Inf Syst Front (2018) 20:21–43
DOI 10.1007/s10796-016-9732-0

ETL workflow reparation by means of case-based reasoning

Artur Wojciechowski[1]

Published online: 7 January 2017
© The Author(s) 2017. This article is published with open access at Springerlink.com

Abstract Data sources (DSs) being integrated in a data warehouse frequently change their structures/schemas. As a consequence, in many cases, an already deployed ETL workflow stops its execution, yielding errors. Since in big companies the number of ETL workflows may reach dozens of thousands and since structural changes of DSs are frequent, an automatic repair of an ETL workflow after such changes is of high practical importance. In our approach, we developed a framework, called *E-ETL*, for handling the evolution of an ETL layer. In the framework, an ETL workflow is semi-automatically or automatically (depending on a case) repaired as the result of structural changes in DSs, so that it works with the changed DSs. *E-ETL* supports two different repair methods, namely: (1) user defined rules, (2) and Case-Based Reasoning. In this paper, we present how Case-Based Reasoning may be applied to repairing ETL workflows. In particular, we contribute an algorithm for selecting the most suitable case for a given ETL evolution problem. The algorithm applies a technique for reducing cases in order to make them more universal and capable of solving more problems. The algorithm has been implemented in prototype *E-ETL* and evaluated experimentally. The obtained results are also discussed in this paper.

Keywords Data source evolution · ETL evolution · ETL repair · Case-based reasoning

✉ Artur Wojciechowski
artur.wojciechowski@cs.put.poznan.pl
http://calypso.cs.put.poznan.pl/projects/e-etl/

[1] Institute of Computing Science, Poznan University of Technology, Poznan, Poland

1 Introduction

A data warehouse (DW) system has been developed in order to provide a framework for the integration of heterogeneous, distributed, and autonomous data storage systems (typically databases) deployed in a company. They will further be called data sources (DSs). The integrated data are subjects of advanced data analysis, e.g., trends analysis, prediction, and data mining. Typically, a DW system is composed of four layers: (1) a data sources layer, (2) an Extraction-Translation-Loading (ETL) layer, (3) a repository layer (a data warehouse) that stores the integrated and summarized data, and (4) a data analytics layer.

The ETL (or its ELT variant) is responsible for extracting data from DSs, transforming data into a common data model, cleaning data, removing missing, inconsistent, and redundant values, integrating data, and loading them into a DW. The ETL/ELT layer, is implemented as a workflow of tasks managed by a dedicated software. An ETL workflow will interchangeably called an ETL process. A task in an ETL workflow will interchangeably called an activity.

An inherent feature of DSs is their evolution in time with respect not only to their contents (data) but also to their structures (schemas) (Wrembel 2009; Wrembel and Bębel 2007). According to Moon et al. (2008) and Sjøberg (1993), structures of data sources change frequently. For example, the Wikipedia schema has changed on average every 9-10 days in years 2003-2008 (Curino et al. 2008a). As a consequence of DSs changes, in many cases, an already deployed ETL workflow stops its execution with errors. As a consequence, an ETL process must be repaired, i.e., redesigned and redeployed.

In practice, in large companies the number of different ETL workflows may exceed 30,000. Thus, a manual modification of ETL workflows is complex, time-consuming, and

prone-to-fail. Since structural changes in DSs are frequent, an automatic or semi-automatic repair of an ETL workflow after such changes is of high practical importance.

Handling and incorporating structural changes to the ETL layer received so far little attention from the research community (Manousis et al. 2013, 2015; Papastefanatos et al. 2009; Rundensteiner 1999). Moreover, none of the commercial or open-source ETL tools existing on the market supports this functionality.

In our research, we have designed and developed a prototype ETL framework, called *E-ETL* (Wojciechowski 2011, 2013a, b) that is able to repair its workflows automatically (in simple cases) or semi-automatically (in more complex cases) as the result of structural changes in data sources. To this end, in Wojciechowski (2015) we proposed an initial version of a repair algorithm for an ETL workflow. The algorithm is based on Case-Based Reasoning method. Our approach consists of two main algorithms, namely: (1) *Case Detection Algorithm* (CDA) for building a case base, and (2) *Best Case Searching Algorithm* (BCSA) for choosing the best case.

This paper extends (Wojciechowski 2015, 2016) with:

- a *Case Reduction Algorithm* (CRA) that improves CDA by making cases more general (i.e. able to solve more problems),
- a test case scenario, for the purpose of validating our approach,
- an experimental evaluation the proposed BCSA.

The paper is organized as follows. Section 2 discusses how Case-Based Reasoning can be applied to an evolving ETL layer. Section 3 describes the *E-ETL* Library of Repair Cases. Section 4 presents a procedure of building a case. Section 5 introduces a method for comparing cases and shows how to apply modifications to an ETL process. Section 6 describes the *Case Reduction Algorithm*. Section 7 presents a test scenario. Section 8 discusses performance tests of the BCSA and their results. Section 9 outlines research related to the topic of this paper. Section 10 summarizes the paper and outlines issues for future development.

2 Case-based reasoning for ETL repair

Case-Based Reasoning (CBR) is inspired by human reasoning and the fact that people commonly solve problems by remembering how they solved similar problems in the past. CBR solves a new problem, say P, by adapting and applying to P previously successful solutions of a problem similar to P (Schank 1983). Further in this paper, a problem and its solution will be called a *case*.

As mentioned before, structural changes in DSs often cause that ETL workflows stop their execution with errors. A desired solution to this problem is to (automatically or semi-automatically) repair such an ETL workflow. According to Case-Based Reasoning, in order to find a solution, a similar case should be found and adapted to the current ETL problem.

Most of the ETL engines work use dataflows to process streams of tuples. Typically, the first activity in the workload reads data from DSs and outputs a collection of tuples. A DS can be described as a collection that includes the set of tuples. The collection elements define elements of the tuple. A database table, spreadsheet, branch in an XML document are examples of collections, whereas table column, column in spreadsheet, node in XML are examples of collection elements.

Such a definition of a DS is independent of the DS type and its model (e.g., relational, object-oriented). Even if the whole DS cannot be described as the collection of tuples (i.e., unstructured DS) it is sufficient that extracted data (i.e., the output of the activity that reads from the DS) can be described as the collection of tuples.

2.1 ETL workflow representation

Typically, different ETL engines use their specific internal representations of ETL processes. These models well describe flows of data between ETL tasks. From these representations the one that is appreciated by the research community is based on graphs, c.f., (Manousis et al. 2013; 2015; Papastefanatos et al. 2010; Papastefanatos et al. 2008; 2009; 2012). For our problem, the graph representation is of a particular value since it well supports modeling the dependencies between the elements of an ETL process, thus is suitable for impact analysis.

Since the *E-ETL* framework is designed to work with various ETL engines, it has to be able to represent various specific representations of ETL processes in a unified manner. For this reason, the *E-ETL* framework uses its own internal representation based on graphs, which permits to work with different external ETL engines.

An ETL process is defined as a directed graph $G = (N, D, SN)$, where N denotes nodes, D - edges, and SN - supernodes.

N of the graph is the set of $Nodes$ that represent activity parameters (inputs, outputs and internal configuration of the activity, e.g. a query condition and collections elements. In order to get elements of the ETL process influenced by a change in $Node$ it is sufficient to get all successors of the $Node$ in the graph $G = (N, D, SN)$. Therefore $Nodes$ represents all elements of the ETL process that can be changed and can modify the execution of the ETL process.

Each $Node$ is described as:

$$Node = (Type_N, Name, DataType, DataTypeLength, Value).$$

$Type_N$ is the type of $Node$ and may have one of five values, namely: (1) *CollectionElement* for collections elements, (2) *Input* for input attributes of the activity, (3) *Output* for output attributes of the activity, (4) *Config* for activity internal configuration (i.e., a query condition), and (5) *Identifier* for $Nodes$ identifying collections (i.e., a name of a database table).

Edges D of the graph is a set of direct dependencies between $Nodes$. If there exist a directed edge from $Node_A$ to $Node_B$, then $Node_B$ directly depends on $Node_A$ (i.e., $Node_B$ consumes data influenced or produced by $Node_A$). Such a direct dependency is denoted as:

$Dep(Node_A, Node_B) \Rightarrow Node_B$ depends on $Node_A$.

If there is a directed path from $Node_A$ to $Node_C$ (e.g., $Node_A \rightarrow Node_B \rightarrow Node_C$), then $Node_C$ indirectly depends on $Node_A$.

SN is the set of $SuperNodes$ that group $Nodes$ into subgraphs: $SuperNode = (N, D, Type_{SN})$. $SuperNodes$ represents ETL process activities, data source collections, and target DW tables. $Type_{SN}$ is the type of $SuperNode$ and may have one of three values, namely: (1) *DataSource* for data source collections, (2) *DataTarget* for tables in a DW, and (3) *Activity* for ETL process activities.

2.2 DS changes and ETL repairs

In the ETL process repair, a case can be described as $Case = (DSCs, Ms)$, where $DSCs$ is the set of changes in data sources and Ms is a recipe for the repair of an ETL workflow.

A change in a data source can be described as a tuple of an action and an element that is affected by the action – $DSC = (action, E)$. Changes in data sources can be divided into two groups, namely: (1) collection changes and (2) collection element changes.

The five following changes are distinguished for Collections:

- addition – $DSC = (add, SuperNode)$, e.g., adding a table into a database,
- deletion – $DSC = (delete, SuperNode)$, e.g., deleting a table or a spreadsheet in an Excel file,
- renaming – $DSC = (alter_{Name}, SuperNode)$, e.g., changing a file or a table name,
- splitting – $DSC = (split, SuperNode)$, e.g., partitioning a table,
- merging – $DSC = (merge, SuperNode)$, e.g., merging partitioned tables.

The five following changes are distinguished for Collection elements:

- addition – $DSC = (add, Node)$, e.g., adding a new column into a table,
- deletion – $DSC = (delete, Node)$, e.g., deleting a column from a spreadsheet,
- renaming – $DSC = (alter_{Name}, Node)$, e.g., changing a node name in an XML document,
- type change – $DSC = (alter_{DataType}, Node)$, e.g., changing a column type form numeric to string,
- type length change – $DSC = (alter_{Length}, Node)$, e.g., changing a column type length from char(4) to char(8).

The repair (Ms) consists of the sequence of modifications of an ETL process that adjust the ETL process to its new state, so that it works without errors on the modified data sources. The modification is described as tuple of: a modifying action and an element affected by the modification – $M = (modification, E)$.

Examples of modifications include:

- addition of an ETL activity – $M = (add, SuperNode)$,
- deletion of an ETL activity – $M = (delete, SuperNode)$,
- addition of an ETL activity parameter – $M = (add, Node)$,
- deletion of an ETL activity parameter – $M = (delete, Node)$,
- modification of an ETL activity parameter name – $M = (alter_{Name}, Node)$,
- modification of an ETL activity parameter value – $M = (alter_{Value}, Node)$,
- modification of an ETL activity parameter data type – $M = (alter_{DataType}, Node)$,
- modification of an ETL activity parameter data type length – $M = (alter_{Length}, Node)$.

2.3 Repair cases

A case is build of elements affected by $DSCs$ and Ms. Therefore, it is possible to extract subgarph $G_C = (N_C, D_C, SN_C)$ from $G = (N, D, SN)$. G_C can be obtained by the algorithm of building a case, which is described in Section 4.

G_C consists of $SuperNodes$ and $Nodes$ affected by $DSCs$ or Ms. In other words, if $SuperNode$ or $Node$ are contained in $DSCs$ or Ms then $SuperNode$ or $Node$ belongs to SN_C or N_C, respectively, as expressed in Formula 1.

$$\begin{aligned}&(\exists_{Dsc \in DSCs} Node \in Dsc) \vee (\exists_{m \in Ms} Node \in m)\\ &\Rightarrow Node \in N_C (\exists_{Dsc \in DSCs} SuperNode \in Dsc)\\ &\vee (\exists_{m \in Ms} SuperNode \in m) \Rightarrow SuperNode \in SN_C \quad (1)\end{aligned}$$

D_C consists of direct dependencies between any of $Nodes$ in N_C. D_C is the subset of D. If there is a

direct dependency $Dep(Node_A, Node_B)$ between any two $Nodes$ in G and these $Nodes$ belongs to G_C then $Dep(Node_A, Node_B)$ belongs to D_C. D_C is defined by Formula 2.

$$Node_A, Node_B \in N_C \wedge \exists_{Dep(Node_A, Node_B) \in D} \Rightarrow Dep(Node_A, Node_B) \in D_C \quad (2)$$

Modification M handles data source change DSC ($handle(DSC, M)$) if there exists a path from any $Nodes$ in DSC to any $Nodes$ in M.

Cases helping to solve a new problem can be found in two dimensions of an ETL project, namely: (1) fragments of the ETL process and (2) time. The first dimension means that a case represents a similar change in a similar DS and modifications in a similar part of the ETL process reading from that DS. The time dimension allows to look for cases in the whole evolution history of the ETL process. Therefore, if there was a change in the past and now a different part of the same ETL process changes in the same way, then the previously repaired change can be a case for the ETL process repair algorithm.

Example 1 Let us assume an ETL process that loads data from several instances of an ERP or CRM system. Every instance can use a different version of an ERP/CRM system, hence each instance (treated as a data source) is similar to the other instances but, not necessarily the same. For each instance, there can be a dedicated ETL subprocess. When there are updates to some of the ERP/CRM systems, the structures of data sources that are associated with them may change. That causes the need of repairing the relevant ETL subprocesses. Changes in one data source and a manual repair of an associated ETL subprocess may be used as a case for a repair algorithm to fix other ETL subprocesses associated with the updated data sources.

Example 2 Let us consider an evolution that takes place in a data source consisting of multiple tables that are logical partitions of the same data set. Let us assume a set of customers data spread across three tables: (1) individual customers, (2) corporate customers, and (3) institutional customers. Each of the tables may include some specific columns, which are important only for one type of a customer but, all of them have also a common set of columns. If there is a change in data sources that modifies the common set of columns (e.g., adding a rating to all three types of customers) then the change of one of these tables and the repair of the ETL process associated with this table can be used as a case for a repair algorithm to fix other ETL parts associated with the other two tables.

Example 3 Let us consider data source D_i, composed of multiple tables containing the same set of columns, e.g., State, City, and Street. Now, the source evolves by replacing columns State, City, and Street with one column containing ID of an address stored in table Addresses. The case built for handling his change can further be be used by a repair algorithm to fix other ETL fragments associated with other tables in D_i changed in the same way.

Since a case is build of a **set** of changes in data sources, it is suitable for handling connected data source changes. The connected data source changes should be handled together. In *Example 3*, three columns were replaced with one new column, i.e., there were three $DSCs$ removing a column and one DSC adding a column. All four $DSCs$ should be included in a case, since they are connected and should be handled together.

Structural changes in DSs may be handled in multiple ways. Reqired modifications of the ETL process may vary on many elements i.e., the design of the ETL process, the domain of the processed data, the quality of data. Even the same problem in the same process may need different solution due to new circumstances i.e., new company guidelines, new functionality available in an ETL tool. Therefore, we cannot guarantee that the solution delivered by means of Case-Based Reasoning methodology is correct. Nevertheless, we can propose to a user a solution, which was applicable in the past, and let a user decide if it is correct.

In companies where thousands of ETL processes are maintained in parallel, the probability that there are several similar processes using similar data sources may be reasonably high. Therefore, the more ETL processes exists, the more feasible is the Case-Based Reasoning for the ETL repair.

3 Library of repair cases in E-ETL

The Case-Based Reasoning method for repairing ETL workflows is based on the Library of Repair Cases (LRC). The LRC is constantly augmented with new cases encountered during the usage of the *E-ETL* framework (Wojciechowski 2011).

3.1 E-ETL framework

E-ETL allows to detect structural changes in DSs and handle the changes at an ETL layer. Changes are detected either by means of Event-Condition-Action mechanism (triggers) or by comparing two consecutive DS metadata snapshots. The detection of a DS schema change causes a repair of the ETL workflow that interacts with the changed DS. The repair of the ETL workflow is guided by a few customizable repair algorithms.

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 processes,
- present to a user the impact analysis of the ETL workflow,
- store versions of the ETL process and history of DS changes.

One of the key features of the *E-ETL* framework is that it stores the whole history of all changes in the ETL process. This history is the source for building the LRC. After every modification of the ETL process a new case is added to the LRC. Thus, whenever a data source change (DSC) is detected and the procedure of evolving the ETL process repairs it by using one of the supported repair methods new cases are generated and stored in the LRC. The repair methods include: the *User Defined Rules*, the *Replacer algorithm*, and the *Case-based reasoning*.

The *E-ETL* framework is not supposed to be a fully functional ETL designing tool but an external extension to complex commercial tools available on the market. The proposed framework was developed as a module external to an ETL engine. Communication between *E-ETL* and the ETL engine is realized by means of the ETL engine API. Therefore, it is sometimes impossible to apply all the necessary modification of the ETL process within the *E-ETL* framework (i.e., new business logic must be implemented). For that reason the *E-ETL* framework has a functionality of updating the ETL process. When a user introduces manually some modification in the ETL process using the external ETL tool, then the *E-ETL* framework loads the new definition of the ETL process and compares it with the previous version. The result of this comparison is a set of all modifications made manually by a user. Those modifications together with detected DSCs also provide repair cases.

E-ETL provides a graphical user interface for visualizing ETL workflows and impact analysis.

3.2 Library scope

For each ETL process, there are three scopes of the LRC, namely: (1) a process scope, (2) a project scope, and (3) a global scope. Since the *case-based reasoning* algorithm works in the context of the ETL process, the first scope is the *process scope* and it covers cases that originate from the same ETL process.

Huge companies maintain ETL projects that consist of hundreds or even several thousands of ETL processes. Within a project there are usually some rules that define how to handle DSCs. Therefore, DSCs from one ETL process probably should be handled in a way similar to other ETL processes from the same ETL project. The *project scope* covers cases that originate from the same ETL project.

The *global scope* covers cases that originate neither from the same ETL project nor the same ETL process. Except cases from different user projects, the global scope consists of cases build into the *E-ETL* framework. The purpose of the built-in cases is to support the Case-Based Reasoning algorithm with solutions for handling the most common DSCs.

4 Case detection in ETL process

As mentioned in Section 2, the case is a pair of: (1) a set of changes in data sources and (2) a set of modifications that adjust an ETL process to the new state of the modified data sources (repair of an ETL process). The modifications are defined on ETL activities, i.e., tasks of an ETL workflow. Both sets that create the case should be complete and minimal.

4.1 Completeness

The **completeness** means that the definition of the case should contain all DSCs and all modifications that are dependent. In other words, the set of DSCs should contain all changes that are handled by the set of modifications. Moreover, the set of modifications should contain all modifications that repair the ETL process affected by the set of DSCs. Formula 3 is a formal representation of the completeness constraint.

$$\begin{aligned} &\forall_{DSC \in Case} handle(DSC, M) \Rightarrow M \in Case \\ &\forall_{M \in Case} handle(DSC, M) \Rightarrow DSC \in Case \end{aligned} \tag{3}$$

If the case is not complete, then two problems may occur, namely:

- an incomplete set of DSCs causes that the set of modifications cannot be applied since it is based on a different state of the DS (not fully changed),
- an incomplete set of modifications causes that ETL process may not be fully repaired.

4.2 Minimality

The **minimality** means that the definition (content) of the case should be as small as possible, without violating the completeness constraint. The minimality assures that:

- the set of DSCs contains only these changes that are handled by the set of modifications;

– the set of modifications contains only these modifications that repair the ETL process affected by the set of DSCs.

Moreover, the minimality means that it is impossible to separate from one case another smaller case that is complete. In order to satisfy the minimality constraint formally, a subgarph $G_C = (N_C, D_C, SN_C)$ of the case should be the connected graph (cf. the definition of G_C in Section 2.3).

If subgrahp G_C is not a connected graph, then it is possible to extract connected subgraphs out of G_C. Depth-first or breadth-first graph traversal algorithms can be used in order to extract connected subgraphs. Every extracted subgraph that is a connected graph represents a minimal case.

If the case is not minimal, then the two following problems may occur:

– *Redundant DSCs*: the Case-Based Reasoning algorithm may incorrectly skip the case, since it will not match a new problem because of redundant DSCs,
– *Non-repairing modifications*: the repair of the ETL process might be excessive and modifications that do not repair the ETL process might be applied.

4.2.1 Redundant DSCs

The first problem occurs in the following example. Let us consider a not minimal case *Case_AB*, that:

– contains two DSCs, namely: *DSC_A* and *DSC_B*,
– contains two modifications, namely: *Mod_A* and *Mod_B*,
– *DSC_A* is handled by modification *Mod_A*,
– *DSC_B* is handled by modification *Mod_B*,
– *Mod_A* is independent of *DSC_B*,
– *Mod_B* is independent of *DSC_A*.

Case *Case_AB* could be decomposed into two minimal (and complete) cases: (1) *Case_A* consisting of *DSC_A* and *Mod_A* and (2) *Case_B* consisting of *DSC_B* and *Mod_B*. New problem *Prob_A* consists of one DSC *DSC_A*. *Case_AB* cannot be used for handling *Prob_A* since *Case_AB* contains *Mod_B* in the set of modifications. *Mod_B* is dependent on *DSC_B* that is not part of *Prob_A*. The algorithm for searching a right case for *Prob_A* skips *Case_AB* although part of it (*Mod_A*) is a desired solution. If there are two minimal cases *Case_A* and *Case_B* instead of the not minimal case *Case_AB* then the algorithm for searching a right case for *Prob_A* can properly choose *Case_A*.

4.2.2 Non-reparing modifications

Let us consider the second not minimal case *Case_AB'*, such that:

– contains one DSC - *DSC_A*,
– contains two modifications, namely *Mod_A* and *Mod_B*,
– *DSC_A* is handled by modification *Mod_A*,
– *Mod_B* is independent of *DSC_A* and does not handle *DSC_A*.

If *Mod_B* is not applicable in the ETL process of *Prob_A* then *Case_AB'* is skipped by the searching algorithm. If *Mod_B* is applicable then the second problem of the not minimal cases may occur. *Mod_B*, which is not necessary and do not handle *Prob_A*, is applied to the ETL process. The application of *Mod_B* is excessive.

Figure 1 shows an example of DSCs and associated ETL process modifications. The exemplary ETL process reads data from the *OnLineSales* table and the *InStoreSales* table. Next, the data are merged by the union activity. The result is joined with data read from the *Customers* table. Finally, the result is stored in the *TotalSales* table.

Let us assume that as a result of a data source evolution, three DSCs, namely: *dsc1*, *dsc2*, and *dsc3* were applied. *dsc1* adds column *Discount* to table *OnLineSales*. *dsc2* adds column *Discount* to table *InStoreSales*. *dsc3* renames table *Customers* to *Clients*.

In order to handle the three DCSs, three ETL process modifications, namely: *mod1*, *mod2*, and *mod3* must be applied. *mod1* adds new attribute *Discount* to activity *Union*. *mod2* adds new activity that replaces old *TotalValue* with *TotalValue - Discount*. It was added between activities *Union* and *Join*. *mod3* updates the component that reads the *Customers* table - it replaces the old table name with new one *Clients*.

For this evolution scenario, two new cases can be defined. The first one (*case1*) consist of two DSCs, namely: *dsc1*, *dsc2* and two ETL process modifications, namely: *mod1* and *mod2*. The second case (*case2*) consists of DSC *dsc3* and modification *mod3*.

These cases are minimal and complete. An addition of *dsc3* or *mod3* to *case1* would violate the minimality constraint. Moreover, a removal of any item from *case1* would violate the completeness constraint. We can remove neither *dsc1* nor *dsc2* from *case1* because *mod1* is connected with them. Moreover, we can remove neither *mod1* nor *mod2* because they are the results of *dsc1* and *dsc2*.

As a result of the minimality constraint, it is possible to extract multiple cases from one set of DSCs and modified ETL process (multiple G_C subgraphs of the same graph G). However, every returned case should consists of different set of DSCs and different set of modifications. As a result of the completes constraint, DSCs and modifications sets should be disjoint between cases extracted from one set of DSCs and modified ETL process. Every G_C that is a subgraph of the same G should consists of different sets of N_C, D_C and

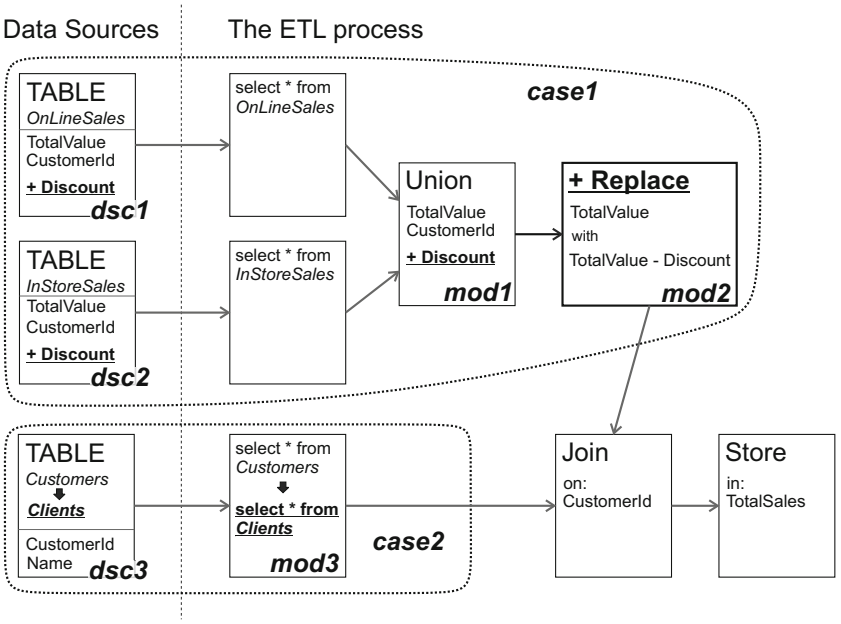

Fig. 1 An example of DSCs and ETL process modifications

SN_C. Sets N_C, D_C, and SN_C should be disjoint between different G_C extracted out of one G.

Another consequence of the completes and minimality constraints are deterministic cases extracted from one set of DSCs and modified ETL process. In other words, from one set of DSCs and modified ETL process we will always get the same cases. This is due to the fact that: (1) we cannot remove any DSC or modification from the case (the completeness constraint) and (2) we cannot add either DSC or modification to the case (the minimality constraint).

4.3 Case detection algorithm

Within the *E-ETL* framework we provide the *Case Detection Algorithm* (CDA) that detects and builds new minimal and complete cases for the ETL repair algorithm, based on Case-Based Reasoning. The algorithm consists of four main steps, presented in Algorithm 1.

- Step 1: DSCs and ETL process modifications are reduced in order to make cases more general. Step 1 is clarified in Section 6.
- Step 2: for each data source that has any change, a new case is initiated. The set of modifications for the new case consists of modifications that concern ETL activities that process data read from a changed data source. In other words, the set of modifications consists of modifications that change $Nodes$ or $SuperNodes$ dependent on (directly or indirectly) $Nodes$ or $SuperNodes$ representing a changed data source. At this point, cases are minimal but not complete.
- Step 3: each case is compared with other cases and if any two cases have at least one common modification then they are merged into one case. The purpose of the third step is to achieve the completeness.
- Step 4: the last *for all* loop in Algorithm 1 calculates the maximum value of the Factor of Similarity and Applicability (FSA) (cf. Section 5.2) that is used to speed up finding the best case. Step 4 is clarified in Section 5.3.

The CDA algorithm requires as an input: (1) a set of all detected data sources that have been changed and (2) an ETL process with all modifications that handle the detected changes. The CDA algorithm may detect multiple cases and return a set of them, in order to satisfy the minimality constraint. However, every returned case consists of different set of DSCs and different set of modifications. As a result of the completes constraint, DSCs and modifications sets are disjoint between cases.

The need to provide the input values into the CDA causes that the algorithm may be executed when all $DSCs$ has been applied to data sources and the ETL process has been repaired. The execution of the CDA after every atomic change has been applied to data sources prevents from detecting connected $DSCs$ and produces only cases with a single DSC. The of the CDA with the not repaired ETL process (as an input) produces cases that do not repair $DSCs$. Moreover, a missing input parameter to the CDA may cause violation of the completeness constraint. In other words, the CDA algorithm should be executed for the whole new version (fully updated version) of an external data source and a repaired ETL process.

Algorithm 1 *Case Detection Algorithm* (CDA)

Require: *data_sources*
Require: *ETL_process_activities*
1. *cases* ← []
   {Step 1 - reduction of DSCs and ETL process modifications}
2. reduce(*data_sources*, *ETL_process_activities*)
   {Step 2 - initiate minimal cases}
3. **for all** *data_source* in *data_sources* **do**
4. **if** *data_source* has *changes* **and** *data_source.changes* **not** in cases **then**
5. create new *case* **and** add to *cases*
6. add *data_source.changes* to *case*
   {detect modified ETL activities that process data from changed data sources}
7. **for all** *activity* in *ETL_process_activities* **do**
8. **if** *activity* process data from *data_source* **and** *activity* has *modifications* **then**
9. add *activity.modifications* to *case*
10. **end if**
11. **end for**
12. **end if**
13. **end for**
    {Step 3 - merge cases in order to achieve completeness}
14. **for all** *caseA* in *cases* **do**
15. **for all** *caseB* in *cases* **do**
16. **if** *caseB* has at least one the same *modification* with *caseA* **then**
17. merge *caseA* to *caseB*
18. remove *caseA* from *cases*
19. **end if**
20. **end for**
21. **end for**
    {Step 4 - compute maximum FSAs}
22. **for all** *case* in *cases* **do**
23. prepare maximum *FSAs* for the *case*
24. **end for**
25. **return** *cases*

5 Choosing the right case

One of the key points in Case-Based Reasoning is to choose the most appropriate case for solving a new problem. In an ETL environment, a new problem is represented by changes in data sources. Therefore, to solve the problem it is crucial to find the case whose set of DSCs is similar to the problem. Moreover, the set of modifications in the case must be applicable to the ETL process where the problem is meant to be solved.

The similarity of sets of DSCs means that although types of DSCs have to be the same, they can modify non-identical data source elements. For example, if a new DSC is an addition of column *Contact* to table *EnterpriseClients*, then a similar change can be an addition of column *ContactPerson* to table *InstitutionClients*. Both changes represent additions of columns but the names of the columns and the names of the tables are different. Therefore, a procedure that compares changes not only has to consider element names but also their types and subelements. If an element is an ETL activity, then subelements are its attributes. An ETL activity attribute does not have subelements. Added columns should have the same types and if they are foreign keys they should refer to the same table.

5.1 Decomposing problems

It is possible to decompose a given problem into smaller problems. Therefore, it is sufficient that the set of DSCs in the case is a subset of DSCs in a new problem. The opposite situation (the set of DSCs in the case is a superset of the set of DSCs in a new problem) is forbidden because it is not possible to apply to the ETL process modifications that base on missing DSCs.

The following example clarifies the possibility of decomposition. Let us consider new problem *Prob_AB* consisting of two DSCs, namely *DSC_A* and *DSC_B*. Let us assume that in the LRC there are three cases, namely *Case_A*, *Case_B*, and *Case_ABC*. *Case_A* has one DSC called *DSC_A* and one modification called *Mod_A*, which handle *DSC_A*. *Mod_A* is dependent on *DSC_A*. *Case_B* has one DSC - *DSC_B* and one modification - *Mod_B*, which handle *DSC_B*. *Mod_B* is dependent on *DSC_B*. *Case_ABC* has three DSCs, namely *DSC_A*, *DSC_B*, and *DSC_C*. These three DSCs are handled by two modifications, namely *Mod_AC* and *Mod_BC*. *Mod_AC* is dependent on *DSC_A* and *DSC_C*. *Mod_BC* is dependent on *DSC_B* and *DSC_C*. The following dependencies are visualized in Fig. 2.

The most desired case for *Prob_AB* has two DSCs, namely *DSC_A* and *DSC_B*, but it is not present in the example LRC. *Case_ABC* has *DSC_A* and *DSC_B*, but *Case_ABC* modifications depend on *DSC_C*, which is not present in *Prob_AB*. Therefore, *Case_ABC* modifications can not be applied in order to solve *Prob_AB*.

Let us assume that *Prob_AB* is decomposed into two problems, namely *Prob_A* and *Prob_B*. *Prob_A* consists of *DSC_A* and *Prob_B* consists of *DSC_B*. *Prob_A* can be solved using *Case_A* (application of modification *Mod_A*). *Prob_B* can be solved using *Case_B* (application of modification *Mod_B*). After decomposing, DSCs of *Prob_AB* are handled by modifications contained in *Case_A* and *Case_B*.

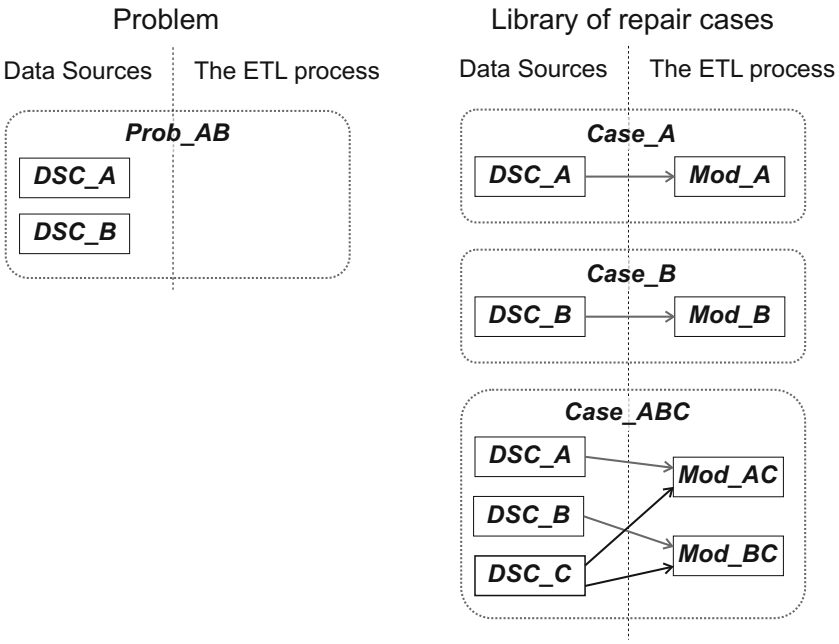

Fig. 2 Example of decomposing problems

5.2 Measuring similarity of cases

In order to repair an ETL process it has to be modified. Not all modifications can be applied to all ETL processes. Let us assume that there is an ETL process modification adding an ETL activity that sorts data by column *Age*. Moreover, in data sources that are a subject of the new problem there is no *Age* column, and therefore the ETL process does not process such a column. The ETL activity that sorts data by *Age* can not be added since this column is not present. It is impossible to add an ETL activity whose functionality is based on a non-existing column. Such a modification of the ETL process is not applicable. Thus, for each case it is necessary to check if the case modifications can be applied to an ETL process concerned by a new problem. In order to check a single modification, the element (i.e., an ETL activity or its attribute) of the ETL process that should be changed by the modification, must exist in ETL process concerned by a new problem.

An exception for that is the modification that adds a new element. What's more, the modified element should process a set of columns similar to the one processed by the case. In other words, the ETL activity should have a set of inputs and outputs similar to those defined in the case. To this end, each modification must contain an information about the surrounding elements and an information about activities between the modified element and a data source. The information about the surrounding elements includes the description of inputs and outputs of the modified activity.

As mentioned before, a case is similar to a new problem if the set of DSCs is similar to the new problem and modification can be applied to the ETL process concerned by a new problem.

It is possible that more than one case will be similar to a new problem. Thus, there is a need for defining how similar is the repair case to the new problem. The more similar is the repair case to the new problem the more probably is the application to solve the problem. Therefore, the case that is the most similar to the new problem should be chosen. To this end, for each case, a Factor of Similarity and Applicability (FSA) is calculated. The higher the value of FSA the more appropriate is the case for the new problem. If DSCs do not have the same types or modifications are not applicable, then FSA equals 0.

The more common features of the data sources (column names, column types, column type lengths, table names, table columns) the higher FSA is. Also the more common elements in the surroundings of the modified ETL process elements the higher FSA is. The scope of the case also influences FSA. Cases form the process scope have the increased value of FSA and cases form the global scope have the decreased value of FSA. If there are two cases with the same value of FSA, the case that has been already used more times is probably the more appropriate one. For that reason, the next element that is part of FSA is the number of accepted and verified by a user usages of the case.

In order to calculate FSA, common features of the data sources and common elements in the ETL process have to be obtained. To this end, a case has to be mapped into a new problem. DSCs defined in the case have to be mapped into DSCs defined in the new problem. Analogically, activities modified by the set of case modifications have to

be mapped into new problem activities. The procedure of mapping assures that for every case DSC, a corresponding new problem DSC is assigned. Assignation of corresponding DSCs is required to create data source collection pairs. Such a collection pair consists of: (1) a data source collection, changed by case DSC and (2) a collection changed by the corresponding new problem DSC. Collection pairs are necessary to calculate similarity of data sources. Analogically, pairs of corresponding activities are created. Activity pairs are necessary to calculate the applicability of modifications.

The FSA is a weighted sum of calculated data sources similarity and applicability of modifications. This sum is modified by the scope and number of usages.

FSA is computed by Formula 4.

$$FSA = (W_{simDs} * Sim_{ds} + W_{App} * App) * Scope * FactorOfUsage \tag{4}$$

Sim_{ds} and App are data sources similarity and applicability. W_{simDs} and W_{App} denote weights that control the importance of Sim_{ds} and App, respectively. The description and formulas for the similarity and the applicability are presented in Sections 5.2.1 and 5.2.2, respectively. The $Scope$ parameter represents the influence of the origin of the case on the FSA, as expressed in Formula 5.

$$Scope = \begin{cases} W_{processScope} & \text{if process scope} \\ W_{projectScope} & \text{if project scope} \\ W_{globalScope} & \text{if global scope} \end{cases}$$
$$W_{globalScope} < W_{projectScope} < W_{processScope} \tag{5}$$

Cases from the same process as a new problem have the highest scope. Cases from the same project have the middle value of the scope. Cases form the global scope have the lowest value of the scope. The last part of the FSA formula is $FactorOfUsage$, computed by Formula 6.

$$FactorOfUsage = W_{MaxUsage} - W_{NrOfUsage}{}^{NrOfUsage} * (W_{MaxUsage} - 1) \tag{6}$$

$NrOfUsage$ is the number of usages (accepted and verified by a user) of the case. $W_{NrOfUsage}$ controls the behavior of $FactorOfUsage$. Its value should be between 0 and 1. The higher is the value of $W_{NrOfUsage}$ the lower is the difference between frequently and infrequently used (accepted) cases. $W_{MaxUsage}$ defines the highest value of $FactorOfUsage$. The value of $W_{MaxUsage}$ must be higher than 1. The highest value of $FactorOfUsage$ is given for cases used very frequently. The lowest value of $FactorOfUsage$ is equal to 1 and it is assigned to never-used cases. $FactorOfUsage$ of never-used cases (value equal to 1) does not influence the value of FSA.

5.2.1 Similarity of data sources

Case data sources are data sources changed by the set of DSCs defined in the case. Analogically, **new problem data sources** are data sources changed by the set of DSCs defined in the new problem.

The similarity of case data sources and new problem data sources is based on a size of common data source elements ($Size_{commonDs}$) and a size of new problem data sources ($Size_{problemDs}$ that are affected by changes). Common elements are obtained by a procedure of mapping a case into a new problem. Every pair of common elements consists of an element from the case and its mapped equivalent from a new problem. The similarity of case data sources and new problem data sources is equal to 1 if a case data sources perfectly matches a new problem data sources. The similarity is equal to 0 if there are no common data source elements. Formula 7 defines the similarity of case data sources and new problem data sources.

$$Sim_{ds} = \frac{Size_{commonDs}}{Size_{problemDs}} \tag{7}$$

The size of the new problem ($Size_{problemDs}$) is a sum of all changed data source collections (tables, spread sheets, sets of records) and all of its elements (table columns, record properties). The size of the new problem is computed by Formula 8.

$$Size_{problemDs} = \sum_{\substack{changed \\ source \\ collections}} (W_{cName} + NrOfCollectionElements) \tag{8}$$

W_{cName} is a weight that controls an importance of a collection name in the similarity of collections. The size of common elements ($Size_{commonDs}$) is based on data source collection pairs. In each collection pair there is a collection from the analyzed case and an equivalent collection from the new problem. $Size_{commonDs}$ is a sum of sizes of collection pairs and is computed by Formula 9.

$$Size_{commonDs} = \sum_{\substack{collection \\ pairs}} Size_{dsCollectionPair} \tag{9}$$

Formula 10 describes the size of each collection pair that is a sum of similarities of collection names ($Sim_{collName}$) and similarities of mapped collection elements.

$$Size_{dsCollectionPair} = W_{cName} * Sim_{collName} + \sum_{elems} Sim_{elem} \tag{10}$$

W_{cName} controls an importance of collection names similarities. $Sim_{collName}$ is a string similarity which is based on

the edit distance (Levenshtein 1966). The string similarity is computed by Formula 11.

$$string\ similarity = \frac{stringLength}{stringLength + editDistance} \tag{11}$$

The similarity of collection elements Sim_{elem} compares equivalent elements. It takes into account elements names ($Sim_{elemName}$), types (Sim_{type}), types sizes (Sim_{size}) and types precisions (Sim_{prec}). The similarity of collection elements is defined by Formula 12.

$$Sim_{elem} = W_{ceName} * Sim_{elemName} + W_{ceType} * Sim_{type} + W_{ceSize} * Sim_{size} + W_{cePrec} * Sim_{prec} \tag{12}$$

W_{ceName}, W_{ceType}, W_{ceSize} and W_{cePrec} are weights that control importances of all Sim_{elem} parts and all together should sum to 1 (otherwise $Size_{problemDs}$ will be invalid). Example values of these weights could be: $W_{ceName} = 0.67$, $W_{ceType} = 0.2$, $W_{ceSize} = 0.08$, $W_{cePrec} = 0.02$. $Sim_{elemName}$ is computed as the string similarity (like $Sim_{collName}$). Sim_{type} compares elements types (string, integer, decimal, float, etc.) as in Formula 13.

$$Sim_{type} = \begin{cases} 1 & \text{if the same} \\ 0 & \text{if different} \end{cases} \tag{13}$$

Sim_{size} defined by Formula 14 and Sim_{prec} defined by Formula 15 are calculated only if elements types are equal.

$$Sim_{size} = \begin{cases} 0 & \text{if } Sim_{type} = 0 \\ \frac{Size_{caseElem} - |Size_{caseElem} - Size_{problemElem}|}{Size_{caseElem}} & \text{if } Sim_{type} = 1 \end{cases} \tag{14}$$

Where $Size_{caseElem}$ and $Size_{problemElem}$ are case and problem collection element type sizes, respectively.

$$Sim_{prec} = \begin{cases} 0 & \text{if } Sim_{type} = 0 \\ \frac{Prec_{caseElem} - |Prec_{caseElem} - Prec_{problemElem}|}{Prec_{caseElem}} & \text{if } Sim_{type} = 1 \end{cases} \tag{15}$$

Where $Prec_{caseElem}$ and $Prec_{problemElem}$ are the case and the problem collection element type precisions, respectively.

5.2.2 Applicability

The applicability of case modifications to a new problem ETL process is based on a size of common parts of the ETL process ($Size_{commonProc}$) and a size of the case ETL process that is affected by changes ($Size_{caseProc}$). The applicability is equal to 1 if the case ETL process perfectly matches to the new problem. Formula 16 defines the applicability.

$$App = \frac{Size_{commonProc}}{Size_{caseProc}} \tag{16}$$

$Size_{caseProc}$ defines a size of changed ETL activities in the analyzed case. For each modified activity three parameters are evaluated: (1) a number of activity input and output elements ($NrInOuts$), (2) a number of activity properties ($NrProps$), and (3) a number of activities that in the shortest path that leads from changed data sources to the analyzed activity ($NrPreds$). $Size_{caseProc}$ is a weighted sum of these parameters computed by Formula 17.

$$Size_{caseProc} = \sum_{\substack{modified \\ activities}} (W_{inOuts} * NrInOuts + W_{prop} * NrProps + W_{pred} * NrPreds) \tag{17}$$

The size of common activities of the ETL process ($Size_{commonProc}$) is a sum of sizes of activities pairs computed by Formula 18. Activity pair consists of one activity form an analyzed case and matched activity form a new problem.

$$Size_{commonProc} = \sum_{\substack{modified \\ activityPairs}} Size_{activityPair} \tag{18}$$

For each single activity pair three parameters are evaluated: (1) a similarity of activity input and output elements (Sim_{InOuts}), (2) a similarity of activity properties (Sim_{prop}), and (3) a similarity of activities that leads from data sources to the analyzed activity (Sim_{pred}). $Size_{activityPair}$ is a weighted sum of these parameters computed by Formula 19. The weights are the same as for $Size_{caseProc}$ (cf., Formula 17).

$$Size_{activityPair} = W_{inOuts} * Sim_{InOuts} + W_{prop} * Sim_{prop} + W_{pred} * Sim_{pred} \tag{19}$$

Formula 20 defines Sim_{InOuts} as a sum of similarities of all inputs and all outputs of the ETL activity. The similarity of a single input or an output is computed in the same way as the similarity of collection elements (Formula 12). Every input and output of the case activity is compared with its matched input or output of the new problem activity. The comparison is done on: name, type, type size, and type precision.

$$Sim_{InOuts} = \sum_{\substack{inputs \\ outputs}} Sim_{elem} \tag{20}$$

Many ETL activities have some properties. For example, an activity that sorts a data stream has a property which tells by which attribute a sort operation should be done. Sim_{prop} is a number of properties that are equal in the case activity and in the new problem activity. Formula 21 defines similarity of properties.

$$Sim_{prop} = \sum_{properties} \begin{cases} 1 & \text{if the same value} \\ 0 & \text{if different value} \end{cases} \quad (21)$$

The last part of $Size_{activityPair}$ is Sim_{pred} which compares a placement of the activities in the ETL processes. This comparison is done by comparing shortest paths from the changed data source. The comparison of paths is done like the comparison element names - string similarity. $editDistance$ of the paths is a number of operations (insertion, deletion, substitution) that are necessary to convert the path that leads to the case activity into the path that leads to the new problem activity. A value of Sim_{pred} is computed by Formula 22.

$$Sim_{pred} = \frac{pathLength}{pathLength + editDistance} \quad (22)$$

5.2.3 Similarity of semantics

In the framework presented in this paper, the computation of FSA is based only on structural properties of data sources and an ETL process. The *E-ETL* analyzes only the structure of data sources (e.g. collection names, collection elements, data types). In the prototype implementation of *E-ETL*, an ETL process is read from a project file (i.e. *.dtsx – SSIS ETL process description file). The file contains:

- metadata about a control flow of the ETL process;
- the type of each activity (e.g., sorting, joining, look-up) in the ETL process;
- the structure of a data set processed by the ETL process.

The metadata usage in *E-ETL* can be easily extended (and is planned as one of the next steps) in order to analyze some semantics of the data sources and ETL processes. One of the possible ways to do this is to take into consideration basic statistics of each collection and collection element. The statistics on a collection include among others:

- the number of tuples in the collection,
- the frequency of additions of new tuples to the collection,
- the frequency of deletions of tuples in the collection,
- the frequency of updates of tuples in the collection.

The statistics on a collection element include among others:

- the number of unique values,
- min, max, average of numeric values,
- min, max, average of the length of text values,
- standard deviation and variance of numeric values,
- standard deviation and variance of the length of text values,
- the percentage of null values,
- histograms of attribute values.

Additionally, the metadata on activities and its parameters can be extended with statistics on an execution of an ETL process. The statistics might include among others: the number of input tuples to the ETL process, the number of output tuples from the process, the number of input and output tuples of each ETL activity.

5.3 Searching algorithm

In the *E-ETL* framework we provide the *Best Case Searching Algorithm* (BCSA) whose purpose is to search for the best case in the LRC. In an unsupervised execution of the *E-ETL* a case with the highest calculated FSA is chosen as the best one and is used for a repair. However, when the execution of the *E-ETL* is supervised by a user then he/she has to accept the usage of the case. In order to handle a situation when a user does not accept a case with the highest FSA, the *E-ETL* presents to a user n best cases so he/she can choose one of them. The value of n can be configured by a user.

The BCSA consists of one main loop that iterates over every case in the LRC. Every case is mapped into a new problem and the FSA is calculated. In order to speed up the execution of the BCSA it is crucial to limit the set of potential cases for the new problem out of the whole LRC.

The set of cases can be limited to only those that have lower or equal number of DSCs (for each change type), as compared to number of DSCs in a new problem. If there are more DSCs in a case than in the new problem, then the case DCSs cannot be mapped into the new problem DCSs. Therefore, cases that have more DSCs than included in the new problem are skipped.

The final step of Algorithm 1 prepares for computing the maximum value of FSAs. Maximum FSA can be achieved if the case perfectly matches a new problem. If a value of a case maximum FSA is lower than the FSA value of the best case that has already been checked, then a calculation of the case FSA can be omitted. Therefore, cases maximum FSAs are used in order to reduce number of cases that have to be checked.

The detailed description of the Best Case Searching Algorithm can be found in Wojciechowski (2016).

5.4 Storing the library of repair cases

The Library of Repair Cases is the root of the storage model and it contains the collection of *Case* objects. *Case* objects

are stored in two database tables, namely: (1) *CaseInfo* and (2) *CaseDetails*. Every row in *CaseInfo* represents data about a single repair case, i.e., a single *Case* objects. The *Case* object contains: (1) a number of accepted case usages required to compute $FactorOfUsage$, (2) a project and process identifiers required to define the case scope, (3) an information about a number of DSCs, (4) a precomputed $Size_{caseDs}$, and (5) a reference to details of the case.

Details of the case are stored in the *CaseDetails* table. Every row in *CaseDetails* contains a serialized data about: (1) changed data sources and (2) modifications of an ETL process.

The LRC is stored in two tables in order to get the best results from the optimization steps of the BCSA (cf. Section 5.3). If a case passes the optimization checks (i.e., the number of DSCs and maximum FSA), then data from *CaseDetails* are read.

A detailed description of the storage of the LRC can be found in Wojciechowski (2016).

6 Case reduction

The aim of reducing a case is to make a case more general. The more general is a case, the higher is the probability that the case will match a new problem. The case can became more general by reducing the number of its elements. It can be done in two ways, namely: (1) a reduction of DSCs and (2) a reduction of a modification. The less DSCs are included in the case, the more new problems can be decomposed to that case. The less modifications are contained in the case, the higher the probability that the modifications are applicable to an ETL process of a new problem. Smaller number of the case elements reduces time needed to search the best case, by reducing the number of elements that have to be matched and compared.

A reduction of the case elements must be done with respect to the completeness constraint. Therefore, every removed element from a case must be restored in the process of applying the case solution to a new problem. Case elements that can be restored represent modifications, which can be created automatically. Only the elements that can be created automatically can be removed from the case.

6.1 Rules

E-ETL includes not only the CBR based algorithm but also the *Defined rules* algorithm. *Defined rules* applies evolution rules, defined by a user, to particular elements of an ETL process. For each element (activity or activity attribute) of an ETL process, a user can define whether this element is supposed to propagate changes, to block them, to ask a user, or to trigger action specific to a detected change. Let us consider a simple example of a DSC, that changes a name of one table column from *ClientId* to *CustomerId*. The *Defined rules* algorithm can propagate column name change through all ETL process activities, by changing these activities to use a new column name.

Rules can be set on every $Node$ and $SuperNode$ of graph G. When $Node_A$, which represents a CollectionElement, is affected by a DSC then for a given DSC an Evolution Event (EE) is created ($DSC \rightarrow EE$). EE contains information about changes of $Node_A$. EE is propagated to all $Nodes$ that directly depend on $Node_A$. According to EE and rules set on $Nodes$ every node is modified (or not if the *block* rule is set) in order to adjust the $Nodes$ to the changed data source. The modifications of $Nodes$ can be described as propagation modifications PMs. In other words, EE propagated to $Nodes$ cause PMs ($EE \rightarrow PMs$).

New EEs are created on the modified $Nodes$ and propagated to the next dependent $Nodes$ ($PMs \rightarrow EEs$). The propagation process repeats recursively until there is no EE to propagate or there is no dependent $Nodes$.

For a large ETL process, defining rules for every activity may be inefficient. Therefore, there is a possibility to define rules for groups of activities and a possibility to define default rules. Default rules can be set for the whole ETL process (e.g., the *propagate* rule in order to propagate all changes). Moreover, default rules can be set for selected types of DSC (e.g., the *block* rule for collection deletion). The definition of the default rules is not linked with a specific ETL process. Therefore, a user can use the same set of the default rules for all ETL processes. Default rules are sufficient to employ the *Defined rules* algorithm in order to reduce a case. The *Defined rules* algorithm is described in details in Wojciechowski (2011, 2013a, b).

Every modification that simply propagates changes from predecessors can be created automatically using the *Defined rules* algorithm. Therefore, a modification that simply propagates changes from predecessors can be removed from a case during the case reduction.

Every DSC that is handled only by removed modifications can be removed from the case. Such DSCs removed from the case are fully handled by the *Defined rules* algorithm. Let us assume that PMs is the set of all modifications created by the *Defined rules* algorithm. $Case_{Reduced} = (DSCs_R, Ms_R)$ is the reduced case. The reduced set of modifications satisfies the following condition: $M \in Ms_R \Rightarrow M \notin PMs$.

The reduction of an already created case may violate the minimality constraint. Two DSCs are contained in the same case if the case contains modifications of at least one ETL process activities that handle these DSCs. If all modifications that handle given two DSCs are reduced then the case is not minimal and can be separated into two cases.

6.2 DSCs reduction

In order to avoid detecting cases that became not minimal, the reduction of DSCs and modification should be executed at the very beginning of the process of detecting cases. Therefore, the first step of Algorithm 1 (described in Section 4.2.1) is the reduction of DSCs and ETL process modifications. The pesudo-code of this first step is presented in Algorithm 2. The algorithm consists of two main steps.

- In the first step, for each ETL process activity that has any modification, a set of automatic modifications (created by the *Defined rules* algorithm) is obtained. If the set of automatic modifications is equal to the set of all activity modifications, then the modifications are removed from the activity. Since such an activity does not include other modifications, it is not further processed in Step 2 of Algorithm 1.
- In the second step, each data source that has any changes is processed. If there is no activity that handles

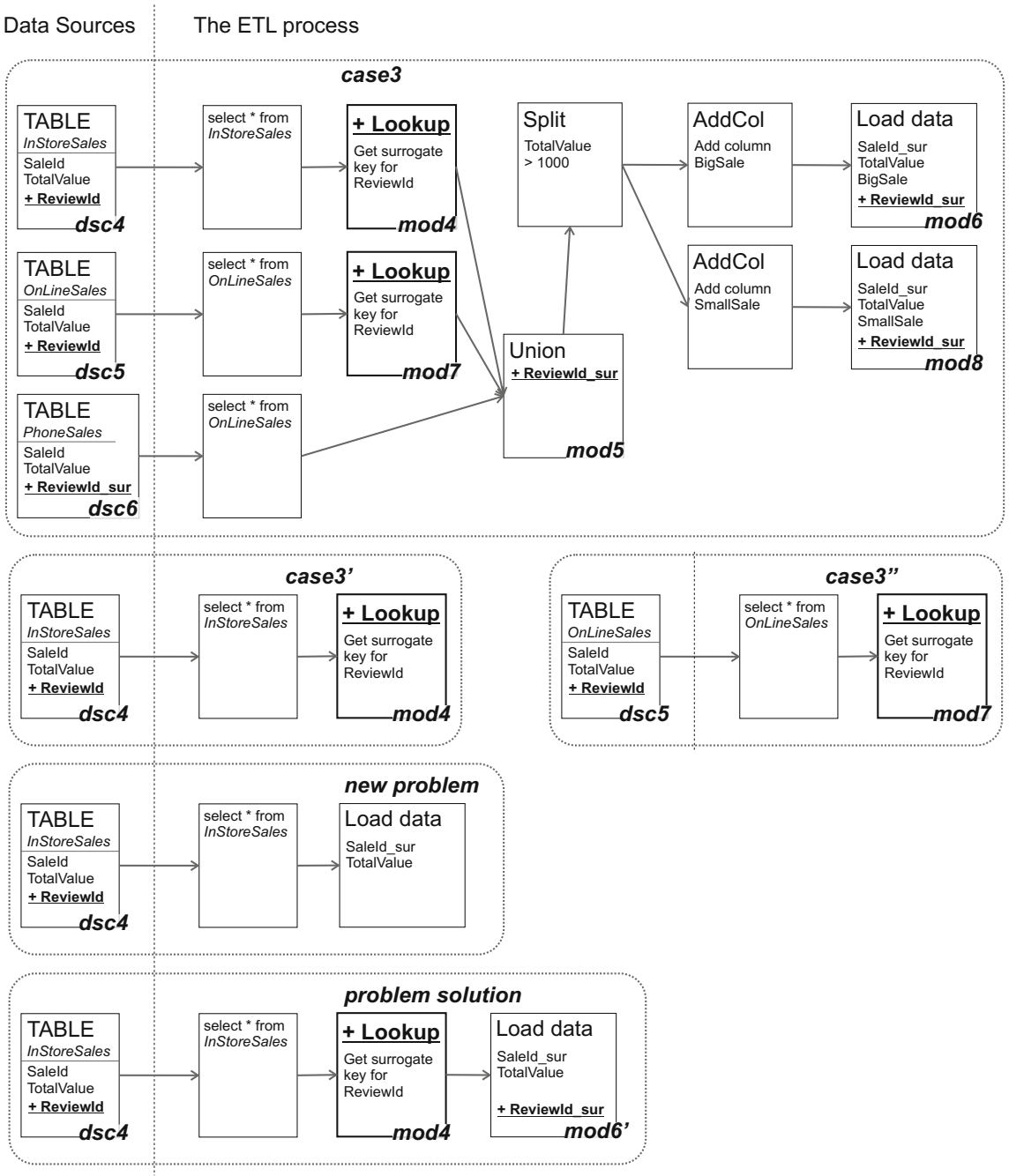

Fig. 3 An example of a case reduction

changes of a given data source, then changes are removed from the data source. Since such a data source does not include other changes it is not further processed in Step 2 of Algorithm 1.

After the execution of Algorithm 2, the set of DSCs and ETL process modifications are reduced. After reducing DSCs and ETL process modifications, the second and the third step of Algorithm 1 create minimal, complete, and reduced cases.

Algorithm 2 The *Case Reduction Algorithm* (CRA)

Require: *data_sources*
Require: *ETL_process_activities*
{Step 1 - Reduce set of modifications}
1. **for all** *activity* in *ETL_process_activities*
2. **if** *activity* has *modifications* **then**
3. *current_modifications* ← *activity.modifications* {get activity current modifications}
4. *default_modifications* ← *activity.get_default_modification* {get activity modifications that would be created as a result of data source changes propagation}
5. **if** *current_modifications* equals *default_modifications* **then**
6. remove *modifications* from *activity*
7. **end if**
8. **end if**
9. **end for**
{Step 2 - Reduce set of DSCs}
10. **for all** *data_source* in *data_sources* **do**
11. **if** *data_source* has *changes* **and** *ETL_process_activities* **not** contains *activity* that handles *data_source.changes* **then**
12. remove *changes* from *data_source*
13. **end if**
14. **end for**

Figure 3 shows an example of the case reduction. The figure consists of four elements, namely: (1) a not-reduced case - *case3*, (2) reduced cases - *case3'* and *case"*, (3) a new problem, and (4) a solution for the new problem.

case3 is based on an ETL process that reads data from three tables, namely: *InStoreSales*, *OnLineSales*, and *PhoneSales*. Next, the data are merged by the union activity. The merged data are split into two types of sales, namely: sales with TotalValue higher than 1000 and sales with TotalValue lower or equal to 1000. Column *BigSale* is added to the first type of sales. Column *SmallSale* is added to the second type of sales. Finally, both types of sales are loaded into a DW.

The not-reduced *case3* consists of three DSCs, namely: *dsc4*, *dsc5*, and *dsc6* and five modifications, namely: *mod4*, *mod5*, *mod6*, *mod7*, *mod8*. *dsc4* adds column *ReviewId* to table *InStoreSales*. *dsc5* adds column *ReviewId* to table *OnlineSales*. *dsc6* adds column *ReviewId_sur* to table *PhoneSales*. *mod4* and *mod7* add *Lookup* activities creating surrogate keys for *ReviewId*.

Since *dsc6* already adds a surrogate key, there is no modification that adds *Lookup* activity for data read from table *PhoneSales*. The new activities are added between existing activities that read data and activity *Union*. *mod5* includes the newly added surrogate key (*ReviewId_sur*) in activity *Union*. *mod6* and *mod8* include surrogate keys (*ReviewId_sur*) in the activities loading data into the DW.

mod4 and *mod7* are modifications that represent new business logic introduced by a user in response to *dsc4* and *dsc5*, respectively. *mod4* and *mod7* cannot be created automatically (by the *Defined rules* algorithm). However, *mod5*, *mod6*, and *mod8* simply propagate the addition of newly created surrogate keys. *mod5*, *mod6*, and *mod8* can be created automatically (by the *Defined rules* algorithm). Therefore, *mod5*, *mod6*, and *mod8* can be removed from *case3*.

The reduction of modifications in *case3* leads to state in which there is no modification that handles *dsc6*. As a result, *dsc6* can be removed from *case3*. In the current state, *case3* consists of two DSCs, namely: *dsc4* and *dsc5* as well as two modifications, namely: *mod4* and *mod7*. *mod4* handles *dsc4* whereas, *mod7* handles *dsc5*. Since there is no single modification that handles both DSCs, *case3* is not minimal and can be split into two separated cases, namely *case3'* and *case3"*. *case3'* consist of one DSC - *dsc4* and one modification - *mod4*. *case3"* consist of one DSC - *dsc5* and one modification - *mod7*. *case3'* and *case3"* represent complete, minimal, and reduced cases.

Let us assume the new problem that occurs in an ETL process that reads data from table *OnLineSales* and loads them into the DW. DSC that has to be handled (*dsc4*) adds column *ReviewId* to table *OnLineSales*. *case3* can not be used for solving the new problem because of the two following incompatibilities:

- *case3* has more (three) DSCs than the new problem (one).
- The activities of the modified ETL process can not be mapped into activities of the ETL process in the new problem. In the new problem, there is only one source table, therefore only one activity *Lookup* can be added. In the new problem, there is no activity *Union* and there is only one activity loading data into the DW, as compared to two such activities in *case3*.

Therefore, sets of DSCs are not similar and modifications contained in *case3* are not applicable to the new problem ETL process.

However, *case3'* perfectly matches the new problem. *case3'* consists of one DSC (*dsc4*), which is the same as

DSC in the new problem. Modification *mod4* contained in *case3'* is applicable to the new problem.

In order to solve the new problem by using *case3'*, *mod4* has to be applied, i.e. the *Lookup* activity creating a surrogate key for *ReviewId* has to be added after activity reading data. Since the new problem is solved by the reduced case, the result of the introduced modifications has to be propagated through the next activities, i.e. addition of column *ReviewId_sur* (the result of activity *Lookup*) has to be propagated to the activity loading data into the DW. Propagation of the new column is done by the *Defined rules* algorithm. The result of the propagation is *mod6'* that includes the surrogate key (*ReviewId_sur*) in the activity loading data into the DW.

7 Use case

In order to illustrate the application of the repair algorithm to the Case-Based Reasoning method, we will present an example use case.

The use case consists of three elements, namely: (1) the LRC, (2) a new problem, and (3) a solution for the new problem. Figure 4 presents the three elements. the LRC consists of four cases, namely: *case1*, *case2*, *case4*, and *case5*. *case1* and *case2* are described in Section 4 and presented in Fig. 1. *case4* and *case5* are shown in Fig. 4.

case4 is based on an ETL process that reads data from table *InStoreSales*. Next, a surrogate key is created for *SaleId*. Finally, data are loaded into a DW. *case4* consists

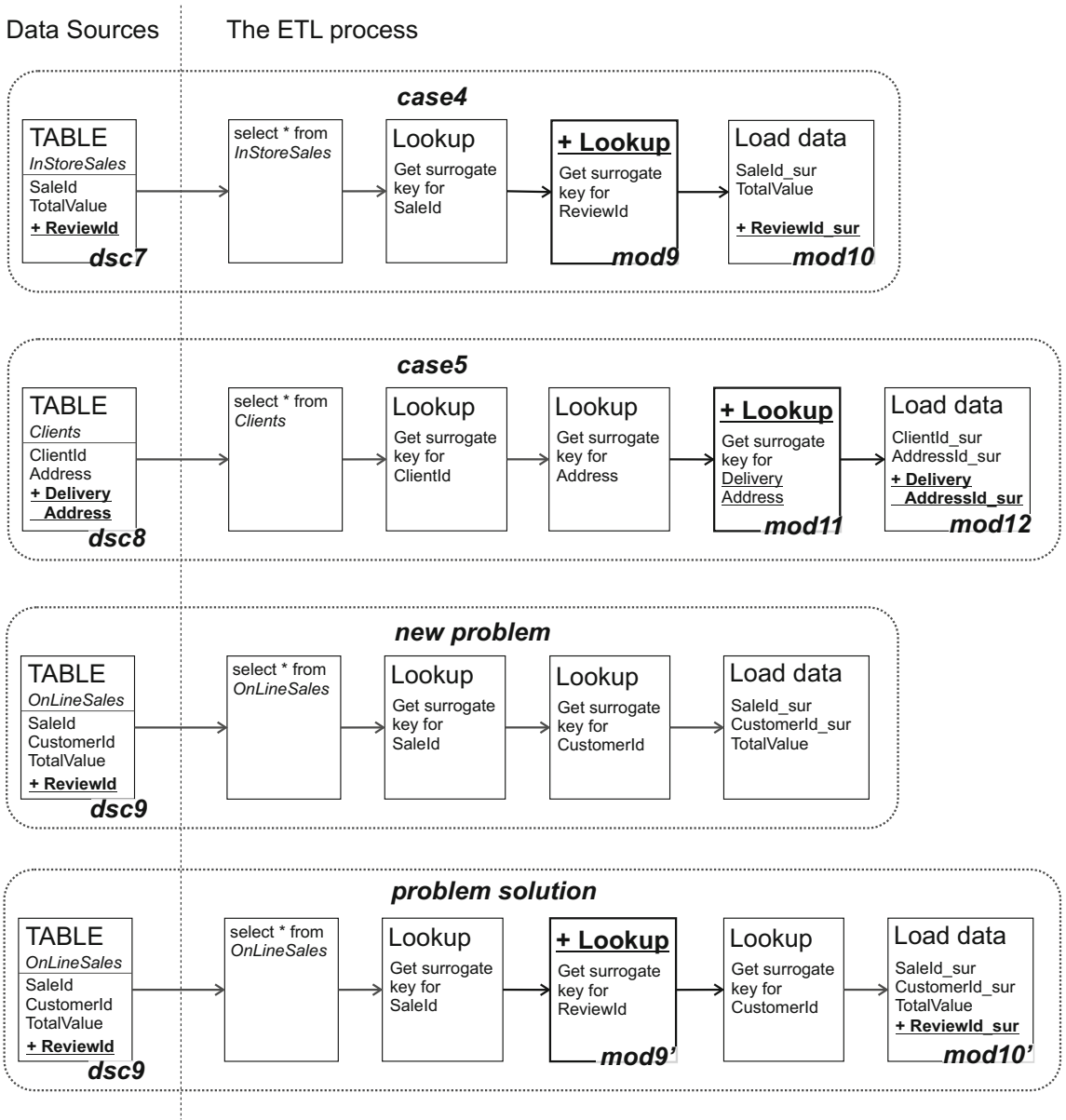

Fig. 4 Use case

of one DSC - *dsc7* and two modifications, namely *mod9*, *mod10*. *dsc7* adds column *ReviewId* to table *InStoreSales*. *mod9* adds *Lookup* activity creating a surrogate key for *ReviewId*. The new activity is added between existing activities *Lookup* and *Load data*. *mod10* includes the newly created surrogate key (*ReviewId_sur*) in the activity loading data into the DW.

case5 is based on an ETL process that reads data from table *Clients*. Next, surrogate keys are created for *ClientId* and *Address*. Finally, data are loaded into the DW. *case5* consists of one DSC - *dsc8* and two modifications, namely *mod11*, *mod12*. *dsc8* adds column *DeliveryAddress* to table *Clients*. *mod11* adds *Lookup* activity creating a surrogate key for *DeliveryAddress*. The new activity is added between existing activities *Lookup* and *Load data*. *mod12* includes the newly created surrogate key (*DeliveryAddressId_sur*) in the activity loading data into the DW.

The new problem in the use case occurs in an ETL process that reads data from table *OnLineSales*. Next, surrogate keys are created for *SaleId* and *CustomerId*. Finaly, data are loaded into a DW. DSC that has to be handled (*dsc9*) adds column *ReviewId* to table *OnLineSales*. In order to find a solution for the new problem, the LRC is searched. *case1* cannot be used because it contains two DSCs, whereas in the new problem there is only one DSC. *case2* cannot be used because it contains DSC renaming a table and in the new problem there is DSC adding a column. Both *case4* and *case5* have one DSC that adds a column. Since *dsc7* matches *dsc9* in the new problem, an applicability of the modifications can be checked. In order to apply *mod9*, an ETL process in the new problem must contain the *Lookup* activity after which the new *Lookup* activity can be added. In order to apply *mod10*, an ETL process in the new problem must contain the *Load data* activity. Required activities must process data read from the changed data source. There are required activities in the new problem ETL process. Therefore, the *case4* solution can be applied in order to solve the new problem. The FSA is calculated for this case.

Since *dsc8* matches *dsc9* in the new problem, an applicability of the modifications can be check. In order to apply *mod11*, an ETL process in the new problem must contain the *Lookup* activity after which the new *Lookup* activity can be added. In order to apply *mod12*, an ETL process in the new problem must contain the *Load data* activity. Required activities must process data read from the changed data source. There are required activities in the new problem ETL process. Therefore, the *case5* solution can be applied in order to solve the new problem. The FSA is calculated for this case.

case4 has more common elements with the new problem than *case5*. The following elements are common: (1) data source table columns (e.g., *SaleId* and *TotalValue*), (2) similar table names (e.g., *InStoreSales* and *OnLineSales*), and (3) similar properties of the activities (e.g., *SaleId* in activity *Lookup*). Therefore, FSA of *case4* is higher than FSA of *case5*. Since the best case is the one with the highest FSA, the best case to solve the new problem is *case4*.

case4 modifications (*mod9* and *mod10*) are applied in order to handle *dsc9*. *mod9'* is based on *mod9*. *mod9'* adds the *Lookup* activity creating a surrogate key for *ReviewId* after the first existing *Lookup* activity. *mod10'* is based on *mod10*. *mod10'* includes the newly created surrogate key (*ReviewId_sur*) in the activity loading data into the DW.

A solution that consists of modifications *mod9'* and *mod10'* is presented to a user. If he/she accepts the solution, then the modifications are stored in the ETL process.

8 Performance evaluation

In the *E-ETL* approach it is up to a user to select a repair scenario from the most suitable scenarios provided by *E-ETL*. For this reason, the most crucial component of *E-ETL* the BCSA algorithm. The goal of this experimental evaluation was to assess the performance of the BCSA. To this end, we conducted a series of experiments. We assessed how execution time of the BCSA is influenced by the following factors:

- the number of DSCs in a new problem,
- the size of data sources modified by DSCs in a new problem,
- the number of activities in an ETL process of a new problem,
- the size of an ETL process of a new problem,
- the length of the longest path in an ETL process of a new problem,
- the number of cases in the LRC.

Even though, there are a few benchmarks related to ETL, e.g., (Council 2016; Alexe et al. 2008), our experiments were based on test scenarios that we developed. The reason for using our own test scenarios is because the existing ETL benchmarks focus on measuring the performance of the whole ETL process, whereas our work deals with the problem of evolving data sources and the evolution of ETL processes. Since the algorithms that we developed do not execute ETL processes but change their definitions, we found the existing ETL benchmarks inadequate to our problem.

Our test scenarios were based on the LRC that was filled with random cases. The cases reflected real schema changes described in multiple research papers, e.g., (Curino et al. 2008b; Qiu et al. 2013; Sjøberg 1993; Skoulis et al. 2014;

Vassiliadis et al. 2015; Wu and Neamtiu 2011). The test cases stored in the LRC contained:

- from 1 to 10 data source changes,
- random data sources (with random: collection names, number of collection elements as well as collection elements names, types, and precisions),
- from 1 to 30 modifications in an ETL process,
- from 3 to 20 activities in the longest path of a modified ETL process.

Searching for the best cases was executed for randomly created problems. The problems contained:

- from 1 to 10 DSCs,
- random data sources (with random: collection names, number of collection elements as well as collection elements names, types, and precisions),
- from 3 to 20 activities in the longest path of the ETL process.

The calculation of the FSA for multiple cases was implemented to run in parallel threads, as it is the most expensive part of the algorithm.

The test were divided into four phases. Each phase consisted of two steps. In the first step, 27,000 new random cases were added to the LRC. In the second step, 18,000 new random problems were generated. For each new problem the algorithm searching the best case was executed. As a consequence, we executed 18,000 searches in the LRC with 27,000, 54,000, 81,000 and 108,000 of random cases. In total, we executed 72,000 searches. Each value presented on the charts below represents an average elapsed time of at least 200 executions.

The tests were run on workstation with processor Intel Xeon W3680 (6 cores, 3.33GHz), 8GB RAM and a SSD disk. The LRC were stored in a MS SQLServer 2012 database. In the following subsections we describe the results of our performance tests.

As mentioned in Section 2, the *E-ETL* framework uses its own internal data model that is independent of any external ETL tool. BCSA searches for the most appropriate case and operates only on the internal data model. A repair of the ETL process is not part of the BCSA. Therefore, BCSA is also independent of an external ETL tool. For these reasons, the chosen ETL tool does not influence the experiments. Since the *E-ETL* framework currently is well connected to Microsoft SQL Server Integration Services via the API, in the experiments we used the Microsoft tool.

In this section, we show the searching performance of the BCSA for: (1) a variable number of DSCs, (2) a variable size of a data source, (3) a variable number of activities in an ETL process, (4) a variable size of an ETL process, (5) a variable length of the longest path in an ETL process, and (6) a variable number of cases in the LRC.

8.1 Searching the best case for a variable number of DSCs

Figure 5 presents the impact of the number of DSCs in a new problem on the BCSA elapsed execution (search) time. As we can observe from the chart, the execution time increases with the increase of the number of DSCs in a new problem. This increase is close to linear.

As mentioned in Section 5.1, the a problem can be decomposed into smaller problems. The more DSCs are in the a problem, the smaller problems can be created by means of a decomposition. For example, if there is a new problem with one DSC, then the CRA has to check only cases with the same DSC. If there is a new problem with two DSCs, namely: *DSC_A* and *DSC_B*, then the CRA has to check cases that contain: (1) only *DSC_A*, (2) only *DSC_B*, as well as both *DSC_A* and *DSC_B*. Therefore, the more DSCs are in the new problem, the more cases have to be compared.

8.2 Searching the best case for a variable size of a data source

Figure 6 shows how the BCSA search time depends on the size of data sources defined in a new problem. The size of a data source is calculated as sum of the number of data source collections and the number of data source collection elements (i.e., the sum of the number of tables and their columns).

From Fig. 6, we observe that for a data source of size about 350 elements, the time increases, and remains rather constant for DS sizes greater than approximately 350. In our opinion, it is because the mechanism of maximum FSA is efficient only for larger data sources. Therefore, a time required for mapping large data sources is compensated by a time gained from reduction of the number of cases required to check.

As mentioned in Section 5.2, a data source defined in a new problem has to be mapped into a data source

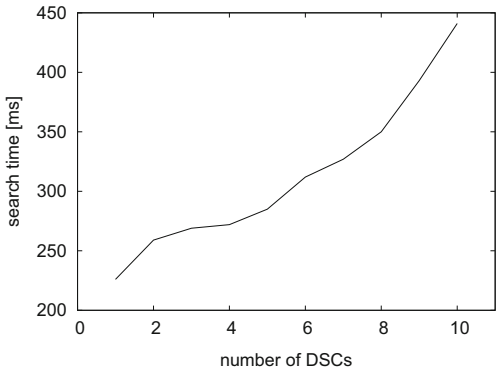

Fig. 5 Searching the best case for a variable number of DSCs

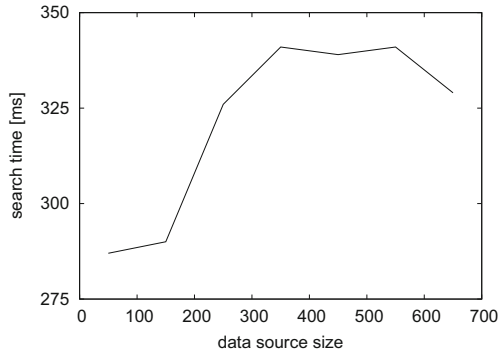

Fig. 6 Searching the best case for a variable size of a data source

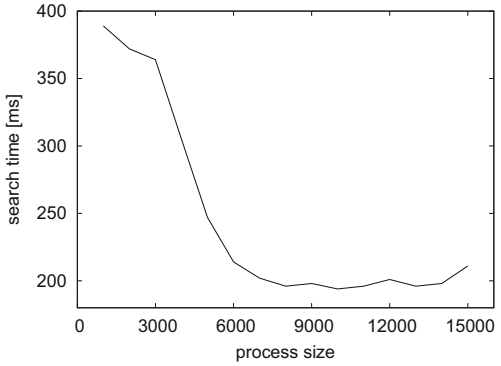

Fig. 8 Searching the best case for a variable size of an ETL process

defined in a case. The more elements exist in a data source defined in a new problem, the more difficult is to create the mapping. Moreover, usually larger data sources contain more DSCs. Therefore, an increase of a data source size causes an increase of the BCSA search time. On the other hand, as mention in Section 5.3, the maximum value of the *Factor of Similarity and Applicability* (FSA) is used to reduce the number of cases to check. The more elements exist in a data source defined in a new problem, the higher is the probability that a case achieves a high FSA score.

8.3 Searching the best case for a variable number of activities in an ETL process

Figure 7 presents an influence of the number of activities in an ETL process defined in a new problem on the search time. The more activities exist in an ETL process defined in a new problem, the higher is the probability that a case achieves a high FSA score. The higher the FSA score of cases already checked, the more efficient is the reduction based on maximum FSAs of cases. Therefore, the more activities exist in an ETL process defined in a new problem, the lower is the execution time of the search algorithm.

8.4 Searching the best case for a variable size of an ETL process

Figure 8 presents an influence of the size of an ETL process defined in a new problem on the search time. The size of an ETL process defined in a new problem is calculated as the sum of the number of activities and the number of activity elements. The size of an ETL process is similar to the number of activities in the ETL process. Therefore, both factors have similar influence on execution time.

In the figure we observe that the higher the size of an ETL process defined in a new problem, the lower is the execution time of the search algorithm.

8.5 Searching the best case for a variable length of the longest path in an ETL process

Figure 9 presents the influence of the longest path in an ETL process defined in a new problem on the execution time. The longer the longest path in an ETL process, the more activities exist in the ETL process. As we observe from the chart, the search time decreases with the increase of the length of the longest path in an ETL process. The longer is the longest path (more activities) in an ETL process defined in a new

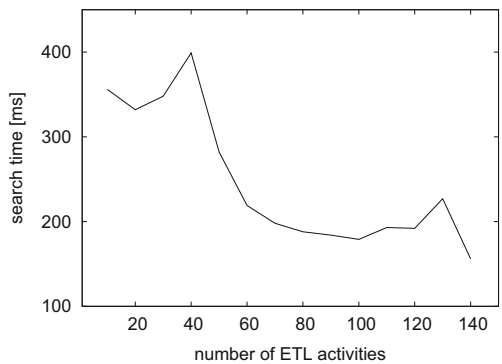

Fig. 7 Searching the best case for a variable number of activities in an ETL process

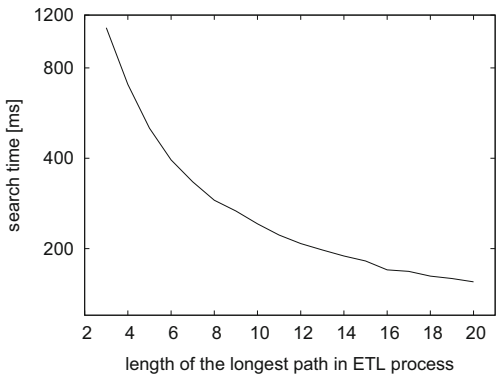

Fig. 9 Searching the best case for a variable length of the longest path in an ETL process

problem, the higher is the probability that a case achieves a high FSA score.

From the figure we observe that the higher the FSA score of cases already checked, the more efficient is the reduction based on maximum FSAs of cases.

8.6 Searching the best case for a variable number of cases in the LRC

Figure 10 presents the influence of the size of the LRC on the execution time. The chart reveals that this dependency is linear, as the higher the number of cases in the LRC, the more case have to be checked.

9 Related work

The work presented in (Schank 1983) is considered to be the origin of Case-Based Reasoning. The idea was developed, extended, and reviewed in (Aamodt and Plaza 1994; Hammond 1990; Kolodner 1992; 1993; Watson and Marir 1994). Since the algorithm presented in this paper uses only a key concept of this methodology, the mentioned publications will not be discussed here. In this section we focus on related approaches to supporting the evolution of the ETL layer.

In (Papastefanatos et al. 2008), the authors proposed several metrics based on the graph model (the same as in *Hecataeus*) for measuring and evaluating the design quality of an ETL layer with respect to its ability to sustain DSCs. In (Papastefanatos et al. 2012), the set of metrics has been extended and tested in a real world ETL project. The authors identified three factors that can be used in order to predict a system vulnerability to DSCs. Those factors are: (1) *schema sizes*, (2) *functionality of an ETL activity*, and (3) *module-level design*. The first factor indicates that if the DS consists of tables with many columns then the ETL process is probably more vulnerable to changes. The second factor indicates that activities that depend on many columns and that output many columns are more vulnerable to changes. According to the last factor, activities that reduce a number of processed attributes should be placed early in the ETL process.

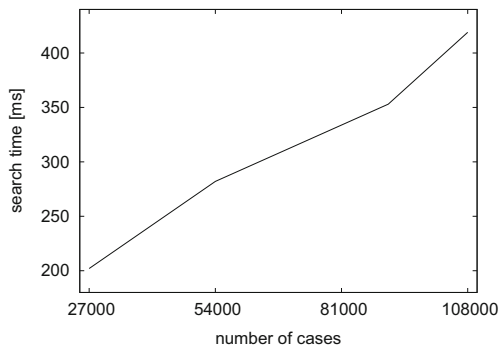

Fig. 10 Searching the best case for a variable number of cases in the LRC

The factors define only the probability that an ETL process will need repairing but the estimation of repair costs was not considered. Some changes (e.g. renaming a column) cause a simple repair of an ETL process and some others (e.g. column deletion) may cause changes of the business logic in the ETL process. The authors analyzed only 7 ETL processes that consist of 58 activities in total. In comparison to enterprise ETL projects it seems to be too small.

In (Wrembel and Bębel 2005) the authors proposed a prototype system that automatically detects changes in DSs and propagates them into a DW. The prototype allows to define changes that are to be detected and associates activities with the changes to be executed in a DW. The main limitation of the prototype is that it does not allow ETL workflows to evolve. Instead, it focuses on propagating DSs changes into a DW. Moreover, the presented solution is restricted to only relational databases. The next drawback is that a change detection mechanism depends on triggers, and as such, in practice it will not be allowed to be installed in a production database.

Detecting structural changes in data sources and propagating them into the ETL layer have not received much attention from the research community. One of the first solution of this problem is *Evolvable View Environment* (EVE) (Nica and Rundensteiner 1999; Rundensteiner et al. 2000; Rundensteiner et al. 1999). EVE is an environment that allows to detect an ETL workflow that is implemented by means of views. For every view, it is possible to specify which elements of the view 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.

Recent development in the field of evolving ETL workflows includes a framework called *Hecataeus* (Papastefanatos et al. 2010; Papastefanatos et al. 2009). In Hecataeus, all ETL activities and DSs are modeled as a graph whose nodes are relations, attributes, queries, conditions, views, functions, and ETL steps. Nodes are connected by 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 DS change event. In a response to an event, Hecataeus can either propagate the event, i.e. modify the graph according to a predefined policy, prompt an administrator, or block the event propagation.

In (Manousis et al. 2013; 2015), the authors proposed a method for the adaptation of *Evolving Data-Intensive*

Ecosystems (EDIE) built on top of *Hecataeus*. An ETL layer is example of such an ecosystem. The key idea of the method is to maintain alternative variants (old versions) of data sources and data flows. Data flow operations can be annotated with policies that instruct whether they should be adapted to an evolved data source or should use the old version of the data source.

Data sources are often external and independent systems (i.e., ERP, CRM or accounting system). An update of the external system often is required by a company needs or by changes in the law (i.e. changes in the tax law). In practice, it is difficult or impossible to maintain alternative variants of external systems for the purpose of using them in an ETL layer.

The aforementioned three approaches, i.e., *EVE*, *Hecataeus*, and *EDIE* are the most similar to *E-ETL*. Table 1 compares them with respect to the features that, in our opinion, are fundamental for managing ETL evolution.

The **manual effort** is a measure of a user work required to be done in order to use a given approach. The manual effort for *E-ETL* is low because it does not require setting up of rules by an ETL developer. In *EVE* and *EDIE*, a developer firstly has to decide for each element of an ETL process how it should behave in response to a DSC. To this end he/she has to set up rules for an element, a group of elements, or apply the set default rules. Even if a developer decided to apply only the default rules, he/she needs to consider if there exist exceptions for each element. In the *E-ETL* framework, there is also a possibility to define propagation rules (*Defined rules*), but these rules are optional. The default rules are sufficient to employ the *Defined rules* algorithm in order to reduce a case.

DSs may be related to each other. A change in a collection or a collection element can have an impact on other collection/collection element. Such changes in multiple DSs should be handled together, i.e. reparation algorithms should **support multiple DSCs**. *EVE*, *Hecataeus*, and *EDIE* manage each DSC separately. In *E-ETL*, if multiple DSCs are handled by the same modification of the ETL process, then all DSCs are contained in one case (the completeness constraint, cf. Section 4). Thus, *E-ETL* supports multiple DSCs.

An ETL processes often contains **User Defined Functions (UDFs) and complex activities**. Such elements have to be handled as black boxes without a knowledge about possible modifications. *EVE*, *Hecataeus*, and *EDIE* support UDFs and complex activities in a very limited scope. If cases in the LRC contain modifications of UDFs or complex activities then *E-ETL* is able to apply such modifications to an ETL process.

A repair of an ETL process often requires the modification of a **Business logic**. For example, an addition of a new column may require not only a propagation of a new attribute but also an addition of a new activity (e.g., an activity that cleans data read from the new column). If the LRC contains cases that include a modification of the Business clogic, then *E-ETL* can introduce such modification of a Business logic during the repair process. The competitors do not offer this functionality.

A DW is typically designed either as a star or a snowflake schema. The key concepts of these design patterns are facts an dimensions. ETL processes extract data from DSs, transform and load the data into a DW, either as facts or as dimensions. As an ETL task for processing facts differs from a task for processing dimensions, a repair of these different tasks should be handled differently. As a result, repair algorithms should support **different approaches to facts and dimensions**.

If a user manually repaired an ETL task for processing facts and dimensions in different ways, then appropriate cases were stored in the LRC. If the LRC contains these cases, then *E-ETL* supports different approaches to facts and dimensions. Whereas in *EVE*, *Hecataeus*, and *EDIE*, a user has to set different rules for facts and dimensions. Different rules for fact and dimensions increase the number or rules, i.e. increase the manual effort.

Table 1 The comparison of the *E-ETL* framework and similar approaches

Feature	*EVE*	*Hecataeus*	*EDIE*	*E-ETL*
Manual effort	high	high	high	low
Support for multiple DSCs	no	no	no	yes
UDFs and complex activities	no	no	no	yes
Business logic	no	no	no	yes
Different approach to facts and dimensions	possible	possible	possible	depends
Cooperation with external ETL tools	no	no	no	yes
Automatic	automatic	automatic	automatic	semi-automatic
Correctness	yes	yes	yes	no
Works without knowledge base	yes	yes	yes	no
Require alternative DS	no	no	yes	no

The *E-ETL* framework is designed from scratch to **cooperate with multiple external ETL tools** (currently with Microsoft SQL Server Integration Services via its API), whereas the competitors are standalone tools restricted only to ETL processes developed as sequences of SQL views.

The **correctness** of an ETL process repair means that an ETL process has been repaired in a proper way and it fulfills user requirements. A repair in *E-ETL* is based on repairs from the past. We cannot guarantee, that a new problem will be solved in the same way as a problem from the past. Therefore, we do not prove the correctness of an ETL process repair. The competitors prove the correctness of a process repair, according to rules defined by a user.

Since, *E-ETL* does not guarantee the correctness, the proposed repairs are presented to a user in order to be accepted or rejected. The competitors **automatically** repair an ETL process.

The *E-ETL* framework requires the LRC, which is a kind of a knowledge base. The more cases in the LRC, the higher is the probability that *E-ETL* will find a proper solution for a new problem. *EVE*, *Hecataeus*, and *EDIE* **work without any knowledge base**.

As mentioned before, only *EDIE* **requires the maintenance of alternative DS** versions.

10 Summary

A structural change in a data source often causes that an already deployed ETL workflow stops its execution with errors. In such a case, the workflow needs to be repaired. A manual repair is complex, time-consuming, and prone-to-fail. Since structural changes in DSs are frequent, an automatic or semi-automatic repair of an ETL process after such changes is of high practical importance.

Unfortunately, none of the commercial or open-source ETL tools existing on the market offers such a functionality. Moreover, the problem of repairing an ETL process received so far little attention from the research community (Manousis et al. 2013; Papastefanatos et al. 2009; Rundensteiner et al. 1999). The proposed solutions are limited to simple DS changes and require defining (for each change) explicit repair rules by a user.

This paper proposes a solution to handling evolving data sources in an ETL layer. To this end, we have developed the *E-ETL* framework for detecting structural changes in DSs and for repairing an ETL process accordingly. The framework repairs semi-automatically an ETL workflow using the repair algorithm. The repair method that we propose is based on Case-Based Reasoning. The main advantage of the presented repair algorithm is that it does not require repair rules. *E-ETL* learns with every case how to solve similar problems. If the Library of Repair Cases contains cases that include a modification of a Business logic, then *E-ETL* can introduce such a modification of a Business logic during the repair process.

This paper contributes: (1) the *Case Reduction Algorithm* for making cases more general, (2) a test case scenario, for the purpose of validating our approach, (3) an experimental evaluation our approach with respect to its performance, for 6 different factors influencing the performance.

As mentioned in Section 5.2.3, *E-ETL* computes the Factor of Similarity and Applicability (FSA) taking into consideration only structural properties of data sources and an ETL process. To this end, *E-ETL* analyzes only the structure of data sources (e.g. collection names, collection elements, data types). In the next release of the FSA computation algorithm we plan to exploit also semantics of the data sources and ETL processes, using for this purpose metadata describing these components and metadata on executions of ETL processes. In a yet further extension of the FSA computation algorithm we plan to exploit ontologies.

Open Access This article is distributed under the terms of the Creative Commons Attribution 4.0 International License (http://creativecommons.org/licenses/by/4.0/), which permits unrestricted use, distribution, and reproduction in any medium, provided you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons license, and indicate if changes were made.

References

Aamodt, A., & Plaza, E. (1994). Case-based reasoning: Foundational issues, methodological variations, and system approaches . *AI Comunications*, *7*(1), 39–59.

Alexe, B., Tan, W.C., & Velegrakis, Y. (2008). Stbenchmark: Towards a benchmark for mapping systems. *Proc. of Int. Conf. on Very Large Data Bases (VLDB)*, *1*(1), 230–244.

Council, T.P.P. (2016). TPC Benchmark DI (TPC-DI). http://www.tpc.org/tpcdi/default.asp. [Online; accessed 01-Feb-2016].

Curino, C., Moon, H.J., Tanca, L., & Zaniolo, C. (2008). Schema evolution in wikipedia - toward a web information system benchmark. In *Proc. of Int. Conf. on Enterprise Information Systems (ICEIS), pp. 323–332*.

Curino, C., Moon, H.J., Tanca, L., & Zaniolo, C. (2008). Schema evolution in wikipedia - toward a web information system benchmark. In *Proc. of Int. Conf. on Enterprise Information Systems (ICEIS), pp. 323–332*.

Hammond, K.J. (1990). Case-based planning: A framework for planning from experience. *Cognitive Science*, *14*(3), 385–443.

Kolodner, J.L. (1992). An introduction to case-based reasoning. *Artificial Intelligence Review*, *6*(1), 3–34.

Kolodner, J.L. (1993). *Case-Based Reasoning*: Morgan-Kaufmann Publishers, Inc.

Levenshtein, V. (1966). Binary codes capable of correcting deletions, insertions and reversals. *Sov. Phys. Dokl.*, *10*, 707–710.

Manousis, P., Vassiliadis, P., & Papastefanatos, G. (2013). Automating the Adaptation of Evolving Data-Intensive Ecosystems. In *Proc. of Int. Conf. on Conceptual Modeling (ER), LNCS, vol. 8217, pp. 182–196*.

Manousis, P., Vassiliadis, P., & Papastefanatos, G. (2015). Impact Analysis and Policy-Conforming Rewriting of Evolving Data-Intensive Ecosystems. *Journal on Data Semantics*, *4*(4), 231–267.

Moon, H.J., Curino, C.A., Deutsch, A., Hou, C., & Zaniolo, C. (2008). Managing and querying transaction-time databases under schema evolution. In *Proc. of Int. Conf. on Very Large Data Bases (VLDB), vol. 1, pp. 882–895*.

Nica, A., & Rundensteiner, E.A. (1999). View maintenance after view synchronization. In *Proc. of Int. Database Engineering and Application Symposium (IDEAS), pp. 215–213*.

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

Papastefanatos, G., Vassiliadis, P., Simitsis, A., & Vassiliou, Y. (2008). Design Metrics for Data Warehouse Evolution. In *Proc. of Int. Conf. on Conceptual Modeling (ER), LNCS, vol. 5231, pp. 440–454*.

Papastefanatos, G., Vassiliadis, P., Simitsis, A., & Vassiliou, Y. (2009). Policy-Regulated Management of ETL Evolution. *J. Data Semantics*, *5530*, 147–177.

Papastefanatos, G., Vassiliadis, P., Simitsis, A., & Vassiliou, Y. (2012). Metrics for the prediction of evolution impact in etl ecosystems: A case study. *J. Data Semantics*, *1*(2), 75–97.

Qiu, D., Li, B., & Su, Z. (2013). An empirical analysis of the co-evolution of schema and code in database applications. *Proc. of Joint Meeting on Foundations of Software Engineering*, *125–135*.

Rundensteiner, E.A., Koeller, A., & Zhang, X. (2000). Maintaining data warehouses over changing information sources. *Communications of the ACM*, *43*(6), 57–62.

Rundensteiner, E.A., Koeller, A., Zhang, X., Lee, A.J., Nica, A., Van Wyk, A., & Lee, Y. (1999). 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*.

Schank, R.C. (1983). *Dynamic Memory: A Theory of Reminding and Learning in Computers and People*.

Sjøberg, D. (1993). Quantifying schema evolution. *Information and Software Technology*, *35*(1), 35–54.

Skoulis, I., Vassiliadis, P., & Zarras, A.V. (2014). Open-source databases: Within, outside, or beyond lehman's laws of software evolution?. In *Proc. of Int. Conf. on Advanced Information Systems Engineering (CAiSE), pp. 379–393*.

Vassiliadis, P., Zarras, A.V., & Skoulis, I. (2015). How is Life for a Table in an Evolving Relational Schema? Birth, Death and Everything in Between. In *Proc. of Int. Conf. on Conceptual Modeling (ER), pp. 453–466*.

Watson, I., & Marir, F. (1994). Case-based reasoning: A review. *The knowledge engineering review*, *9*(4), 327–354.

Wojciechowski, A. (2011). E-ETL: Framework for managing evolving ETL processes. In *Proc. of ACM Information and Knowledge Management Workshop (PIKM), pp. 59–66*.

Wojciechowski, A. (2013). E-ETL: Framework for managing evolving ETL processes. In *Proc. of Conf. Advances in Databases and Information Systems Workshops (ADBIS), pp. 441–449*.

Wojciechowski, A. (2013). E-ETL: Framework for managing evolving ETL processes. *Foundations of Computing and Decision Sciences*, *38*(2), 131–142.

Wojciechowski, A. (2015). E-ETL Framework: ETL process reparation algorithms using Case-Based Reasoning. In *Proc. of Conf. Advances in Databases and Information Systems Workshops (ADBIS), pp. 321–333*.

Wojciechowski, A. (2016). On handling evolution of ETL layer by means of Case-Based Reasoning. Tech. Rep. RA-8/16, Poznan University of Technology, Institute of Computing Science. http://calypso.cs.put.poznan.pl/projects/e-etl/ifip2016.pdf.

Wrembel, R. (2009). A survey on managing the evolution of data warehouses. *International Journal of Data Warehousing & Mining*, *5*(2), 24–56.

Wrembel, R., & Bębel, B. (2005). The Framework for Detecting and Propagating Changes from Data Sources Structure into a Data Warehouse. *Foundations of Computing & Decision Sciences*, *30*(4), 361–372.

Wrembel, R., & Bębel, B. (2007). Metadata management in a multiversion data warehouse. *Journal on Data Semantics*, *8*, 118–157. LNCS 4380.

Wu, S., & Neamtiu, I. (2011). Schema evolution analysis for embedded databases. In *Proc. of Int. Conf. on Data Engineering (ICDE), pp. 151–156*.

Artur Wojciechowski is a Ph.D. student in the Faculty of Computing, at Poznan University of Technology, in Poland. His main research focuses on data warehouse systems, ETL tools and data processing technologies. He received a Bachelor and Master of Science degree from Poznan University of Technology. He works as a software designer and developer.