MAIME - Maintenance Manager for ETL

Darius̆ Butkevičius, Philipp D. Freiberger, Frederik M. Halberg, Jacob B. Hansen, Søren Jensen, and Michael Tarp

Dept. of Computer Science, Aalborg University, Denmark
{dbutke16, pfreib15, fhalbe12, jh12, sajens12, mtarp09}@student.aau.dk

ABSTRACT

Maintaining ETL (Extract, Transform, and Load) processes is a time-consuming and error-prone task. External Data Sources (EDSs) often change their schema which potentially leaves the ETL processes that extract data from those EDSs invalid. Repairing these ETL processes is time-consuming and tedious, therefore this paper examines the idea of introducing automation into the process of ETL process maintenance.

We propose MAIME as a tool to semi-automatically maintain ETL processes. MAIME is made to work with SQL Server Integration Services (SSIS) and uses a graph model which serves as a layer of abstraction on top of SSIS Data Flow tasks (ETL processes). We introduce a graph alteration algorithm which propagates detected EDS schema changes through the graph. Modifications done to a graph are directly applied to an ETL process. The maintenance process of MAIME can be configured by an administrator so that different types of EDS schema changes and SSIS transformations are handled differently.

MAIME is shown to be able to maintain a subset of transformations for SSIS Data Flow tasks automatically. The evaluation involved both maintaining an ETL process with MAIME and doing it manually in the SSIS Designer and comparing the results of the two. This showed that for maintaining ETL processes, the amount of user inputs was decreased by a factor of 9.5 and the time spent was reduced by a factor of 9.8.

Keywords

Business Intelligence, Data Warehouse, ETL process, ETL maintenance, EDS schema change, SSIS

1. INTRODUCTION

Business Intelligence (BI) is an essential set of techniques and tools for a business to provide analytics and reporting in order to give decision support. The BI techniques and tools often use a data warehouse (DW) containing the data for the decision making of a business. The term data warehousing refers to the process of building, populating, and utilizing a DW in business decisions [1].

A DW usually contains transformed data from one or more External Data Sources (EDSs). In order to populate a DW, a process involving three steps is used: Extract, Transform, and Load (ETL). An ETL process extracts data from one or more EDSs, transforms the data into a desirable format by cleansing it (i.e., correcting or removing corrupt/inaccurate data), deriving new values, etc., and finally load it into a DW.

Constructing ETL processes can be a comprehensive task, as it consists of several parts: (1) understanding the structure and contents of the EDSs, (2) designing and implementing the data flow from sources (EDSs) to destinations (DWs) and performing complex transformations on the data, and (3) handling any errors such as missing values or data validation.

While the construction of ETL processes is a well-known phenomenon, maintaining ETL processes after construction has received less academic attention. We worked with a consultant from a major bank in Denmark who detailed that the most time-consuming task when working with ETL resides in maintaining ETL processes. This maintenance process is mostly done in reaction to EDS schema changes, which according to the consultant needs to be resolved for hundreds of tables for the quarterly releases. EDS schema changes include: Additions, deletions, renaming, and other modifications (e.g., data type changes, table splitting, table merging) on databases, tables, and columns. An EDS schema change can render multiple ETL processes invalid and require manual work to repair. This is a rather tedious task that can be very time-consuming if it has to be repeated for multiple ETL processes. EDS schema changes can be nontrivial to resolve, as transformations of an ETL process depend upon each other. Thus even a small EDS schema change can affect several parts of an ETL process. The impact of an EDS schema change depends on both the type of the EDS schema change and how much an ETL process uses the changed EDSs.

We wish to work with the problem of maintaining ETL pro-

cesses in reaction to EDS schema changes. To summarize, maintaining ETL processes requires manual work, can be very time consuming, and can be quite error-prone. To remedy these problems we propose the tool MAIME which is capable of: (1) detecting schema changes in EDSs and (2) semi-automatically adjusting affected ETL processes.
MAIME works with SQL Server Integration Services (SSIS) [2] and supports Microsoft SQL Server [3]. SSIS is chosen as the ETL platform since it is in the top three of the most used tools by businesses for data integration [4]. SSIS also has an easy to use graphical tool (SSIS Designer [5]) and includes an API, which allows access to modify ETL processes through third party programs like MAIME. To maintain ETL processes, we formalize and implement a graph model as a layer of abstraction on top of SSIS Data Flow tasks. By doing so, we only have to modify the graph which then automatically modifies the corresponding SSIS Data Flow task.
Running ETL processes is often an extensive and time-consuming task, and a single ETL process not being able to execute could cause a considerable amount of time wasted since the administrator would have to repair the ETL process and run the ETL processes once again. MAIME is based on this concept and wants to repair ETL processes at all costs, this includes deleting transformations of the ETL processes if needed.

Achievements: MAIME was shown to be able to successfully repair ETL processes in response to EDS schema changes. For the implemented set of transformations, a comparison was made between resolving EDS schema changes in MAIME and doing it manually in the SSIS Designer tool. The evaluation shows that MAIME is on average 9.8 times faster and required 9.5 times less input from the user to maintain.

Paper Organization: Section 2 provides an overview of SSIS, how we detect EDS schema changes, and the architecture of MAIME. Section 3 formalizes the graph model of MAIME and describes its usage. Section 4 describes important details of the implementation. Section 5 shows an evaluation of how much easier it is to maintain ETL processes when using MAIME rather than doing it manually. Section 6 covers related work. Section 7 concludes on the paper and provides directions for future work. Finally, we present acknowledgements and provide the appendix.

2. OVERVIEW

In this section, we first give an overview of the SSIS components covered by our solution. Afterwards, we explain how we detect EDS schema changes and which options we have considered. Finally, we give a description of the architecture of MAIME.

2.1 Overview of SSIS Components

This section provides an overview of the internal details of SSIS and a list of transformations we include in MAIME.
Knight et al. [2] provide an overview of the SSIS architecture. To briefly summarize, a *Package* is a core component of SSIS which connects all of its tasks together. The package is stored in a file which the SSIS engine can load and execute the contained Control Flows. The *Control Flow* of a package controls the order and execution of its contained tasks. Several types of tasks exist for a Control Flow (e.g. *Execute SQL task* and *Script task*), but this paper only focuses on *Data Flow tasks*. A Data Flow task can extract data from one or more sources, transform the data, and load it into one or more destinations. It can be observed that the actions performed in a Data Flow task resemble those of an ETL process. We therefore regard a Data Flow task as an ETL process.
A Data Flow task can make use of several types of *transformations*, referred to as *data flow components* in SSIS. For this paper, the scope has been reduced to only consider transformations we deem to be some of the most commonly used transformations in SSIS. The graphical tool that Microsoft provides for constructing SSIS packages categorizes a set of transformations as being common. MAIME covers a subset of these common transformations, which have been chosen based on our experience with SSIS. The transformations chosen are: *Aggregation*, *Conditional Split*, *Data Conversion*, *Derived Column*, *Lookup*, *Sort*, and *Union All*. To extract and load data we use *OLE DB Source* and *OLE DB Destination*, respectively. For convenience, we extend the term transformations to include OLE DB Source and OLE DB Destination even though they do not transform data.

2.2 EDS Schema Change Detection

The first problem of maintaining ETL processes is to detect EDS schema changes. To do this we present three different approaches along with their strengths and weaknesses. In the end of this section, we present the chosen approach and explain why this has been chosen. Currently, MAIME is limited to only work with Microsoft SQL Server. An obvious extension would be to include additional database management systems. We therefore also take into account approaches that are extensible.

2.2.1 Third Party Tools

There currently exist a lot of tools available for managing database changes, for example: RedGate [6], DB Comparer [7], and Visual Studio's Schema Comparison tool [8]. Besides some of the tools not being free, almost all the tools we looked at were standalone tools not meant to be integrated into another solution.
Strengths: Comparison is delegated to a (presumably) well tested tool.
Weaknesses: RedGate is a rather expensive tool for this project. DB-Comparer is outdated as the last supported version was Microsoft SQL Server 2008. Visual Studio's Schema Comparison tool does not provide an API.

2.2.2 Data Definition Language Triggers

Data Definition Language (DDL) triggers [9] are primarily used in response to an SQL statement beginning with keywords such as: CREATE, ALTER, or DROP. We can apply these triggers to a database and thereby record any schema change that has occurred to the database, table, or column. This alone is not enough to maintain an ETL process. For a column we need to know which state (name, data type, etc.) the column was in when the ETL process was created. We therefore need either a snapshot of the state of the column when the column was used in an ETL process or an indicator of where in the list of records the column was used and then look at all future modifications from that indicator. The same principle applies to databases and tables.

Strengths: We can receive real-time updates of schema changes. Triggers can be customized to only react to specific types of changes in the schema.
Weaknesses: Triggers might produce some overhead by notifying about every change right away. In addition to constructing DDL triggers, we also need to store snapshots or infer the previous state of a database, table, or column based on the current state and modifications done to it after the last time it was used in an ETL process.

2.2.3 *Information Schema Views*

Information schema views [10] can be used to obtain metadata about databases, tables, and columns. By extracting and storing metadata snapshots of databases we are able to compare different states of the database schemas to get an overview of which changes have been made since the last available snapshot. By having this comparison, we can then apply these changes to the ETL process. The following is an example of how the information schema views can be used to extract metadata:
"`SELECT * FROM AdventureWorks2014.INFORMATION_SCHEMA.Columns`"
Which among other things provides information for all columns and their data types in AdventureWorks2014.
Strengths: It is easy to setup the capturing process at compile time as no additional setup by the database administrator is required. It can be customized which data is captured.
Weaknesses: Information schema views differ between database vendors, so it would be necessary to implement a solution for every supported database.

2.2.4 *Chosen Method*

We chose to use the information schema views mainly since it was the easiest approach as we chose to tailor MAIME only for Microsoft SQL Server. DDL triggers require setup by a database administrator, which is undesirable if it can be avoided. Compared to using the DDL trigger approach, we also avoid overhead produced by storing every change. For information schema views, we only need to compare the current version with the version used during the last execution of MAIME. We could have used a third party tool if we had found one that fit our needs. Thus information schema views best fit MAIME's needs by storing exactly the information which is needed to detect schema changes.
A description of how the chosen method is implemented is provided in Section 4.2.

2.3 Architecture

As shown in Figure 1, MAIME consists of three components: *EDS Change Manager* (ECM), *Maintenance Manager* (MM), and a *GUI*. First we introduce the context of MAIME, namely EDS, Metadata Store, Log, and Configuration, followed by a brief description of the three components. Finally, a general use case of MAIME is introduced.

2.3.1 *Context of MAIME*

First we provide short descriptions for MAIME's surrounding elements that are necessary to describe the ECM, the MM, and the GUI components.
EDSs correspond to the external data sources of which schema changes are detected. In the figure the *Metadata Store* corresponds to a local directory. Each captured EDS metadata snapshot is stored as a JSON file in the Metadata Store. An example of a snapshot is shown in the Appendix C.

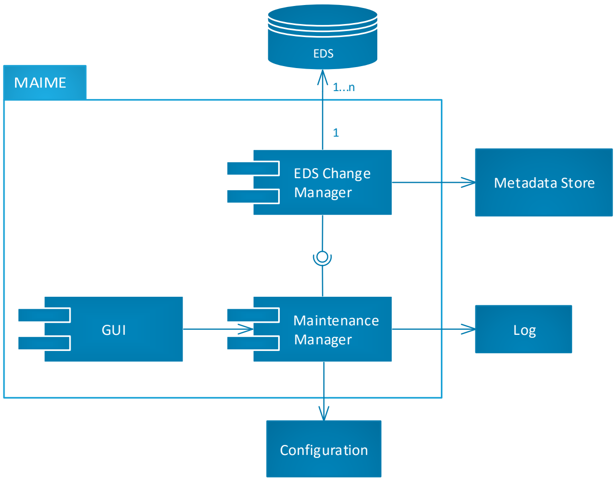


Figure 1: Architecture of MAIME.

The *Log* is a local directory which stores every change that is applied by the MM so that an administrator is able to analyze the maintenance of ETL processes. Log entries are stored in separate files grouped by date.
The *Configuration* is a JSON file which stores the configurations that are defined by an administrator before executing maintenance of ETL processes. The administrator configurations are further described in Section 3.2.

2.3.2 *EDS Change Manager*

The ECM is able to capture metadata of multiple EDSs by making use of SQL Server's information schema views. EDS metadata snapshots are captured and stored in the Metadata Store. Before repairing an ETL process, the current version of the EDS metadata snapshot is compared with the latest metadata snapshot that is stored in the Metadata Store. As a result, a list of EDS schema changes is produced which is accessible to the MM through an interface as we can see in Figure 1. The structure of an EDS metadata snapshot and more details of the ECM are presented in Section 4.2.

2.3.3 *GUI*

The GUI is accessed by an administrator to maintain ETL processes. It involves: (1) Selecting ETL processes to maintain, (2) adjusting administrator configurations, (3) confirming or denying changes by answering to prompts, (4) visualizing applied changes to the ETL processes, and (5) displaying Log entries for the applied changes.

2.3.4 *Maintenance Manager*

The core logic of MAIME resides in the MM. Therefore, the MM requires a more detailed overview than other components. This overview is shown in Figure 2. The *ETL Updater* is responsible for loading the ETL processes and creating corresponding graphs. One *Graph* is responsible for transitively updating one underlying ETL process. The ETL Updater contains multiple Graphs and operates on them when the graph alteration algorithm, which is explained in Section 3.3, is called. The graph alteration algorithm is called for each EDS schema change that is provided by the Metadata Provider.

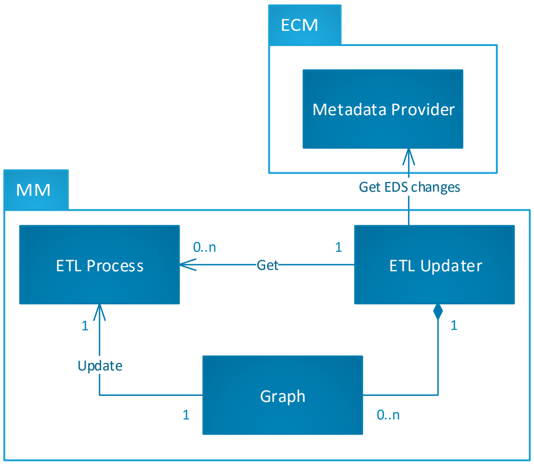


Figure 2: Detailed view of MM.

Modifications made during the graph alteration algorithm to the graphs are done semi-automatically. How exactly this is done depends on how the administrator defines the configurations which are described in Section 3.2.

2.3.5 *Use Case of ETL Processes Maintenance*

Figure 3 depicts a basic use case of maintaining ETL processes with MAIME. The prerequisite of this scenario is to have metadata snapshots stored in the Metadata Store for the chosen EDSs. An administrator uses the GUI to adjust administrator configurations and select ETL processes (1-5). Then the administrator is able to initiate the maintenance of the ETL processes (6). The GUI delegates maintenance of ETL processes to the MM by passing the directory containing the ETL processes (7). The MM loads ETL processes from the provided directory and creates corresponding graphs for each ETL process (8). Then the MM deduces EDSs from MAIME graphs by finding all the distinct sources used in all the vertices corresponding to OLE DB Source and Lookup transformations (9). As a last preparation action, it gets latest EDS schema changes (10, 11). The reparation involves an iteration over all the graphs and calling the graph alteration algorithm for each retrieved EDS schema change (12). The algorithm is described in Section 3.3. After all the ETL processes are updated, the ETL processes are saved into new files (13) and the administrator can see MAIME's graphs with changes applied to the ETL processes (14, 15).

In a more extensive scenario the GUI can also be used to prompt the administrator if modifications need to be confirmed. In this case the graph alteration algorithm execution (12), which is shown in Figure 3, is interrupted by a call to the GUI, which in turn prompts the administrator to confirm a change to an ETL process.

3. GRAPH MODEL

To analyze ETL processes and the effects of EDS schema changes, the ETL processes are modeled using a graph structure. As mentioned before, the purpose of this graph is to create a layer on top of SSIS which provides a level of abstraction. This means that whenever we make a change in our graph, corresponding changes are also applied to the underlying SSIS package. Afterwards, we save the SSIS package which is now in an executable state. By formally defining our graph model as the basis of our implementation, we can present an easier structure compared to working directly with internal details of a package to perform adjustments according to EDS schema changes. The structure of our graph is guided by the structure of SSIS Data Flow tasks.

An advantage of using graphs for modeling an ETL process lies in the capability of easily representing dependencies between columns for all transformations and handling cascading changes using graph traversal.

3.1 Formal Definition of Graph Model

Each ETL process is represented as an acyclic *property graph* $G = (V, E)$ where a vertex $v \in V$ represents a transformation and an edge $e \in E$ contains the columns that are transferred from one vertex to another. A property graph is a labeled, attributed, directed multi-graph [11]. Each vertex and edge can have multiple properties. A *property* is a key-value pair describing what a vertex or edge represents. A multi-graph allows for multiple edges between the same two vertices. Currently, this is not implemented. To access a given property, e.g., *name* of vertex v_1, we use the following dot notation: $v_1.name$.

The set C is said to be all columns that are used in an ETL process. A column in the graph represents a column in an EDS, or a column created during the ETL process. Each *column* $c \in C$ is a 3-tuple $(id, name, type)$ where *id* is a unique number identifying the column throughout the entire ETL process. *id* was included because *name* and *type* are not enough to uniquely identify a column, as two columns can have the same name and type in the same ETL process. *name* is the name of the column, and *type* is the data type of the column. For simplicity sake, we here regard the data type of a column as simply being the type of the column in SSIS such as boolean, currency, float, etc. In Section 4.2.1, we describe what other properties of a column we use such as IsNullable, IsUnique, etc. These other properties are not included in our model nor used in the implementation yet. A complete list of supported data types can be found in [12].

Each edge $e \in E$ is a 3-tuple $(v_1, v_2, columns)$ where the edge represents a connection between two vertices and $columns \subseteq C$ is the set of the columns being transferred from a vertex v_1 to another vertex v_2. Putting columns on the edge is particularly advantageous for transformations such as Aggregate which can have multiple outgoing edges, where each edge can transfer a different set of columns.

A vertex v represents a transformation with a list of properties which depends on the type of transformation. The only properties that all vertices have in common regardless of their transformation type are: *name*, *type*, and *dependencies*. The only exception is OLE DB Destination which does not have the *dependencies* property, which is explained in Section 3.1.2. *name* is a unique name used to identify v. *type* denotes which kind of transformation v represents (e.g. Conditional Split or Aggregate).

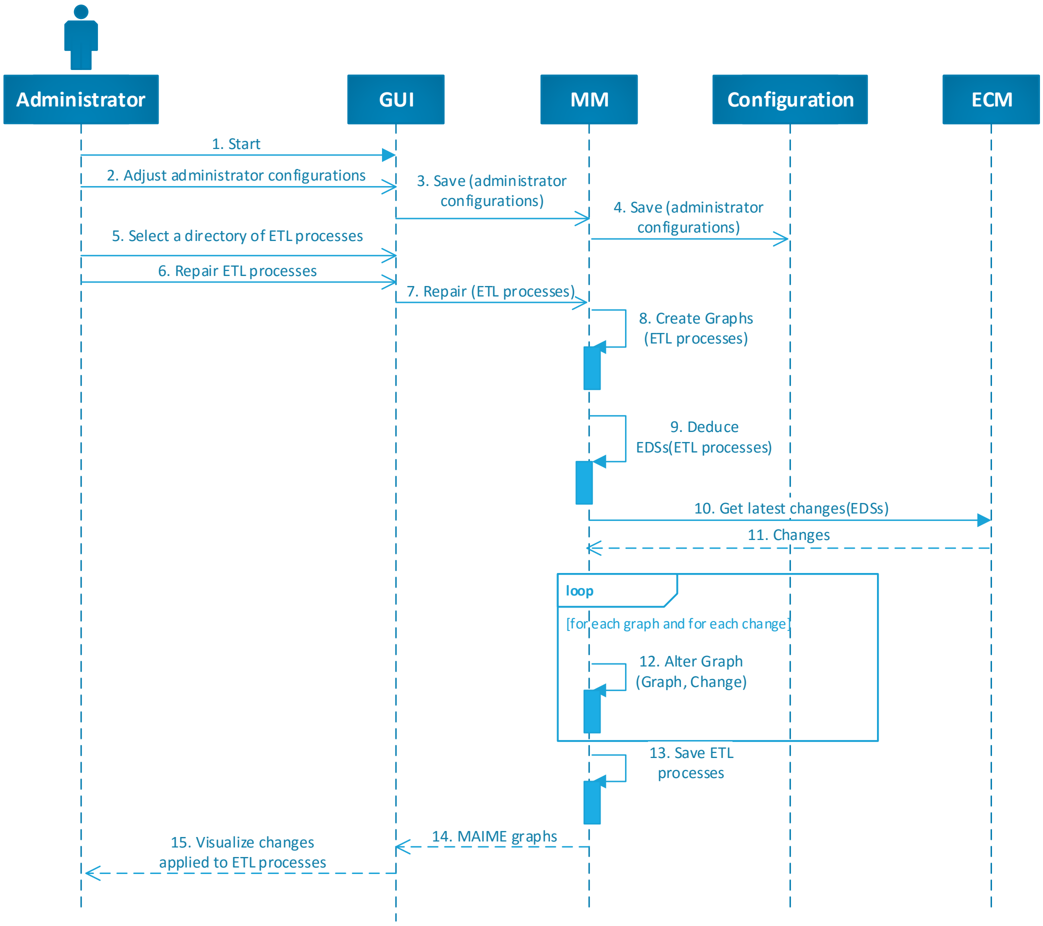

Administrator
GUI
MM
Configuration
ECM
1. Start
2. Adjust administrator configurations
3. Save (administrator configurations)
4. Save (administrator configurations)
5. Select a directory of ETL processes
6. Repair ETL processes
7. Repair (ETL processes)
8. Create Graphs (ETL processes)
9. Deduce EDSs(ETL processes)
10. Get latest changes(EDSs)
11. Changes
loop
[for each graph and for each change]
12. Alter Graph (Graph, Change)
13. Save ETL processes
14. MAIME graphs
15. Visualize changes applied to ETL processes

Figure 3: Basic use case of MAIME.

dependencies shows how columns depend on each other. If for example we extract a column c from an EDS and have an Aggregate transformation that takes the average of c and outputs c' we say that $c' \mapsto \{c\}$. In the case of the Aggregate transformation, c' originates from c, and is therefore dependent on c. Any modifications such as column deletion or column data type change to c can result in a similar change to c'. Formally, *dependencies* is a mapping from an *output column* $o \in C$ to a set of *input columns* $i = \{c_1, \ldots, c_n\} \subseteq C$. We say that o is dependent on $\{c_1, \ldots, c_n\}$ and denote this as: $o \mapsto \{c_1, \ldots, c_n\}$. The output columns are defined as the columns that a vertex sends to another vertex through an outgoing edge, such as an OLE DB Source transferring columns to a Conditional Split. The input columns are defined as the columns that a vertex receives from another vertex through an ingoing edge, such as an Aggregate receiving columns from an OLE DB Source.

The main purpose of dependencies is to detect whether an EDS schema change has any cascading effects. We therefore need to know which parts of the graph are affected and have to be modified. Due to this, the graph contains some trivial dependencies if a transformation does not affect a column, e.g., a dependency such as: $c \mapsto \{c\}$ where c is dependent on itself. This is referred to as the column being dependent on itself. An example of an output column depending on multiple input columns could be a Derived Column transformation, where a derived output column o is dependent on the input columns i_1 and i_2 if they were used to derive the value of o. Thus $o \mapsto \{i_1, i_2\}$ which shows that if an EDS schema change affects i_1 or i_2 then it might also affect o.

Additional properties of a vertex are defined by the type of the vertex. For instance, *v.conditions* is a necessary property for *Conditional Split*, whereas *Union All* does not involve conditions, and thus does not have this property. Table 1 shows all properties specific to each transformation, number of ingoing, and outgoing edges.

The following sections describes the definitions of properties in more detail and how *dependencies* is specified for each type of transformation.

3.1.1 OLE DB Source

An OLE DB Source is used to extract data from a table or view of an EDS. An OLE DB Source represents an entry point of the graph, providing data for upcoming transformations, i.e. a vertex with no ingoing edges.

The additional properties of an OLE DB Source are:

database is the name of the database being extracted from.

Table 1: Table of transformations, their properties, and number of allowed ingoing and outgoing edges

Transformation	Specific properties	Ingoing edges	Outgoing edges
OLE DB Source	`database`, `table`, and `columns`	0	1
OLE DB Destination	`database`, `table`, and `columns`	1	0
Aggregate	`aggregations`	1	1+
Conditional split	`conditions`	1	1+
Data conversion	`conversions`	1	1
Derived column	`derivations`	1	1
Lookup	`database`, `table`, `joins`, `columns`, and `outputcolumns`	1	2
Sort	`sortings` and `passthrough`	1	1
Union all	`inputedges` and `unions`	1+	1

table is the name of the table being extracted from. *columns* is the list of columns extracted from the table.
dependencies for OLE DB Source is trivial, as for each column $c \in columns$, $c \mapsto \emptyset$. Each column is dependent on nothing in the OLE DB Source, as this is the first time it appears in the graph.

3.1.2 OLE DB Destination

An OLE DB Destination represents an ending point in the graph and is responsible for loading data into a DW.
OLE DB Destination has the same properties as OLE DB Source (which is *database*, *table*, and *columns*), where the database name and table name represents the location where *columns* are loaded to.
Since OLE DB Destination does not have any outgoing edges, *dependencies* does not exist for OLE DB Destination.

3.1.3 Aggregate

An Aggregate applies aggregate functions such as GROUP BY, SUM, or AVERAGE to the values of columns, which results in new outputs with the aggregated values.
Formally, the only property specific to the Aggregate vertex is defined as:
$aggregations = \{(f_1, input_1, output_1, dest_1), \ldots, (f_n, input_n, output_n, dest_n)\}$ where $f_i \in \{SUM, COUNT, COUNT\ DISTINCT, GROUP\ BY, AVG, MIN, MAX\}$ is the aggregate function used, $input_i \in C$ is the input column that the i-th output column is computed from, and $output_i \in C$ is the result of the i-th aggregation and output of aggregate i, respectively. $dest_i \in V$ is the vertex receiving the output column $output_i$. A given destination $dest$ can appear in multiple tuples of $aggregations$, which shows that the $dest$ receives multiple columns through a single edge.
For each tuple i in $aggregations$, we define the dependency: $output_i \mapsto \{input_i\}$.

3.1.4 Conditional Split

A Conditional Split applies expressions to rows and can thereby route them to different destinations based on what conditions they satisfy. Expressions are for example: $SUBSTRING(title, 1, 3) == "Mr."$ or $salary > 30000$.
Formally, the only property specific to the Conditional Split vertex is defined as:
$conditions = \{(expr_1, p_1, dest_1), \ldots, (expr_n, p_n, dest_n)\}$ where $expr_i$ is a predicate that specifies what rows are routed to $dest_i$. $p_i \in \mathbb{N}^+$ is a priority which indicates in which order the conditions are evaluated in (where 1 is the highest priority). The set of priorities $\{p_1, \ldots, p_n\}$ is a gap-free series of numbers that range from 1 to n, where n is the number of conditions. $dest_i \in V$ is the vertex that receives rows (that have not been taken by a higher priority condition) for which $expr_i$ holds.
In Conditional Split the set of output columns on each outgoing edge is equal to the set of input columns. This is because a Conditional Split transformation is not able to add, delete, or otherwise change columns, but is only able to filter rows. Thus, each column depends on itself.

3.1.5 Data Conversion

A Data Conversion converts the data type of a column and creates a new column with the new data type.
Formally, the only specific property of the data conversion vertex is defined as:
$conversions = \{(input_1, output_1), \ldots, (input_n, output_n)\}$ where $input_i \in C$ is the input column of the vertex and $output_i \in C$ is a new column that has been created through the Data Conversion from $input_i$.
Each of the tuples in $conversions$ describes that $output_i \mapsto \{input_i\}$. The specific conversions of data types are present in the type of $output_i$. Despite its name, a Data Conversion does not convert a column, but rather creates a new column rather than replacing the existing column such that both $input_i$ and $output_i$ are output columns.
dependencies is simple for Data Conversion, as a dependency is made for each new $output_i$ such that $output_i \mapsto \{input_i\}$ for all tuples in $conversions$ and since all input columns are sent to next vertex, they each depend on themselves.

3.1.6 Derived Column

A Derived Column creates new columns or replaces existing input columns based on input columns and expressions applied to those inputs.
Formally, the only property specific to the Derived Column vertex is defined as:
$derivations = \{(expr_1, output_1), \ldots, (expr_n, output_n)\}$ where $expr_i$ is an expression used for computing the new values of $output_i$, and $output_i \in C$ is a newly created column.
dependencies for Derived Column is similar to Data Conversion in that each input column is dependent on itself. For all output columns in *derivation* we define the dependency $output_i \mapsto \{c_{i,1}, \ldots, c_{i,j}\}$ where $\{c_{i,1}, \ldots, c_{i,j}\}$ is the set of all columns used in $expr_i$.

3.1.7 Lookup

A Lookup extracts additional data from a database by joining with given input columns.
Formally, the properties of a Lookup vertex are defined as:

$database$ is the name of the database we lookup additional columns from. $table$ is the name of the table being looked up. $columns \subseteq C$ represents all columns in the table being looked up. $outputcolumns \subseteq columns$ is the columns extracted from the table.
$joins = \{(input_1, lookup_1), \ldots, (input_n, lookup_n)\}$ where $joins$ is derived from an equi-join of the Lookup. $input_i$ is a column from the preceding vertex used for the join condition and $lookup_i$ is a column from $columns$, i.e., $join_i \in joins$ defines an equality between two columns $input_i$ and $lookup_i$ and the condition is of the form $input_i = lookup_i$. The equi-join is used to extract the columns $outputcolumns$.
Each input column has a dependency to itself. Each new column in $outputcolumns$ is dependent on all the input columns used in the join conditions of the Lookup. In other words, for each output column $output \in outputcolumns$ we have that $output \mapsto \{input_1, \ldots, input_n\}$ where $input_i$ is the input column of the i-th tuple in $joins$.

3.1.8 Sort

A Sort sorts the input rows in either ascending or descending order and creates new output columns for the sorted rows. It is possible to sort on multiple columns where a certain priority has to be given.
Formally, the properties of a Sort vertex are defined as:
$sortings = \{(input_1, output_1, sorttype_1, order_1), \ldots (input_n, output_n, sorttype_n, order_n)\}$ where $input_i \in C$, $output_i \in C$, $sorttype_i \in \{ascending, descending\}$, and $order_i \in \mathbb{N}^+$ indicates in which order the columns are sorted (where 1 is the highest priority). The set of orders $\{order_1, \ldots, order_n\}$ is a gap-free series of numbers that range from 1 to n, where n is the number of tuples in $sortings$
$passthrough = \{c_1, \ldots, c_j\}$ is the set of columns passed through the transformation. This does not include columns that are used for sorting.
For dependencies, each column $c \in passthrough$ is dependent on itself.

3.1.9 Union All

A Union All combines rows from multiple vertices.
Formally, the properties of a Union All vertex are defined as:
$inputedges = (e_1, \ldots, e_j)$, where e_i is the i-th ingoing edge of the Union All, and j is the amount of ingoing edges.
$unions = \{(output_1, input_1), \ldots, (output_n, input_n)\}$ where $input_i = (c_1, \ldots, c_j)$. $c_i \in C \cup \epsilon$ where ϵ is a convention used to indicate that for a given union, no input is taken from the corresponding edge. For example, the tuple $(output_i, input_i)$ where $input_i = (c_1, \epsilon, c_3)$ indicates that the unioned $output_i$ uses c_1 from e_1, c_3 from e_3, but no column is used from e_2. $output_i$ is the column generated by unioning the columns in the i-th $input$ tuple. A column from a table can only be part of a single union.
$dependencies$ for Union All only makes use of the $output$ columns from $unions$ as no columns are passed through Union All. For the i-th tuple of $unions$, $output_i \mapsto \{c_1, \ldots, c_n\}$ such that the new column, which is a result of the union, is dependent on all columns that was used for the union. This dependency is derived for all tuples in $unions$.

3.1.10 Graph Example

This section gives an illustration of how ETL processes are structured through our graph model. An example of an ETL process that is illustrated in MAIME is shown in Figure 4 and the corresponding schema diagram of an EDS is shown in Figure 5. Figure 4 shows a simplified structure that for example does not include the type of the vertex. This is not shown since it is obvious from the name of the vertex.
The example itself consists of an OLE DB Source, Lookup, Derived Column, and OLE DB Destination, which is represented by the four vertices: 1, 2, 3, and 4 respectively. Each box connected to a vertex contains the properties of that vertex and each box connected to an edge contains the properties of that edge. The OLE DB Source extracts the columns: ID, $Name$, and Age from a local database named `DBTesting` from the table `Person`. Lookup extracts $TotalAmount$ from the `Sale` table by joining ID from the `Person` table with $CustomerID$ from the `Sale` table. Derived Column derives the new column $AmountTimes10$ from $TotalAmount$ with the derivation $TotalAmount * 10$. Finally, the OLE DB Destination loads the columns: ID, $Name$, Age, and $AmountTimes10$ into the DW.
As an example for a dependency we have $AmountTimes10 \mapsto \{TotalAmount\}$ in Derived Column, this dependency originates from the derivation of $TotalAmount*10$ which involves the column $TotalAmount$. This can be used to indicate that any change to $TotalAmount$ can affect $AmountTimes10$, e.g., if $TotalAmount$ is deleted, $AmountTimes10$ can no longer be derived and therefore also needs to be deleted.

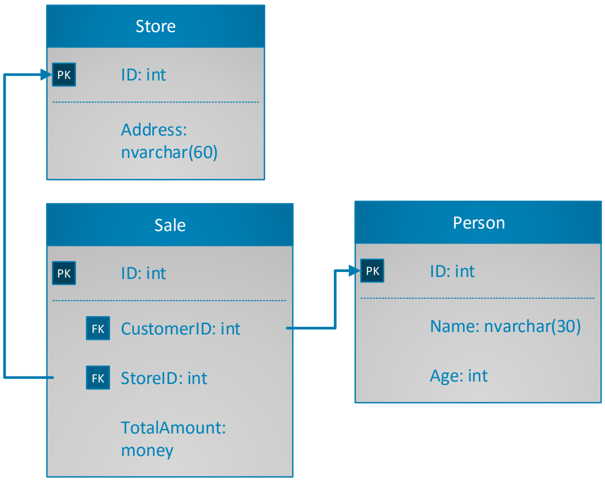


Figure 5: Schema diagram for the EDS used in Figure 4.

3.2 Administrator Configurations

This section describes how an administrator can configure which modifications are allowed to be made throughout the graph in case of EDS schema changes. An administrator can configure what actions are allowed to be performed in an ETL process when a specific EDS schema change happens. As an example, this means that an administrator can define that only renames should be done automatically, and the program should block all other modifications. If the program is given free reign, however, it can make any number of modifications to ensure that ETL processes execute successfully.
In the following section we describe the *configurations* that an administrator can specify before executing MAIME.
There are two types of configurations: (1) *EDS schema*

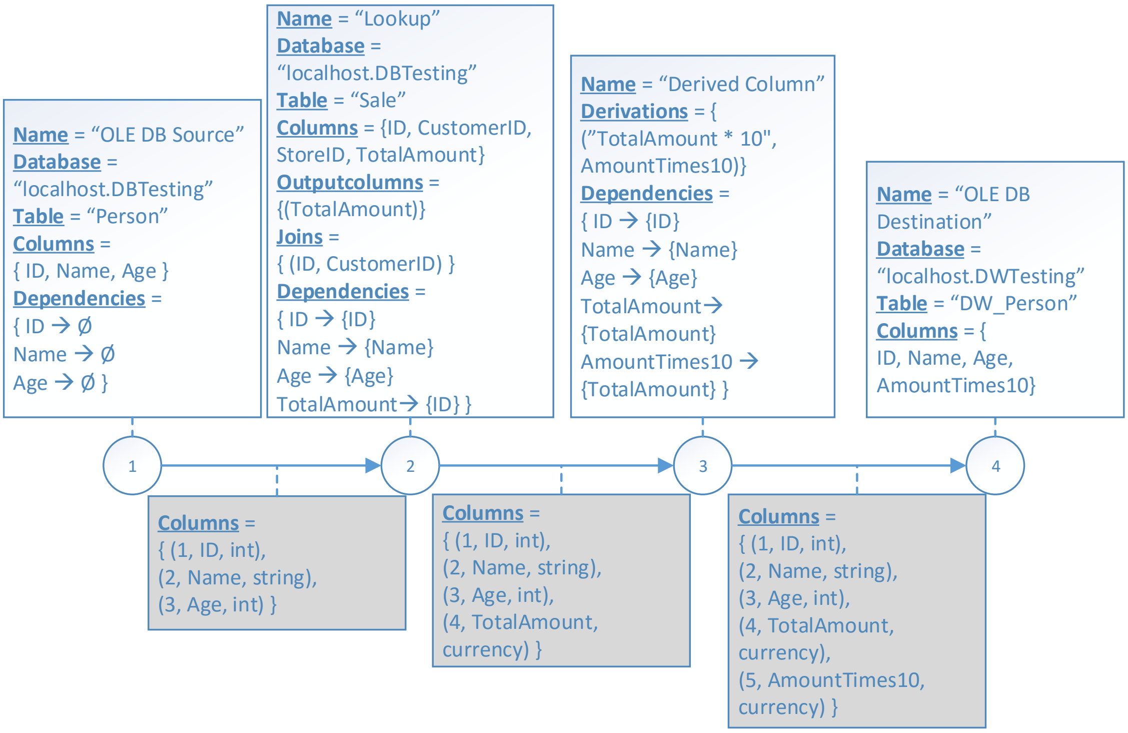

Name = "OLE DB Source"
Database = "localhost.DBTesting"
Table = "Person"
Columns = { ID, Name, Age }
Dependencies = { ID → Ø
Name → Ø
Age → Ø }
Name = "Lookup"
Database = "localhost.DBTesting"
Table = "Sale"
Columns = {ID, CustomerID, StoreID, TotalAmount}
Outputcolumns = {(TotalAmount)}
Joins = { (ID, CustomerID) }
Dependencies = { ID → {ID}
Name → {Name}
Age → {Age}
TotalAmount → {ID} }
Name = "Derived Column"
Derivations = { ("TotalAmount * 10", AmountTimes10)}
Dependencies = { ID → {ID}
Name → {Name}
Age → {Age}
TotalAmount → {TotalAmount}
AmountTimes10 → {TotalAmount} }
Name = "OLE DB Destination"
Database = "localhost.DWTesting"
Table = "DW_Person"
Columns = { ID, Name, Age, AmountTimes10}
1
2
3
4
Columns = { (1, ID, int), (2, Name, string), (3, Age, int) }
Columns = { (1, ID, int), (2, Name, string), (3, Age, int), (4, TotalAmount, currency) }
Columns = { (1, ID, int), (2, Name, string), (3, Age, int), (4, TotalAmount, currency), (5, AmountTimes10, currency) }

Figure 4: Illustration of graph of MAIME. This graph contains OLE DB Source, Lookup, Derived Column, and OLE DB Destination. The data type names are simplified for the purpose of readability. When we refer to columns in the properties of vertices, we refer only to the name of the column for simplicity.

change configurations, and (2) *Advanced configurations*. To properly understand the configurations, we need to first explain the different kinds of EDS schema changes.

3.2.1 EDS Schema Changes

For simplification, we only consider certain column related EDS schema changes. The schema changes that we consider are the following:

Addition which represents a new column in a schema.

Deletion which represents a deleted column in a schema.

Rename which represents a renamed column in a schema.

Data Type Change which represents a column where the data type has changed. This also includes any property change of the column such as: Allows null, length, scale, precision, and so on.

This means that changes such as table name changes, table deletions, and foreign key constraint changes are not directly included. This of course reduces the amount of changes MAIME is able to handle, but in return, it reduces the complexity that MAIME has to deal with. However, it might for example be possible to represent a table deletion as a sequence of column deletions.

3.2.2 EDS Schema Change Configurations

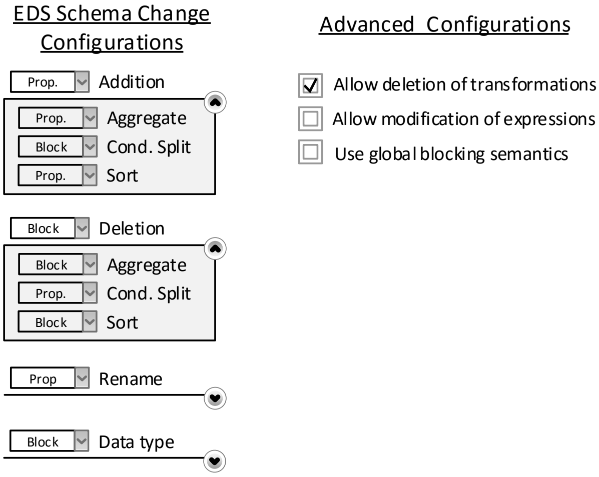
For each combination of EDS schema change $ch \in \{Addition, Deletion, Rename, Data\ Type\ Change\}$ and vertex type t, a policy $p(t, ch) \in \{Propagate, Block, Prompt\}$ can be specified by an administrator.

Propagate
A propagate policy defines that whenever $p(t, ch) = Propagate$, then reparation of vertices of type t are allowed for EDS schema changes of type ch.

Block
For a given EDS schema change ch and vertex type t, we say that if $p(t, ch) = Block$, then for every vertex v where $v.type = t$, we have that the alteration algorithm (explained in Section 3.3) is not allowed to make modifications to v or to any successor v_{succ} of v, even if $p(v_{succ}.type, ch) = Propagate$.

Prompt
If $p(t, ch) = Prompt$, then the choice of whether to block or propagate the change is deferred to runtime, where an administrator is prompted to make that choice just before the reparation is performed.

How an administrator can configure the EDS schema change configurations are shown on the left side of Figure 6. The administrator can either set a policy for the entire type of an EDS schema change, or the administrator can be more specific and choose a policy for each type of vertex for a given EDS change. For example, the administrator can configure that for all Conditional Split vertices, deletion of columns should be propagated, but all other EDS changes should be blocked.



EDS Schema Change Configurations

Prop. Addition
- Prop. Aggregate
- Block Cond. Split
- Prop. Sort

Block Deletion
- Block Aggregate
- Prop. Cond. Split
- Block Sort

Prop Rename

Block Data type

Advanced Configurations

☑ Allow deletion of transformations
□ Allow modification of expressions
□ Use global blocking semantics

Figure 6: The configurations that the administrator is able to specify.

3.2.3 Advanced Configurations

MAIME is based on the idea that we should repair ETL processes such that all of them will be able to execute afterwards. In this section we explain the more difficult cases of maintaining ETL processes, and explain which configurations the administrator can enable to handle these cases. The possible options for advanced configurations can be seen on the right side of Figure 6

Considering the case of a deletion of a column, we need to take into account that both columns and vertices can be dependent on the deleted column. Columns such as a derived column is dependent on the columns that it was derived from. Vertices are dependent on columns if the columns are a part of that vertex's input, expression, condition, or used in some other way by the vertex.

To explain this concept further, we give an example of a Conditional Split with two columns a and b with one expression: $a < 20$ && $b < 40$. The expression reflects that the vertex is dependent on a and b, since if either a or b is deleted, the expression would be invalid, which consequently invalidates the vertex. Maintaining this case could involve modifying the expression, which is explained further below with *Allow modification of expressions*. In the case where the condition only has one column such as $a < 20$, we refer to the Conditional Split as being *fully dependent* on a, since if a is deleted, the vertex would no longer be valid and can not be maintained. What it means for a vertex to be *fully dependent* on an column is explained more in depth in section 3.3. Below are descriptions of the three advanced configurations available in MAIME as illustrated on the right side on Figure 6.

Use Global Blocking Semantics

The *Use global blocking semantics* option dictates whether or not the alteration algorithm should terminate if the policy for any vertex is Block. If *Use global blocking semantics* is enabled, the semantics of the *Block* policy changes. If $p(t, ch) = Block$ and *Use global blocking semantics* is enabled, then every ETL process containing a vertex of type t will not be considered for reparation whenever the EDS schema change is of type ch.

Allow Deletion of Transformations

The *Allow deletion of transformations* option allows deletion of vertices in a graph. Since we aim to make ETL processes run successfully, we give the option of allowing MAIME to modify graphs to such an extent that it might cause large portions of a graph to be removed, but the corresponding ETL process would be able to execute. Figure 7 shows an example of a process where the Sort vertex uses column a for sorting. If a is deleted from the EDS, it would render the vertex invalid. With *Allow deletion of transformations* option enabled, the alteration algorithm attempts to repair the process by deleting the Sort vertex and any successive vertices, thereby making the process executable. The algorithm described in Section 3.3 further explains this principle.

Allow Modification of Expressions

The *Allow modification of expressions* option allows the alteration algorithm to modify expressions in vertices (such as Conditional Split and Derived Column) in the event of deletions or data type changes of columns. Consider our previous example of a Conditional Split with input columns a and b, and one condition with the expression $a < 20$ && $b < 40$, where an EDS schema change of b being deleted occurs. If *Allow modification of expressions* is enabled, the alteration algorithm would attempt to change the expression: $a < 20$ && $b < 40$ to $a < 20$. This would make the process run successfully without removing the whole expression, but the altered semantics of the expression might be undesirable. If *Allow modification of expressions* is disabled, then the alteration algorithm would instead remove the expression. Modification of expressions is always allowed if the EDS schema change is the renaming of a column. This is because it is fairly simple to preserve the original semantics by replacing the old name of the renamed column with the new name in the expression.

3.3 Graph Alteration

This section details how our graph model adapts to EDS schema changes. As put forth in Section 3.2, we have the following types of policies that the administrator can specify: Propagate, Block, and Prompt. Only a propagation (which can also occur through a prompt) alters the graph. Therefore, this section goes into depth on how to propagate an EDS schema change through the graph. How propagation is handled for a given type of EDS schema change is specific to each type of transformation. A table of actions to perform when propagating EDS schema changes through the various transformations is located in Appendix A. However, just applying actions from the propagation table to each vertex independently is not enough, as a change in one vertex can also affect successive vertices. This idea will be further explored in the algorithm in Algorithm 1.

The `Alter-graph` algorithm takes in a single EDS schema change ch and our property graph G. To be able to handle multiple EDS schema changes, the `Alter-graph` algorithm is called for each EDS schema change. Note that in the

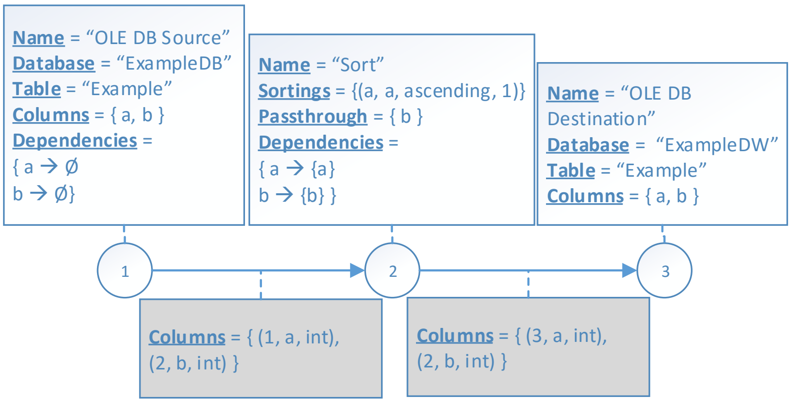


Figure 7: Small example of a graph containing an OLE DB Source, Sort, and OLE DB Destination.

Algorithm 1: Algorithm for propagating EDS schema changes to the graph

```
    Name: Alter-graph
    Input: EDS-Change $ch$, Graph $G$
    Output: Graph $G$
1   List $L$ = topological-sort($G$)
2   if Use global blocking semantics is enabled then
3       foreach Vertex $v \in L$ do
4           Policy $p$ = lookup-policy($v.type$, $ch.type$)
5           if $p$ = Block then
6               return G
7   foreach Vertex $v \in L$ (in topological order) do
8       Policy $p$ = lookup-policy($v.type$, $ch.type$)
9       if $p$ = Block then
10          Remove all successors of $v$ from $L$
11          continue
12      Update $v$'s dependencies to not include deleted
        columns
13      foreach Column $c \in v$'s outgoing edges do
14          if $v.dependencies(c) = \emptyset$ AND $v.type \neq$ OLE
            DB Source then
15              if $v$ is fully dependent on $c$ then
16                  Delete $v$ and $v$'s ingoing and outgoing
                    edges from $G$
17                  Break inner loop
18              else
19                  Delete $c$ from $v$'s corresponding outgoing
                    edges and dependencies
20      if $v.type$ is OLE DB Destination AND $v$ has no
        ingoing edges then
21          Delete $v$
22      if $v$ was not deleted AND $ch$ affects $v$ then
23          $G$ = alter($G$, $v$, $ch$)
24  return $G$
```

pseudo-code, we only consider the case where *Allow deletion of transformations* is enabled, and neither Prompt nor data type change is considered.
The order in which we traverse the graph matters, as visiting a vertex affects successive vertices. For this, topological sorting is used on Line 1 to order the vertices, such that when a vertex is visited, it is guaranteed that all its predecessors have been visited beforehand. The List of topologically sorted vertices is referred to as L. On Lines 2-6, the algorithm handles the case where *Use global blocking semantics* is enabled. This entails checking if there exists a vertex in L with the Block policy defined for the EDS change type ch. In case such a vertex exists, the algorithm returns an unchanged graph. On Line 7, we start traversing each vertex v of L. On Lines 8-11, we check if the policy for v given ch is Block. If this is the case, we can disregard this branch of the graph for the traversal, which is why we remove all successors of v from the sorted list of vertices. Line 12 updates the dependencies of a vertex. This is done by going through each dependency in $v.dependencies$ for a given vertex v and checking that all of the involved columns are still present in the ingoing edges. The reason for this is that some columns might have been removed from v's ingoing edges when traversing the preceding vertices, such that $v.dependencies$ refers to columns that no longer exist in v's input.

As stated before, it is not sufficient to just look at vertices independently when going through the output columns of each vertex. Lines 13-19 show the case for deleting columns with no dependencies. As an example, consider a vertex v whose preceding transformation is a Derived Column with the input column c, which is used to derive a new column d through the expression $d = c + 42$. Now, v will receive c and d as input columns, but if the EDS schema change (ch) is a deletion of c, not only will c not be available to v anymore, d will also be deleted, as it is no longer possible to derive it from the Derived Column without c. The way to find out that d is no longer computable is by seeing that d is dependent on $\emptyset$ (Line 14). This signifies that d was dependent on column(s) which have been deleted. The exception is an OLE DB Source, which is the only vertex for which *dependencies* maps columns to $\emptyset$, as explained in Section 3 (Note OLE DB Source is the only transformation in Table 1 that allows no ingoing edges).

Beginning on Line 15 the case whether it is possible to delete only the given column d or if it is necessary to delete the entire vertex v is considered. This narrows down to whether v is *fully dependent* on d or not. We say that a vertex v_1 is fully dependent on some column c_1, when v_1 would be rendered invalid if c_1 was deleted. For instance, if v_1 is an Aggregation and c_1 is the only remaining column that is being used for aggregations, then v_1 is fully dependent on c_1. The deletion of c_1 would result in $v_1.aggregations = \emptyset$, which is not a valid transformation. What qualifies a vertex as being fully dependent on a column is specific for each type of transformation, we provide a table of all these specifications in Appendix B. Lines 16-17 show the case of deleting a whole vertex and all of its ingoing and outgoing edges, if they are no longer used. This iteration of the loop breaks because it would not make sense to continue iterating over the output columns of a vertex that has been deleted. The other case of only deleting the given column is shown in Lines 18-19.

Since a vertex of type OLE DB Destination does not have any outgoing edges, it is not considered in the previous loop on Lines 13-19. However, we still want to delete the vertex if it is invalid, i.e., if it has no ingoing edge. This is performed on Lines 20-21. Afterwards on Lines 22-23, if v was not deleted at an earlier point in the algorithm, then the corresponding propagation action for the EDS schema change and transformation type is invoked. The propagation action is looked up in Table 5 described in Appendix A.

Finally, the fully altered graph is returned on Line 24.

4. IMPLEMENTATION

This section describes the implementation of MAIME. This is explained by going through components of the architecture and explaining the essential implementation details of each one.

The two most impactful parts of MAIME are the ECM and the MM. The ECM creates, stores, and compares metadata snapshots. The MM is responsible for creating the graph, and propagating EDS schema changes to the graph model. The GUI component will not be examined further with respect to the implementation, since it has been implemented as described in Section 3.2. However, before we go through the components, we will give an overview of which features from our model have been implemented.

MAIME was developed in C# for Microsoft SQL Server 2014 Developer Edition and with the SSIS Designer as the data integration tool. The SSIS Designer is part of the SQL Server Data Tools (v. 14.0.60203.0). The codebase for the implementation includes 7697 lines of code, 85 classes, and 410 members.

4.1 Status of MAIME

This section gives a brief overview of which features are implemented in MAIME. For the advanced configurations, *Allow deletion of transformations* and *Use global blocking semantics* are implemented. We did not implement *Allow modification of expressions.* We focused our attention primarily on *Allow deletion of transformations* and *Use global blocking semantics*, since they seemed to be the most impactful configurations for MAIME. Currently, *Allow deletion of transformations* is always enabled. The Propagate and Block policies are implemented as described in Section 3.2.2. Prompt is not yet implemented in the prototype. Table 2 shows which transformations have been implemented with respect to each type of EDS schema change, and which are left as future work.

The only deviation from the description of transformations in the graph model in Section 3.1 is Lookup. Currently in the implementation, Lookup is fully dependent on all columns in any of the join conditions of the Lookup. This means that if we have the join conditions: $a = d$ and $b = c$, the Lookup is fully dependent on a, b, c, and d, such that if any of these columns are deleted, the Lookup is invalid. In actuality, Lookup should only be fully dependent on a column if it is part of *all* join conditions. However, this requires us to implement further support for the analysis of SQL statements, as unlike other transformations, a large part of the Lookup transformation is represented internally as an SQL statement in SSIS. Regarding visualization of the graph model, we did not find any API that was expressive enough to illustrate our structure, and has therefore not been implemented.

4.2 EDS Change Manager

This section describes the essential implementation details of the ECM.

Metadata is gathered by querying the information schema views of an EDS and preserving it in the Metadata Store for persistent storage. When fetching metadata, an administrator provides the servername, authentication details, and the name of the database for which the metadata is gathered. The metadata is stored as a snapshot in a JSON file; an example can be found in Appendix C. Each snapshot is stored in an individual file in a separate folder for each EDS. The file can later be deserialized by MAIME to compare it with the current metadata of the EDS to find any changes to the EDS. Detecting which type of change has occurred is closely related to the unique identifier of the column, table, or database. If a column has been renamed, we can see that the name property of the column has been changed, but the unique identifier stays the same. However, if the column has been deleted and a column with the same properties but different name has been added, an administrator could mistake it is for rename, while we are actually able to detect that this is an addition and deletion since the unique identifier is not the same for the two columns. This principle is similar with table and database.

4.2.1 Structure of EDS Metadata Snapshot

The created snapshots contain information regarding the databases, tables, and columns of an EDS. For a database we store: a unique identifier (ID), name of the database, and a list of tables in the database. For a table we store: an ID, name of the table, and a list of columns in the table.

The column is a little more extensive than the previous two since a column contains a lot of properties. We considered which properties would be useful for MAIME, and we found that almost everything could be used given the right scenario. We have therefore chosen to extract a lot of properties, but only a few are actually used in MAIME at the moment. For a column we store: an ID, name, data type *(length, scale, precision, precisionradix, charactermaxlength, charactersetcatalog, charactersetname, charactersetschema and datetimeprecision)*, and whether it is a primary key, unique and nullable.

Table 2: Overview of which transformations have been implemented with respect to each type of EDS schema change.

	Addition	Deletion	Rename	Data Type Change
OLE DB Source	✗	✓	✓	✗
OLE DB Destination	✗	✓	✓	✗
Aggregate	✗	✓	✓	✗
Conditional Split	✗	✓	✓	✗
Data Conversion	✗	✗	✗	✗
Derived Column	✗	✓	✓	✗
Lookup	✗	✓	✓	✗
Sort	✗	✗	✗	✗
Union All	✗	✗	✗	✗

4.3 Maintenance Manager

The MM includes the different parts for graph creation, administrator configurations, and graph alteration. In the following sections there is given more detail about how the graph is implemented in MAIME and how changes are propagated throughout the graph.

4.3.1 Graph

This section goes into detail of how the graph is represented internally in MAIME.

As it was outlined in Section 2.3, our graph is implemented as a layer on top of SSIS. Therefore, our graph constructs have references to the corresponding SSIS constructs. To be more exact, each vertex refers to a SSIS component (transformation), each edge refers to a SSIS path, and each column has a corresponding reference to a SSIS IDTS object. The SSIS IDTS object covers both input and output columns in SSIS. By having these references, we can easily propagate changes to the underlying ETL process during execution of the graph alteration algorithm that is described in Section 3.3.

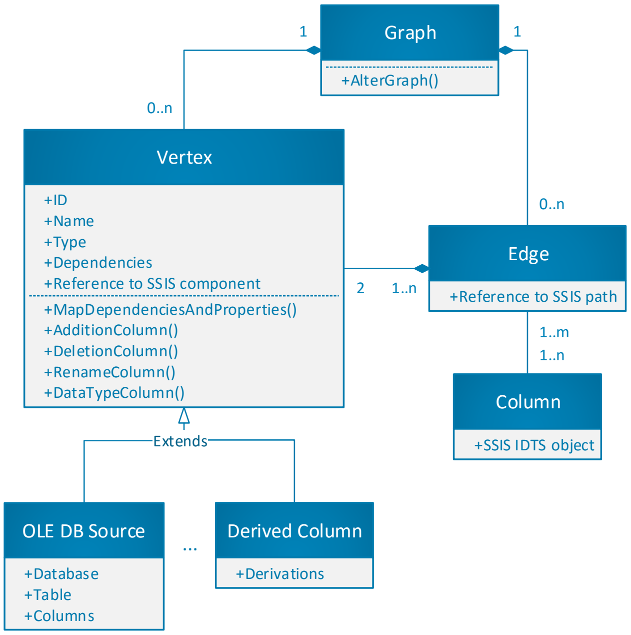


Figure 8: Detailed view of the MAIME graph.

The implementation of the graph is described based on the class diagram in Figure 8. For simplicity, only OLE DB Source and Derived Column are shown in the bottom of the diagram. The graph contains a list of vertices and a list of edges. Each edge contains two vertices and has a list of the columns which are transferred through it. Vertex is an abstract class which acts as an extension point for different transformation types. Each transformation overrides the `MapDependenciesAndProperties` method which allows each type of transformation to have their own specific dependencies and properties. For example, for the Derived Column transformation, we define dependencies based on the expression used to create the derivation. The other abstract methods `RenameColumn`, `AdditionColumn`, `DeletionColumn`, and `DataTypeColumn` are implemented as described in the Section 3.3.

4.3.2 Transition from SSIS to MAIME Graph

This section explains how the graph of MAIME is constructed by the MM from the internal details of SSIS.

Implementing Graph Transition

In order to realize the graph of MAIME as described in Section 3.1, we have to translate the components from SSIS to the graph of MAIME. Initially, a SSIS package is loaded by parsing its `.dtsx` file conforming to XML specification. Then, we extract its SSIS Data Flow tasks. Note that every Data Flow task is stored as a graph in a `.dtsx` file. This is referred to as a *SSIS graph*.

Construction of a MAIME graph is done in two iterations. During the first iteration we extract one Data Flow task and go through its components, creating corresponding vertices with references to the Data Flow task components. With the set of vertices V and references to the SSIS components, we can extract information for each vertex such as $v.name$ and $v.type$. Afterwards, every SSIS path is translated into an edge. This gives us the set of edges E of our graph. It is important to note that columns are not stored in the edges in this iteration, since they do not exist on SSIS paths. After the first iteration is complete, we have created the basic structure of the graph of MAIME.

In the second iteration, we deduce dependencies, assign columns to edges, and create transformation specific properties by overriding the abstract method `MapDependenciesAndProperties`.

How columns from our graph model are implemented is of interest, as it is important to know how we identify columns uniquely. For a given column c, $c.id$, $c.name$, and $c.type$ are trivially resolved as these can be taken directly from SSIS once a column is first encountered when traversing the graph in topological order. *c.id* is able to be directly mapped to a property in SSIS called *LineageID*. In a Data Flow task, each column has at least one unique LineageID. The LineageID is used to uniquely identify the origin of the column, which can for example originate from an OLE DB source if the column is extracted from an EDS. It can also come from a Derived Column transformation for a newly derived column, or from an Aggregate transformation for the newly created columns. The LineageID is important to propagate changes from the graph model to the Data Flow task in SSIS as it allows us to identify the columns in the SSIS package.

5. EVALUATION

This evaluation is conducted to test the efficiency of MAIME by seeing how much time and how many user inputs are required for MAIME to repair an ETL process compared to repairing an ETL process manually. In Section 5.1 we first describe the methodology of how the evaluation is done. Afterwards in Section 5.2, we go through one of our test cases in detail to give insight into how they are structured. Later in Section 5.3 we present the results of the evaluation.

5.1 Methodology

In the evaluation we compare how much time and how many user inputs are required to resolve a series of EDS schema changes manually compared to doing the same changes in MAIME. A user input is defined to be both a mouse click and a keystroke done by the user (both are recorded separately).

The SSIS packages that the evaluation has been conducted on have been created by ourselves. We created them because the prototype only implements a subset of the transformations available in SSIS, and can therefore not maintain any transformation outside this subset. We did not find any well known, publicly available SSIS packages that conformed to these constraints. Each of our SSIS packages has a single Data Flow task for this evaluation. Multiple instances of the transformations were included and used in various packages to increase the likelihood that all cases have been tested.

The user performing the evaluation is given the EDS schema changes and two versions of the ETL process: One in its initial state, and another in the desired result state. The user then performs maintenance of each package three times. The first evaluation gives an indication of how long it takes to maintain an ETL process without knowing every step to maintain the ETL process. The second and third attempt are performed in order to provide a best-case scenario, since the user learns the quickest way to maintain the ETL process through repetition. MAIME also repairs the package three times. The timer of the evaluation starts when the user begins the repair the ETL process after having seen the given information (EDS schema changes and the two versions of the ETL process). During the test, the duration and amount of user inputs used is recorded using a software tool. This is done for both the manual approach and the MAIME approach.

For the manual approach, the user repairs the ETL process with the SSIS Designer tool, which is a graphical tool from Microsoft. The task is to transform the broken SSIS package into the desired result state. To conduct the evaluation, we used Visual Studio 2015 Community edition [13] and SQL Server Data Tools (v. 14.0.60203.0) [14].

For the MAIME approach, the interaction required by the user involves: (1) accepting the configurations and currently loaded ETL processes and connected EDSs, and (2) initializing the maintenance process.

The EDS schema changes on columns that are used for the evaluation include: Deletion and renaming. Addition and data type change for columns are not evaluated since they have not been implemented. A schema diagram is shown in Figure 9 for the EDS that is used for the SSIS packages before any EDS schema changes happen.

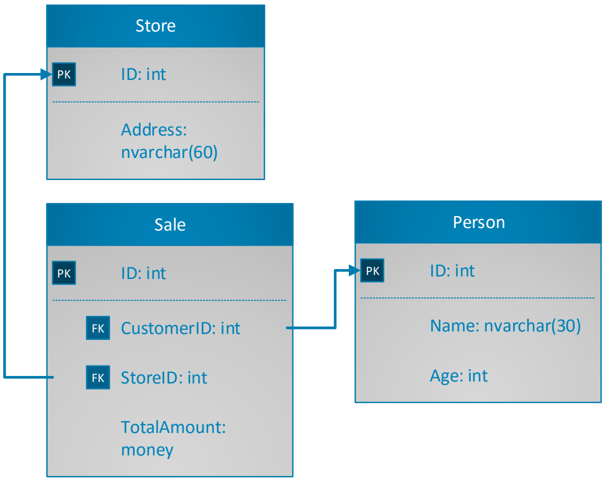


Figure 9: Figure 5 shown again for convenience. Schema diagram of the EDS that the evaluation is conducted on.

For each SSIS package that we compare, both MAIME and the manual work end up having the same end result, i.e. the same state of the maintained SSIS package. Only the process of achieving that end result differs.

The EDS schema change configurations of MAIME for all evaluations are all set to propagate. For the advanced configurations, *Allow deletion of transformations* is enabled, *Allow modification of expressions* is disabled, and *Use global blocking semantics* is disabled.

The user doing the evaluation is an 8th semester computer science student from Aalborg University with roughly 5 months of experience with SSIS and ETL. In an ideal evaluation the user would have more SSIS experience. We wish to point out the potential bias in this evaluation, as the user doing the evaluation is an author of this article.

The technical specifications of the computer used for conducting the evaluations are: i7-2600 CPU at 3.40 GHz, 8.00 GB DDR3 RAM. It runs with Windows Server 2012 using Microsoft SQL Server 2014 Developer Edition and MAIME was executed using .NET Framework 4 (v. 4.0.30319).

5.2 Evaluation Example

This section gives an elaborate explanation of one of our test cases and a given list of EDS schema changes. First we describe the SSIS package for the test case, followed by the

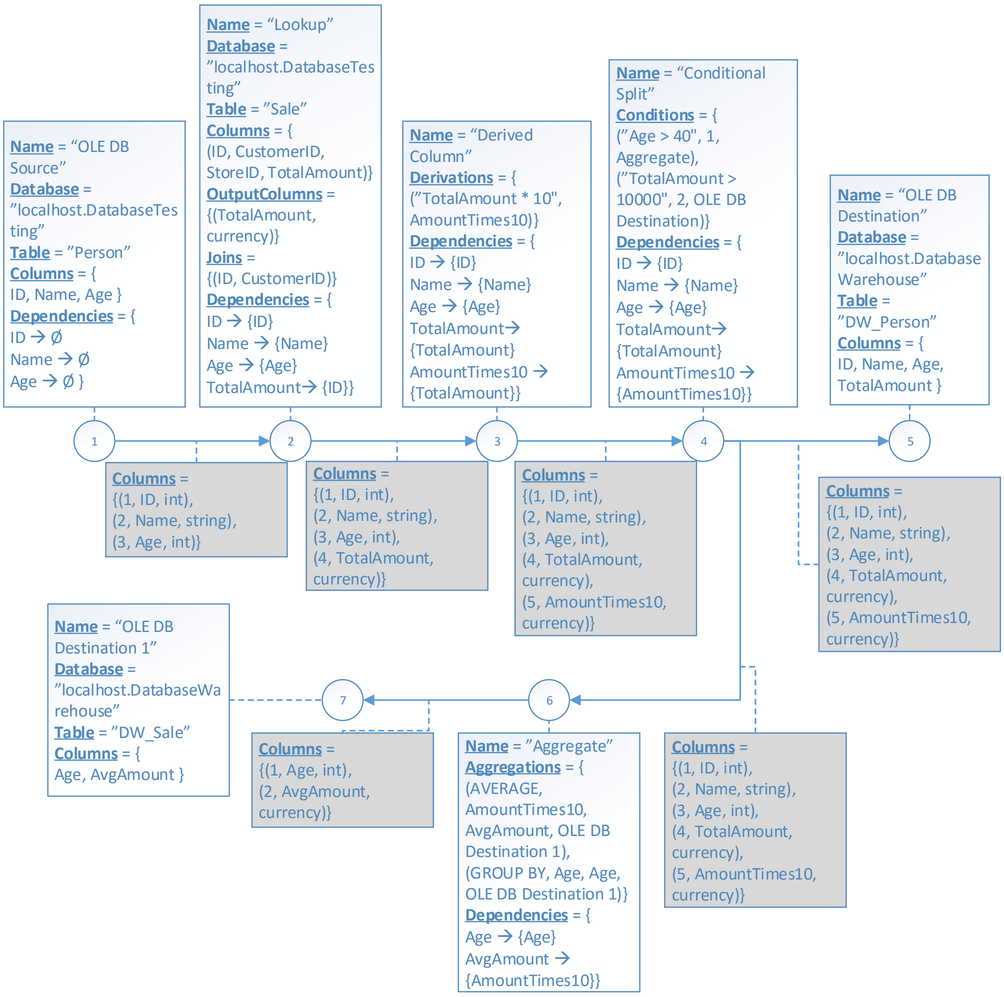

Name = "OLE DB Source"
Database = "localhost.DatabaseTesting"
Table = "Person"
Columns = { ID, Name, Age }
Dependencies = { ID → Ø Name → Ø Age → Ø }
Name = "Lookup"
Database = "localhost.DatabaseTesting"
Table = "Sale"
Columns = { (ID, CustomerID, StoreID, TotalAmount)}
OutputColumns = {(TotalAmount, currency)}
Joins = {(ID, CustomerID)}
Dependencies = { ID → {ID} Name → {Name} Age → {Age} TotalAmount→ {ID}}
Name = "Derived Column"
Derivations = { ("TotalAmount * 10", AmountTimes10)}
Dependencies = { ID → {ID} Name → {Name} Age → {Age} TotalAmount→ {TotalAmount} AmountTimes10 → {TotalAmount}}
Name = "Conditional Split"
Conditions = { ("Age > 40", 1, Aggregate), ("TotalAmount > 10000", 2, OLE DB Destination)}
Dependencies = { ID → {ID} Name → {Name} Age → {Age} TotalAmount→ {TotalAmount} AmountTimes10 → {AmountTimes10}}
Name = "OLE DB Destination"
Database = "localhost.Database Warehouse"
Table = "DW_Person"
Columns = { ID, Name, Age, TotalAmount }
1
2
3
4
5
Columns = {(1, ID, int), (2, Name, string), (3, Age, int)}
Columns = {(1, ID, int), (2, Name, string), (3, Age, int), (4, TotalAmount, currency)}
Columns = {(1, ID, int), (2, Name, string), (3, Age, int), (4, TotalAmount, currency), (5, AmountTimes10, currency)}
Columns = {(1, ID, int), (2, Name, string), (3, Age, int), (4, TotalAmount, currency), (5, AmountTimes10, currency)}
Name = "OLE DB Destination 1"
Database = "localhost.DatabaseWarehouse"
Table = "DW_Sale"
Columns = { Age, AvgAmount }
7
6
Columns = {(1, Age, int), (2, AvgAmount, currency)}
Name = "Aggregate"
Aggregations = { (AVERAGE, AmountTimes10, AvgAmount, OLE DB Destination 1), (GROUP BY, Age, Age, OLE DB Destination 1)}
Dependencies = { Age → {Age} AvgAmount → {AmountTimes10}}
Columns = {(1, ID, int), (2, Name, string), (3, Age, int), (4, TotalAmount, currency), (5, AmountTimes10, currency)}

Figure 11: The Data Flow task from Figure 10 shown in MAIME. Types of vertices are omitted, as the type of transformation should be clear from the name.

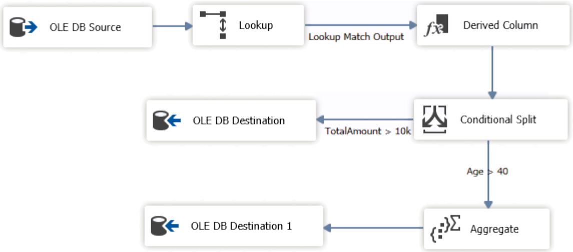


Figure 10: The Data Flow task of the SSIS package for test case 1 before any modifications in the SSIS Designer.

EDS schema changes. The two remaining test cases can be found in Appendix D.

Figure 10 shows the SSIS package containing the transformations: OLE DB Source, Lookup, Derived Column, Conditional Split, Aggregate, and two OLE DB Destinations. Note that the labels on the SSIS paths have no effect on the execution of the task, they are merely user-defined descriptions. To better understand this package, we present it with MAIME's graph model as shown in Figure 11.

OLE DB Source extracts the columns: ID, $Name$, and Age from the Person table.

Lookup extracts the $TotalAmount$ from the Sale table by joining $Sale.CustomerID$ with $Person.ID$.

Derived Column derives the new column $AmountTimes10$ which is derived from the derivation: $(TotalAmount * 10, AmountTimes10)$.

Conditional Split has two outgoing edges and two conditions: $\{(Age > 40, 1, Aggregate), (TotalAmount > 10000, 2, OLE\ DB\ Destination)\}$.

Aggregate has the aggregations: $\{(AVERAGE, AmountTimes10, AvgAmount, OLE\ DB\ Destination\ 1), (GROUP\ BY, Age, Age, OLE\ DB\ Destination\ 1)\}$.

OLE DB Destination 1 loads $AvgAmount$ and Age into the DW table SalesData.

OLE DB Destination loads the ID, Age, and $AmountTimes10$ into the DW table PersonSalesData.

The EDS changes that are applied to this evaluation are: (1) Renaming of the column Age to $RenamedAge$ in the $Person$ table and (2) deletion of the column $TotalAmount$ in the $Sale$ table. The state of the graph after it has been maintained is show in Figure 12 in the SSIS designer tool, and with MAIME's graph in Figure 13. Figure 13 shows that Age is successfully renamed on all edges. The deletion of column $TotalAmount$ is slightly more complicated since the condition $TotalAmount > 10000$ on Conditional Split involves the column $TotalAmount$. This condition is no longer valid, which results in the removal of the outgoing edge containing it. We therefore delete OLE DB Destination, as it no longer has any ingoing edges. Derived Column derives the column $AmountTimes10$ from $TotalAmount$ and can therefore no longer be derived. We delete this derivation, but still retain our Derived Column transformation since it can exist without doing any derivations. Ideally, MAIME would delete Derived Column and connect Lookup and Conditional Split. However, this can be rather difficult to do in certain cases, and is further discussed as future work in Section 7.1. The Aggregate transformation takes the average of $AmountTimes10$, which no longer exists, and this aggregation is therefore also deleted. OLE DB Destination 1 no longer loads the aggregated AvgAmount from the Aggregate transformation into the DW. Both $TotalAmount$, $AmountTimes10$, and AvgAmount are deleted from the edges.

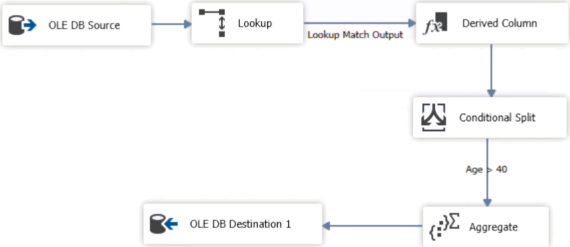


Figure 12: The Data Flow task of the SSIS package for test case 1 after the EDS schema changes of renaming Age to RenamedAge and deleting TotalAmount.

5.3 Results

This section shows the results from evaluating the time consumption and number of user inputs needed for resolving EDS schema changes with MAIME and for doing it manually with the SSIS Designer tool. The results for the time consumption are shown in Table 3 while the results for the number of user inputs are shown in Table 4.

For time-consumption, MAIME took on average 4 seconds across all 3 test cases while manually resolving changes took 39.3 seconds on average across all 3 cases for the third attempt. This means MAIME was on average of 9.8 times faster for resolving EDS schema changes.

For user input, MAIME required on average 4 user inputs, while manually resolving changes required 38 user inputs on average for the third attempt across all 3 cases. This means MAIME on average required 9.5 times less inputs for resolving EDS schema changes.

For the manual work, it took significantly more time the first time the user had to maintain the ETL process. The user did, however, get progressively quicker at maintaining the ETL processes for the second and third attempt, with the best result being from the third attempt. The number of both mouse clicks and keystrokes showed similar results. On the basis of these evaluations, MAIME has been shown to be able to significantly reduce the amount of time and user input needed for maintaining ETL processes.

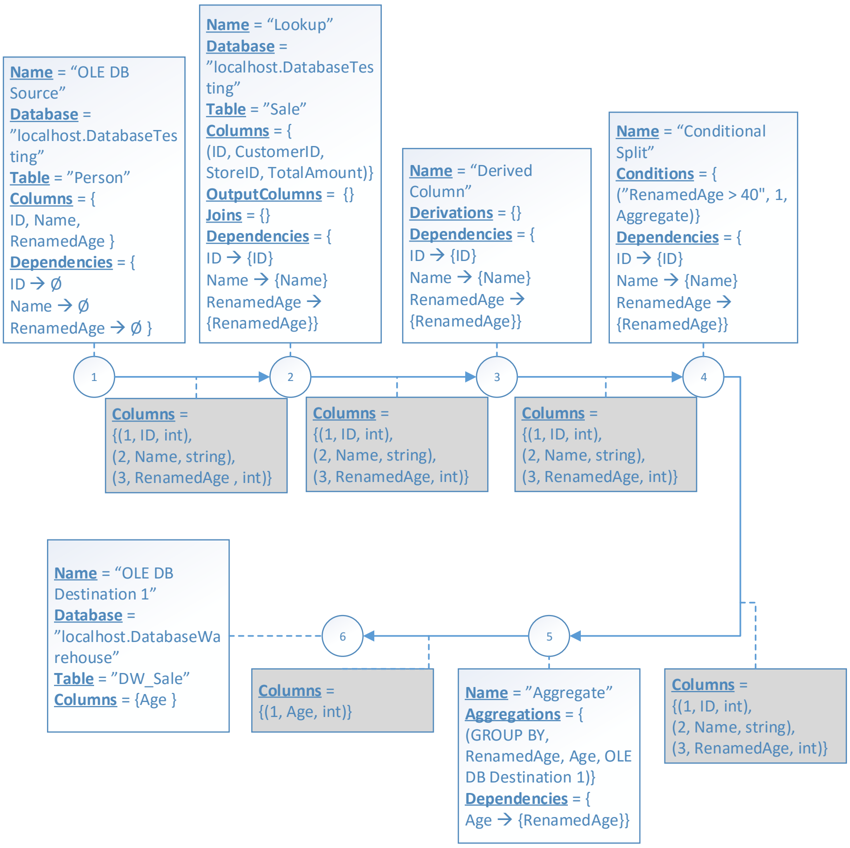
In a real life scenario, a company would have to maintain a multitude of ETL processes, which for each could take on average 39.3 seconds to repair as our evaluation showed. However, MAIME would not for every ETL process use 4 seconds, as shown in the evaluation, as this result is derived from the program having to first load the ETL processes, detect EDS schema changes, and then maintain the ETL processes, not to mention the time spent clicking buttons in the GUI. The reparation algorithm of MAIME uses less than a second, and we would therefore argue that the evaluation results are heavily favored towards repairing the ETL process manually.

6. RELATED WORK

Related work exists on the topic of maintaining ETL processes when the schema of EDSs change.

The framework *Hecataeus* by Papastefanatos et al. [15–17] analyzes ETL processes by detecting when a change happens to the schema of the EDSs (they call it an evolution event), and proposes changes to the ETL processes based on defined *policies*.

The Hecataeus framework abstracts ETL processes as SQL queries and views which are used in a graph to represent the activities. An activity resembles an ETL transformation. Hecataeus' graph is captured by an *ETL Summary* which can have multiple subgraphs for each activity. Each activity can further be broken down, as it includes nodes or entire subgraphs for relations, SQL queries, conditions, and uses views for instance for input and output. The types of evolution events taken into account are addition, deletion, and modification. An administrator can annotate nodes and edges of the graph with policies for each type of evolution event, while in MAIME policies are provided for each EDS change type and not specifically for each node or edge. This aspect could be preferable in MAIME if the administrator had to repair a lot of ETL processes. The Hecataeus policies dictate how ETL processes should be adjusted when an evolution event occurs. There are 3 policies: (1) *Propagate* readjusts the graph to reflect the new semantics according to the evolution event throughout the rest of the graph, (2) *Block* tries to retain the old semantics, and (3) *Prompt* asks the administrator to choose either Propagate or Block. Both Propagate and Prompt are similar in MAIME, whereas Block attempts to retain the semantics. To describe the Block policy, consider the example where the following SQL statement is used: `SELECT * FROM tablename`. This extracts all columns from the table *tablename*. If a new column is added to the table, extracting all column would result in different outcome and to retain the semantics with the Block policy, Hecataeus instead change the SQL statement to: `SELECT` $c_1, \ldots, c_n$ `FROM tablename`. This means that Hecateaus specifies that not all columns should be selected and can thereby exclude the newly added column in



Name = "OLE DB Source"
Database = "localhost.DatabaseTesting"
Table = "Person"
Columns = { ID, Name, RenamedAge }
Dependencies = {
ID → ∅
Name → ∅
RenamedAge → ∅ }
Name = "Lookup"
Database = "localhost.DatabaseTesting"
Table = "Sale"
Columns = { (ID, CustomerID, StoreID, TotalAmount)}
OutputColumns = {}
Joins = {}
Dependencies = {
ID → {ID}
Name → {Name}
RenamedAge → {RenamedAge}}
Name = "Derived Column"
Derivations = {}
Dependencies = {
ID → {ID}
Name → {Name}
RenamedAge → {RenamedAge}}
Name = "Conditional Split"
Conditions = { ("RenamedAge > 40", 1, Aggregate)}
Dependencies = {
ID → {ID}
Name → {Name}
RenamedAge → {RenamedAge}}
1
2
3
4
Columns = {(1, ID, int), (2, Name, string), (3, RenamedAge , int)}
Columns = {(1, ID, int), (2, Name, string), (3, RenamedAge, int)}
Columns = {(1, ID, int), (2, Name, string), (3, RenamedAge, int)}
Name = "OLE DB Destination 1"
Database = "localhost.DatabaseWarehouse"
Table = "DW_Sale"
Columns = {Age }
6
5
Columns = {(1, Age, int)}
Name = "Aggregate"
Aggregations = { (GROUP BY, RenamedAge, Age, OLE DB Destination 1)}
Dependencies = {
Age → {RenamedAge}}
Columns = {(1, ID, int), (2, Name, string), (3, RenamedAge, int)}

Figure 13: The altered Data Flow task from Figure 12 shown in MAIME. Types of vertices are omitted, as the type of transformation should be clear from the name.

Table 3: Time results for resolving EDS schema changes for the three test cases for MAIME and doing it manually. Test case results for the three repetitions are comma separated.

Time Elapsed	MAIME	Manual
Test Case 1	4 sec, 4 sec, 4 sec	187 sec, 159 sec, 59 sec
Test Case 2	4 sec, 4 sec, 4 sec	154 sec, 60 sec, 49 sec
Test Case 3	4 sec, 4 sec, 4 sec	23 sec, 13 sec, 10 sec

the list of selected columns $c_1, \ldots, c_n$.

The Hecataeus framework provides a thorough set of constructs for modeling databases and ETL processes as graphs, but there are some distinctions between their work and ours. The work on Hecataeus does not provide an explanation of how they construct their graph initially or how EDS schema changes are propagated to the ETL process. Furthermore, it is not explained how an evolution event is discovered, besides remarking that changes in an ETL process can be discovered by noticing failures to execute. A point worth mentioning is that MAIME is less time-consuming as an administrator does not have to annotate as many vertices and edges. On the other hand this makes Hecataeus more customizable. A significant difference compared to Hecataeus is that they model ETL processes through SQL queries and views while we make use of commonly used transformations for our graph model and thus abstract away from working with SQL. Finally, a distinction between Hecataeus and MAIME is that in Hecateaus the administrator has to define policies for each graph and thereby prepare for any upcoming EDS schema changes. However, MAIME is rather a program that you execute after the EDS schema changes have happened to try to repair broken ETL processes by comparing with a previous metadata snapshot.

Another framework is *E-ETL* [18, 19] by Wojciechowski which is also able to semi-automatically adapt structural changes of EDSs in ETL processes. Compared to the Hecataeus framework, a more detailed specification of how changes in EDSs are observed are given. These are observed by either comparing two successive snapshots or using schema triggers. In order to adapt the ETL processes, there are different methods which define how reparations are propagated. The work on E-ETL also briefly remarks that the framework is able to handle multiple ETL tools by having a model translator for each supported tool. E-ETL also has the same three policies as Hecataeus, with the exception of

Table 4: User Input results for resolving EDS schema changes for the three test cases for MAIME and doing it manually. Test case results for the three repetitions are comma separated.

User Inputs	MAIME	Manual
Test Case 1	Keystrokes: 0, 0, 0 Mouse clicks: 4, 4, 4	Keystrokes: 23, 15, 12 Mouse clicks: 88, 85, 38
Test Case 2	Keystrokes: 0, 0, 0 Mouse clicks: 4, 4, 4	Keystrokes: 9, 9, 8 Mouse clicks: 92, 48, 45
Test Case 3	Keystrokes: 0, 0, 0 Mouse clicks: 4, 4, 4	Keystrokes: 0, 0, 0 Mouse clicks: 16, 11, 11

the *Block* policy. In E-ETL the *Block* policy ignores the EDS schema change and does not attempt to modify the graph. The E-ETL framework has multiple algorithms for resolving an EDS schema change. These include: (1) *Defined rules* in which the administrator can define rules himself for nodes and edges, (2) *Standard rules* are the default rules which will be used if the administrator did not define anything, and (3) *Alternative scenarios* where case-based reasoning is used by adjusting the ETL process based on solutions to similar problems experienced previously. How these algorithms are used together is not specified.

While E-ETL expanded on some of the areas that Hecataeus lacked such as managing EDS changes and translating the graph, E-ETL is lacking explanations on their graph model (referred to as their *internal metamodel*). Like Hecataeus, E-ETL models ETL processes through SQL queries. The graph model has a *Super node* for each ETL process. An example of a Super node is shown in Figure 14. For a given SQL query, only the involved columns and constants are included, and their dependencies are represented through directed edges. When columns are dependent on each other is not clearly defined.

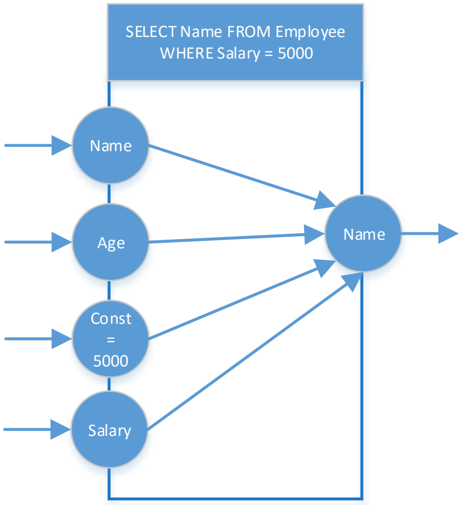


Figure 14: An example of how the query `SELECT Name FROM Employee WHERE Salary = 5000` is modeled in E-ETL.

Related work exists on how to conceptually model ETL processes using transformations that are commonly used in ETL processes. Trujillo et al. [20] do this by defining a small set of transformations that can be modeled through UML diagrams. Examples of the mechanism include *Aggregation*, *Conversion*, and *Join*, which are similar to the transformation Aggregate, Data Conversion, and Union All, respectively. This is in contrast to other frameworks, such as Hecataeus and E-ETL, where ETL processes are described by SQL queries and views, and represented with graphs. Using transformations rather than SQL queries can make conceptual modeling simpler and more maintainable, as each transformation has a clear responsibility and provides a higher level of abstraction compared to SQL queries, but at the cost of expressiveness. SQL queries are more expressive than both the approach of Trujillo et al. and MAIME, since we currently do not handle SQL queries and can therefore not use the functionality of SQL.

Trujillo et al. provide a set of general ETL processes that are not specific to any platform, while our transformations are specific to SSIS. Furthermore, instead of using UML diagrams we use property graphs to more closely relate to the internal graphs of SSIS. Trujillo et al. is focused on a theoretical framework whereas we also implemented our model.

Finally for the last comparison, we briefly investigated the Business Intelligence Markup Language (BIML) which is an XML dialect that can be used to generate SSIS packages [21]. It can be used to create templates and reusable code snippets to make the creation of SSIS packages easier and less time consuming. This would require that all of the ETL processes are written in BIML, which could prove to be problematic since no API for BIML is freely available.

MAIME is a tool that has a well-defined formalization of the graph as Hecataeus does, but at the same time includes the details from E-ETL of how to tie the formal graph model together with the implementation of an ETL process of a given ETL tool, which in our case is SSIS. Furthermore, MAIME also provides an openly available and working prototype unlike Hecataeus and E-ETL.

7. CONCLUSION AND FUTURE WORK

In this paper we introduced MAIME, a prototype that in reaction to EDS schema changes semi-automatically maintains ETL processes in the form of SSIS Data Flow tasks. ETL processes can be complex and time-consuming to repair manually. We presented MAIME as a tool to reduce the amount of errors and time spent on maintaining ETL processes. To accomplish this task, we introduced and implemented a graph model as a layer on top of SSIS Data Flow tasks, which simplifies the handling of EDS schema changes.

As we have seen in the evaluation in Section 5, MAIME required 9.5 times less input from the user and was 9.8 times faster compared to doing it with the SSIS Designer tool.
On the basis of these results, we conclude that MAIME has

been able to ease the burden of maintaining ETL processes in the test cases. Depending on the amount of processes that can be repaired automatically with MAIME, the amount of work and time required for an administrator to repair ETL processes manually are reduced greatly.

7.1 Future Work

Due to time constraints, we have left some aspects of maintaining ETL processes to be resolved in future projects. In this section, we give an overview of possible extensions to MAIME.

We disregarded propagating changes to the DW, e.g., adding a column to the relevant DW schema. However, an administrator might want to do the proper adjustments to the DW himself, in order to capture the correct semantics of an EDS schema change. For example, when a column was inserted in an EDS, it might be reasonable to insert it into the DW. In an ideal case, the semantics of an EDS schema change should be analyzed before applying a change to the DW. It is also possible to leave an administrator the possibility of configuring what to do in the DW when different types of EDS schema changes occur.

In collaboration with a consultant from a Danish bank, we have discovered the need to version ETL processes, especially to allow importing data from a previous EDS backup. Taking into consideration that multiple changes could have occurred to the ETL processes and the EDSs, it would require us to store previous versions of the ETL processes, versions of the EDSs and their data, and the DWs structures. We have analyzed several works on the DW schema versioning [22–25] and considered that these could serve as starting points for future work . However, in spite of the importance of versioning, it was not our primary concern for this project and we therefore chose to ease our burden by leaving versioning of ETL processes, DWs, and EDSs for future work.

We focused on reparation of ETL processes and put less emphasis on how the semantics of ETL processes are changed by our reparation. However, in some cases it may lead to unintended outcomes. To illustrate this point, consider an example of a SSIS Data Flow task consisting of an OLE DB Source, a Conditional Split, and an OLE DB Destination transformation. The OLE DB Source outputs columns x and y, the Conditional Split has the condition $x > 20$, and the OLE DB Destination receives x and y. If x was deleted, the graph alteration algorithm of MAIME would delete the Conditional Split and the OLE DB Destination from the ETL process cascadingly, leaving only the OLE DB Source (Given that the option *Allow deletion of vertices* is enabled). This occurs since the Conditional Split only uses column x, and since x is deleted, the transformation is invalid. The end result of this ETL process is essentially useless, and we propose in the future to instead only delete Conditional Split and connect OLE DB Source and OLE DB Destination, which would then save y to the DW. The idea behind this solution is to attempt to retain as much functionality from the ETL process as possibly.

Another direction for future work is to continue implementing the SSIS tasks (transformations, script, etc.) available in SSIS and adding support for more EDS schema changes. Currently, we extract only one SSIS Data Flow task from the current SSIS package. An improvement would be to extract all SSIS Data Flow tasks from each SSIS package. We support the following common SSIS transformations for column deletion and column rename: OLE DB Source, OLE DB Destination, Aggregate, Conditional Split, Derived Column, and Lookup. In the future, support for more EDS schema changes such as addition, data type change, nullable or unique constraints could be added. Furthermore, common SSIS transformations like Sort, Union All, Data Conversion could be implemented. Implementing additional SSIS features also includes implementing the SSIS Control Flow tasks which can for example contain loops for Data Flow tasks, which greatly influences the ETL process.

Acknowledgements

We would like to thank our consultant Xuegang ”Harry” Huang for his help of providing essential insight of the problem domain.

References

[1] Ralph Kimball and Margy Ross. *The data warehouse toolkit: The complete guide to dimensional modeling.* John Wiley & Sons, 2011.

[2] Brian Knight, Erik Veerman, Jessica M. Moss, Mike Davis, and Chris Rock. *Professional Microsoft SQL Server 2012 Integration Services.* John Wiley & Sons, 2012.

[3] SQL server. URL `https://www.microsoft.com/en/server-cloud/products/sql-server/`. (Last accessed 21. April 2016).

[4] Eric Thoo and Mark A. Beyer. Gartner’s magic quadrant for data integration tools. 2014.

[5] Ssis designer. URL `https://msdn.microsoft.com/en-us/library/ms137973.aspx`. (Last accessed 31. May 2016).

[6] Redgate software. URL `http://www.red-gate.com/`. (Last accessed 12. May 2016).

[7] DB Comparer. URL `http://dbcomparer.com/`. (Last accessed 12. May 2016).

[8] Schema compare. URL `https://msdn.microsoft.com/en-us/library/hh272690(v=vs.103).aspx`. (Last accessed 12. May 2016).

[9] DDL triggers. URL `https://msdn.microsoft.com/en-us/library/ms175941(v=sql.120).aspx`. (Last accessed 12. May 2016).

[10] Information schema views. URL `https://msdn.microsoft.com/da-dk/library/ms186778.aspx`. (Last accessed 12. May 2016).

[11] Marko A. Rodriguez and Peter Neubauer. Constructions from dots and lines. *Bulletin of the American Society for Information Science and Technology*, 2010. doi: 10.1002/bult.2010.1720360610.

[12] Integration services data types. URL `https://msdn.microsoft.com/en-us/library/ms141036(v=sql.120).aspx`. (Last accessed 27. May 2016).

[13] Visual studio 2015. URL `https://www.visualstudio.com/`. (Last accessed 25. May 2016).

[14] SQL Server Data Tools. URL `https://msdn.microsoft.com/en-us/library/mt204009.aspx`. (Last accessed 25. May 2016).

[15] George Papastefanatos, Panos Vassiliadis, Alkis Simitsis, and Yannis Vassiliou. What-if analysis for data warehouse evolution. *DaWaK*, 2007.

[16] George Papastefanatos, Panos Vassiliadis, Alkis Simitsis, Timos Sellis, and Yannis Vassiliou. Rule-based management of schema changes at ETL sources. *ADBIS*, 2010.

[17] George Papastefanatos, Panos Vassiliadis, Alkis Simitsis, and Yannis Vassiliou. Policy-regulated management of ETL evolution. *Journal on Data Semantics XIII*, 2009.

[18] Artur Wojciechowski. E-ETL: Framework for managing evolving ETL workflows. *Foundations of Computing and Decision Sciences*, 38(2), 2013. doi: 10.2478/fcds-2013-0005.

[19] Artur Wojciechowski. E-ETL framework: ETL process reparation algorithms using case-based reasoning. *ADBIS*, 2015.

[20] Juan Trujillo and Sergio Luján-Mora. A uml based approach for modeling ETL processes in data warehouses. In *Conceptual Modeling-ER 2003*, pages 307–320. Springer, 2003.

[21] Business Intelligence Markup Language. URL `https://www.varigence.com/biml`. (Last accessed 1. June 2016).

[22] Tadeusz Morzy and Robert Wrembel. On querying versions of multiversion data warehouse. In *Proceedings of the 7th ACM International Workshop on Data Warehousing and OLAP*, DOLAP '04, pages 92–101, 2004.

[23] Bartosz Bębel, Johann Eder, Christian Koncilia, Tadeusz Morzy, and Robert Wrembel. Creation and management of versions in multiversion data warehouse. In *Proceedings of the 2004 ACM Symposium on Applied Computing*, SAC '04, pages 717–723, 2004.

[24] Matteo Golfarelli, Jens Lechtenbörger, Stefano Rizzi, and Gottfried Vossen. Schema versioning in data warehouses: Enabling cross-version querying via schema augmentation. *Data & Knowledge Engineering*, 59(2): 435 – 459, 2006.

[25] Danijela Subotić, Patrizia Poščić, and Vladan Jovanović. Data warehouse schema evolution: State of the art. *Central European Conference on Information and Intelligent Systems*, 2014.

APPENDIX

A. PROPAGATION TABLE

This section contains which propagation action should be taken on the graph model of MAIME for a given EDS schema change and a given transformation. As argued in Section 3.2, only a propagation action can alter the graph, and thus is the only action that needs to be specified. For all the propagation actions, it is assumed that the configuration *Allow modification of expressions* is disabled. All propagation actions except those for data type change are specified in Table 5.

B. FULLY DEPENDENT QUALIFICATIONS

This section describes when we consider a transformation to be fully dependent on an output column, as this is used in Algorithm 1. As argued previously, this is specific to each type of transformation, which is represented as a vertex v. Table 6 specifies what it means for each type of transformation to be fully dependent on an output column c.

C. METADATA SNAPSHOT FILE

Code listing 1 is an example of a metadata snapshot file used by MAIME. The file is formatted as JSON. Note that `$id` is an id used to retain references when the snapshot is serialized. The corresponding `$ref` refers to these id numbers, and results in references when the JSON file is deserialized.

Listing 1: Example of metadata snapshot

```
{
  "$id": "1",
  "Database": {
    "$id": "2",
    "ID": 14,
    "Name": "DatabaseTesting",
    "Tables": {
      "$id": "3",
      "245575913": {
        "$id": "4",
        "Columns": {
          "$id": "5",
          "1": {
            "$id": "6",
            "CharacterMaxLength": 0,
            "CharacterSetCatalog": null,
            "CharacterSetName": null,
            "CharacterSetSchema": null,
            "DataType": 10,
            "DateTimePrecision": 0,
            "ID": 1,
            "IsNullable": false,
            "Name": "ID",
            "Ordinal": 1,
            "Precision": 10,
            "PrecisionRadix": 10,
            "Scale": 0,
            "IsPrimaryKey": false,
            "IsUnique": false,
            "Table": {
              "$ref": "4"
            }
          },
          "2": {
            "$id": "7",
            "CharacterMaxLength": 30,
            "CharacterSetCatalog": null,
            "CharacterSetName": "UNICODE",
            "CharacterSetSchema": null,
            "DataType": 14,
            "DateTimePrecision": 0,
            "ID": 2,
            "IsNullable": true,
            "Name": "Name",
            "Ordinal": 2,
            "Precision": 0,
            "PrecisionRadix": 0,
            "Scale": 0,
            "IsPrimaryKey": false,
            "IsUnique": false,
            "Table": {
              "$ref": "4"
            }
          },
          "3": {
            "$id": "8",
            "CharacterMaxLength": 0,
            "CharacterSetCatalog": null,
            "CharacterSetName": null,
            "CharacterSetSchema": null,
            "DataType": 10,
            "DateTimePrecision": 0,
            "ID": 3,
            "IsNullable": true,
            "Name": "Age",
            "Ordinal": 3,
            "Precision": 10,
            "PrecisionRadix": 10,
            "Scale": 0,
            "IsPrimaryKey": false,
            "IsUnique": false,
            "Table": {
              "$ref": "4"
            }
          }
        },
        "FullName": "[dbo].[Person]",
        "Name": "Person",
        "ID": 245575913,
        "Database": {
          "$ref": "2"
        }
      },

    }
  },
  "creationDate": "2016-05-20T09:17:24.8850188+02:00"
}
```

D. REMAINING TEST CASES FOR EVALUATION

This appendix shows the two other SSIS packages that were tested in connection with our evaluation from Section 5. Test case 2 before EDS schema changes is shown in Figure 15 while Figure 16 shows the SSIS package after reparation.

Table 5: Propagation actions for all combinations of EDS chema change and transformation.

EDS change	**Transformation**	**Propagate action**
Addition	Aggregate	Write to log that new column c is available here.
	Conditional Split	Pass new column c through without doing any transformations.
	Data Conversion	Pass new column c through without doing any transformations.
	Derived Column	Pass new column c through without doing any transformations.
	Lookup	If new column c first appears in table being looked up, include c in $v.outputcolumns$. Otherwise, if c originates from preceding vertex, pass c through without doing any transformations.
	OLE DB Destination	Write to log that new column c is available here.
	OLE DB Source	Include new column c in $v.columns$
	Sort	Pass new column c through without doing any transformations.
	Union All	Write to log that new column c is available here.
Deletion	Aggregate	Delete all tuples $(f_i, input_i, output_i, dest_i) \in v.aggregations$ where the deleted column $c = input_i$. Remove all deleted columns $output_i$ from all outgoing edges.
	Conditional Split	Delete all tuples $(expr_i, p_i, dest_i) \in v.conditions$ where the deleted column c occurs in $cond_i \in v.conditions$. Adjust p_i for all other conditions such that p ranges from 1 *to* n where n is the number of conditions. Delete all edges between the Conditional Split and $dest_i$ for all deleted conditions. Remove c from all outgoing edges.
	Data Conversion	Delete all tuples $(input_i, output_i) \in v.conversions$ where the deleted column $c = input_i$. Delete $input_i, output_i$ from all outgoing edges.
	Derived Column	Delete all tuples $(expr_i, output_i) \in v.derivations$ where the deleted column c occurs in $expr_i$. Delete c and all deleted $output_i$ from all outgoing edges.
	Lookup	Delete all join conditions $join_i \in v.joins$ where the deleted column c occurs in. Remove c from $v.columns$, $v.outputcolumns$, and outgoing edges. If no joins are left then delete Lookup and all of its ingoing and outgoing edges.
	OLE DB Destination	Remove c from $v.columns$.
	OLE DB Source	Delete column c from $v.columns$. Delete c from the outgoing edges.
	Sort	Delete all tuples $(input_i, output_i, sorttype_i, order_i) \in v.sortings$ where deleted column $c = input_i$. Delete c from $v.passthrough$ and from the outgoing edge.
	Union All	For all tuples $(output_i, input_i) \in v.unions$, replace all occurrences of c in $input_i$ with ϵ. If all columns in $input_i$ are ϵ, then remove the i-th entry in $v.unions$ and remove $output_i$ from the outgoing edge.
Rename	Aggregate	c is already renamed.
	Conditional Split	For renamed column c, if c occurs in $expr_i$ for any tuple $(expr_i, p_i, dest_i) \in v.conditions$ then rewrite $expr_i$ to reflect new column name.
	Data Conversion	c is already renamed.
	Derived Column	For the renamed column c, rename all occurrences of c in all $expr_i$ that exist in tuples $(expr_i, output_i) \in v.derivations$.
	Lookup	For renamed column c, if c occurs in any join condition $join_i \in v.joins$ or if a column in $v.outputcolumns$, rename all occurrences of c in the graph.
	OLE DB Destination	c is already renamed.
	OLE DB Source	Rename all occurrences of c in the graph.
	Sort	c is already renamed.
	Union All	c is already renamed.

Table 6: Table describing what it means for each type of transformation to be fully dependent on an output column.

Transformation Type	**Vertex v is fully dependent on output column c if:**
Aggregate	There is only one tuple $(f, input, output, dest) \in v.aggregations$, and $output = c$.
Conditional Split	$\forall(expr, p, dest) \in v.conditions$, c is used in $expr$.
Data Conversion	Not applicable. The transformation works without any conversions.
Derived Column	Not applicable. The transformation works without any derivations.
Lookup	$\forall j \in v.joins$, c appears in j.
OLE DB Destination	Not applicable. Never evaluated in Algorithm 1.
OLE DB Source	Not applicable. Never evaluated in Algorithm 1.
Sort	$\forall(input, output, sorttype, order) \in v.sortings$, $c = output$..
Union All	There is only one tuple $(output, input) \in v.unions$, and $c = output$.

The EDS schema changes that were applied to test case 2 are: Deletion of `Sale.TotalAmount`, rename of `Person.ID` to `Person.RenamedID`, and rename of `Store.Address` to `Store.RenamedAddress`. Test case 3 before EDS schema changes is shown in Figure 17 while Figure 18 shows the SSIS package after reparation. The EDS schema changes that were applied to test case 3 are: Deletion of `Person.Name` and rename of `Person.Age` to `Person.RenamedAge`.

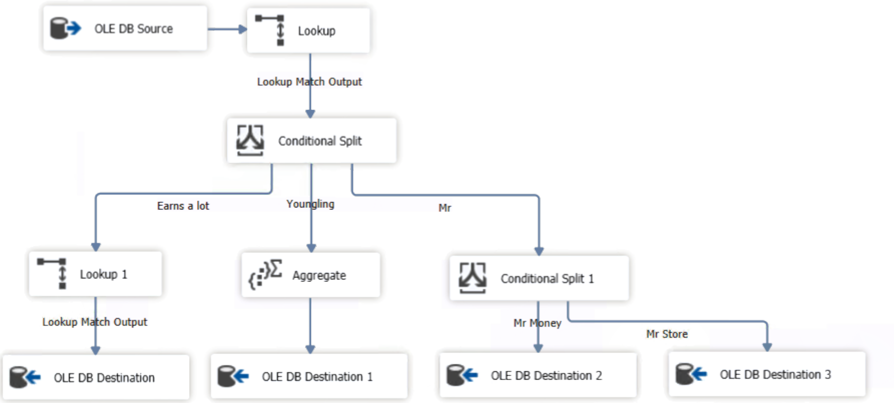


Figure 15: The Data Flow task of the SSIS package for test case 2 before any EDS schema changes.

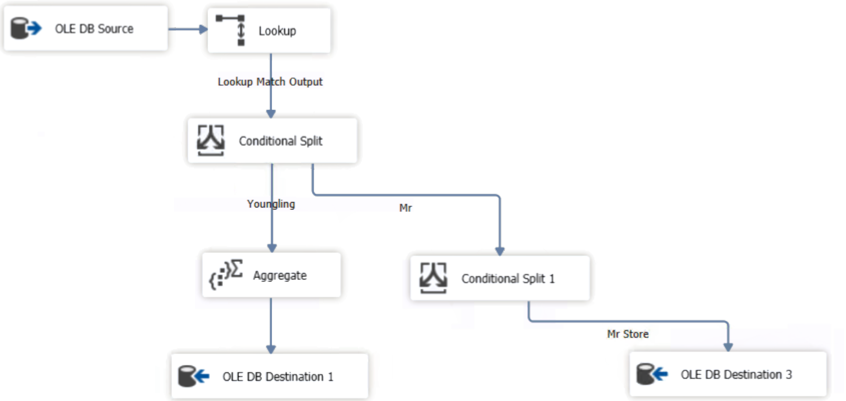


Figure 16: The Data Flow task of the SSIS package for test case 2 after the EDS schema changes.

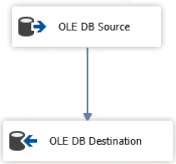


Figure 17: The Data Flow task of the SSIS package for test case 3 before any EDS schema changes.

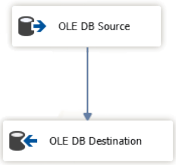


Figure 18: The Data Flow task of the SSIS package for test case 3 after the EDS schema changes.