Refactoring ETL Flows in The Wild

Dolev Adas[1], Ohad Eytan[1], Guy Khazma[2§], Josep Sampé[1], and Paula Ta-Shma[1]

[1]*IBM Research, Israel*
Email: {Dolev.Adas, ohad, Josep.Sampe.Domenech}@ibm.com, paula@il.ibm.com

[2]*Department of Computer Science, University of Toronto at Ontario, Canada*
Email: guykhazma@cs.toronto.edu

***Abstract*—In modern data-driven ecosystems, Extract, Transform, Load (ETL) flows serve as the backbone of data integration pipelines. These flows facilitate the seamless movement of data across disparate systems and formats, streamlining processes that range from data acquisition to preparation for analysis. However, the pervasive use of ETL flows introduces a pressing challenge—how to bound the maintenance cost of an ever-expanding number of flows. In this paper, we describe an end-to-end prototype for ETL flow refactoring, aimed at reducing the maintenance cost, which keeps the human in the loop for refactoring decisions. Our prototype adopts and significantly extends the gSpan Frequent Subgraph Mining (FSM) algorithm to apply it to real-world ETL use cases in the context of the IBM DataStage™ data integration tool. We report on real customer workloads, share their statistics and evaluate the performance of our prototype. We found potential for up to 32% maintenance cost reduction on the use cases we analyzed after removing duplicate flows. We also share an anonymized version of the workloads with the research community.**

***Index Terms*—data flows, subflows, ETL, data integration, frequent subgraph mining**

I. Introduction

Extract-Transform-Load (ETL) describes the process of identifying and extracting data from various sources followed by transformations, eventually loading the result to data target store(s). Its purpose is to bridge the gap between transactional (OLTP) and analytical (OLAP) systems, which are separate because of their differing requirements - low latency simple transactional updates on smaller datasets versus high throughput complex queries on much larger datasets. ETL transfers data between these systems and has been a key area of data processing in the industry for over 25 years [1].

Analysts predict the size of the ETL software market to triple between 2022 and 2030 [2]. In recent years, the trend to separate transactional and analytics systems has only become stronger. For example, analytics data is increasingly stored in cheap scalable but immutable object storage systems. Moreover, Machine Learning (ML), a modern form of data analytics, is data hungry and requires purpose fit formats.

This long history coupled with a bright future drives a need to modernize while managing existing legacy workloads. Such workloads can consist of tens to hundreds of thousands of ETL flows, authored over many years by diverse sets of authors exposed to developer churn. In addition, these workloads often contain many duplicate or similar flows because of a lack of versioning and subflow support in current or previous tool versions. This leads to increasingly brittle legacy code which is hard to change - an important contributing factor to the high maintenance cost of ETL implementations [2].

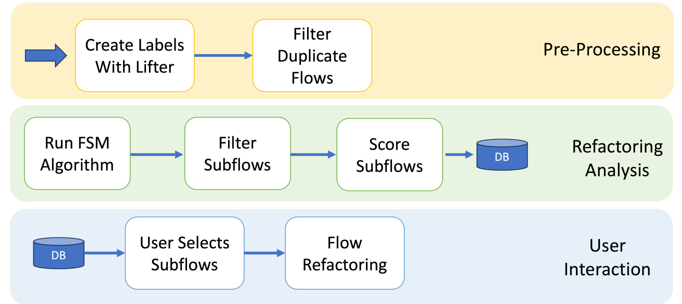


Fig. 1. Refactoring ETL Flows Workflow

To reduce this maintenance cost, we focus on semi-automated refactoring of ETL flow workloads expressed using the IBM DataStage™ data integration tool[1] [3]. We view a data integration flow as a directed acyclic graph (DAG) (Figure 2), and look for frequent common subgraphs across the flows in a workload to be factored out as shared subflows (Figure 3). This is analogous to refactoring programming language code by factoring out common code segments as reusable functions. Note that DAGs are used to model workloads in a wide variety of tools such as DBT [4], Airflow [5] and DVC [6] so our work is applicable beyond DataStage™ and data integration.

The search for frequent common subgraphs is known as Frequent Subgraph Mining (FSM), and we leverage an FSM algorithm called gSpan [7], and further adapt it in many ways to apply it to our real-world industry use case. Our work builds on, extends and complements gSpan by adding capabilities required for ETL refactoring, resulting in an end-to-end prototype. Our refactoring workflow is depicted in Figure 1.

We address three main themes. Firstly, we show there is significant opportunity in real ETL workloads for a reduction in maintenance cost by identifying common subflows and refactoring accordingly (up to 32% in the workloads

[§]Work done while working at IBM Research.

[1]Note: this paper presents research results and does not reflect product roadmap.

979-8-3503-2445-7/23/$31.00 ©2023 IEEE

we analyzed). Secondly, we describe the extensions to FSM algorithms that are needed to provide an end-to-end refactoring workflow. Finally, we demonstrate that such a workflow is feasible in practice in terms of run time. We now cover our main contributions in detail:

End-to-end Refactoring Workflow: We implemented an end-to-end refactoring workflow (Figure 1) which keeps the human-in-the-loop to make final refactoring decisions. This workflow includes pre-processing steps such as identifying duplicate flows, running the FSM algorithm, filtering and scoring subflows according to their perceived benefit, user selection of subflows for refactoring and then performing the refactoring itself. Our prototype includes a novel user interface which enables the user to visualize and select subflows for refactoring.

The Lifter: The subflow feature of DataStage™ can be parameterized similarly to the way programming language functions can accept parameters. This means that if we only look for identical subgraphs, like only looking for completely identical code segments, we will miss many opportunities for refactoring. Instead, we need to look for subgraphs which are equivalent modulo what can be parameterized. This is done by mapping flows using a function we call the *lifter* to a more abstract representation (Figure 5).

gSpan Improvements: To the best of our knowledge there is no open source implementation of gSpan which works correctly on directed graphs so we extended an existing implementation for undirected graphs to support directed graphs. We also added support for concurrent execution and improved memory management.

Scoring and Filtering: FSM and gSpan in particular often identifies a very large number of frequent common subgraphs (candidates). In order to identify those with the best potential to reduce maintenance cost we filter to *closed* subgraphs (Figure 9) and score candidates with the aim of reducing the total number of workload stages.

Opportunity for Maintenance Cost Reduction We share statistics of 6 real workloads including min/max/average flow sizes (Table I) and distributions of flow sizes (Figure 8). In addition, we analyze the frequency of detected subflows in these datasets according to their *size* and *support* (see section IV) (Figure 10). We found their potential maintenance cost reduction (measured as total number of stages) to be up to 32% after removing duplicate flows (Figure 11).

Experimental Results: We analyze tradeoffs between various factors such as maximal subflow size and minimum support and the running time of our algorithm (Figures 6 and 7). We also show how concurrent execution across several threads improves performance (Figure 12). On our 6 workloads, reasonable bounds on subflow size and minimum support result in running times of several minutes which is sufficient for our purposes.

Anonymized Workloads: We contributed an anonymized version our workloads to the research community.

Paper Outline: The rest of the paper is organized as follows. Section II surveys the related work, Section III covers the DataStage™ integration tool, motivates our refactoring goals and presents our workflow. Section IV describes the gSpan FSM algorithm and presents our enhancements, Section V applies gSpan with our extensions to refactoring ETL workloads and Section VI provides an experimental evaluation. We close with a discussion on future work in Section VII and our conclusions in Section VIII.

II. Related Work

Throughout the years, ETL workloads received significant attention from the research community with topics ranging from schema matching, data cleansing and quality to engineering aspects and more [1].

The existence and usefulness of common patterns in ETL flows has been recognized and explored in [8] where it was advocated to make use of common patterns in order to improve the physical optimization and scheduling of ETL flows. Mining of ETL workflows to find frequent patterns is explored in [9]. The authors used the FSG algorithm [10] to mine frequent patterns and an optimized version of the VF2 algorithm [11] to recognize frequent patterns in arbitrary ETL workflow. The method was evaluated on 25 workflows from the TPC-DI benchmark [12] and explored for quality-based analysis of ETL flows and derivation of a conceptual representation. In our work, the main goal is refactoring of ETL flows in order to reduce maintenance cost. This is demonstrated by analyzing 6 real-world datasets of more than 30K client flows in total, where each one has hundreds/thousands of flows. In addition, we chose to leverage the gSpan algorithm for frequent subgraph mining as it was shown to outperform FSG [7]. Our usage of the gSpan algorithm involves modifying it to support directed graphs. Such a modification was explored in [13] and is claimed to be supported in several open-source libraries [14], [15]. However, these implementations do not guarantee correctness due to lack of testing and indeed we encountered such issues. Our implementation builds on [15] and adapt it to our requirements. In particular, we fix the support for directed acyclic graphs by using virtual edges and test its correctness over a diverse set of graphs.

Discovering repetitive logic in ETLs and refactoring it into a standalone ETL that is referenced by other ETLs is also mentioned in [16] where a variant of the gSpan algorithm called cgSpan is presented. cgSpan limits the total number of frequent subgraphs found by detecting only closed frequent graphs. While refactoring of ETL workflows serve as the motivation for developing cgSpan the paper does not discuss the application of this method on ETL workflows and evaluate it only on standard datasets used in subgraph mining.

Similar refactoring requirements were discussed in the context of Automotive Model-Based Development [17] [18] [19]. Unlike our work, their use case deals with finding all maximal subgraphs within one graph.

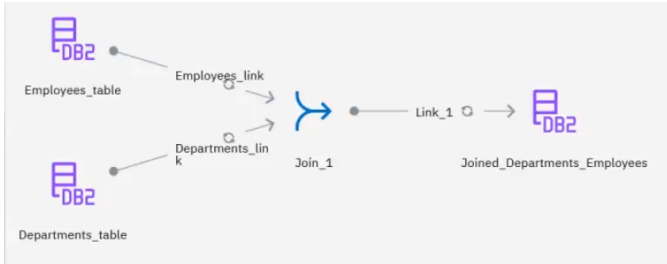


Fig. 2. Example of a simple flow taking data from two sources, joining them, and saving to a target database.

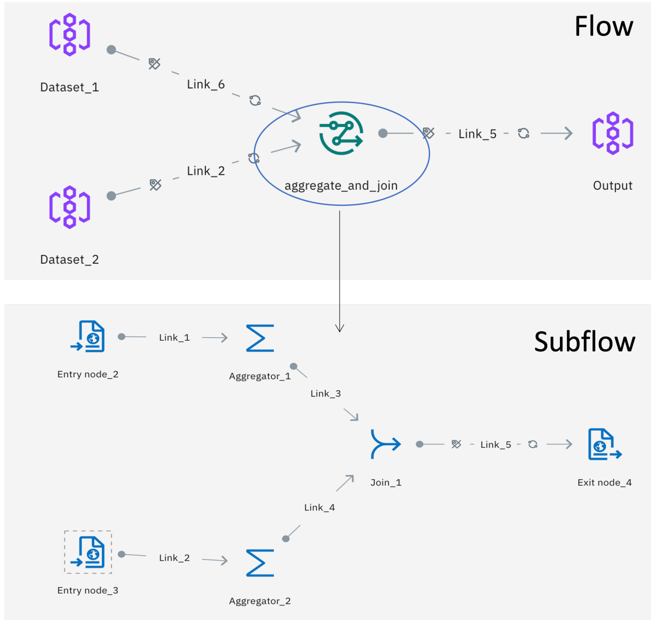


Fig. 3. Example of a flow with a subflow.The subflow is surrounded by a circle in the main flow.

III. Data Flows Refactoring

A. Data Flows and Subflows in IBM DataStage™

IBM DataStage™ [3] is a data integration tool providing its users the ability to create and run ETL and ELT jobs. The job design is primarily done using a UI where the user can add binding and execution nodes (called stages) and connect them with edges, creating a data flow from sources to targets. Figure 2 shows an example of a simple flow.

Subflows [20] are a feature of DataStage™ that enables making part of the job design reusable. A user can use subflows to make common data flow components available throughout the project. Figure 3 shows an example of a simple flow with a subflow.

B. Motivation

Maintenance of large flows is costly and slows development time. Repetitions within flows lead to numerous issues. When the same subflow is scattered across multiple flows in a project, it becomes harder to maintain and update. Since alterations must be made in multiple locations, amplifying the chances of introducing errors. Similarly, large flows hamper comprehensibility and hinder collaboration. It makes the flows convoluted and harder to understand. Flow refactoring emerges as the solution to these challenges. By eliminating duplication and breaking down complex flows into more manageable components, refactoring improves the overall quality of the flows in the project.

Despite these clear advantages of subflows, for various reasons, many users tend not to use them enough. Even if they comprehend that and want to start using it, manually refactoring large datasets of flows is a difficult and tedious task. We aim to provide an automated tool that will allow users to run an analysis of their datasets - that could run for several minutes in the background, get a list of potential subflows, and then explore and choose interactively what to refactor and get it done.

C. Workflow

Our refactoring workflow is depicted in Figure 1. It consists of three phases: the pre-processing phase, the refactoring analysis phase, and the user interaction phase. During the pre-processing phase, which is offline and operates in the background, a label is generated for each node using a function called *Lifter*. This function takes node parameters as input and produces a label. Parameters can range from just the stage type to all node parameters. A detailed explanation of the Lifter is provided in V-A. The next step is to eliminate duplicate graphs, as this will create redundant results in the FSM algorithm.

During the refactoring analysis phase, which also operates in the background, we identify potential subflows for refactoring. The first step in this phase runs the FSM algorithm. Given the significant number of resulting subflows, the subsequent action filters the results. The last step in this phase scores each subflow to form an ordered list of subflows recommended for refactoring. For a good user experience, the offline phases described above should take up to several minutes.

The user interaction phase is an online phase. The initial step requires the user to select the flows for refactoring from the ordered list generated in the refactoring analysis phase. To facilitate this process, we implement an interactive tool that enables users to examine each subflow and view associated statistics on the flows.

We proceed with the flow refactoring task. A new flow is generated based on the chosen subflow and redirects the original flows to it. Notice that the user can choose to refactor all occurrences of a subflow across all graphs in the project or only a specific subset of occurrences. Figure 4 shows an example of two flows before and after refactoring.

IV. Algorithm

We now describe FSM algorithms, and in particular gSpan, and detail our extensions to gSpan. We use gSpan to identify subflows for refactoring.

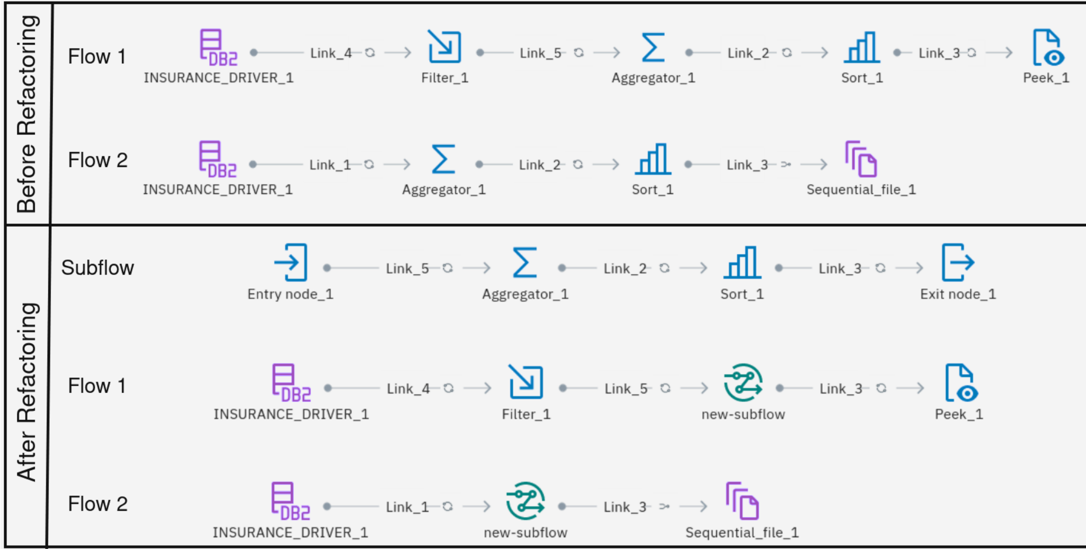


Fig. 4. An Example of refactoring a subflow of size two - containing an aggregation stage and a sort stage - with support of two.

A. *Preliminaries*

The concepts used throughout this paper are listed below. Each concept is accompanied by references to the original definition.

Definition IV.1 (Directed Labeled Graphs)**.** A directed labeled graph $G = (V, E, L, l)$ where (1) V is a set of vertices (2) $E \subseteq V \times V$ is a set of edges and $e = (v, v')$ denotes an edge from v to v'. (3) L is a set of labels (4) $l : V \rightarrow L$ is a function assigning labels to the vertices.

Definition IV.2 (Subgraph Isomorphism)**.** A directed labeled graph $H = (V', E', L, l')$ is isomorphic to a subgraph of $G = (V, E, L, l)$ if there exists a subgraph $G_0 = (V_0, E_0, L, l_0) | V_0 \subseteq V, E_0 \in E \cap (V_0 \times V_0)$ such that there exists a bijection $f : V_0 \rightarrow V'$, such that $\forall u \in V_0, l_0(u) = l'(f(u))$ and $(u, v) \in E_0 \Leftrightarrow (f(u), f(v)) \in E'$.

Definition IV.3 (Support)**.** [7] [Definition. 3] Given a graph dataset $\mathcal{D} = \{G_0, G_1, ..., G_n\}$, $support(g)$ denotes the number of graphs $G \in \mathcal{D}$ in which g is isomorphic to a subgraph of G.

Definition IV.4 (Frequent Subgraph Mining)**.** [7] [Definition. 3] Given a minimum support threshold *minSup*, the set of *frequent subgraph* (FS) includes all the subgraphs g such that $support(g) \geq minSup$.

B. *Frequent Subgraph Mining*

Frequent Subgraph Mining is a fundamental task in graph analysis that aims to discover recurring patterns within a set of graphs [21]. These patterns, often referred to as subgraphs, represent structurally meaningful relationships present across the dataset. Identifying these frequent subgraphs provides valuable insights into the underlying structure and interactions within the data. Applications for frequent subgraph mining span various domains, including bioinformatics, social network analysis, and more.

The task of identifying common subgraphs involves defining a minimum support threshold, beyond which subgraphs are considered frequent. The challenge lies in efficiently exploring the vast space of possible subgraphs and patterns to uncover those that satisfy this frequency criterion.

The gSpan algorithm [7], standing for Graph-based Substructure Pattern Mining, has emerged as one of the most widely used and influential techniques for Frequent Subgraph Mining. Recognized for its efficiency, scalability, and versatility, gSpan addresses the core challenge of efficiently discovering frequent subgraphs within a graph dataset [22].

The gSpan algorithm is designed to systematically explore the space of subgraphs in a manner that efficiently identifies frequent patterns. Here's an overview of its key components and steps:

1. *Graph Database and Minimum Support Threshold*. gSpan takes a collection of labeled graphs as input, forming the database. Each graph consists of nodes representing entities and edges representing relationships. The user sets a minimum support threshold, determining the minimum frequency required for a subgraph to be considered frequent.

2. *DFS Tree and Pruning*. The algorithm employs a Depth-First Search traversal (DFS-traversal) strategy to generate subgraph patterns. It maintains a DFS tree, where each node represents a partial subgraph, and edges indicate extensions by adding nodes and edges. Pruning techniques are applied to discard subgraphs with frequencies below the minimum support threshold, thus focusing the exploration on relevant patterns.

3. *Canonical Form Representation*. To enhance efficiency, gSpan employs a canonical form representation for subgraphs called DFS-Code. A DFS-traversal of a graph defines an

order in which the edges are visited. The concatenation of edge representations in that order is the graph's DFS-Code. This form ensures that isomorphic subgraphs have a unique representation, allowing for easy comparison and pruning of duplicates. Canonical form aids in reducing the search space and eliminates redundancy.

4. *Recursive Exploration and Reporting*. Starting with a single-node seed subgraph, gSpan iterates through each node in the graph, expanding the seed by adding edges and nodes to form larger subgraphs. The algorithm recursively explores the search space, ensuring that each extended subgraph is checked for frequency and transformed into its canonical form. When a frequent subgraph is encountered, it is reported as a discovered pattern.

The gSpan algorithm has garnered significant attention in the research community, leading to the development of multiple implementations across various programming languages, including Python, Java, and C++. For our specific project requirements and the convenience of seamless integration, we opted to work with the Java implementation of gSpan [15]. However, in order to tailor the implementation to meet our specific needs and unique requirements, certain adaptations and enhancements were necessary. The changes are described in the next section.

C. Enhancements to the gSpan Java Implementation

In this section, we describe the key modifications we introduced to the Java implementation of the gSpan algorithm [15] to enhance its performance, scalability, and applicability to our needs and requirements. Our enhancements include support for Directed Acyclic Graphs (DAGs), parallelism for accelerated execution, improved input handling, propagation of original graph information, and efficient memory usage through subgraph dumping.

1. *Support for DAGs*. While the original gSpan java implementation exclusively supported undirected graphs, we have enabled the algorithm to process DAGs.

Originally, gSpan Java uses an array of Vertices as the internal graph representation, with each vertex containing an array of Edges in the form of $(From, To, Label)$, where $From$ and To are indices pointing to the graph array. In this Java version of gSpan, graphs are treated as undirected. Consequently, every edge in the graph definition is added as two separate edges in the internal representation. For example, consider the graph definition $A \leftrightarrow B \leftrightarrow C$. This is internally represented as both $A \rightarrow B \rightarrow C$, and $A \leftarrow B \leftarrow C$, allowing the algorithm to traverse the graph in both directions. When reporting a subgraph, it doesn't matter whether gSpan reports it using the edge $A \rightarrow B$ or $B \rightarrow A$, as it's undirected, and both representations are correct solutions.

To enable gSpan Java to process DAGs, we adopt the same approach to the undirected version, with one crucial difference: we add edges in both directions, however, we distinguish those edges that do not exist in the original graph definition as *virtual edges*. For example, given the DAG definition $A \rightarrow B \rightarrow C$, we initially include the original edges $A \rightarrow B \rightarrow C$ in the internal representation, and then we incorporate the reverse edges $A \leftarrow B \leftarrow C$ while designating them as *virtual edges*. gSpan then processes the graphs as if they were undirected. When reporting a subgraph, we identify *virtual edges* and convert them to their corresponding original edges. For instance, if gSpan identifies a subgraph with the edge $B \rightarrow A$ and this edge is tagged as *virtual edge*, we interchange the nodes to $A \rightarrow B$ before reporting the subgraph.

2. *Parallelism for Accelerated Execution*. The original gSpan algorithm operates sequentially within a single process. To address this limitation and harness modern multicore processors, we have integrated parallelism into the gSpan algorithm. Our implementation allows users to specify the number of threads to utilize for concurrent execution. This enhancement significantly reduces the overall execution time by distributing the workload across multiple threads. For debugging purposes, users can specify a single thread, reverting to the original sequential behavior.

3. *Improved Input Handling*. Our modification expands the input capabilities of the gSpan implementation. The original version only accepted a single-file dataset path for graph loading. In contrast, our enhancement allows users to provide an array of jGraphT [23] graphs as the input dataset. This flexibility streamlines the usage of the algorithm with custom graph data structures and facilitates integration into diverse projects.

4. *Propagation of Original Graph Information*. One limitation of the original gSpan implementation was the absence of mapping between discovered subgraphs and their corresponding nodes/edges in the original graph. To address this, our enhancement ensures the propagation of original graph information throughout the analysis. This means that the nodes and edges found by the algorithm are mapped back to their counterparts in the original graph, enabling users to establish the connection between subgraphs and their context.

5. *Efficient Memory Usage through Subgraph Dumping*. In scenarios where the gSpan algorithm discovers a substantial number of subgraphs, memory consumption becomes a concern. To mitigate this, we have introduced a background thread dedicated to subgraph dumping. As subgraphs are generated, this thread efficiently flushes them to disk. This approach drastically reduces the memory footprint of the algorithm, enabling its application on large datasets where millions of subgraphs can be produced.

V. Application to ETL : Adjustments & Trade-offs

In this section, we present our algorithm surrounding the FSM algorithm described earlier. First, we describe the lifter and the pre-processing algorithm. Following this, we discuss the trade-offs in the selection of the parameters for the FSM algorithm. Additionally, we describe the processes of scoring and filtering the results generated by the FSM algorithm.

A. Pre-processing

Each stage in DataStage™ is represented as a JSON document containing the stage parameters in addition to its

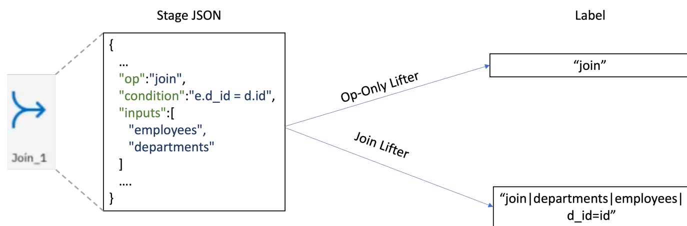


Fig. 5. A Lifter maps a stage to a normalized string label to be used in the FSM algorithm. In the above example, the OP-ONLY lifter extracts only the operation while the JOIN lifter is specifically designed for the join stage and takes into account the join inputs and condition.

connections to other stages. To use gSpan, we model each flow as a directed labeled graph by using a *Lifter*. The Lifter is a function that gets as an input the stage parameters and maps them to the label that is used in the frequent subgraph algorithm. An important aspect of the Lifter is to normalize the labels for different stages such that two stages which are semantically equivalent for the purpose of mining are mapped to the same label. For example, a common normalization method is to sort the inputs lexicographically in order to be agnostic to the order in the user's input. Figure 5 shows an example of two lifters for the join stage from figure 2. The OP-ONLY lifter uses only the stage operation as the label. The JOIN lifter, on the other hand, takes into account the join stage parameters such as the join condition and the inputs (sorted lexicographically).

Using lifters at different levels of granularity enables mining the flows at various resolutions. Coarser granularity lifters generate more opportunities for finding frequent subgraphs but require us to ensure that we can still define common subflows for them using subflow parameterization, as well as adhering to best practices such as avoiding over parameterization. In addition, variations of our use case can benefit from different lifters. For example, cross platform pattern mining can use the OP-ONLY lifter to discover structural shared subflows across clients in order to build a common shared library. Conversely, using a stage specific lifter on client flows can help identify patterns which are unique to this client. Our implementation enables defining lifters in a pluggable way.

The granularity of the lifter that is used may introduce duplicate *identical* flows by mapping their stages to the same string labels even though they differ in some parameters which are not taken into account by the lifter. Therefore, before running the FSM algorithm, we de-duplicate the flows and include only one representative for each class. To identify duplicated flows we use a similar method to the one described in [24] for finding similar subexpressions. The method makes use of a Merkle Tree [25] where each node identifier is its label. A fingerprint is computed for each node using a cryptographic hash function which combines the identifier of each node and the identifiers of its children recursively starting from the root of the sub-tree down to the leaves. Identical flows will have the same fingerprint for the sink node. DataStage™ flows are DAGs and thus may have multiple sinks so we add a dummy sink that connects all existing sinks.

B. Limiting the number of identified subgraphs

The subgraph isomorphism problem is an NP-complete problem. We point out that the runtime of gSpan is exponential and its output is typically exponential in the size of its input. In our experiments we found that the run time of gSpan is closely linked to the number of subflows identified (Figure 6 and 7).

We identified two significant factors that lead to a reduction in the number of subflows identified, consequently decreasing the overall run time. The first factor is the maximal number of nodes allowed for a subflow. As shown in Figure 6, increasing this number exponentially increases the number of subflows found and the run time. The number of flows for each dataset is in Table I. It's evident that the size of each dataset directly influences the quantity of generated subflows and consequently affects the runtime. The second important parameter is the support threshold. As shown in Figure 7, as we increase the support, the number of subflows decreases exponentially, reducing the run time accordingly. Dataset 1 comprises a small set of similar flows, resulting in the creation of 15 subflows This explains why we do not observe a noticeable correlation between the small number of subflows and the runtime, which remains consistently under a second across all support values. Similarly, this same explanation is applicable to the weak correlation found in dataset 2, which is also relatively small in size.

C. Scoring and Filtering

To further reduce the number of subgraphs presented to the end user, we filter the results produced from gSpan. We used closed graph filter [26]. A graph g is closed in a database if there exists no proper supergraph of g (i.e., $g' \supset g$) that has the same support.

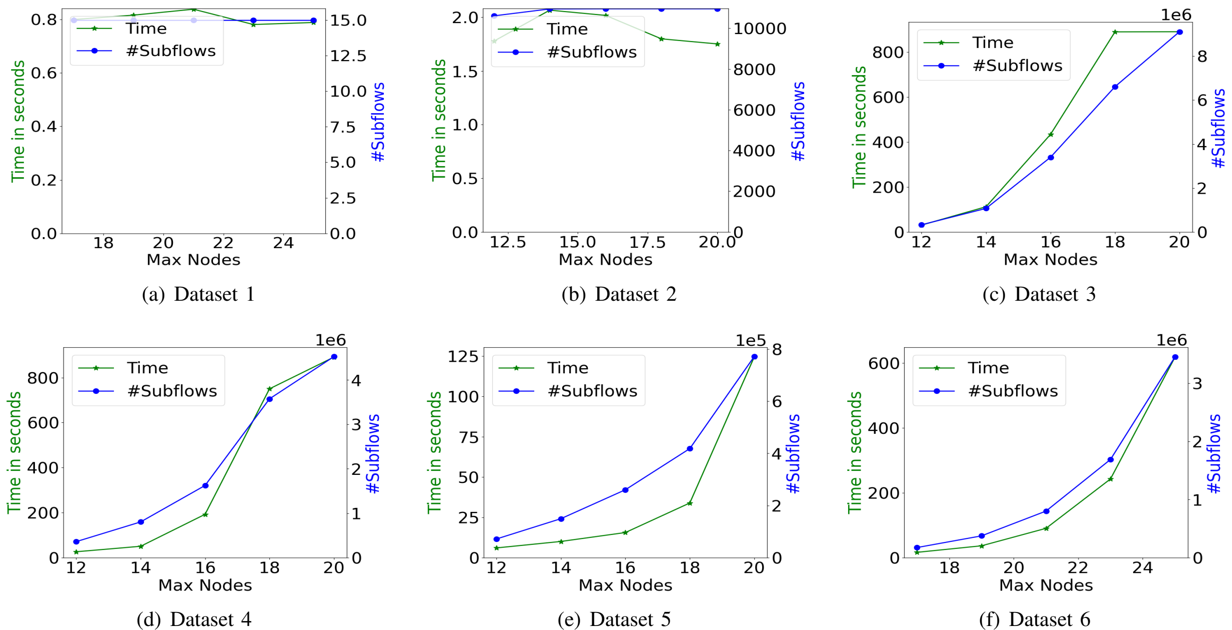
We continued with ranking the subflows, creating an ordered list that allows the end user to select the subflows to refactor easily. The subflows are ranked based on their size (the number of nodes) multiplied by the subflow support. This approach considers small subflows with high support and larger subflows with lower support as potential candidates for refactoring. We use this ranking function to filter further the number of subflows displayed.

VI. EXPERIMENTAL EVALUATION

In this section, we evaluate our proposed methods on real-world datasets that came from IBM's DataStage™ service. We first describe the datasets and share some statistics, and then present the results of the subflow analysis.

A. Datasets

We collected six datasets from various IBM clients, each having hundreds to thousands of flows, combining more than 30,000 flows in total. We measured the size of each flow (i.e., the number of stages in the flow) and report the maximal, minimal, average, and median values in Table I together with



(a) Dataset 1 (b) Dataset 2 (c) Dataset 3

(d) Dataset 4 (e) Dataset 5 (f) Dataset 6

Fig. 6. Execution time measurements and the number of subflows found. We fixed the 'support' parameter and vary the 'maximal node number'. The support threshold for each dataset is 3. The green line on the right y-axis represents the runtime measurements, while the blue line on the left y-axis represents the number of subflows.

TABLE I
STATISTICS OF THE COLLECTED DATASETS

Dataset Name	Number of Flows	Size of Flow			
		Min	Max	Avg	Med
Dataset 1	743	2	50	7.82	6
Dataset 2	762	2	85	8.32	5
Dataset 3	12,158	1	86	6.95	5
Dataset 4	11,466	1	192	7.69	5
Dataset 5	4,693	2	88	7.03	4
Dataset 6	7,010	2	83	4.47	4

TABLE II
DUPLICATES AND UNIQUE FLOWS

Dataset Name	Total Flows	Duplicates Flows	Unique Flows
Dataset 1	743	8	734
Dataset 2	762	51	688
Dataset 3	12,158	825	10829
Dataset 4	11,466	414	10571
Dataset 5	4,693	459	4031
Dataset 6	7,010	1093	3775

the entire distribution of sizes in Figure 8. One can see that although the majority of the flows are small (e.g., about 61% of the flows have 5 or less stages), there are significant amounts of more extensive flows (e.g., 37% of the flows have $6-29$ stages), and a noteworthy long tail of very large flows (e.g., about 2% of the flows have 30 stages or more). This indicates that there is a potential for an effective refactoring process.

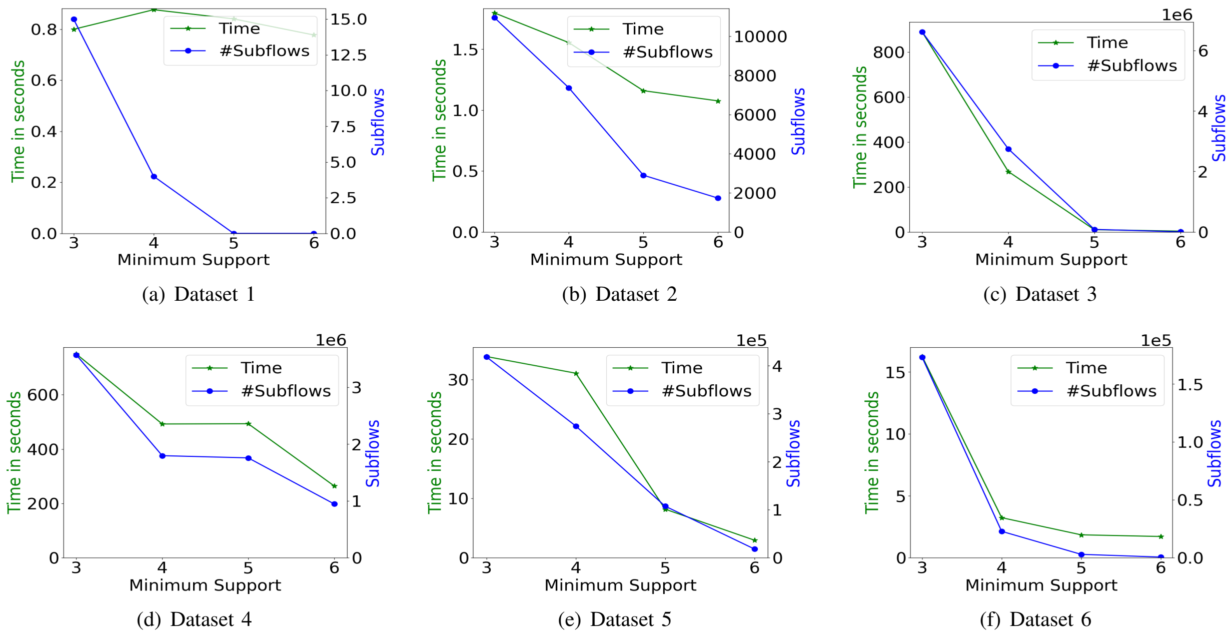
As part of the pre-processing phase, described in Section III-C, we identified and eliminated duplicate flows. These are flows that appear twice or more in the user workload. Their numbers together with the count of the remaining flows are displayed in Table II. Our analysis is done only on the set of the unique flows.

To provide transparency and research opportunities to the community, we released an anonymized version of our datasets. We used two different lifters and hashed the results in order to keep details protected while allowing FSM algorithms to analyze flow structure. The datasets and more details on the process are available in [27]. As far as we know, this would be the first open real-world ETL flows dataset for FSM.

B. Results

The prototype we built aimed to expose end-users to potential subflows in order to save maintenance costs by refactoring them to smaller components. An optional future usage is to suggest these components to users as they write new flows. Consequently, we are more interested in small and medium subflows than huge ones. In addition, targeting end-users as the consumers of our tool, runtime performance is crucial. Although we don't aim to provide a dynamic online experience, we cannot allow days or hours of processing. Several minutes is the acceptable scope. For these reasons, and following the discussion in Section V, unless stated otherwise, we limited the analysis to subflows of size smaller than ten stages and looked for all subflows with support of two or more. We ran our measurements on a 2.3 GHz 8-Core Intel Core i9 CPU with 64 GB of memory.

As stated before, our analysis was conducted using "unique flows", with $n-1$ instances of each duplicate flow being excluded (see Table II). This approach was adopted due to the potential implications of duplicate entries, which could notably increase processing times and inflate the total count of discovered subflows. For instance, if two flows contain



(a) Dataset 1 (b) Dataset 2 (c) Dataset 3

(d) Dataset 4 (e) Dataset 5 (f) Dataset 6

Fig. 7. Execution time measurements and the number of subflows found. We vary the 'support' parameter while keeping the 'maximal node number' (subflow size) fixed. The size of the subflows is 17 for datasets 1,6 and 18 for datasets 2-5. The green line on the right y-axis represents the runtime measurements, while the blue line on the left y-axis represents the number of subflows.

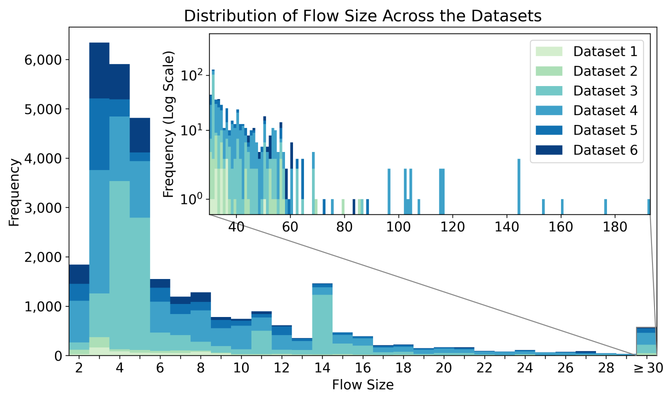


Fig. 8. The distribution of flow size in our six datasets. The tail of flows that are equal or greater than 30 stages are aggregated in the last bar of the outside plot, and broken into details in the inside plot (notice the log scale on the y axis)

the same DAG, the result of the analysis would include all subflows that contain $>= 2$ nodes, which, depending on the flow size, can be thousands or hundreds of thousands of subflows. In this sense, we implemented a mechanism to know in advance which flows are duplicate by calculating their canonical representation, allowing us to exclude them for the analysis, and eventually including them back in the final result. However, while $n-1$ instances of each duplicate flow were eliminated from the analysis, flows that exhibit less than 100% similarity were kept for examination. This, for the same reason explained above, results in an extensive number of subflows being detected, as illustrated in Figure 9, where the algorithm identified millions of subflows in most of the datasets. To manage this abundance of subflows, we employed the "closed frequent subgraph filter" detailed in Section V-C. This filter substantially reduces the final count of subflows, retaining only the interesting ones.

The total number of subflows discovered across all datasets together with number of subflows after applying the *closed frequent subgraphs filter* is depicted in Figure 9. This figure also encompasses the reduction ratio, showcasing a substantial reduction of up to 95% in the count of relevant subflows in the final output.

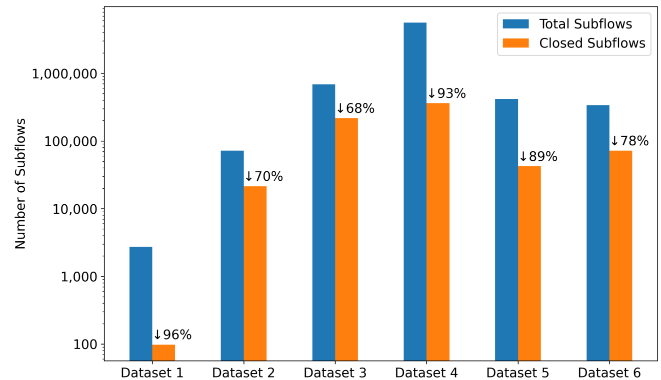


Fig. 9. The total number of subflows (TS) vs the total number of closed subflows (TCS). The percentage of reduction from TS to TCS is shown above the bars. Note the logarithmic scale.

While this reduces the number of subflows substantially, the technique is still limited when common subflows are larger than $MaxNodes$. In that case, all subgraphs of size $MaxNodes$ are considered closed since the larger subflows are not discovered, so the user will get a vast number of partially overlapping subflows. To avoid this, we omit all subflows with size equal to $MaxNodes$ and present only

smaller subflows, effectively limiting subflow sizes to be at most $MaxNodes - 1$.

The number of subflows for each size and support, is presented in Figure 10. As expected, small subflows with low support (the bottom-left corner of the Figure) are the most common, while bigger subflows (top part) and those with high support (right part) are rarer. Nevertheless, they exist, and together they make up a large portion of the subflows, i.e., subflows of size 5 or greater, and subflows with support 5 and more, are about 33% of all the subflows.

To quantify the potential benefit from the subflows further, we define the *maintenance size* as the number of stages in the flows and subflows of an entire dataset $\mathcal{D}$:

$$MS(\mathcal{D}) = \sum_{f \in flows} |f| + \sum_{s \in subflows} |s|$$

where $|x|$ is the number of stages in x. Then, refactoring a subflow of size n and support of k, reduces the size of the flows by $k \cdot (n-1)$ and adds a new subflow of size n (for simplicity, we ignore the cost of inputs and outputs stages). Summing it up, the total *potential maintenance size* becomes:

$$PMS(\mathcal{D}) = MS(\mathcal{D}) + \sum_{s \in new-subflows} |s| - k_s \cdot (|s| - 1)$$

where k_s is the support of subflow s. These measurements for our datasets are shown in Figure 11. For all of the datasets except for the smallest one, we see a substantial opportunity, with a reduction of more than 10% of the maintenance size, and up to 32% for the largest dataset. We point out that this is an optimistic measurement - since some subflows are overlapping so you can't use them together and for other reasons you may not use some subflows - but it gives us a rough estimation of what can be achieved.

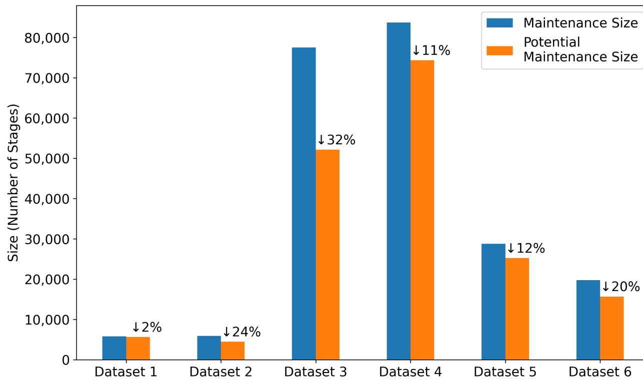


Fig. 11. The Maintenance Size (MS) and Potential Maintenance Size (PMS) of our six datasets. The percentage of reduction from MS to PMS is shown above the bars.

Finally, we evaluated the runtime performance, using our newly integrated parallelism feature described in Section IV-C, experimenting with up to four threads. The results of those runs are presented in Figure 12. Remarkably, our findings revealed that employing three threads yielded the most favorable results, where none of the runs exceeded 15 minutes of execution, falling inside our goal range of several minutes. The main reason for this is that using three threads is enough to enable the algorithm to swiftly traverse all the smaller extensions while concurrently processing the largest extensions in parallel. In this sense, further increasing the number of threads would not yield substantial benefits, as they would remain underutilized. Although we have no guarantees that other datasets would be the same, it seems that overall this will not be a limiting issue.

VII. Discussion and Future Work

Customers with workloads in production may have concerns regarding refactoring. Risk can be significantly reduced by integrating such a refactoring tool with rigorous test suites to verify no change in behavior and performance. This is a topic for future work. We also point out that one could use our tool to reduce maintenance costs without refactoring. For example, when updating part of a flow (with or without subflows), our tool could be extended to identify additional instances of that part and suggest propagating the same update to them.

This paper focuses on design time analysis but similar techniques could be applied to run time analysis. For example, it may be possible to identify multiple flows which share similar operators and are also applied to the same inputs. Such flows might be refactored to build a common intermediate result, which could improve flow execution run time performance as well as reduce maintenance costs. Both improvements reduce TCO and thereby contribute to energy efficiency and sustainability.

VIII. Conclusions

In this paper, we address the ever increasing burden of maintaining ETL flows. We introduce a novel approach that identifies shared patterns in diverse flows and refactors by creating subflows. Our framework is powered by an FSM tool called gSpan which detects shared patterns across a dataset of graphs. Through the integration of features such as parallelism, DAG support, and the propagation of original graph information, we extend gSpan to accommodate specific requirements of ETL use cases. Moreover, our filtering and scoring techniques prioritize relevant subflows and recommend them for refactoring. Our method enhances the maintainability of ETL flows, reducing TCO and improving sustainability. Our evaluation, spanning various real-world datasets, demonstrates the robustness and adaptability of our approach, showing a reduction in maintenance costs of up to 32%. Finally, we contributed an anonymized version of our workloads to the research community.

Acknowledgments

We would like to thank Gal Lushi and Roee Shlomo for their work on an earlier stage of this project. Thanks also to Michael Factor, Ronen Kat, Niels Pardon, Bruno Wassermann and the external reviewers for providing useful review feedback. This research was partially supported by EU Horizon Framework grant agreement 101070186 (TEADAL).

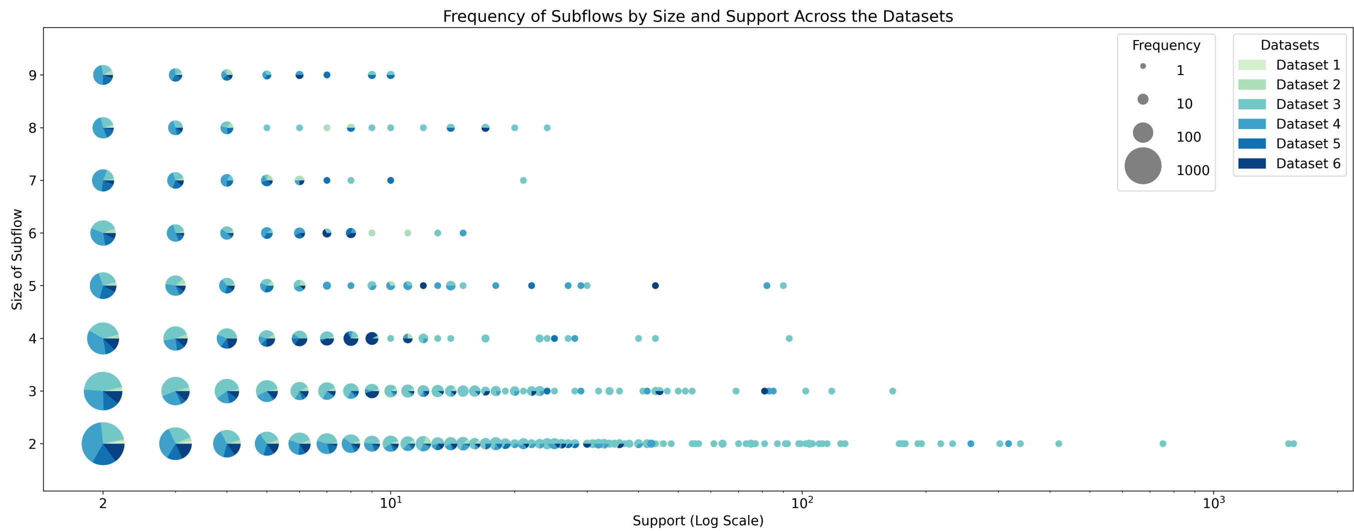


Fig. 10. The frequency of detected subflows by the support and size of the subflow (i.e., point of size S at (X, Y) implies that there are S subflows of size Y with support of X).

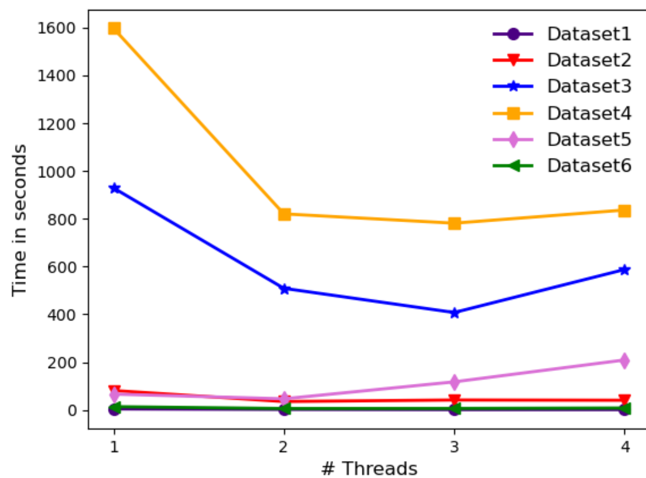


Fig. 12. Execution time measurement in seconds.

References

[1] A. Simitsis, S. Skiadopoulos, and P. Vassiliadis, "The History, Present, and Future of ETL Technology," 2023. [Online]. Available: https://ceur-ws.org/Vol-3369/invited1.pdf

[2] SkyQuest, "Global ETL Software Market Insights," https://www.skyquestt.com/report/etl-software-market.

[3] "IBM DataStage™," https://www.ibm.com/products/datastage.

[4] "DAG (DBT Documentation)," https://docs.getdbt.com/terms/dag.

[5] "DAGs (Airflow Documentation)," https://airflow.apache.org/docs/apache-airflow/stable/core-concepts/dags.html.

[6] "DAG (DVC Documentation," https://dvc.org/doc/command-reference/dag.

[7] X. Yan and J. Han, "gSpan: graph-based substructure pattern mining," in *2002 IEEE International Conference on Data Mining, 2002. Proceedings.*, 2002, pp. 721–724.

[8] P. Vassiliadis, A. Simitsis, and E. Baikousi, "A taxonomy of ETL activities," in *Proceedings of the ACM twelfth international workshop on Data warehousing and OLAP*, 2009, pp. 25–32.

[9] V. Theodorou, A. Abelló, M. Thiele, and W. Lehner, "Frequent Patterns in ETL Workflows: An Empirical Approach," *Data & Knowledge Engineering*, vol. 112, 09 2017.

[10] M. Kuramochi and G. Karypis, "An efficient algorithm for discovering frequent subgraphs," *IEEE transactions on Knowledge and Data Engineering*, vol. 16, no. 9, pp. 1038–1051, 2004.

[11] P. F. C. Sansone, M. Vento, L. Cordella, P. Foggia, C. Sansone, and M. Vento, "An improved algorithm for matching large graphs," in *Proc. of the 3rd IAPR-TC-15 International Workshop on Graph-based Representations*, vol. 57, 2001.

[12] M. Poess, T. Rabl, H.-A. Jacobsen, and B. Caufield, "TPC-DI: the first industry benchmark for data integration," *Proceedings of the VLDB Endowment*, vol. 7, no. 13, pp. 1367–1378, 2014.

[13] C. W.-k. Leung, "Technical notes on extending gSpan to directed graphs," *Technical report, Technical Report, Management University, Singapore*, 2010.

[14] Q. Chen and N. Karpov, "betterenvi/gSpan: a Python implementation of gSpan," https://github.com/betterenvi/gSpan.

[15] T. Zhu, "TonyZZX/gSpan.Java: a Java implementation of gSpan," https://github.com/TonyZZX/gSpan.Java.

[16] Z. Shaul and S. Naaz, "cgSpan: Closed Graph-Based Substructure Pattern Mining," in *2021 IEEE International Conference on Big Data (Big Data)*. Los Alamitos, CA, USA: IEEE Computer Society, dec 2021, pp. 4989–4998.

[17] F. Deissenboeck, B. Hummel, E. Jürgens, B. Schätz, S. Wagner, J.-F. Girard, and S. Teuchert, "Clone detection in automotive model-based development," in *Proceedings of the 30th international conference on Software engineering*, 2008, pp. 603–612.

[18] F. Deissenboeck, B. Hummel, E. Juergens, M. Pfaehler, and B. Schaetz, "Model clone detection in practice," in *Proceedings of the 4th International Workshop on Software Clones*, 2010, pp. 57–64.

[19] J. Chen, T. R. Dean, and M. H. Alalfi, "Clone detection in Matlab Stateflow models," *Software Quality Journal*, vol. 24, pp. 917–946, 2016.

[20] "IBM Subflows in DataStage," https://www.ibm.com/docs/en/cloud-paks/cp-data/4.7.x?topic=reusable-subflows.

[21] C. Jiang, F. Coenen, and M. Zito, "A survey of frequent subgraph mining algorithms," *The Knowledge Engineering Review*, vol. 000, pp. 1–31, 01 2004.

[22] S. U. Rehman, S. Asghar, Y. Zhuang, S. Fong *et al.*, "Performance evaluation of frequent subgraph discovery techniques," *Mathematical problems in engineering*, vol. 2014, 2014.

[23] D. Michail, J. Kinable, B. Naveh, and J. V. Sichi, "JGraphT—A Java Library for Graph Data Structures and Algorithms," *ACM Trans. Math. Softw.*, vol. 46, no. 2, may 2020. [Online]. Available: https://doi.org/10.1145/3381449

[24] P. Michiardi, D. Carra, and S. Migliorini, "Cache-based multi-query optimization for data-intensive scalable computing frameworks," *Information Systems Frontiers*, vol. 23, pp. 35–51, 2021.

[25] R. C. Merkle, "Protocols for public key cryptosystems," in *Secure communications and asymmetric cryptosystems*. Routledge, 2019, pp. 73–104.

[26] X. Yan and J. Han, "CloseGraph: Mining Closed Frequent Graph Patterns," 01 2003, pp. 286–295.

[27] O. Eytan, "Releasing Anonymized ETL Flow Datasets for FSM," https://ibiz/datastage-fsm-dataset, 2023.