Frequent Patterns in ETL Workflows: An Empirical Approach

Vasileios Theodorou[a,*], Alberto Abelló[a], Maik Thiele[b], Wolfgang Lehner[b]

[a] *Universitat Politécnica de Catalunya, Barcelona, Spain*
[b] *Technische Universität Dresden, Dresden, Germany*

Abstract

The complexity of Business Intelligence activities has driven the proposal of several approaches for the effective modeling of Extract-Transform-Load (ETL) processes, based on the conceptual abstraction of their operations. Apart from fostering automation and maintainability, such modeling also provides the building blocks to identify and represent frequently recurring patterns. Despite some existing work on classifying ETL components and functionality archetypes, the issue of systematically mining such patterns and their connection to quality attributes such as performance has not yet been addressed. In this work, we propose a methodology for the identification of ETL structural patterns. We logically model the ETL workflows using labeled graphs and employ graph algorithms to identify candidate patterns and to recognize them on different workflows. We showcase our approach through a use case that is applied on implemented ETL processes from the TPC-DI specification and we present mined ETL patterns. Decomposing ETL processes to identified patterns, our approach provides a stepping stone for the automatic translation of ETL logical models to their conceptual representation and to generate fine-grained cost models at the granularity level of patterns.

Keywords: ETL, patterns, empirical, graph matching

1. Introduction

Extract, Transform and Load (ETL) processes are crucial business processes that require heavy investments for their design, deployment and maintenance. With data being increasingly recognized as a key asset for the success of any enterprise, interest is growing for the development of more sophisticated models and tools to aid in data process automation and dynamicity. According to a recent Gartner report [1], the data integration tool market is growing with an impressive rate, with an increase of 10.5% from 2014 to 2015 and an expected total market revenue of $4 billion in 2020.

In this context, ETL requirements are becoming more advanced and demanding with expectations such as self-service BI [2] and on-the-fly data processing making ETL projects even more complex. Moreover, ETL users and developers with different backgrounds, using

*Corresponding author

Email addresses: vasileios@essi.upc.edu (Vasileios Theodorou), aabello@essi.upc.edu (Alberto Abelló), maik.thiele@tu-dresden.de (Maik Thiele), wolfgang.lehner@tu-dresden.de (Wolfgang Lehner)

different models and technologies form a confusing landscape of ETL frameworks and processes that is hard to analyze and harness. The reply from academia has been the proposal of models (e.g., using the business process modeling notation (BPMN) [3] or the unified modeling language (UML) [4]) that classify ETL functionalities in different levels of abstraction, creating common ground for the description of ETL operations and fostering design automation and analysis. On the direction of translating conceptual and logical operations to physical implementations, the design and implementation of large ETL flows is detached from the use of specific technologies and using tested structures and best practices, it can become more reliable, efficient and simple. On the opposite direction, mapping physical to logical and conceptual models allows for the concise representation and reuse of components, as well as different layers of analysis and comparison of ETL flows.

The proposed ETL modeling as well as the ETL logical view generated by different open-source and proprietary tools, expose an ETL workflow perspective that opens the door for the identification and specification of ETL patterns. Although there have been some approaches on ETL patterns, considering most used ETL tasks or the morphology of the complete ETL flow, a bottom up methodology that can identify patterns and apply customized analysis on a given generic set of ETL processes is still missing, making the practical exploitation of ETL patterns difficult.

In this work, we introduce our empirical approach for the identification of ETL structural patterns that are significant for different types of analysis. Based on the different types of ETL operations, we logically model the ETL workflows using labeled graphs and employ graph algorithms to identify candidate patterns and to recognize them on different workflows. For the identification phase, we use frequent subgraph discovery techniques and for the recognition phase, we introduce an algorithm that can perform very well for ETL flows and can scale for the cases of multiple and large flows.

Our approach can be used for the (pre-)evaluation of alternative ETL workflows at design time, without the need for their execution or simulation, but solely by decomposing them to recognized patterns for which quality characteristics are measured during a supervised learning phase. In addition, it can generate fine-grained cost models at the granularity level of patterns. It can also be used to classify reusable and well-defined ETL steps regardless of the implementation technology in order to i) automatically derive more understandable conceptual modeling and visualization of ETLs and ii) improve the reusability and reliability of ETL components in cases of alternative pattern implementations by exposing their characteristics. To showcase our approach, we implement a set of realistic ETL processes defined in the *TPC-DI benchmark* and we show experimental results from applying our methodology on them.

From the same domain (i.e., the TPC-DI benchmark), we adopt as our running toy example an ETL process (see Figure 1) that populates the *FactCashBalances* table during the *Historical Load* phase. In Figure 1, we show the logical view of the ETL process, as it is viewed from the implementation that we developed using the Pentaho Data Integration (PDI) open source tool. The ETL process extracts data from a plain-text file in the Staging Area (*CashTransaction.txt*) and processes one of its fields to remove *time* and keep only *date* information. Consequently, data are joined with *DimAccount* table to obtain the corresponding keys for customers and accounts, after irrelevant fields from both sources have been projected out and data have been sorted. Similarly, data are joined with the *DimDate*

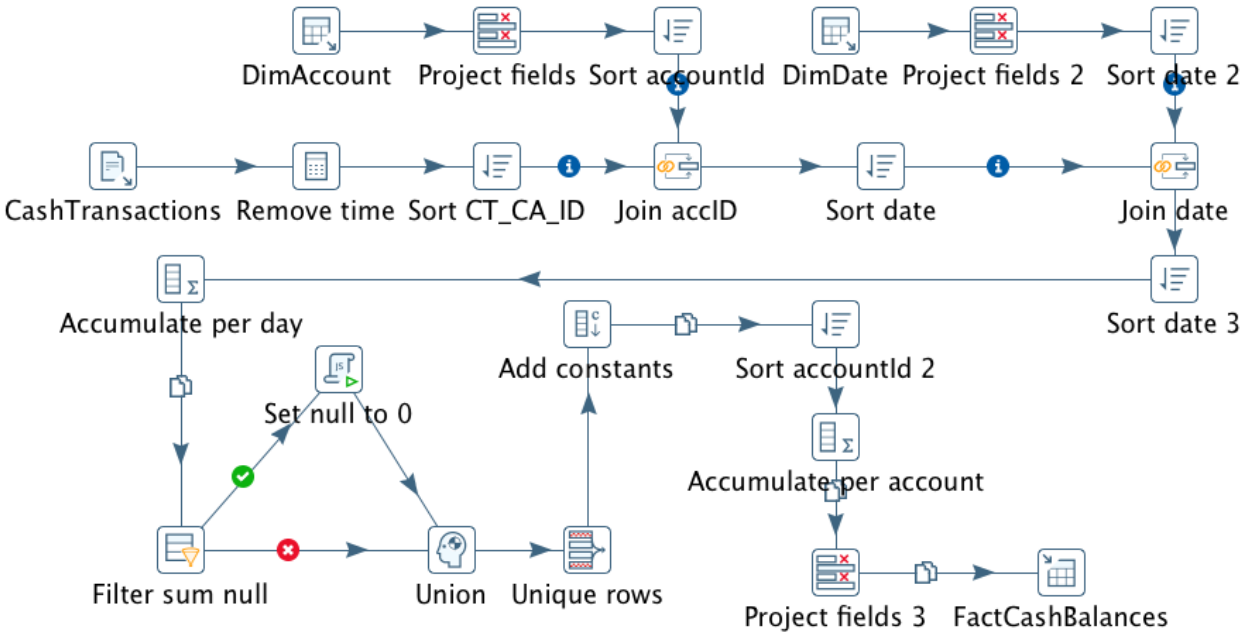

Figure 1: ETL flow example: TPC-DI FactCashBalances population

dimension table to obtain corresponding date information and then they are aggregated on a daily basis. After null values have been replaced by zeros —only for the rows that contain null values— distinct rows are kept and constants (e.g., *effectiveDate*) are added to each row. Finally, data are aggregated per account and after keeping only relevant fields, they are loaded to the *FactCashBalances* table of the DW.

Contributions. Our main contributions are as follows.

- A pattern-based analysis of ETL workflows, using the main control flow patterns from the Workflow Patterns Initiative [1] as a guide.
- A novel empirical approach for mining ETL structural patterns, using frequent subgraph discovery algorithms.
- Adaptation of the VF2 graph matching algorithm with optimizations to perform very well on ETL workflows.
- Presentation of the most frequent ETL patterns, identified in 25 implemented ETL processes from the TPC-DI framework.

Outline. The rest of the paper is structured as follows. Section 2 discusses related work on ETL patterns and Section 3 formalizes the models of ETL frequent patterns and sets the theoretical background for our approach. Section 4 illustrates our architecture and the algorithms that we employ for pattern mining and pattern recognition and Section 5 shows

[1] `http://www.workflowpatterns.com/`

results from applying our methodology on implemented ETLs. Finally, Section 6 presents two interesting use cases of a provided structured methodology while Section 7 concludes the paper.

2. Related Work

There has been considerable work in the area of ETL modeling, in an effort to promote automation through the definition of structural abstractions and systematic methodologies for the design and analysis of ETL models. In [5], ETL activities are formally defined and classified and in [6], such activities are modeled as the interacting steps of workflows in a multi-layered view that bridges conceptual to logical modeling and exposes quality characteristics (denoted as QoX). Similarly to the latter, the authors in [3] adapt the BPMN representation for the conceptual view of the ETL flow and describe a systematic approach of code generation from BPMN models. In the same direction, [4] proposes a UML-based framework where ETL activities constitute the block units of UML activity diagrams among which there is control flow.

The modeling of ETL processes using well defined, reusable components interacting as workflow activities has set the foundations for their pattern-based analysis and design. In the Business Process Management (BPM) community, such analysis has already taken place, with significant work conducted as part of the Workflow Patterns Initiative [7, 8]. This work examines workflows and various different Workflow Management Systems (WfMSs) and identifies a set of recurring features (i.e., patterns). It takes under consideration the modeling languages used for the design and the modeling notation of business process models and extracts a number of patterns to describe mostly control-flow and data-flow semantics commonly offered by WfMSs. In [9], there is a practical illustration of how such patterns can be introduced and integrated into an existing business process model and in [10], the application of patterns on a business process is linked to the strategic decision making level, through its mapping to business goals and non-functional requirements (NFRs).

When it comes to patterns in ETL activities, in [11] there is a profiling of ETL workflows with models called *Butterflies*, based on their form with regards to the distribution of activities relatively to the beginning (i.e., data extraction) and the ending (i.e., data loading) part(s) of the ETL process. Such categorization captures the idea of linking discrete ETL components to ETL requirements, e.g., by providing some indication about their computational or memory needs based on their structural morphology. However, it is not described in detail how the ETL archetypes presented can co-exist as different parts of the same ETL process, nor is any methodology proposed for the quantification of the relationships between Butterflies recognition and implications on the workflow.

More recently, the authors in [12] further extend their line of research [13] on ETL patterns and propose the grouping of ETL operations to abstract their functionality and form known generic ETL activities. To this end, they formulate a pallet of most used data warehousing tasks in real world and propose the design of ETL processes by using workflows that comprise of customizations of these patterns. Although their work fosters reusability and correctness, one important limitation stems from the definition of universal patterns in a top down approach. In this respect, the pattern-based analysis of random ETL workflows that have been designed or implemented with the use of different technologies might not

easily lead to their decomposition in a-priory classified components that are useful for the specific analysis. In other words, this approach assumes that the ETL workflow models comply to some arbitrary-built pattern classification, whereas we advocate that it would make more sense for patterns to be dynamically constructed in an ad-hoc fashion, based on the type of analysis on one hand and the type of examined workflows on the other.

To our knowledge, there is currently no work gathering ETL patterns in an evidence-based manner. Thus, our work is the first one to introduce the idea of mining frequent ETL components to identify valid patterns for the purpose of the analysis, instead of relying on experience or expertise to define generic, universal motifs.

3. ETL Patterns

Abstracting ETL processes on a logical level allows for the identification of recurring structures among the produced workflow models, that can indicate patterns. In this respect, we delve into the well-studied area of *workflow patterns* (WP) [7] and examine their application on ETL workflows, in order to drive insights for the definition of our pattern model, which we subsequently present.

3.1. Workflow Patterns for ETL Flows

In this subsection, we present the basic workflow control-flow patterns [7] that describe control-flow semantics commonly offered by various workflow management systems and we position them in the context of ETL flows.

ETL workflows are data-intensive flows where atomic tasks correspond to ETL operations, for which pipelining plays a crucial role and the smallest unit of data that can flow between them is a tuple. In this regard, we conceptually relate data-flow to control-flow by assuming that the control-flow dependencies refer to processing of data (i.e., tuples) by ETL operations. Of course, we should not exclude the case of blocking operations, where the unit of data that is expected by one operation in order to complete its execution, is a dataset, i.e., a set of tuples that all need to pass from one operation to the other. However, this simply implies some restrictions on the task completion which again produces some tuple(s) as an output to the succeeding operation(s), and thus does not change the generality of our approach. Hence, for one specific tuple, the task (i.e., ETL operation) activation is when this tuple enters this specific task for processing and the task completion is when this task has completed processing this specific tuple or the set of tuples in which it participates, if it is the case of a blocking operation. Following this concept, we perform the analysis below that defines the different workflow patterns in the context of ETL processes:

- Sequence
  Description: A task in a process is enabled after the completion of a preceding task in the same process.
  In ETL context: An ETL operation in a flow begins its execution right after the completion of the execution of a preceding operation in the same flow. The inputs of ETL operations are datasets and thus the smallest token that flows through the ETL is a tuple. In this regard, the Sequence pattern can be regarded in a tuple-by-tuple fashion, translating to: *one tuple will be processed by one ETL operation after*

its processing by a preceding operator has completed. This definition of *sequence* is broad enough to cover both the cases i) when one operation does not have to process all tuples of a dataset before their processing by succeeding operations, allowing for pipelining and ii) when the processing semantics of the operation denote a blocking operator (e.g., *sorter* or *aggregator*).

- Parallel Split
  Description: The divergence of a branch into two or more parallel branches each of which execute concurrently.
  In ETL context: Two or more succeeding ETL operations begin their execution right after the completion of the execution of a preceding operation. The inputs of all these succeeding operations are identical datasets, coming as copies of the output of the preceding operation. For example, multiple ETL operations might perform the same processing of the same datasets at the same time implementing redundant execution. This case is useful i) for improving the reliability of the ETL process, so that even if some component fails, there are others executing identical tasks and the process does not need to terminate with errors and ii) for improving the correctness of the process by crosschecking the output results from identical tasks. Another example of parallel split in ETL processes is when (parts of) the same datasets need to be loaded to different output data sources (e.g., for loading surrogate keys correspondence).

- Synchronization
  Description: The convergence of two or more branches into a single subsequent branch such that the thread of control is passed to the subsequent branch when all input branches have completed execution.
  In ETL context: Two or more ETL operators are succeeded by the same ETL operator, which requires input from all of them in order to begin its execution. Datasets coming from the preceding operators are thus combined in some way by the succeeding operator. Examples of Synchronization within an ETL flow include different types of Joins where the left and right parts of the join operation come from different incoming flows.

- Exclusive Choice
  Description: The divergence of a branch into two or more branches such that when the incoming branch is enabled, the thread of control is immediately passed to precisely one of the outgoing branches based on a mechanism that can select one of the outgoing branches.
  In ETL context: Only one of two or more ETL operations that succeed an ETL operation begins its execution right after the completion of the execution of the preceding operation. The output of the preceding operation is directed (routed) to precisely one of the candidate succeeding operations, based on defined conditions and/or policies. As an example, different tuples can be routed to different operations based on condition evaluations or simply in a round robin fashion.

- Simple Merge
  Description: The convergence of two or more branches into a single subsequent branch

ETL Operator	Related WP
Filter, Single Value Alteration, Project, Field Addition, Aggregation, Sort	Sequence
Router	Exclusive Choice
Splitter	Parallel Split
Join	Synchronization
Union	Simple Merge

Table 1: ETL operators

such that each execution completion of an incoming branch results in the thread of control being passed to the subsequent branch.
In ETL context: Two or more ETL operators are succeeded by the same ETL operator, which begins its execution every time it receives input from any of the preceding operators. Datasets coming from different operators do not need to be combined with each other, but have to conform to specific constraints for their unified processing by the same succeeding operation, e.g., common schema. An example of Simple Merge in ETL flows is the union of datasets coming from multiple different operations.

3.2. ETL Patterns model

Based on the basic WP for ETL, as they were defined above, we can derive a classification of ETL operators that depends on their control-flow semantics. If we further enrich this classification with the processing semantics of each operator, we obtain the classification of Table 1.

This basic set of operators constitutes the building blocks of a vast number of ETL processes logical models, making this number even greater if we consider that operators such as *Single Value Alteration* and *Field Addition* can be based on User Defined Functions (UDF) written in different programming languages. Taking under consideration the topologies that are formed by the way that different operators are connected inside an ETL flow, combined with the type of each operator, we derive structures that are candidate ETL patterns. In order for a candidate pattern to be considered an ETL pattern, it needs to satisfy the following two conditions:

1. It has to occur frequently, i.e., its support with respect to all the (examined) ETL flows has to exceed a threshold value s.
2. It has to be significant for the conducted analysis, i.e., it has to exhibit some important or differentiating behavior that can lead to its characterization, e.g., concerning its functional contribution to the complete workflow or its performance deviation from other parts of the workflow. The former characterization can be performed by an ETL expert and the latter can be assessed by conducting performance analysis.

An ETL *operation* **o** is an atomic processing unit responsible for a single transformation over the input data, having specific processing semantics **ps** over the input data and a specific branch structure defined as the way it connects with neighboring nodes (i.e., operations). We logically model an ETL flow as a directed acyclic graph (DAG) consisting of a set of nodes,

which are ETL operations ($\mathbf{O}$), while the graph edges ($\mathbf{E}$) represent the directed control flow among the nodes of the graph ($o_1 \prec o_2$). Formally:

$o = \mathbf{ps}$
$ETL = (\mathbf{O}, \mathbf{E})$, such that:
$\forall e \in \mathbf{E} : \exists (o_1, o_2), o_1 \in \mathbf{O} \wedge o_2 \in \mathbf{O} \wedge o_1 \prec o_2$

This abstract definition of the ETL flow and its operations allows for the analysis of ETLs independent of the technologies that are used for their implementation and thus enables the mining of ETL patterns from a large number of ETL flows that can easily map to our model. Based on the characteristics of each operator $\mathbf{o}$, it can be mapped to one *label* $\mathbf{l}$ from a predefined set $\mathbb{L}$ through the surjective function *label*. Formally:
$label : O \mapsto \mathbb{L}$

A *Pattern Model* $\mathbf{PM}$ would then be a DAG where its nodes $\mathbf{PN}$ have a specific label $\mathbf{l}$ and a specific branch structure. Formally:
$pn = \mathbf{l}$
$PM = (\mathbf{PN}, \mathbf{E})$, such that:
$\forall e \in \mathbf{E} : \exists (pn_1, pn_2), pn_1 \in \mathbf{PN} \wedge pn_2 \in \mathbf{PN} \wedge pn_1 \prec pn_2$

We assume that only coherent structures make sense for our analysis and thus pattern models are *connected graphs*, i.e., graphs for which, if we ignore directionality there is a path from any of their nodes to any other node in the graph. We should note here that based on different analysis requirements there can be different definitions of mappings (i.e, mapping functions and sets of symbols), mapping one operator to one label. For instance, an operator can be mapped to a label, based solely on its input and output cardinality or based on its operation type. We have found that the latter case can produce useful results and hence that is the analysis that we use in our work. Thus, the labels that we use for our analysis are within a set $\mathbb{OT}$, where $\mathbb{OT} \subseteq \mathbb{L}$ and each element $ot \in \mathbb{OT}$ refers to the *operation type* of the operation and hence can take values from the classification of ETL operators in Table 1.

In Figure 2, we present the conceptual model of the pattern model and how it relates to the ETL flow. A more detailed description of Figure 2 is as follows: Using a mapping function, we can *semantically annotate* the elements (*EFGElements*) of an ETL Flow Graph, i.e., the edges (*EFGDirectedEdges*) and the nodes (*EFGOperationNodes*) by assigning them with corresponding labels and thus producing *Annotated EFG Elements*. A collection of such elements (i.e., a graph containing annotated edges and annotated nodes) can then form an *Annotated EFG Elements Subgraph* $\mathbf{aees} \in \mathbb{AEES}$, which is an occurrence of a pattern model $\mathbf{pm} \in \mathbb{PM}$ iff each and every element $\mathbf{e}$ from the subgraph corresponds to an element $\mathbf{pe}$ from the pattern model, through a bijective function *occurrenceOf*. Formally:
$occurrenceOf : \mathbb{AEES} \mapsto \mathbb{PM}$.

Notice that according to this model, edges between ETL operators can also be mapped to labels. This assumption has been made for completeness and because there can be some practical cases of ETL models where edges can be differentiated according to the manner that datasets flow from the source node to the target node (e.g., *copy-edges* can refer to the case when all outgoing edges from one source node copy the same datasets to all target nodes and *distr-edges* to the case when datasets are distributed among target nodes). However, for our analysis we consider all edges to be of the same type, i.e., that there is only one label to

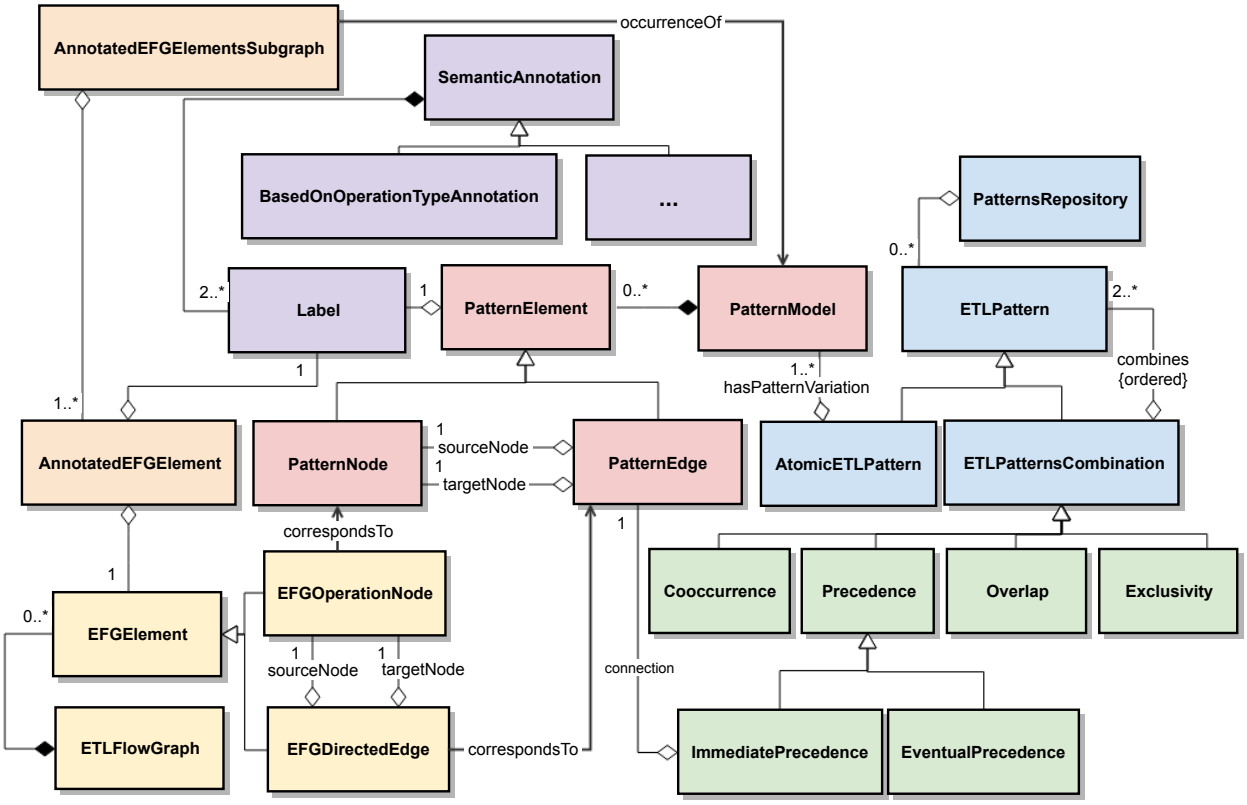

Figure 2: ETL Pattern Conceptual Model

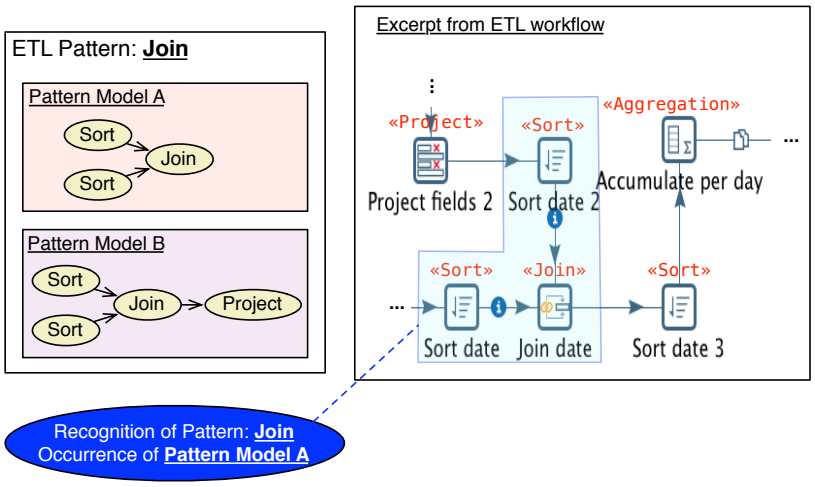

Figure 3: Pattern Model and Pattern Occurrence on an ETL workflow

characterize all edges.

As mentioned above, for our analysis we use the semantic annotation (i.e., labeling) based on operation type (*BasedOnOperationTypeAnnotation*), but there can be different kinds of semantic annotation (i.e., subclasses of the class *SemanticAnnotation*), based on the conducted analysis. As is illustrated in Figure 2, the same pattern (*AtomicETLPattern*) can have variations, resulting to different corresponding pattern models in Figure 3). For instance, two different pattern models can correspond to the same ETL functionality, so if the analysis purpose is the clustering of operations based on their functionality, these two models will constitute variations of the same pattern. An example is illustrated in Figure 3), where we show how the ETL pattern *Join* can have two different pattern models (i.e., *Pattern Model A* and *Pattern Model B*) and how by finding the occurrence of one of these models (Pattern Model A) on the excerpt from our example ETL process from Figure 1, we can recognize it as an instance of the ETL pattern. Furthermore, a combination of two or more patterns (*ETLPatternsCombination*) can itself be a pattern. In this respect, two (or more) patterns can be combined in the following ways, forming a new pattern:

- *Overlap*: Patterns can overlap, with their pattern models sharing elements or with elements of one pattern model located inside the other. This case also includes *pattern nesting*, where one pattern is located inside the other.
- *Precedence*: One pattern is located (right) after the other.
- *Cooccurrence*: Both patterns occur in the same ETL flow.
- *Exclusivity*: Only one of the patterns can occur in the ETL flow and not the other(s).

It should be noticed that the participation of ETL patterns in ETL Pattern Combinations entails a concrete role for each pattern in the combination and this is denoted by the characterization of the *combines* association as *ordered*. Despite our approach allowing for the occurrence of overlapping patterns and patterns one after the other (i.e., precedence),

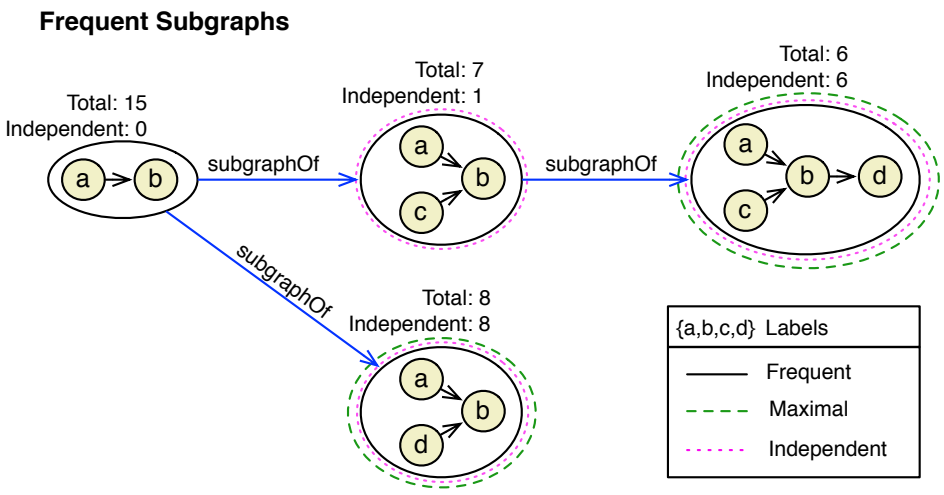

Figure 4: Maximal and Independent Frequent Subgraphs

we do not consider the case of the combinations themselves being patterns (i.e., we only consider *Atomic ETL Patterns*).

3.3. Frequent ETL Patterns

As mentioned above, one of the conditions for a candidate pattern to be considered a pattern is that its support has to exceed some predefined value. In other words, its corresponding pattern model(s) have to occur frequently over the entire set of examined ETL workflows. Since both the ETL workflows and the pattern models are represented using graphs, the problem of mining such patterns can be examined under the prism of *frequent subgraph mining*, which is a subclass of *frequent itemset discovery* [14], where the goal is to discover frequently occuring subgraphs within a set of graphs or a single large graph, with frequency of occurrence above a specified threshold value.

For the purpose of our analysis, we are not interested in frequent subgraphs that *always* appear inside other, bigger frequent subgraphs (i.e., pattern nesting). In this respect, we define a maximality condition: A frequent subgraph (SG1) is *maximal* when there exists no frequent subgraph (SG2) of bigger size, where (SG1) is a proper subgraph of (SG2). Formally:

$isMaximal(SG1) \iff isFrequent(SG1) \wedge \nexists SG2$ such that $isFrequent(SG2) \wedge SG1 \subset SG2$

On the contrary, we define *independent frequent subgraphs* as frequent subgraphs that occur at least once not nested inside the occurrence of another frequent subgraph. Formally: $isIndependent(SG1) \iff isFrequent(SG1) \wedge \exists occurrence(SG1)$ such that ($\nexists SG2$ such that $isFrequent(SG2) \wedge occurrence(SG1) \subset occurrence(SG2)$)

Figure 4 shows the distinction of these two concepts through an example. The two numbers on each frequent subgraph respectively denote the total number of occurrences and the number of independent occurrences (i.e., not inside the occurrence of another frequent subgraph).

4. Architecture

In this section, we present our approach on mining ETL patterns and subsequently, recognizing instances of them on arbitrary ETL workflows. For the latter, we present our algorithm and we conduct an analysis regarding its complexity.

4.1. Pattern mining

The first step for mining patterns of interest is the identification of reoccurring structures over a set of ETL workflows. As mentioned above, we can view our problem as an application of *frequent itemset discovery*, where the goal is to discover frequently occuring "items" within a set of "baskets". Since we logically model ETL workflows as graphs, it is natural for this stage to use graph mining techniques, i.e., techniques for extracting statistically significant and useful knowledge from graph structures. Given a set of graph representations of ETL workflows, the reoccurring structures of interest are graphs, that occur as subgraphs of the initial graphs more frequently than a specified threshold percentage. Thus, we can employ the use of algorithms from the well studied area of *Frequent Subgraph Mining* (FSM). In [18], there is a detailed review of this mature research area, including the main research challenges and the most interesting proposed solutions.

For our experiments, we decided to use the *FSG* algorithm [19], because of 1) its computational efficiency and 2) fast and reliable results from testing that we conducted using its available implementation. This algorithm generates candidate frequent subgraphs in a bottom-up approach, starting with initial subgraphs of one edge and adding one edge at each step while checking for the frequency criterion. It performs significant pruning to the problem space while searching for patterns, taking advantage of the fact that *if a graph is frequent, then all of its subgraphs are also frequent.*

Two parameters can change the output of the algorithm. Firstly, the support threshold, i.e., the minimum number of ETL flows that need to contain a subgraph for it to be accounted as frequent, can vary. In addition, we can select whether we are interested only in *maximal* subgraphs (see subsection 3.3).

After the identification of frequent patterns, the next step is their filtering in order to maintain only the patterns of some value. In this respect, a first filtering is performed by keeping only *independent* subgraphs (see subsection 3.3). To this end, all the instances of all the frequent subgraphs are recognized within the initial set of ETL workflows, using the pattern recognition algorithm defined below (subsection 4.2) and graph-subgraph relationships among these instances are analyzed. Subsequently, frequent subgraphs are classified as variations of ETL patterns, based on the conducted analysis. For instance, experts can classify these subgraphs according to their conceptual functionality, or performance evaluation can be conducted to classify subgraphs based on their isolated performance as compared to the performance of the complete ETL. The results of such analyses are then stored in a repository of ETL patterns.

4.2. Pattern recognition

Once a knowledge base of ETL patterns has been built, occurrences of those patterns can be recognized in any arbitrary ETL workflow. The workflow first needs to be transformed to its graph representation and then the task is reduced to finding a correspondence between the

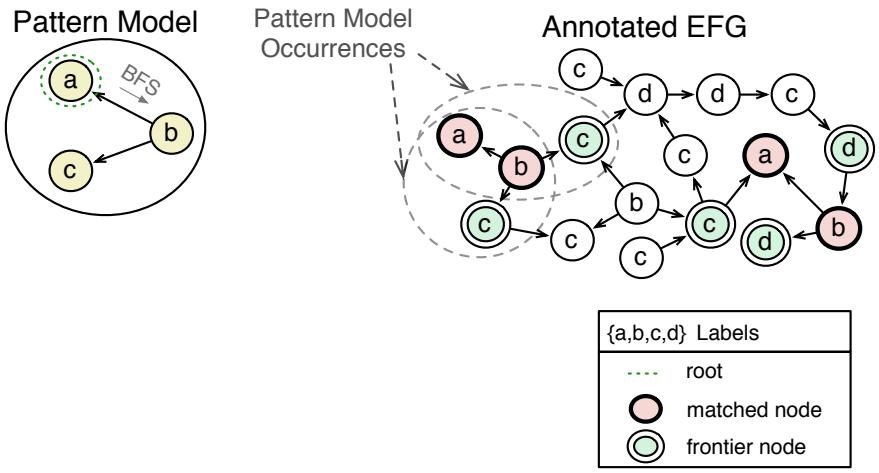

Figure 5: Execution of Find Pattern Model Occurrences Algorithm

model (i.e., the ETL pattern model) and part(s) of the ETL workflow graph representation. Since both the model and the examined ETL are modeled as graphs, this is a typical use case for a graph matching algorithm. In [20], there is an interesting comparison between the implementation in a common codebase of five state-of-the-art subgraph isomorphism algorithms and in [21], there is a comprehensive review of different techniques that have been proposed for this NP-complete problem, the most popular being the Ullmann's algorithm [22] and the VF2 algorithm [23]. After studying these algorithms, we decided to adapt the VF2 algorithm with some optimizations (Algorithm 1), maintaining different data structures while searching for pattern model occurrences. Our algorithm can perform very well for graphs with the cheracteristics of ETL workflows — i) very small branching factors, ii) number of different *labels* comparable to the average graph size.

Taking under consideration the VF2 heuristic of adding to the search space only adjacent nodes while generating candidate matches, our algorithm iterates over the nodes of the pattern model in a breadth-first search (BFS) manner while at the same time matching them with nodes from the annotated EFG, that satisfy certain conditions (see Figure 5). The candidate node matches are searched only within adjacent (i.e., neighboring) nodes from the already matched nodes, which we call *frontier nodes* and for a specific candidate pattern match, if there is no adjacent node that satisfies the conditions, it is dismissed. New candidate pattern matches commence after every iteration, until the pattern model has been fully traversed or the set of candidate pattern matches is empty.

One practical heuristic that we use to speed up the execution of the algorithm by keeping the number of initial candidate node matches on the annotated EFG as small as possible, is the execution of preparation steps, through which the root of the BFS, (i.e., the node of the pattern model from which the iteration will start) is selected to be annotated with the label from the pattern model that is least frequently found on the annotated EFG. The EFG is traversed once and the number of occurrences of each label is stored in *count* —a HashMap object that maps labels to integers (i.e., labels' frequencies). One of the nodes of the pattern model that have the label with the minimum frequency in *count*, is selected as

the subsequent BFS starting point (i.e., the root).

Algorithm 1 Find All Pattern Model Occurrences

Input: ETL, PM, root ▷ *root* is a node from the PM
Output: $\mathbb{PO}$

1: $\mathbb{PO}, \mathbb{PO}_{\not\models}$,m,f $\leftarrow \emptyset$; candOc $\leftarrow$ []; ▷ initializations
2: **for each** $o_i \in$ ETL **do** ▷ iterate nodes of ETL
3: **if** (sameProperties(o_i,root)) **then** ▷ check for match
4: m $\leftarrow$ {[o_i,root]}; ▷ add to set of matches for specific occurrence
5: f $\leftarrow$ *neighbors*(o_i); ▷ add adjacent nodes to the frontier
6: $\mathbb{PO}$.*add*([m,f]); ▷ add candidate occurrence to $\mathbb{PO}$
7: **end if**
8: **end for**
9: **for each** $pn_n \in$ BFS_order(PM,root) **do** ▷ iterate nodes of PM
10: **for each** $co_l \in \mathbb{PO}$ **do** ▷ iterate candidate occurrences in $\mathbb{PO}$
11: **for each** $o_m \in co_l.f$ **do** ▷ iterate nodes from frontier
12: **if** (sameProperties(o_m,pn_n)) **then** ▷ check for match
13: m $\leftarrow co_l.m \cup \{[o_m, pn_n]\}$ ▷ add to matches
14: f $\leftarrow (co_l.f \setminus \{o_m\}) \cup$ *um-neighbors*(o_m); ▷ update frontier
15: $\mathbb{PO}_{\not\models}$.*add*([m,f]); ▷ add updated occurrence to $\mathbb{PO}_{\not\models}$
16: **end if**
17: **end for**
18: **end for**
19: $\mathbb{PO} \leftarrow \mathbb{PO}_{\not\models}$; $\mathbb{PO}_{\not\models} \leftarrow \emptyset$; ▷ copy and initialize for next iteration
20: **end for**
21: **return** $\mathbb{PO}$;

Given an **ETL**, a pattern model **PM** and its **root** (see for example the node with label *a* from the pattern model in Figure 5), Algorithm 1 returns a set $\mathbb{PO}$ of pattern occurrences, that contains all the occurrences of PM in ETL. Each element of this set consists of two parts: i) *m*, which is a set of matches, matching one node from ETL to one node from PM and ii) *f*, which is a set of nodes from the ETL, to maintain all the frontier nodes for one pattern occurrence, i.e., the search space for the subsequent iterations. First, the algorithm iterates over all the nodes of the ETL (Step 2) and finds all the nodes that have same properties as *root* (Step 3). All these nodes then act as the initial node matches around which pattern occurrences are searched (see for example the search space of the annotated EFG in Figure 5, where pattern occurrences are searched only around the two nodes with label *a*). The matches are added to the set *m* (Step 4) and the neighbors of these nodes (i.e., their adjacent nodes in the ETL) are added to the set *f* that maintains the *frontier* for the search around each candidate occurrence (Step 5). Candidate pattern occurrences with these characteristics are added to the set $\mathbb{PO}$. Subsequently, the pattern model is traversed in a BFS order (Step 9), ignoring the directionality of the graph and excluding the root since it has already been matched. At this point, we use a second heuristic to speed up the execution of the algorithm: The order by which the pattern model is traversed depends on the frequency of its labels on the ETL, which we have already collected in the *count* object.

Thus, when a pattern node has multiple unvisited neighbors (i.e., adjacent nodes that have not yet been visited by the BFS), the order by which they are visited depends on their label —from least to most frequent label in the ETL. For each candidate occurrence, each node from its frontier is checked for having the same properties with the current node from the pattern model (Step 12). If it does, then the matches and the frontier of the current candidate occurrence are updated (Steps 13 and 14). Regarding the frontier update, the *um-neighbors*, i.e., the adjacent nodes to the matched node from the ETL that have not already been matched to a pattern node, are added to the existing nodes in the frontier and the matched node is removed from the frontier (Step 14). Subsequently, a new candidate occurrence with these updated parts is added to a set $\mathbb{PO}_{\not\in}$. In every iteration this set replaces the old set $\mathbb{PO}$ that gets initialized to the empty set (Step 19). We should note that despite the pattern model graph being traversed ignoring its directionality, when the check for same properties between a pattern model node and an ETL node takes place, the directionality is taken under consideration.

4.2.1. Algorithm complexity

Although subgraph isomorphism is well-known NP-hard problem [24], in practice, our algorithm can execute very fast because of the particularities of ETLs and the heuristics that we use. The adjacent nodes for each node are maintained inside two hashmap objects that map labels to nodes —one hashmap for the *incoming* adjacent nodes and one for the *outgoing* adjacent nodes [25]. If n is the size of the ETL graph, then each of these two objects is of maximum size $(n-1)$ for each ETL node (a node cannot be adjacent to itself and a node cannot be found in two different buckets of the hashmap because it only has one label), thus the space complexity of the algorithm is: $2*(n-1)*n = \mathcal{O}(n^2)$.

When it comes to the time complexity, due to the use of the hashmap objects, the check for same properties between the nodes of the pattern model and the ETL can take place in constant time t. Thus, if m is the size of the PM, the time complexity is: $t * \sum_{i=1}^{m} N_i$, where N_i is the running number of candidate occurrences (i.e., the size of $\mathbb{PO}$, see Step 10 of Algorithm 1) during each iteration of the BFS. We should note that N_1 is the number of initial candidate occurrences which is determined by the selection of the pattern model *root* element. The growth rate $\frac{N_{k+1}}{N_k}$ of the solution space depends on the fanout of the nodes of the ETL graph on one hand; and on the distribution of the different labels on the ETL nodes, on the other. In other words, the search space grows by being multiplied by the number of neighbors of each node in the candidate occurrence frontier, which is being matched (see Step 14 of Algorithm 1), but it also shrinks at the same time by pruning nodes that do not match the corresponding node from the PM. In the worst case of the ETL being a clique where all the labels of the pattern model and all the labels of the ETL graph are the same one label, there will be no pruning and thus, every time a new node from the pattern model is visited, the number of candidate occurrences will multiply by $(n-p)$ where p is the number of already matched nodes, until all m nodes are visited. Thus, in the worst case the total number of candidate occurrences during the last iteration will be: $N_m = \prod_{i=1}^{m}(n-i) = \mathcal{O}(\frac{(n-1)!}{(n-1-m)!})$. However, according to our experience with implementing ETL workflows from the TPC-DI benchmark, this is hardly a realistic case for ETL graphs, where the branching factor is close to 1. In addition, the existence of a number of different labels in real ETL graphs, guarantees that a lot of pruning takes place, especially in the common case where no (unmatched) node

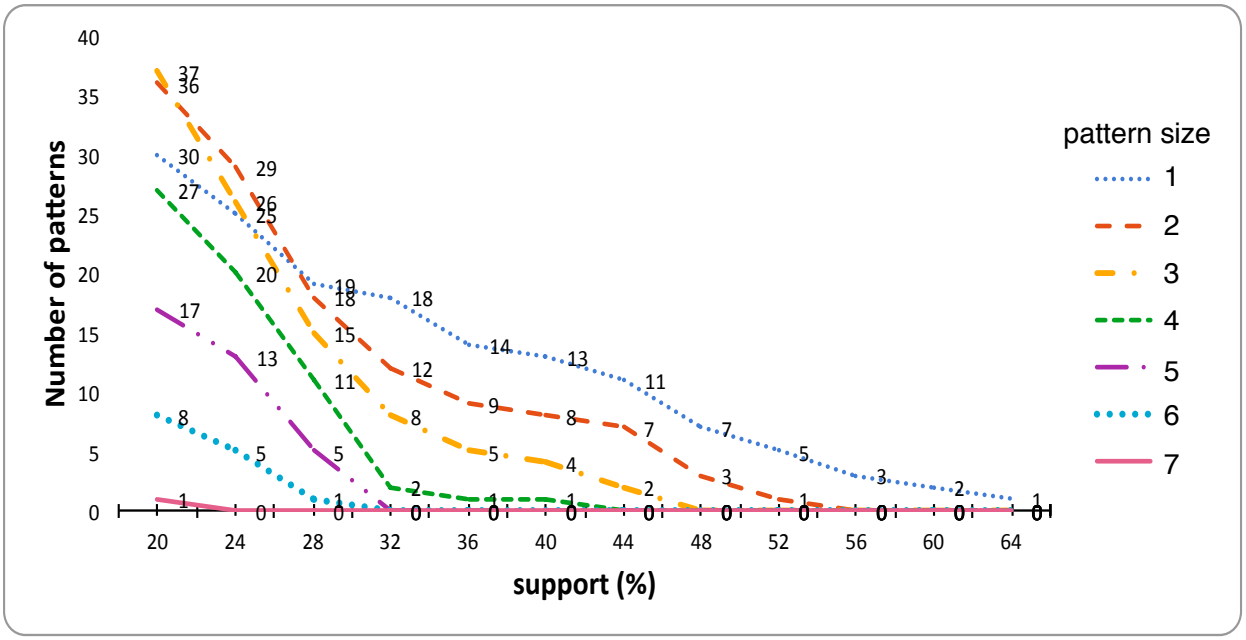

Figure 6: Number of (non-maximal) frequent patterns identified for different support values and for different pattern sizes

of specific label is adjacent to a candidate occurrence, which can very easily be checked, with the bucket corresponding to this label being empty. Furthermore, the size m of the meaningful pattern models is usually very small (< 10).

5. Results

In this section, we show results obtained from the application of the algorithms to 25 ETL processes from the TPC-DI benchmark[2] that we implemented using the Pentaho Data Integration open source tool[3].

5.1. Mined ETL Patterns

In this subsection, we present our results from mining frequent patterns during the *Learning Phase* of our approach. To this end, we used the *FSG* algorithm [19] on the graph representation of the 25 TPC-DI ETLs, using its available implementation[4]. In Figure 6, we show the number of frequent patterns of different size (i.e., number of edges) that we obtain, using different values for support (i.e., the minimum proportion of ETL workflows that need to contain a subgraph for it to be accounted as a frequent pattern). It should be noticed that the FSG algorithm executed in less than 2 *msec* for all these cases. As expected, since we are not imposing the maximality constraint, as the support increases, the number of identified

[2] `http://www.tpc.org/tpcdi/`
[3] Full implementation available at: `https://github.com/AKartashoff/TPCDI-PDI/`
[4] `http://glaros.dtc.umn.edu/gkhome/pafi/overview`

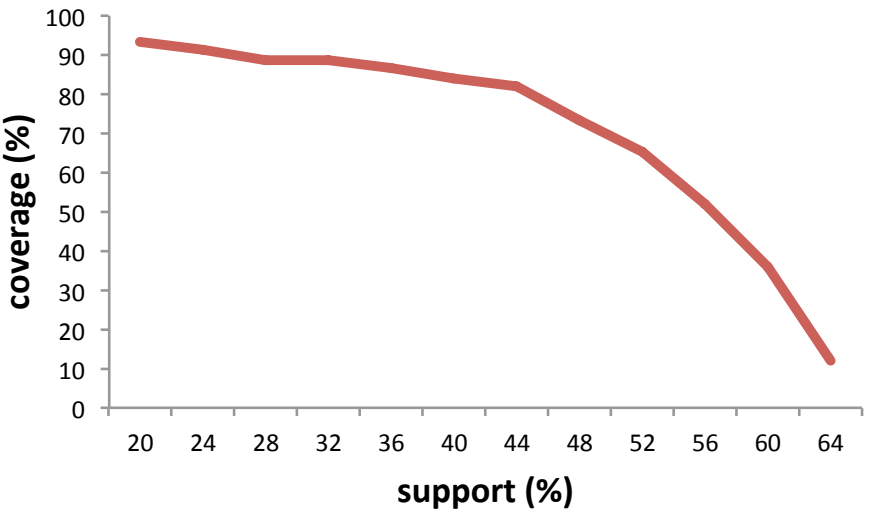

Figure 7: Coverage of ETL workflows for different support values

patterns decreases. In other words, all the patterns that are identified with some support s will also be identified with any other smaller support, plus additional patterns that do not satisfy the frequency criterion for s. We can also infer from Figure 6 that in general, as the size of the patterns decreases, the number of identified patterns increases. However, as can be seen, there are some noticeable exceptions to this rule. The curve for pattern size 1 crosses both with the curve for pattern size 2 and with the curve for pattern size 3. The reason is that the same pattern of size 1 can be a subgraph of two or more patterns of size 2. As an example, let us consider a set of three labels $\{a, b, c\}$ and the identification of the following frequent patterns of size 2: i) $[a - a - b]$, ii) $[a - b - a]$, iii) $[a - a - c]$ and iv) $[a - c - a]$. Each subgraph of these *four* frequent patterns will also be a frequent pattern for the same support s. However, there are only *three* distinct subgraphs of size 1 among these patterns —i) $[a - a]$, ii) $[a - b]$ and iii) $[a - c]$— and it is not necessary that there exist more frequent patterns of size 1 with these labels. The same explanation can be given for the case of the curve for pattern size 2 crossing with the curve for pattern size 3. The reason that the rule holds for greater s values, is that this explained behavior is outgrown by the tendency of larger patterns to be more difficult to find frequently. Finally, we can observe that beyond some support value, there is no frequent pattern identified.

In Figure 7, we show for different support values, the coverage of all the ETL workflows from the patterns identified, i.e., the percentage of ETL operations that take part in pattern model occurrences. Pattern identification was conducted by our implementation of the algorithm proposed in Subsec. 4.2[5]. As we can observe, the coverage decreases as the support value increases, which is an expected behavior since the overall number of identified patterns decreases as well (see Figure 6). This decrease appears to be non-linear and especially beyond some value s ($\approx 45\%$) for which coverage is $\approx 80\%$, it appears to decrease faster and faster as support increases. Another interesting observation is that for a small value of s,

[5]Full implementation available at: `https://github.com/theovas/etl-patterns/`

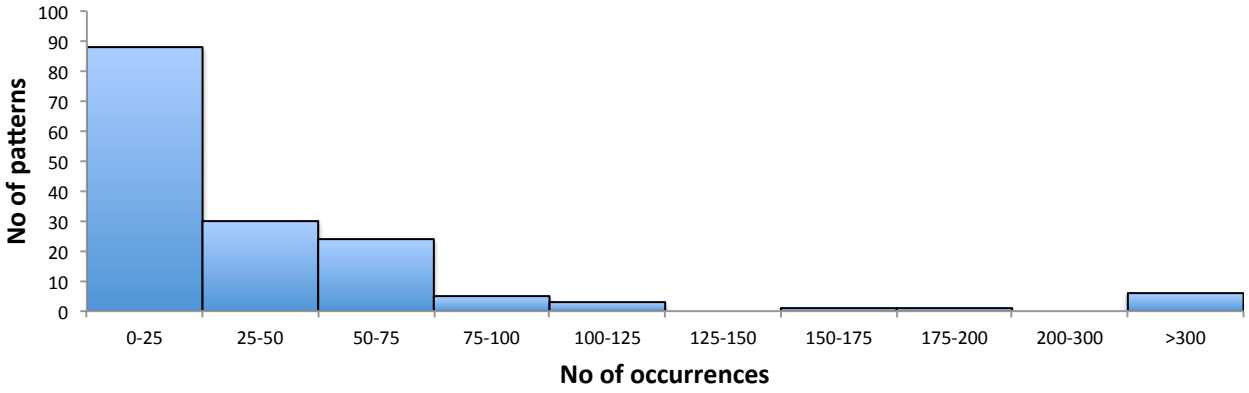

Figure 8: Number of patterns w.r.t. their number of occurrences

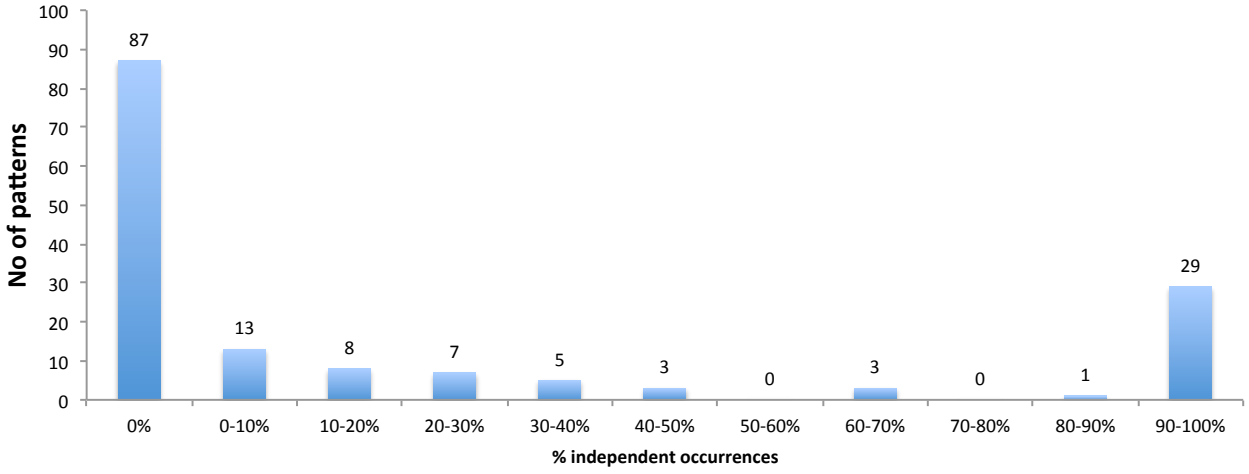

Figure 9: Number of patterns w.r.t. their frequency of independent occurrences

coverage reaches a very high value, above 93%. This validates our claim that it is possible to map, if not the complete, at least a very large part of an ETL workflow to pattern model occurrences, facilitating the translation to its conceptual representation, as is explained in Subsection 6.3.

Using the support value 20% (i.e., pattern model occurs at least in 5 of the 25 ETLs), we obtained 156 pattern models of different sizes, from 1 to 7 edges. Subsequently, we employed our pattern recognition algorithm to find all the occurrences of each of these pattern models on the 25 ETLs and the results about the number of occurrences for all the pattern models are shown in Figure 8. It is clear that some pattern models occur much more frequently than the others, but there is also a big difference between all occurrences and only independent occurrences for each pattern model. This is illustrated in Figure 9, where we show the number of pattern models for different ranges of % of independent occurrences. We can see that there are 87 pattern models with 0% independent occurrence i.e., out of all their occurrences, they never occur independently, not nested inside the occurrence of another pattern model. These pattern models are irrelevant for our analysis and thus we only keep

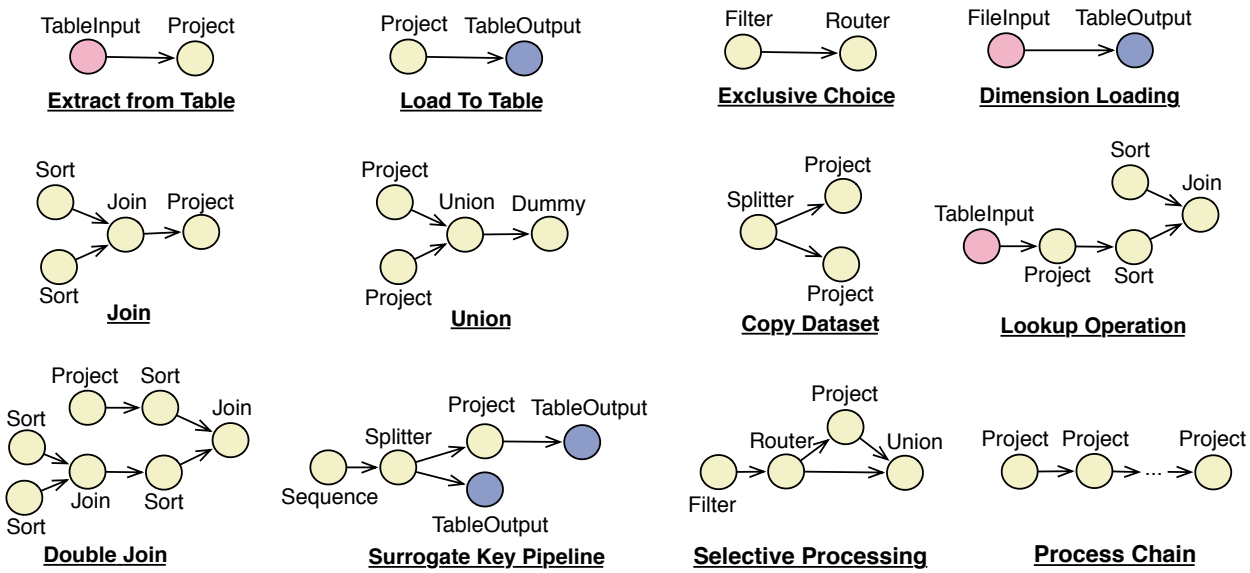

Figure 10: Frequent ETL pattern models

the remaining $156 - 87 = 69$ pattern models.

After examining these 69 pattern models playing the role of the ETL expert, we concluded with a set of ETL patterns that we characterized based on their functionality, the most interesting of which are illustrated in Figure 10. For each pattern, we show only one pattern model —the one occurring most frequently— but as explained above, each pattern can have two or more variations, as was the case with the patterns that we identified. The variations that we identified for each pattern, only differ in a very small number of model nodes (usually only one node is different) and thus it is possible to implement clustering algorithms based on distance criteria and automatically cluster the different pattern models as pattern variations. We also observed that there are frequent pattern models of small size, which can be combined to synthesize bigger frequent pattern models. As we look from frequent pattern models of smaller size to frequent pattern models of bigger size, the conceptual functionality of the pattern model and its contribution to the ETL process becomes more obvious and concrete.

5.2. Performance Evaluation of Graph Matching Algorithm

In this subsection, we present the performance results from running the implementation of our graph matching algorithm (Algorithm 1). To this end, we implemented a synthetic ETL generator[6] that generates ETLs of preferred size and uses statistical characteristics of the 25 TPC-DI ETLs as follows: We parsed the TPC-DI ETLs and stored, for each operation type, i) the average number of succeeding operations and ii) the percentage of succeeding occurrences for each operation type. For example, we found that out of all the 293 succeeding operations of all the operations of operation type *Project*, only 12 operations are of type *TableOutput* and thus the probability of our generator generating an operation

[6]Implementation available at: `https://github.com/theovas/etl-patterns/blob/master/utils/SyntheticFlowGenerator.java`

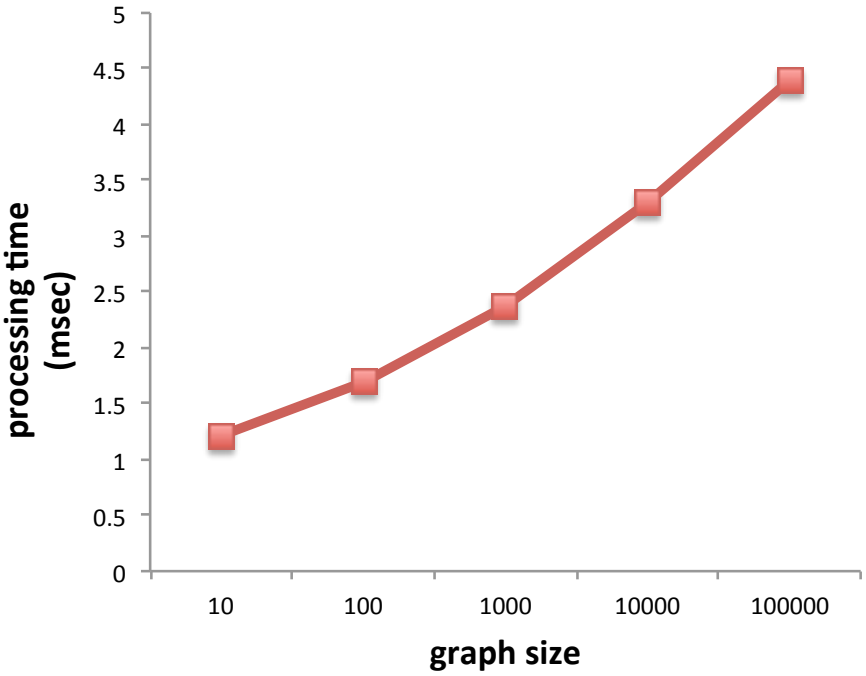

Figure 11: Performance of the graph matching algorithm for ETLs of different sizes (Y-axis in log scale)

of type *TableOutput* after an operation of operation type *Project* is set to: $12/239 = 5\%$. In order to be able to adjust the size of the produced ETL, our generator can modify the percentage of new operations that are of joining type being generated, as opposed to merging parts of the flow with existing operations of joining type. In the same respect, the probability of output operations (i.e., operations that have zero fanout) can be modified dynamically. We generated ETLs of different sizes and for each ETL, we executed our graph matching algorithm for all of the 156 frequent pattern models on a row. The performance results from these experiments, carried under an OS X 64-bit machine, Processor 965 Intel Core i5, 1.7 GHz and 4GB of DDR3 RAM, are shown in Figure 11. As can be seen, our algorithm performs very well (almost linearly) for ETLs sharing characteristics with the TPC-DI ETLs, even for graphs of size 10^5.

5.3. Granular ETL Performance Evaluation

In subsection 6.2, we claimed that with the use of our approach, ETL workflows can be evaluated with regards to their quality characteristics at the granular level of patterns. In this subsection, we show how such an evaluation can take place for the ETL performance. Thus, we implemented a process that isolates pattern model occurrences (**pmo**) and executes them multiple times to obtain their average execution time, using as input data that are generated from the provided TPC-DI data generator[7] with scale factor 1. This process corresponds to the learning phase that we mentioned in subsection 6.2. To this end, the output of each *incoming* operator to the **pmo** (i.e., all the operators that are not part of the **pmo** but are adjacent to at least one node from the **pmo** with direction towards that node) is stored into

[7] `http://www.tpc.org/tpc_documents_current_versions/current_specifications.asp`

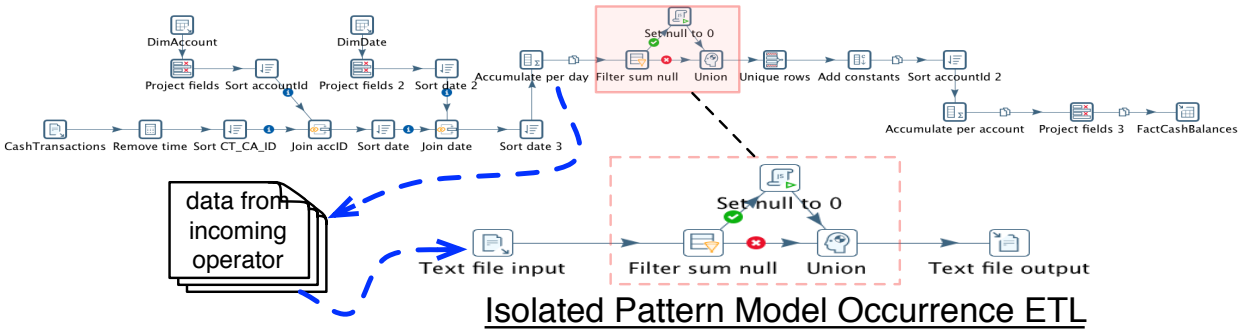

Figure 12: Creating an ETL by isolating a pattern model occurrence

a text file and an *file input* operator is added to the **pmo**, that reads this text file and passes on its data to the pattern nodes. In addition, we added *file output* operators for each edge for which the source is a node from the **pmo** but the target is a node that is not in the **pmo**, as can be seen in Figure 12.

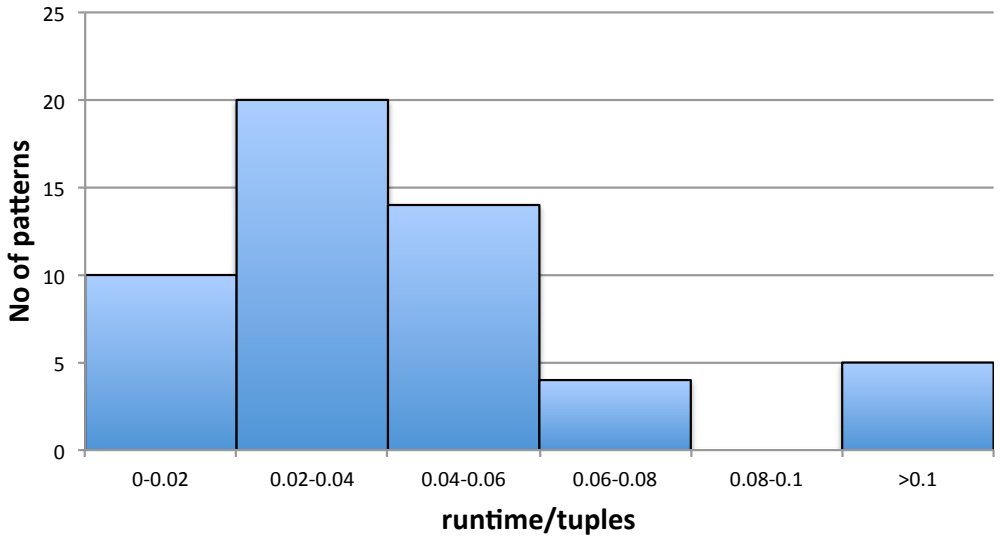

Figure 13: Histogram of number of patterns w.r.t. average values of runtime divided by input size

Executing pattern model occurrences from 6 TPC-DI ETLs, we found that the results were adequate to expose the fluctuation in the performance of different pattern models. Results are shown in Figure 13, where we show a histogram, the y-axis being the number of patterns and the x-axis being the average value for runtime (in *msec*) divided by input size (i.e., the sum of all the tuples coming from the generated *file input* files). As we can observe, *five* pattern models have outstanding values for this measure compared to the others. Three of these pattern models are variations of the *Surrogate Key Pipeline* pattern and the other two are variations of the *Union* pattern. Examining the first three, we observed that they all included a sequence generation operator (i.e., for adding to each tuple an integer to act as a surrogate key), right before an output operator for loading data to a database. These results

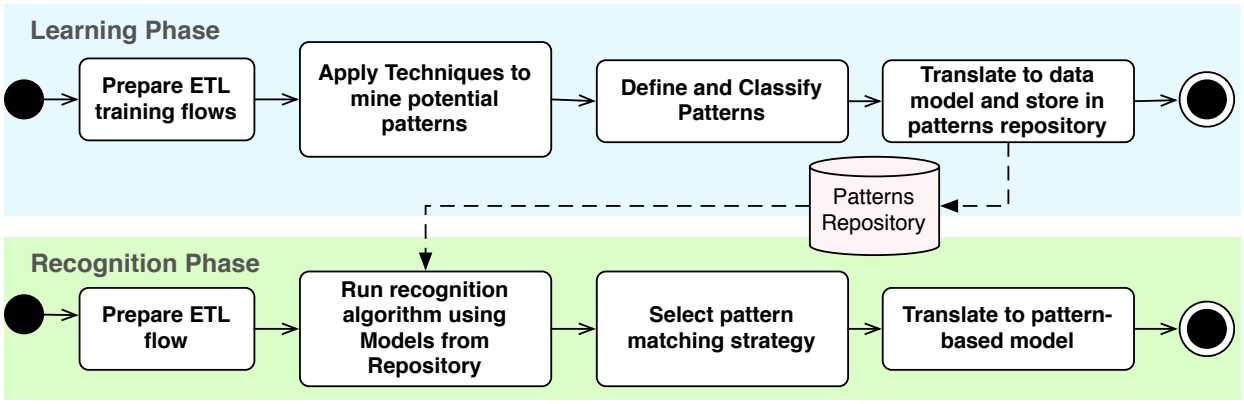

Figure 14: Process Architecture of ETL Workflow Patterns Analysis

are a clear indication of an anti-pattern, since with this design data cannot be processed in a parallel fashion and can possibly be resolved by pushing back the sequence generation operator earlier in the flow.

6. ETL Patterns Use Cases

The identification of patterns within ETL processes and their definition and classification can be used in various ETL projects with contributions spanning from more efficient ETL quality analysis to more usable and reusable ETL models. In this section, we first present a structured methodology according to which our approach can be utilized in context of ETL design. Subsequently, we present two use cases that expose the value of using ETL flows patterns that are derived from our data-driven approach.

6.1. Process Architecture

Our process architecture is depicted in Figure 14 and consists of two phases —the *learning phase* and the *recognition phase*. During the learning phase, a training set of ETL workflows is used to mine ETL patterns and store them in a *patterns repository*. The first step is the modeling of the ETL workflows as the ETL structures defined in Section 3. Subsequently, graph algorithms can be applied on the ETL structures to identify reoccurring structures with frequency above a specified support threshold. After the identification of the frequent structures, analysis takes place to define relevant structures as patterns and to classify them according to their functionality. It is during this step that some frequent structures might be dismissed after being considered irrelevant for the conducted analysis (e.g., non-independent patterns). Finally, ETL patterns are stored in a repository, after being translated to an appropriate data model. During the recognition phase, one ETL flow is modeled as an ETL structure and graph matching algorithms are executed to find occurrences of patterns, from the repository to the ETL structure. A pattern matching strategy is selected to disambiguate the cases where different patterns can be recognized on the same part of the ETL and subsequently, the ETL operations are mapped to corresponding pattern occurrences.

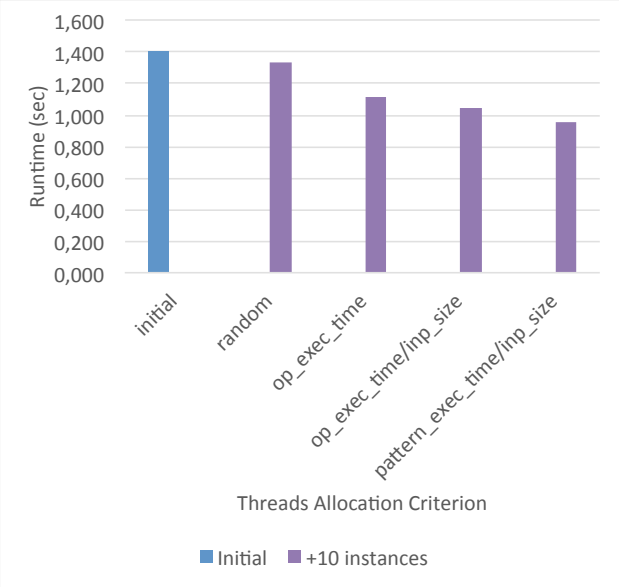

Figure 15: Adding extra instances to DimBroker

6.2. Quality-based Analysis of ETL flows

One interesting application of our bottom-up approach is the more granular evaluation of quality characteristics of ETL processes. In [16], we have gathered quality measures and metrics from literature and we have illustrated how they can be used to evaluate ETL processes with respect to different quality dimensions. Using our approach, such evaluation can take place in a more granular level than the complete ETL workflow—at the level of patterns. What is more, such evaluation can take place without the need for execution of the ETL workflow under test, but simply by statically examining its logical model and by recognizing pattern occurrences, similarly to Figure 17. During a learning phase, the average quality performance of different pattern models can be obtained from the isolated evaluation of their occurrences on a training set of ETLs. Subsequently, after pattern models have been characterized based on their performance, their occurrence on ETL models can signify the existence of parts of the ETL with corresponding performance implications. For instance, different parts of the ETL can be predicted as more, or less costly in terms of consumption of resources, creating a heatmap of the different parts of the ETL. This kind of analysis can also be used for the identification and avoidance of antipatterns [17] during the ETL design phase.

In order to showcase potential benefits from the use of our described methodology, we applied it to two of the ETL processes from the TPC-DI benchmark that we implemented using the Pentaho Data Integration (PDI) tool—*DimBroker* and *Prospect*. Transformations created with PDI are multi-threaded and transformation steps run in parallel, leveraging multiple CPU cores. Thus, by increasing the number of threads assigned to steps, the execution time of the ETL workflow can decrease. The purpose of this evaluation has been to illustrate that on one hand, the distribution of additional threads on different parts of the ETL workflow based on different criteria, can influence its execution time (i.e., performance) and on the other, that making this decision based on the recognition of patterns on the

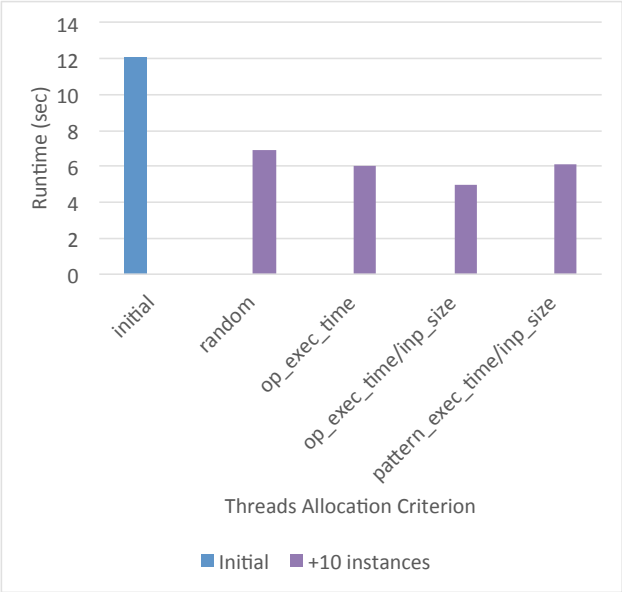

Figure 16: Adding extra instances to Prospect

workflow model can have a positive impact.

Hence, we employ four different criteria for the introduction of 10 additional instances (i.e., threads) for the execution of operations on the ETL workflow: i) allocation of additional instances to random operations (*random*), ii) allocation of additional instances to operators with the highest average execution time (*op_exec_time*), iii) allocation of additional instances to operators with the highest average execution time divided by size of input data (*op_exec_time/inp_size*) and iv) allocation of additional instances to operators inside pattern instances with the highest average execution time divided by size of input data (*pattern_exec_time/inp_size*). For ii) and iii), we followed the same methodology as described in Subsec. 5.3 where instead of pattern instances, each (logical) ETL operation was isolated and its average execution time from 10 executions was measured. For iv), we used the results from Sec. 5, where ETL patterns were mined from the implemented TPC-DI flows and their average execution time was measured, taking under consideration the size of their input datasets. Using our algorithm from Subsec. 4.2, we recognized all ETL pattern instances on the two ETL processes and we selected the instances of the most costly patterns, on operations of which to add additional instances. The execution results on each ETL are shown in Fig. 15 and Fig. 16. As shown, there is performance improvement when adding extra instances for all criteria, with better results than random using any of the criteria ii), iii) and iv). However, it should be noted that for criterion iv), the decision was based solely on performance analysis conducted during a learning phase as opposed to criteria ii) and iii), which were based on performance measures coming from execution of operations of the specific ETL workflows under test. It is therefore illustrated, how our proposed pattern analysis makes performance improvement possible by simply examining the static model of ETL workflows.

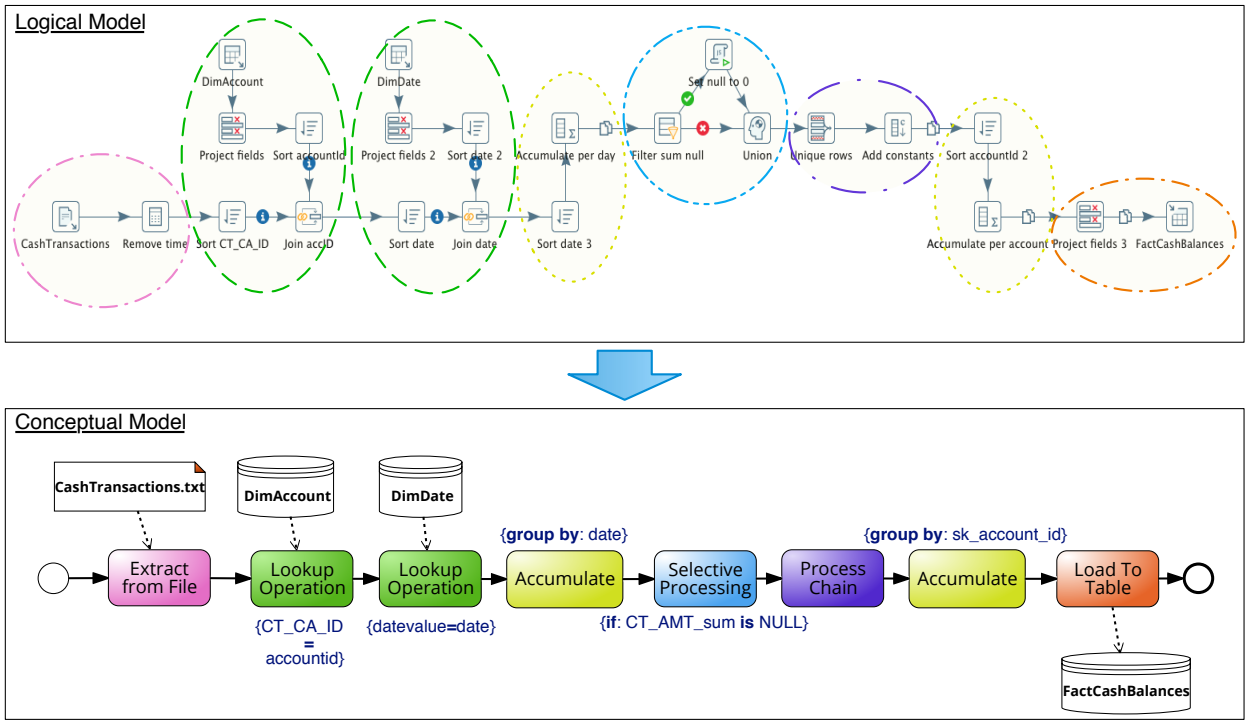

Figure 17: Example of translating logical representation of an ETL process to BPMN

6.3. Conceptual Representation of ETL Flows

The value of using conceptual representations of ETL processes has been recognized in several works [6, 3, 15], where the proposed modeling notation is BPMN, mainly because of its expressiveness and the support of modeling data artifacts and the data flow of the ETL process. These works have mostly focused on the advantages of a conceptual model during the design phase, for modeling abstractions of process functionalities and automating their translation to concrete implementations. Using our bottom-up approach, it is possible to work the other way round, identifying ad-hoc patterns on arbitrary ETL processes and thus populating libraries of such abstractions in the context of any specific business environment and used ETL technologies. It is then straightforward to conceptually model any ETL process in the same context, by decomposing its different parts to the identified patterns.

In Figure 17 we show an example of such a decomposition, translating the logical model of our running example (see Figure 1) from its logical model to a conceptual representation in BPMN. The patterns used are mined from ETL processes within the same domain and context, i.e., TPC-DI ETLs implemented using the Pentaho Data Integration tool. The conceptual representation is much more concise and understandable than the logical view and the translation is completely automated.

7. Summary and Outlook

In the Big Data era, expectations from data-intensive processes are becoming more and more demanding, pushing for solutions that foster agility. In this direction, several ap-

proaches have been proposed for the effective modeling of ETL processes, raising the conceptual level of ETL activities and focusing on the reuse of commonly occurring components during ETL design. However, the frameworks introduced so far heavily rely on expertise to define some universal abstractions that attempt to be applicable for the analysis of arbitrary ETL workflows. In this paper, we introduced a novel empirical approach for pattern-based analysis of ETL workflows in a bottom-up manner. We formally defined an ETL pattern model and we illustrated how it can be instantiated using a training set of ETL workflows to extract frequently reoccurring structural motifs. The graph representation that we adopt enables the use of graph algorithms, such as frequent subgraph discovery algorithms for the mining phase and graph matching algorithms for the recognition phase. For the latter, we adapted the VF2 algorithm with some optimizations and we showed through experiments how it performs very well for ETL workflows. In addition, we presented the most frequent ETL patterns that we identified in implemented processes from the TPC-DI framework, as well as the results from different configurations of the used algorithms. Results show high efficiency and effectiveness of our approach and future work can delve deeper into the evaluation and pattern-based benchmarking of a larger number of realistic ETL workflows to build a solid Knowledge Base of ETL patterns and their characteristics.

Acknowledgements. This research has been funded by the European Commission through the Erasmus Mundus Joint Doctorate "Information Technologies for Business Intelligence - Doctoral College" (IT4BI-DC).

References

[1] M. A. Beyer, E. Thoo, E. Zaidi, R. Greenwald, Magic quadrant for data integration tools, Tech. rep., Gartner (August 2016).

[2] A. Abelló, J. Darmont, L. Etcheverry, M. Golfarelli, J. Mazón, F. Naumann, T. B. Pedersen, S. Rizzi, J. Trujillo, P. Vassiliadis, G. Vossen, Fusion cubes: Towards self-service business intelligence, IJDWM 9 (2) (2013) 66–88.

[3] J.-N. Mazón, E. Zimányi, Z. El Akkaoui, J. Trujillo, A BPMN-based design and maintenance framework for ETL processes, Int. J. Data Warehous. Min. 9 (3) (2013) 46–72.

[4] L. Muñoz, J.-N. Mazón, J. Pardillo, J. Trujillo, Modelling ETL processes of data warehouses with uml activity diagrams, in: OTM 2008, Springer Berlin Heidelberg, 2008, pp. 44–53.

[5] P. Vassiliadis, A. Simitsis, S. Skiadopoulos, Conceptual modeling for ETL processes, DOLAP '02, 2002, pp. 14–21.

[6] K. Wilkinson, A. Simitsis, M. Castellanos, U. Dayal, Leveraging business process models for ETL design, in: ER 2010, 2010, pp. 15–30.

[7] W. M. P. Van Der Aalst, A. H. M. Ter Hofstede, B. Kiepuszewski, A. P. Barros, Workflow patterns, Distrib. Parallel Databases 14 (1) (2003) 5–51.

[8] N. Russell, A. H. M. ter Hofstede, D. Edmond, W. M. P. van der Aalst, Workflow Data Patterns: Identification, Representation and Tool Support, Springer Berlin Heidelberg, Berlin, Heidelberg, 2005, pp. 353–368.

[9] T. Gschwind, J. Koehler, J. Wong, Applying Patterns during Business Process Modeling, Springer Berlin Heidelberg, 2008, pp. 4–19.

[10] G. Mussbacher, D. Amyot, S. A. Behnam, Towards a pattern-based framework for goal-driven business process modeling, SERA 2010 (2010) 137–145.

[11] A. Simitsis, P. Vassiliadis, U. Dayal, A. Karagiannis, V. Tziovara, Benchmarking ETL workflows, in: TPCTC 2009, 2009.

[12] B. Oliveira, O. Belo, Task clustering on ETL systems - a pattern-oriented approach, in: DATA 2015, 2015, pp. 207–214.

[13] B. Oliveira, O. Belo, A. Cuzzocrea, A pattern-oriented approach for supporting ETL conceptual modelling and its yawl-based implementation, in: DATA 2014, Vienna, Austria, 29-31 August, 2014, 2014, pp. 408–415.

[14] M. Salmenkivi, Frequent Itemset Discovery, Springer US, Boston, MA, 2008, pp. 322–323.

[15] B. Oliveira, O. Belo, BPMN patterns for ETL conceptual modelling and validation, in: ISMIS 2012, Springer Berlin Heidelberg, 2012, pp. 445–454.

[16] V. Theodorou, A. Abelló, W. Lehner, M. Thiele, Quality measures for ETL processes: from goals to implementation, Concurrency and Computation: Practice and Experience 28 (15) (2016) 3969–3993.

[17] C. U. Smith, L. G. Williams, Software performance antipatterns, WOSP '00, ACM, New York, NY, USA, 2000, pp. 127–136.

[18] C. Jiang, F. Coenen, M. Zito, A survey of frequent subgraph mining algorithms, Knowledge Eng. Review 28 (1) (2013) 75–105.

[19] M. Kuramochi, G. Karypis, Frequent subgraph discovery, ICDM '01, Washington, DC, USA, 2001, pp. 313–320.

[20] J. Lee, W.-S. Han, R. Kasperovics, J.-H. Lee, An in-depth comparison of subgraph isomorphism algorithms in graph databases, Proc. VLDB Endow. 6 (2) (2012) 133–144.

[21] D. Conte, P. Foggia, C. Sansone, M. Vento, Thirty years of graph matching in pattern recognition, International Journal of Pattern Recognition and Artificial Intelligence 18 (03) (2004) 265–298.

[22] J. R. Ullmann, An algorithm for subgraph isomorphism, J. ACM 23 (1) (1976) 31–42.

[23] L. P. Cordella, P. Foggia, C. Sansone, M. Vento, An improved algorithm for matching large graphs, in: In: 3rd IAPR-TC15 Workshop on Graph-based Representations in Pattern Recognition, Cuen, 2001, pp. 149–159.

[24] R. Shamir, D. Tsur, Faster subtree isomorphism, in: Theory of Computing and Systems, Proc. of the Fifth Israeli Symposium on, 1997, pp. 126–131.

[25] S. Sakr, E. Pardede, Graph Data Management: Techniques and Applications, 1st Edition, IGI Publishing, Hershey, PA, 2011.