rename* and *Collection element Rename* changes. If this rule is set, then *SuperNode* will be modified in such way that input elements will change to new names but output elements remain unchanged. An appropriate mapping will be done inside *SuperNode*. The *Merge* rule that can be set for the *Collection Split* change instructs that *E-ETL* should try to merge back the split collection. The *Convert* and the *Cast* rules can be set accordingly for the *Collection element Type* and *Collection element Length* change. The rules instruct that a type or a length of the type of the attribute should be converted or casted, respectively.

Every ETL activity represented by *SuperNode* can work in a different way. For example, it can be a simple SQL query, or it can just count duplicated elements. For a simple SQL query, a change like adding attribute may modify both input and output *Nodes*. However, for an activity that counts elements, a similar change may modify only the input *Nodes*. The output remains just as one numeric value. Activities based on SQL queries are similar and can be handled by rewriting the query. Contrary to this, activities that are based on ETL tool built-in functionality (i.e. fuzzy lookup) are more complex and each of them has its own parameters set that can be modified. Therefore, for every type of activity there must be a method for handling all types of changes. Since every ETL development environment can have a different set of available ETL activities and they can work in a different way, the handling methods are specific for every ETL development environment.

7 Related Work

The research and technological developments in the area of handling structural changes of EDSs in the DW architecture have mainly focused on managing changes in a DW. In this field, the five following approaches can be distinguished: (1) materialized view adaptation, (2) schema and data evolution, (3) temporal schema and data extensions, (4) partial versioning of schema and data, and (5) the Multiversion Data Warehouse approach. Since they are not directly related to the topic of this paper, they will not be described here. An overview of research problems and approaches can be found in [5, 6].

Detecting structural changes in EDSs and propagating them into the ETL layer have not received much attention from the research community. One of the first solution of this problem was Evolvable View Environment (EVE) presented in [2]. EVE is the environment that allows the evolution of an ETL workflow implemented by means of views. For every view it is possible to specify which elements of the views may change. It is possible to determine whether a particular attribute, both in the *select* and *where* clauses, can be omitted, or replaced by another attribute. Another possibility is that for every table, which is referred by a given view, a user can define whether this table can be omitted or replaced by another table.

The *E-ETL* versus EVE. *E-ETL* also employ a similar solution for handling missing elements (the *Replacer* algorithm). However, the *E-ETL* extends to this solution. *E-ETL* works with different ETL engines, whereas EVE works with ETL workflows developed as sequences of SQL queries. This difference implies that in *E-ETL* method for replacing missing elements can be applied not only for views, tables and their columns but also for ETL activities and their attributes.

Recent developments in the field of evolving ETL workflows include a framework called *Hecataeus* [1, 7]. In Hecataeus, all ETL activities and EDSs are modeled as a graph whose nodes are relations, attributes, queries, conditions, views, functions, and ETL steps. Nodes are connected with edges that represent relationships between different nodes. The graph is annotated with rules that define the behavior of an ETL workflow in response to a certain EDS change event. In a response to an event, Hecataeus can either propagate the event, i.e. modify the graph according to a predefined policy, or prompt an administrator, or block the event propagation.

***E-ETL* versus Hecataeus.** The *E-ETL* framework, presented in this paper, is related to Hecataeus. However, *E-ETL* differs from Hecataeus with respect to:

- *E-ETL* has extended set of evolution rules;
- *E-ETL* has introduced new algorithms for repairing ETL workflow;
- *E-ETL* detects structural changes in EDSs either by means of schema triggers or by comparing two consecutive snapshots of EDS metadata (no information

was provided how Hecataeus detects structural changes);

- *E-ETL* can be connected to any ETL engine and development environment that offers API, whereas Hecataeus needs a specific ETL engine that models ETL tasks by means of graphs;
- *E-ETL* support ETL workflows built of several complex operations (i.e. the operation of removing duplicates that may be available only in the external ETL tool), whereas Hecataeus work with ETL workflows developed as sequences of SQL queries;
- *E-ETL* can work with different types of EDS (i.e. data base, XML files, spreadsheet, record files), whereas Hecataeus supports only data bases as EDSs.

In [4] authors proposed a prototype system that can automatically detect changes in EDSs and propagate them into a DW. The prototype allows to define changes that are to be detected and associates with the changes actions executed in a DW. The main limitation of the prototype is that it does not allow ETL workflows to evolve. Instead of that it focuses on propagating EDSs' changes into a DW. Moreover, the presented solution is restricted to only relational databases as EDSs. The next drawback of this prototype is a detection of changes which depends on triggers mechanism that can be not allowed to be installed in an EDS. Although, the *E-ETL* project is based on that developments, all mentioned shortcomings are not present in the *E-ETL* framework. Previous works on *E-ETL* were presented in [8].

8 Summary

This paper presents the *E-ETL* framework for detecting structural changes in EDSs and repairing an ETL workflow accordingly do detected changes. The framework repairs automatically an ETL workflow using evolution rules defined by a user. The *E-ETL* framework is also able to present to a user possible consequences of future changes (impact analyses). Currently we are implementing the presented framework. We are also preparing tests in an environment including structural changes that appeared in the real production DW systems, outlined in Section 5. Furthermore, we focus on developing a language for defining structural changes that are to be detected and propagated, and for repairing algorithms. *E-ETL* API is currently under development for communicating with Microsoft ETL engine, i.e., SQL Server Integration Services.

The approaches outlined in Section 7 handle structured changes in EDSs. However, as stressed in [9, 10] even ordinary content (data) changes of an EDS may cause structural changes at a DW or changes to the structure of dimension data in a DW. Neither Hecataeus nor EVE nor [4] nor *E-ETL* supports handling appropriately such content changes. In future, we will work on handling such kinds of content changes at the ETL layer and on correctly propagating them into a DW.

References

[1] G. Papastefanatos, P. Vassiliadis, A. Simitsis, T. Sellis, and Y. Vassiliou, "Rule-based Management of Schema Changes at ETL sources," in *Proc. of Conf. Advances in Databases and Information Systems Workshops (ADBIS)*, pp. 55–62, Springer, LNCS 5968, 2010.

[2] E. A. Rundensteiner, A. Koeller, X. Zhang, A. J. Lee, A. Nica, A. Van Wyk, and Y. Lee, "Evolvable View Environment (EVE): Non-Equivalent View Maintenance under Schema Changes," in *Proc. of ACM Int. Conf. on Management of Data (SIGMOD)*, pp. 553–555, ACM Press, 1999.

[3] A. Wojciechowski, "E-ETL: Framework For Managing Evolving ETL Processes," in *Proc. of Conf. Advances in Databases and Information Systems Workshops (ADBIS)*, vol. 185 of *Advances in Intelligent Systems and Computing*, pp. 441–449, Springer, 2013.

[4] R. Wrembel and B. Bebel, "The Framework for Detecting and Propagating Changes from Data Sources Structure into a Data Warehouse," *Foundations of Computing & Decision Sciences*, vol. 30, no. 4, pp. 361–372, 2005.

[5] A. Wojciechowski and R. Wrembel, "Research Problems of the ETL Technology.," *Foundations of Computing and Decision Sciences*, vol. 35, no. 5, pp. 283–306, 2010.

[6] R. Wrembel, "On handling the evolution of external data sources in a data warehouse architecture," in *Data Mining and Database Technologies: Innovative Approaches* (D. Taniar and L. Chen, eds.), IGI Group, 2011.

[7] G. Papastefanatos, P. Vassiliadis, A. Simitsis, and Y. Vassiliou, "Policy-Regulated Management of ETL Evolution.," *J. Data Semantics*, pp. 147–177, 2009.

[8] A. Wojciechowski, "E-ETL: Framework For Managing Evolving ETL Processes.," in *Proc. of Ph.D. Students in Information and Knowledge Management Workshop (PIKM)*, pp. 59–66, ACM Press, 2011.

[9] J. Eder, C. Koncilia, and T. Morzy, "The COMET Metamodel for Temporal Data Warehouses," in *Proc. of Int. Conf. on Advanced Information Systems Engineering (CAiSE)*, pp. 83–99, Springer-Verlag, 2002.

[10] E. A. Rundensteiner, A. Koeller, and X. Zhang, "Maintaining data warehouses over changing information sources," *Communications of the ACM*, vol. 43, no. 6, pp. 57–62, 2000.

Presented at the 16th East-European Conference on Advances in Databases and Information Systems, September 17-20, 2012, Poznań, Poland