Veer: **Verifying Equivalence of Workflow Versions in Iterative Data Analytics**

Sadeem Alsudais, Avinash Kumar, and Chen Li
Department of Computer Science, UC Irvine, CA 92697, USA
{salsudai,avinask1,chenli}@ics.uci.edu

ABSTRACT

Data analytics using GUI-based workflows is an iterative process in which an analyst makes many iterations of changes to refine the workflow, generating a different version at each iteration. In many cases, the result of executing a workflow version is equivalent to a result of a prior executed version. Identifying such equivalence between the execution results of different workflow versions is important for optimizing the performance of a workflow by reusing results from a previous run. The size of the workflows and the complexity of their operators often make existing equivalence verifiers (EVs) not able to solve the problem. In this paper, we present "Veer," which leverages the fact that two workflow versions can be very similar except for a few changes. The solution divides the workflow version pair into small parts, called *windows*, and verifies the equivalence within each window by using an existing EV as a black box. We develop solutions to efficiently generate windows and verify the equivalence within each window. Our thorough experiments on real workflows show that Veer is able to not only verify the equivalence of workflows that cannot be supported by existing EVs but also do the verification efficiently.

CCS CONCEPTS

• **Theory of computation → Semantics and reasoning**.

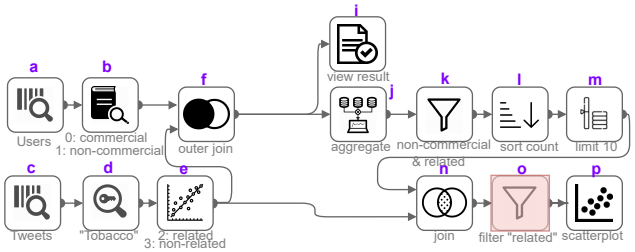
KEYWORDS

iterative data analysis, workflow equivalence verification

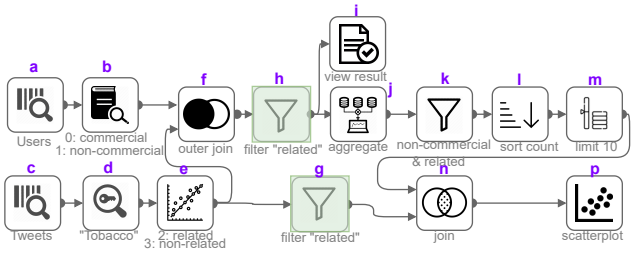
ACM Reference Format:
Sadeem Alsudais, Avinash Kumar, and Chen Li. 2018. Veer: Verifying Equivalence of Workflow Versions in Iterative Data Analytics . In *Proceedings of Make sure to enter the correct conference title from your rights confirmation emai (Conference acronym 'XX)*. ACM, New York, NY, USA, 21 pages. https://doi.org/XXXXXXX.XXXXXXX

1 INTRODUCTION

Big data processing platforms, especially GUI-based systems, enable users to quickly construct complex analytical workflows [5, 15, 32]. These workflows are refined in iterations, generating a new version at each iteration, before a final workflow is constructed, due to the nature of exploratory and iterative data analytics [18, 52]. For example, Figure 1 shows a workflow for finding the relevant Tweets by the top k non-commercial influencers based on their tweeting rate on a specific topic. After the analyst constructs the initial workflow version (a) and executes it, she refines the workflow to achieve the desired results. This yields the following edit operations highlighted in the figure, 1) deleting the filter 'o' operator, 2) adding the filter 'g' operator, and 3) adding the filter 'h' operator.



(a) Version 1: Initial workflow with sinks s_i of all users' tweets, and s_p of top k non-commercial influencers' relevant tweets. The highlighted operator indicates that it is deleted in a subsequent version.



(b) Version 2: Refined version to optimize the workflow performance and filter on relevant tweets of all users. The highlighted operators are newly added in the new version.

Figure 1: Example workflow and its evolution in two versions.

There has been a growing interest recently in keeping track of these workflow versions and their execution results [4, 15, 29, 49, 51]. In many applications, these workflows have a significant amount of overlap and equivalence [4, 28, 55, 56]. For example, 45% of the daily jobs in Microsoft's analytics clusters have overlaps [28]. 27% of 9,486 workflows to detect fraud transactions from Ant Financial have overlaps, 6% of which is equivalent [55]. In the running example, the edits applied on version (a) that led to a new version (b) had no effect on the result of the sink labeled 'p'. Identifying such equivalence between the execution results of different workflow versions is important. The following are two example use cases.

Use case 1: Optimizing workflow execution. Workflows can take a long time to run due to the size of the data and their computational complexity, especially when they have advanced machine learning operations [7, 32, 55]. Optimizing the performance of a workflow execution has been studied extensively in the literature [19, 40].

Conference acronym 'XX, June 03–05, 2018, Woodstock, NY

ACM ISBN 978-1-4503-XXXX-X/18/06…$15.00
https://doi.org/XXXXXXX.XXXXXXX

One optimizing technique is by leveraging the iterative nature of data analytics to reuse previously materialized results [18, 28].

Use case 2: Reducing storage space. The execution of a workflow may produce a large number of results and storing the output of all generated jobs is impractical [20]. Due to the nature of the overlap and equivalence of consecutive versions, one line of works [3, 18] periodically performs a view de-duplication to remove duplicate stored results. Identifying the equivalence between the workflow versions can be used to avoid storing duplicate results and helps in avoiding periodic clean-up of duplicate results.

These use cases show the need for effective and efficient solutions to decide the equivalence of two workflow versions. We observe the following two unique traits of these GUI-based iterative workflows. (*T1*) these workflows can be large and complex, with operators that are semantically rich [5, 18, 52]. For example, the top 8 workflows in Alteryx's workflows hub [6] had an average of 29 operators, with one of the workflows containing 102 operators, and comprised of mostly non-relational operators. Real workflows in Texera [46] had an average size of 23 operators, and most of them had visualization and UDF operators. Some operators are user-defined functions (UDF) that implement highly customized logic including machine learning techniques for analyzing data of different modalities such as text, images, audios, and videos [52]. For instance, the workflows in the running example contain two non-relational operators, namely a Dictionary Matcher and a Classifier. (*T2*) Those adjacent versions of the same workflow tend to be similar, especially during the phase where the developer is refining the workflow to do fine tuning [19, 52]. For example, 50% of the workflows belonging to the benchmarks that simulated real iterative tasks on video [52] and TPC-H [19] data had overlap. The refinements between the successive versions comprised of only a few changes over a particular part of the workflow. Thus, we want to study the following:

Problem Statement: Given two similar versions of a complex workflow, verify if they produce the same results.

Limitations of existing solutions. Workflows include relational operators and UDFs [32]. Thus, we can view the problem of checking the equivalence of two workflow versions as the problem of checking the equivalence of two SQL queries. The latter is undecidable in general [1] (based on the reduction from First-order logic). There have been many Equivalence Verifiers (EVs) proposed to verify the equivalence of two SQL queries [13, 55, 56]. These EVs have *restrictions* on the type of operators they can support, and mainly focus on relational operators such as SPJ, aggregation, and union. They cannot support many semantically rich operators common in workflows, such as dictionary matching and classifier operators in the running example, and other operators such as unnest and sentiment analyzer. To investigate their limitations, we analyzed the SQL queries and workflows from 6 workloads, and generated an equivalent version by adding an empty filter operator. Then, we used EVs from the literature [13, 50, 55, 56] to test the equivalence of these two versions. Table 1 shows the average percentage of pairs for each workload that can be verified by these EVs, which is very low.

Table 1: Limitations of existing EVs to verify equivalence of workflow versions from real workloads.

Workload	# of pairs	AVG. % of pairs supported by existing EVs
Calcite benchmark [9]	232	34.81%
Knime workflows hub [30]	37	2.70%
Orange workflows [39]	32	0.00%
IMDB sample workload [26]	5	0.00%
TPC-DS benchmark [47]	99	2.02%
Texera workflows [46]	105	0.00%

Our Approach. To solve the problem of verifying the equivalence of two workflow versions, we leverage the fact that the two workflow versions are almost identical except for a few local changes (*T2*). In this paper, we present Veer[1], a verifier to test the equivalence of two workflow versions. It addresses the aforementioned problem by utilizing existing EVs as a black box. In §3, we give an overview of the solution, which divides the workflow version pair into small parts, called "windows", so that each window satisfies the EV's restrictions in order to push testing the equivalence of a window to the EV. Our approach is simple yet highly effective in solving a challenging problem, making it easily applicable to a wide range of applications.

Why not develop a new EV? A natural question arises: why do we choose to use existing EVs instead of developing a new one? Since the problem itself is undecidable, any developed solution will inherently have limitations and incompleteness. Our goal is to create a general-purpose solution that maximizes completeness by harnessing the capabilities of these existing EVs. This approach allows us to effectively incorporate any new EVs that may emerge in the future, ensuring the adaptability and flexibility of our solution.

Challenges and Contributions. During the exploration of the proposed idea, we encountered several challenges in developing Veer: 1) How can we enhance the completeness of the solution while maintaining efficiency and effectively handling the incompleteness of the EVs? 2) How do we efficiently handle workflow versions with a single edit and perform the verification? 3) How can we effectively handle workflow versions with multiple edits, and can the windows overlap? We thoroughly investigate these challenges and present the following **contributions**.

(1) We formulate the problem of verifying the equivalence of two complex workflow versions in iterative data analytics. To the best of our knowledge, Veer is the first work that studies this problem by incorporating the knowledge of user edit operations into the solution (§2).
(2) We give an overview of the solution and formally define the "window" concept that is used in the equivalence verification algorithm (§ 3).
(3) We first consider the case where there is a single edit. We analyze how the containment between two windows is related to their equivalence results, and use this analysis to derive the concept of "maximal covering window". We provide complexity analysis (§4).
(4) We study the general case where the two versions have multiple edits. We analyze the challenges of using overlapping

[1]It stands for "Versioned Execution Equivalence Verifier."

windows, and propose a solution based on the "decomposition" concept. We discuss the correctness and the completeness of our algorithm (§5).
(5) We provide a number of optimizations in Veer$^+$ to improve the performance of the baseline algorithm (§ 7).
(6) We report the results of a thorough experimental evaluation of the proposed solutions. The experiments show that the proposed solution is not only able to verify workflows that cannot be verified by existing EVs, but also able to do the verification efficiently (§ 9).

2 PROBLEM FORMULATION

In this section, we use an example workflow to describe the setting. We also formally define the problem of verifying equivalence of two workflow versions. Table 2 shows a summary of the notations used in this section.

Table 2: Notations used for a single workflow.

Notation	Description
W, DAG	A data processing workflow
$\mathbb{D}_w = \{D_1, \ldots, D_l\}$	A set of data sources in the workflow
$\mathbb{S}_w = \{s_1, \ldots, s_n\}$	A set of sinks in the workflow
$\mathcal{M}$	An edit mapping between two versions
δ_j	A set of edit operations to transform DAG v_j to v_{j+1}
$\oplus$	Applying aggregated edit operations on a workflow version
$\mathcal{V}_w = [v_1, \ldots, v_m]$	A list of workflow versions

Data processing workflow. We consider a data processing workflow W as a directed acyclic graph (DAG), where each vertex is an operator and each link represents the direction of data flow. Each operator contains a computation function, we call it a *property* such as a predicate condition, e.g., Price < 20. Each operator has outgoing links, and its produced data is sent on each outgoing link. An operator without any incoming links is called a Source. An operator without any outgoing links is called a Sink, and it produces the final results as a table to be consumed by the user. A workflow may have multiple data source operators denoted as $\mathbb{D}_W = \{D_1, \ldots, D_l\}$ and multiple sink operators denoted as $\mathbb{S}_W = \{s_1, \ldots, s_n\}$.

For example, consider a workflow in Figure 1a. It has two source operators "Tweets" and "Users" and two sink operators s_i and s_p to show a tabular result and a scatterplot visualization, respectively. The OuterJoin operator has two outgoing links to push its processed tuples to the downstream Aggregate and Sink operators. The Filter operator's properties include the boolean selection predicate.

2.1 Workflow Version Control

A workflow W undergoes many edits from the time it was first constructed as part of the iterative process of data analytics [34, 51]. A workflow W has a list of versions $V_W = [v_1, \ldots, v_m]$ along a timeline in which the workflow changes. Each v_j is an immutable version of workflow W in one time point following version v_{j-1}, and contains a number of edit operations to transform v_{j-1} to v_j.

Definition 2.1 (Workflow edit operation). We consider the following edit operations on a workflow:

- An addition of a new operator.
- A deletion of an existing operator.
- A modification of the properties of an operator while the operator's type remains the same, e.g., changing the predicate condition of a Select operator.
- An addition of a new link.
- A removal of an existing link. [2]

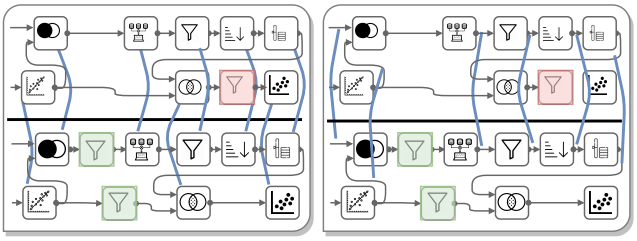
A combination of these edit operations is a *transformation*, denoted as δ_j. The operation of applying the transformation δ_j to a workflow version v_j is denoted as $\oplus$, which produces a new version v_{j+1}. Formally,

$$v_{j+1} = v_j \oplus \delta_j. \quad (1)$$

In the running example, the analyst makes edits to revise the workflow version v_1 in Figure 1a. In particular, she (1) deletes the Filter_o operator; (2) adds a new Filter_h operator; (3) and adds a new Filter_g operator. These operations along with the necessary link changes to add those operators correspond to a transformation, δ_1 and applying it on v_1 will result in a new version v_2, illustrated in Figure 1b.

Workflow edit mapping. Given a pair of versions (P, Q) and an edit mapping $\mathcal{M}$, there is a corresponding transformation from P to Q, which aligns every operator in P to at most one operator in Q. Each operator in Q is mapped onto by at most one operator in P. A link between two operators in P maps to a link between the corresponding operators in Q. Those operators and links in P that are not mapped to any operators and links in Q are assumed to be deleted. Similarly, those operators and links in Q that are not mapped onto by any operators and links in P are assumed to be inserted.

Figure 2 shows an example edit mapping between the two versions v_1 and v_2 in the running example. As Filter_y from v_1 is deleted, the opertor is not mapped to any operator in v_2.



(a) Mapping of operators. **(b) Mapping of links.**

Figure 2: Example of an edit mapping between version v_1 and v_2. Portions of the workflows are omitted for clarity.

[2]We assume links do not have properties. Our solution can be generalized to the case where links have properties.

2.2 Workflow's Execution and Results

A user submits an execution request to run a workflow version. The execution produces *result* of each sink in the version.

Assumption. *Multiple executions of a workflow (or a portion of the workflow) will always produce the same results* [3].

Result equivalence of workflow versions. The execution request for the version v_j may produce a sink result equivalent to the corresponding sink of a previous executed version v_{j-k}, where $k < j$. For example, in Figure 1b, executing the workflow version v_2 produces a result of the scatterplot sink s_2 equivalent to the result of the corresponding scatterplot of v_1. In particular, v_2's edit is pushing down the Filter operator and the scatterplot result remains the same. Notice however that the result of s_i in v_2 is not equivalent to the result of s_i in v_1 because of the addition of the new Filter_h operator. Now, we formally define "sink equivalence."

Definition 2.2 (Sink Equivalence and Version-Pair Equivalence). Consider two workflow versions P and Q with a set of edits $\delta = \{c_1 \dots c_n\}$ and the corresponding mapping $\mathcal{M}$ from P to Q. Each version can have multiple sinks. For each sink s of P, consider the corresponding sink $\mathcal{M}(s)$ of Q. We say s is *equivalent to* $\mathcal{M}(s)$, denoted as "$s \equiv \mathcal{M}(s)$," if for every instance of data sources of P and Q, the two sinks produce the same result. We say s is *inequivalent to* $\mathcal{M}(s)$, denoted as "$s \not\equiv \mathcal{M}(s)$," if there exists an instance of data sources of P and Q where the two sinks produce different results. The two versions are called *equivalent*, denoted as "$P \equiv Q$", if each pair of their sinks under the mapping is equivalent. The two versions are called *inequivalent*, denoted as "$P \not\equiv Q$, if any pair of their sinks under the mapping is inequivalent.

In this paper, we study the problem where the two versions have a single sink. We generalize the solution to the case of multiple sinks in an application of this work.

Expressive power of workflows and SQL queries. Data-processing workflows may involve complex operations, such as topic modeling and sentiment analysis on unstructured data. Workflow DAGs can be viewed as a class of SQL queries that do not contain recursion. Thus, the problem of testing the equivalence of two workflow versions can be treated as testing the equivalence of two SQL queries.

2.3 Equivalence Verifiers (EVs)

An equivalence verifier (or "EV" for short) takes as an input a pair of SQL queries $\mathbf{Q_1}$ and $\mathbf{Q_2}$. An EV returns `True` when $\mathbf{Q_1} \equiv \mathbf{Q_2}$, `False` when $\mathbf{Q_1} \not\equiv \mathbf{Q_2}$, or `Unknown` when the EV cannot determine the equivalence of the pair under a specific table semantics [13, 16, 23, 50, 55, 56]. For instance, UDP [13] and Equitas [56] are two EVs. The former uses U-expressions to model a query while the latter uses a symbolic representation. Both EVs internally convert the expressions to a first-order-logic (FOL) formula and then push the formula to a solver such as an SMT solver [16] to decide its satisfiability. An EV requires two queries to meet certain requirements (called "restrictions") in order to test their equivalence. We will discuss these restrictions in detail in Section 4.2.

[3] This assumption is valid in many real-world applications as we detail in the experiment Section 9

Problem Statement. *Given an EV and two workflow versions P and Q with their mapping $\mathcal{M}$, verify if the two versions are equivalent.*

3 VEER: VERIFYING EQUIVALENCE OF A VERSION PAIR

In this section, we first give an overview of Veer for checking equivalence of a pair of workflow versions (Section 3.1). We formally define the concepts of "window" and "covering window" (Section 3.2).

3.1 Veer: Overview

To verify the equivalence of a pair of sinks in two workflow versions, Veer leverages the fact that the two versions are mostly identical except for a few places with edit operations. It uses existing EVs as a black box. Given an EV, our approach is to break the version pair into multiple "windows," each of which includes local changes and satisfies the EV's restrictions to verify if the pair of portions of the workflow versions in the window is equivalent, as illustrated in Figure 3. We consider different semantics of equivalence between two tuple collections, including sets, bags, and lists, depending on the application of the workflow and the given EV. Veer is agnostic to the underlying EVs, making it usable for any EV of choice.

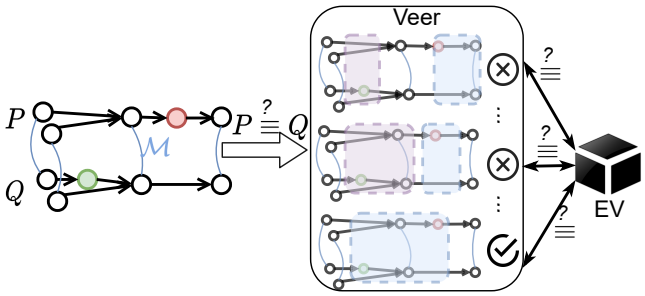


Figure 3: Overview of Veer. Given an EV and two versions with their mapping, Veer breaks (decomposes) the version pair into small windows, each of which satisfies the EV's restrictions. It finds different possible decompositions until it finds one with each of windows verified as equivalent by the EV.

Next we define concepts used in this approach.

3.2 Windows and Covering Windows

Definition 3.1 (Window). Consider two workflow versions P and Q with a set of edits $\delta = \{c_1 \dots c_n\}$ from P to Q and a corresponding mapping $\mathcal{M}$ from P to Q. A *window*, denoted as ω, is a pair of sub-DAGs $\omega(P)$ and $\omega(Q)$, where $\omega(P)$ (respectively $\omega(Q)$) is a connected induced sub-DAG of P (respectively Q). Each pair of operators/links under the mapping $\mathcal{M}$ should be either both in ω or both outside ω.

The operators in the sub-DAGs $\omega(P)$ and $\omega(Q)$ without outgoing links are called their *sinks*. Recall that we assume each workflow has a single sink. However, the sub-DAG $\omega(P)$ and $\omega(Q)$ may have more than one sink. This can happen, for example, when the window contains a Replicate operator. A *neighbor* of a window is either an

operator before a source operator of the window or an operator after a sink of the window. Figure 4 shows a window ω, where each sub-DAG includes the Classifier operator and two downstream operators Left-Outerjoin and Join, which are two sinks of the sub-DAG.

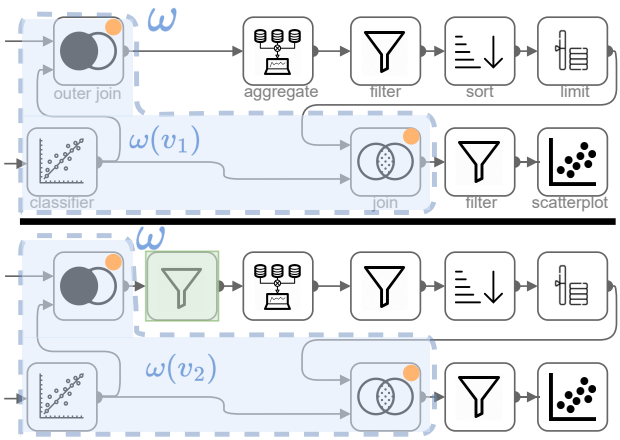


Figure 4: An example window ω and each sub-DAG of $\omega(v_1)$ and $\omega(v_2)$ contains two sinks (shown as "●").

Definition 3.2 (Covering window). Consider two workflow versions P and Q with a set of edits $\delta = \{c_1 \dots c_n\}$ from P to Q and a corresponding mapping $\mathcal{M}$ from P to Q. A *covering window*, denoted as ω_C, is a window to cover a set of changes $C \subseteq \delta$. That is, the sub-DAG in P (respectively sub-DAG in Q) in the window includes the sources if any (respectively targets, if any) operators/links of the edit operations in C.

When the edit operations are clear in the context, we will simply write ω to refer to a covering window. Figure 5 shows a covering window for the change of adding the operator Filter_h to v_2. The covering window includes the sub-DAG $\omega(v_1)$ of v_1 and contains the Aggregate operator. It also includes the sub-DAG $\omega(v_2)$ of v_2 and contains the Filter_h and Aggregate operators.

Definition 3.3 (Equivalence of the two sub-DAGs in a window). We say the two sub-DAGs $\omega(P)$ and $\omega(Q)$ of a window ω are *equivalent*, denoted as "$\omega(P) \equiv \omega(Q)$," if they are equivalent as two stand-alone DAG's, i.e., without considering the constraints from their upstream operators. That is, for every instance of source operators in the sub-DAGs (i.e., those operators without ancestors in the sub-DAGs), each sink s of $\omega(P)$ and the corresponding $\mathcal{M}(s)$ in $\omega(Q)$ produces the same results. In this case, for simplicity, we say this window is equivalent.

Figure 6 shows an example of a covering window ω', where its sub-DAGs $\omega'(v_1)$ and $\omega'(v_2)$ are equivalent.

Notice that for each sub-DAG in the window ω, the results of its upstream operators are the input to the sub-DAG. The equivalence definition considers all instances of the sources of the sub-DAG, without considering the constraints on its input data as the results of upstream operators. For instance, consider the two workflow

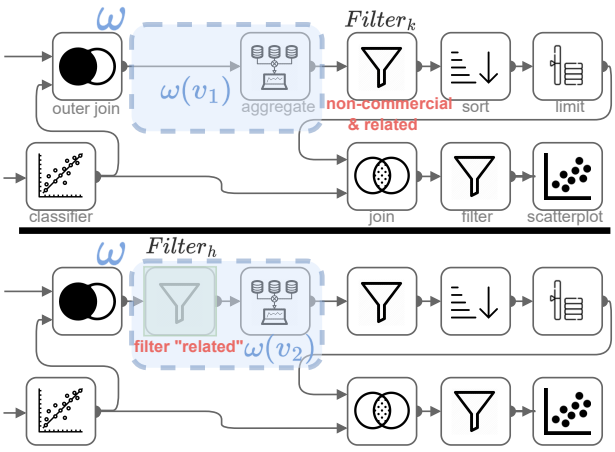


Figure 5: A covering window ω for adding Filter_h.

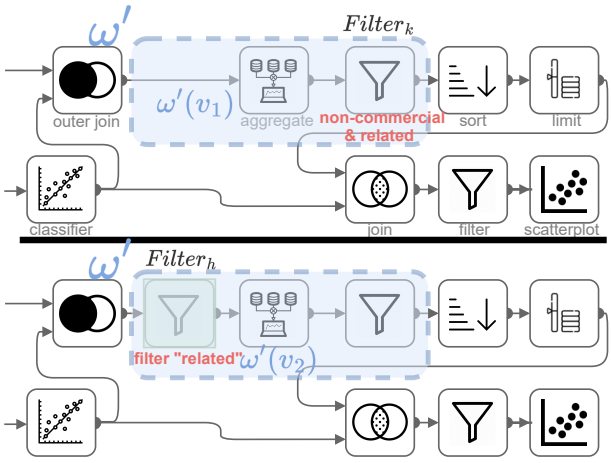


Figure 6: An example covering window ω' showing its pair of sub-DAGs are equivalent.

versions in Figure 7. The two sub-DAGs of the shown window ω are clearly not equivalent as two general workflows, as the top sub-DAG has a filter operator, while the bottom one does not. However, if we consider the constraints of the input data from the upstream operators, the sub-DAGs in ω are indeed equivalent, because each of them has an upstream filter operator with a predict $age < 50$, making the predicate $age < 55$ redundant. We use this definition of sub-DAG equivalence despite the above observation, because we treat the sub-DAGs in a window as a pair of stand-alone workflow DAGs to pass to the EV for verification (see Section 4.1).

Definition 3.4 (Window containment). We say a window ω is *contained* in a window ω', donated as $\omega \subseteq_w \omega'$, if $\omega(P)$ (respectively $\omega(Q)$) of ω is a sub-DAG of the corresponding one in ω'. In this case, we call ω a *sub-window* of ω', and ω' a *super-window* of ω.

For instance, the window ω in Figure 5 is contained in the window ω' in Figure 6.

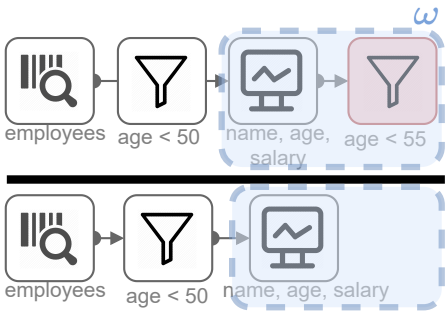
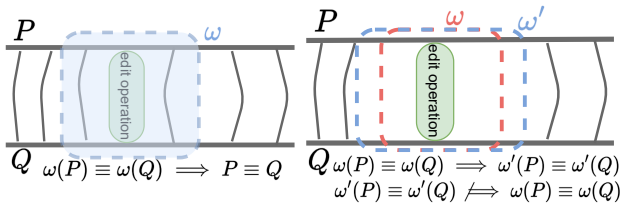


Figure 7: Two sub-DAGs in the window ω are not equivalent, as sub-DAG equivalence in Definition 3.3 does not consider constraints from the upstream operators. But the two complete workflow versions are indeed equivalent.



(a) Using a covering window to check the equivalence of two versions.

(b) Subsumption of windows and their relation to version equivalence.

Figure 8: Conceptual examples to explain the relation between a "covering window" and version pair equivalence.

4 TWO VERSIONS WITH A SINGLE EDIT

In this section, we study how to verify the equivalence of two workflow versions P and Q with a single change c of the corresponding mapping $\mathcal{M}$ from P to Q. We leverage a given EV γ to verify the equivalence of two queries. We discuss how to use the EV to verify the equivalence of the version pair in a window (Section 4.1), and discuss the EV's restrictions (Section 4.2). We present a concept called "maximal covering window", which helps in improving the performance of verifying the equivalence (Section 4.3), and develop a method to find maximal covering windows to verify the equivalence of the two versions (Section 4.4).

4.1 Verification Using a Covering Window

We show how to use a covering window to verify the equivalence of a version pair.

Lemma 4.1. *Consider a version pair (P, Q) with a single edit c operation between them. If there is a covering window $\omega = (\omega(P), \omega(Q))$ of the edit operation such that the sub-DAGs of the window are equivalent, then the version pair is equivalent.*

Proof. Suppose $\omega(P) \equiv \omega(Q)$. From the definition of a covering window, every operator in one sub-DAG of the window ω has its corresponding mapped operator in the other sub-DAG of the window, and the change c is included in the window. This means that the sub-DAGs of P and Q that precede the window ω are isomorphic (structurally identical) and the sub-DAGs of P and Q that follow the window are isomorphic as shown in Figure 8a. Following the assumption that multiple runs of a workflow produce the same result, this infers that given an instance of input sources $\mathbb{D}$, the sub-DAGs before the window would produce equivalent results according to definition 2.2. This result becomes the input source for the window ω and given that the sub-DAGs in ω are equivalent, this means that each sink of $\omega(P)$ is equivalent to the corresponding sink (according to the mapping) of $\omega(Q)$. Hence, the output of the window, which is the input to the pair of sub-DAGs following the window, is identical, and since the operators are isomorphic, the result of the sub-DAGs following the window is equivalent. Thus, $P \equiv Q$. □

Based on this lemma, we can verify the equivalence of a pair of versions as follows: We consider a covering window and check the equivalence of its sub-DAGs by passing each pair of sinks and the sink's ancestor operators in the window (to form a query pair) to an EV. If the EV shows that all the sink pairs are equivalent, then the two versions are equivalent.

A key question is how to find such a covering window. Notice that the two sub-DAGs in Figure 5 are not equivalent. However, if we include the downstream Filter_k in the covering window to form a new window ω' (shown in Figure 6) with a pair of sub-DAGs $\omega'(P)$ and $\omega'(Q)$, then the two sub-DAGs in ω' are equivalent. This example suggests that we may need to consider multiple windows in order to find one that is equivalent.

4.2 EV Restrictions and Valid Windows

We cannot give an arbitrary window to the EV, since each EV has certain restrictions on the sub-DAGs to verify their equivalence.

Definition 4.2 (EV's restrictions). Restrictions of an EV are a set of conditions such that for each query pair if this pair satisfies these conditions, then the EV is able to determine the equivalence of the pair without giving an Unknown answer.

We will relax this definition in Section 8, discuss the consequences of relaxing the definition, and propose solutions. There are two types of restrictions.

- Restrictions due to the EV's explicit assumptions: For example, UDP and Equitas support reasoning of certain operators, e.g., Aggregate and SPJ, but not other operators such as Sort.
- Restrictions that are derived due to the modules used by the EV: For example, Equitas [56], Spes [55], and Spark Verifier [23] use an SMT solver [16] to determine if a FOL formula is satisfiable or not. SMT solver is not complete for determining the satisfiability of formulas when their predicates have non-linear conditions [8]. Thus, these EVs require the predicate conditions in their expressions to to be linear to make sure to receive an answer from the solver.

As an example, the following is an example of the explicit and *derived* restrictions of the Equitas [56] to test the equivalence of two queries [4].

[4] Applications that wish to use Veer need to extend it to include their EV of choice if it is not Equitas or Spes, and incorporate the restrictions specific to those EVs.

R1. The table semantics has to be set semantics.[5]
R2. All operators have to be any of the following types: SPJ, Outer join, and/or Aggregate.
R3. The predicate conditions of SPJ operators have to be linear.
R4. Both queries should have the same number of Outer join operators, if present.
R5. Both queries should have the same number of Aggregate operators, if present.
R6. If they use an Aggregate operator with an aggregation function that depends on the cardinality of the input tuples, e.g., COUNT, then each upstream operator of the Aggregate operator has to be an SPJ operator, and the input tables are not scanned more than once.

Definition 4.3 (Valid window w.r.t an EV). We say a window is *valid* with respect to an EV if it satisfies the EV's restrictions.

In order to test if a window is valid, we pass it to a "validator", in which checks if the window satisfies the EV restrictions or not.

4.3 Maximal Covering Window (MCW)

A main question is how to find a valid covering window with respect to the given EV using which we can verify the equivalence of the two workflow versions. A naive solution is to consider all the covering windows of the edit change c. For each of them, we check its validity, e.g., whether they satisfy the constraints of the EV. If so, we pass the window to the EV to check the equivalence. This approach is computationally costly, since there can be many covering windows. Thus our focus is to reduce the number of covering windows that need to be considered without missing a chance to detect the equivalence of the two workflow versions. The following lemma helps us reduce the search space.

Lemma 4.4. *Consider a version pair (P, Q) with a single edit c between them. Suppose a covering window ω of c is contained in another covering window ω'. If the sub-DAGs in window ω are equivalent, then the sub-DAGs of ω' are also equivalent.*

Proof. Suppose $\omega(P) \equiv \omega(Q)$. Suppose a window ω' consists of the sub-DAGs of the entire version pair, i.e. $\omega'(P) = P$ and $\omega'(Q) = Q$. This means that $\omega \subseteq \omega'$ as $\omega(P) \subseteq \omega'(P)$ and $\omega(Q) \subseteq \omega'(Q)$. Given that the sub-DAGs in ω are equivalent, from Lemma 4.1, we can infer the version pair is equivalent, which means the sub-DAGs in the window ω' are equivalent. □

Based on Lemma 4.4, we can just focus on covering windows that have as many operators as possible without violating the constraints of the EV. If the EV shows that such a window is not equivalent, then none of its sub-windows can be equivalent. (A subtle case is when the EV does not know if the window ω' is equivalent, but can verify that ω is equivalent. We will discuss this case in Section 8.) Based on this observation, we introduce the following concept.

Definition 4.5 (Maximal Covering Window (MCW)). Given a workflow version pair (P, Q) with a single edit operation c, a valid covering window ω is called *maximal* if it is not properly contained by another valid covering window.

[5]In this work, the application determines the desired table semantics, and Veer decides to use an EV that supports the specified table semantics requested by the application by checking the restriction.

The change c may have more than one MCW, For example, suppose the EV is Equitas. Figure 9 shows two MCWs to cover the change of adding the Filter_h operator. One maximal window ω_1 includes the change Filter_h and Left Outerjoin on the left of the change. The window cannot include the Classifier operator from the left side because Equitas cannot reason its semantics. Similarly, the Aggregate operator on the right cannot be included in ω_1 because one of Equitas restrictions is that the input of an Aggregate operator must be an SPJ operator and the window already contains Left OuterJoin. To include the Aggregate operator, a new window ω_2 is formed to exclude Left OuterJoin and include Filter on the right but cannot include Sort because this operator cannot be reasoned by Equitas.

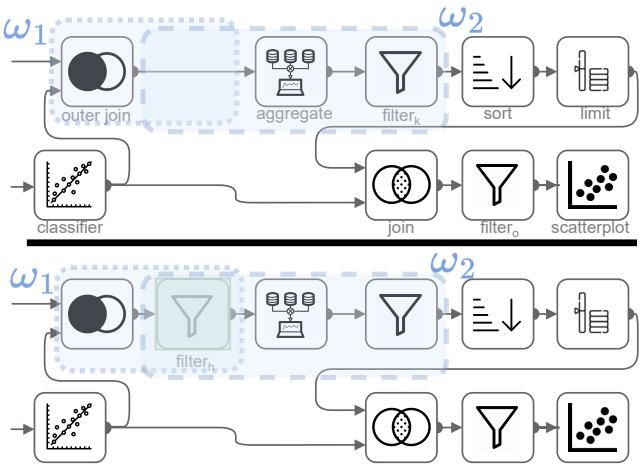


Figure 9: Two MCW ω_1 and ω_2 satisfying the restrictions of Equitas to cover the change of adding Filter_h to v_2.

The MCW ω_2 is verified by Equitas to be equivalent, whereas ω_1 is not. Notice that one equivalent covering window is enough to show the equivalence of the two workflow versions.

4.4 Finding MCWs to Verify Equivalence

Next we study how to efficiently find an MCW to verify the equivalence of two workflow pairs. We present a method shown in Algorithm 1. Given a version pair P and Q and a single edit operation c based on the mapping $\mathcal{M}$, , the method finds an MCW that is verified by the given EV γ to be equivalent.

We use the example in Figure 10 to explain the details of Algorithm 1. The first step is to initialize the window to cover the source and target operator of the change only (line 1). In this example, for the window ω_1, its sub-DAG $\omega_1(v_2)$ contains only Filter_h and its corresponding operator using the mapping $\mathcal{M}$ in $\omega_1(v_1)$. Then we expand all the windows created so far, i.e., ω_1 in this case (line 2). To expand the window, we enumerate all possible combinations of including the neighboring operators on both $\omega_1(v_1)$ and $\omega_1(v_2)$ using the mapping. For each neighbor, we form a new window and check if it has not been explored yet. If not, then we check if the newly formed window is valid (lines 5-6).

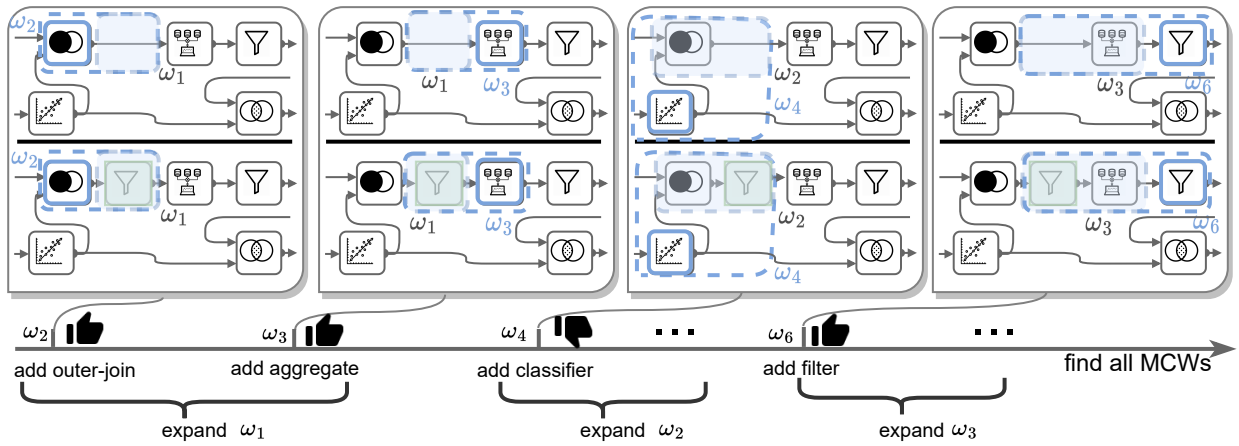


Figure 10: Example to illustrate the process of finding MCWs for the change of adding Filter_h to v_2.

Algorithm 1: Verifying equivalence of two workflow versions with a single edit

Input: A version pair (P, Q); A single edit c; A mapping $\mathcal{M}$; An EV γ
Output: A flag to indicate if they are equivalent

```
// a True value to indicate the pair is equivalent, a False value to indicate the
pair is not equivalent, or Unknown when the pair cannot be verified
1  ω ← create an initial window to include the source and the
     corresponding target (operator/link) of the edit c
2  Ω = {ω} // initialize a set for exploring widows
   // using memoization, a window is explored only once
3  while Ω is not empty do
4      ω_i ← remove one window from Ω
5      for every neighbor of ω_i do
6          if adding neighbor to ω_i meets EV's restrictions then
7              add ω'_i (including the neighbor) to Ω
8      end
9      if none of the neighbors were added to ω_i then
           // the window is maximal
10         if ω_i is verified equivalent by the EV then
11             return True
12         if ω_i is verified not equivalent by the EV and the
             window is the entire version pair then
13             return False
14 end
15 return Unknown
```

In this example, we create the two windows ω_2 and ω_3 to include the operators Outer-join and Aggregate in each window, respectively. We add those windows marked as valid in the traversal list to be further expanded in the following iterations (line 7). We repeat the process on every window. After all the neighbors are explored to be added and we cannot expand the window anymore, we mark it as maximal (line 9). Then we test the equivalence of this maximal window by calling the EV. If the EV says it is equivalent, the algorithm returns TRUE to indicate the version pair is equivalent (line 10). If the EV says that it is not equivalent and the window's sub-DAGs are the complete version pair, then the algorithm returns `False` (line 13). Otherwise, we iterate over other windows until there are no other windows to expand. In that case, the algorithm returns `Unknown` to indicate that the version equivalence cannot be verified as in line 15.

Some EVs [13, 55, 56] return `False` to indicate that the equivalence of the version pair cannot be verified, but it does not necessarily mean that the pair is inequivalent. We take note of these EVs, and in the algorithm mentioned above, we only report `False` if the EV is capable of proving the inequivalence of the pair, such as COSETTE [14].

5 TWO VERSIONS WITH MULTIPLE EDITS

In the previous section, we assumed there is a single edit operation to transform a workflow version to another version. In this section, we extend the setting to discuss the case where multiple edit operations $\delta = \{c_1 \dots c_n\}$ transform a version P to a version Q. A main challenge is finding covering windows for multiple edits (Section 5.1). We address the challenge by decomposing the version pair into a set of *disjoint* windows. We formally define the concepts of "decomposition" and "maximal decomposition" (Section 5.2). We explain how to find maximal decompositions to verify the equivalence of the version pair and prove the correctness of our solution (Section 5.4). We analyze the completeness of the proposed algorithm (Section 5.5).

5.1 Can we use overlapping windows?

When the two versions have more than one edit, they can have multiple covering windows. A natural question is whether we can use covering windows that overlap with each other to test the equivalence of the two versions. We will use an example to show that we cannot do that. The example, shown Figure 11, is inspired from the NY Taxi dataset [37] to calculate the trip time based on the duration and starting time. Suppose the Select_x and Select_z operators are deleted from a version v_1 and Select_y operator is

added to transform the workflow to version v_2. The example shows two overlapping windows ω and ω', each window is equivalent.

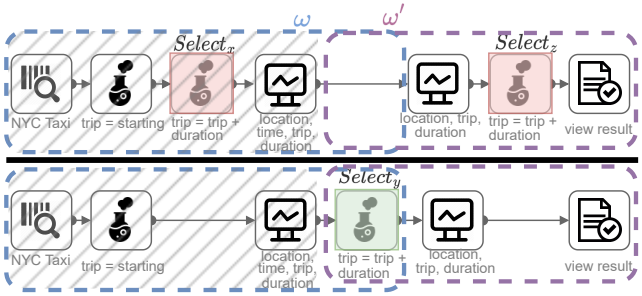


Figure 11: In this example, the blue window ω is equivalent and the purple window ω' is also equivalent. But the version pair is not equivalent. The shaded gray area is the input to window ω'.

We cannot say the version pair is equivalent. The reason is that for the pair of sub-DAGs in ω' to be equivalent, the input sources have to be the same (the shaded area in grey in the example). However, we cannot infer the equivalence of the outcome of that portion of the sub-DAG. In fact, the pair of sub-DAGs in the shaded area in this example produce different results. This problem does not exist in the case of a single edit, because the input sources to any *covering* window (in a single edit case) will always be a one-to-one mapping of the two sub-DAGs and there is no other change outside the covering window. The solution in Section 4 finds *any* window such that its sub-DAGs are equivalent and cannot be directly used to solve the case of verifying the equivalence of the version pair when there are multiple edits.

To overcome this challenge and enable using windows to check the equivalence of the version pair, we require the covering windows to be disjoint. In other words, each operator be included in one and only one window. A naive solution is to do a simple exhaustive approach of decomposing the version pair into all possible combinations of disjoint windows. Next, we formally define a version pair decomposition and how it is used to check the equivalence of a version pair.

5.2 Version Pair Decomposition

Definition 5.1 (Decomposition). For a version pair P and Q with a set of edit operations $\delta = \{c_1 \dots c_n\}$ from P to Q, a *decomposition*, θ is a set of windows $\{\omega_1, \dots, \omega_m\}$ such that:

- Each edit is in one and only one window in the set;
- All the windows are disjoint;
- The union of the windows is the version pair.

Figure 12 shows a decomposition for the three changes in the running example. The example shows two covering windows ω_1 and ω_2, each covers one or more edits [6]. Next, we show how to use a decomposition to verify the equivalence of the version pair by generalizing Lemma 4.1 as follows.

[6]For simplicity, we only show covering windows of a decomposition in the figures throughout this section.

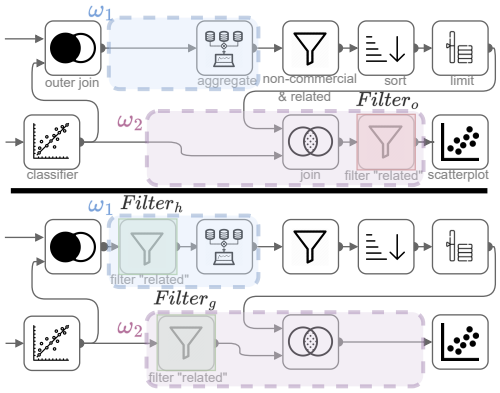


Figure 12: A decomposition θ with two covering windows ω_1 and ω_2 that cover the three edits.

Lemma 5.2. *(Corresponding to Lemma 4.1) For a version pair P and Q with a set of edit operations $\delta = \{c_1 \dots c_n\}$ to transform P to Q, if there is a decomposition θ such that every covering window in θ is equivalent, then the version pair is equivalent.*

Proof. Suppose every covering window ω_i in a decomposition θ is equivalent. Every other window that is not covering, its sub-DAGs are structurally identical, according to Definition 3.2. Given an instance of input sources $\mathbb{D}$, we can have the following two cases. (CASE1:) the input is processed by a pair of structurally identical sub-DAGs that are in a non-covering window. In this case, the pair of sub-DAGs produce an equivalent result since every operator is deterministic according to Assumption 2.2. (CASE2:) the input is processed by a pair of sub-DAGs in a covering window. In this case, the pair of sub-DAGs produce equivalent result because we assumed each covering window is equivalent. In both cases, the output acts as the input to the following portion of the sub-DAGs (either non-covering or a covering window). This propagation continues along the pair of DAGs until the end, thus the version pair produces equivalent results as shown in Figure 13. □

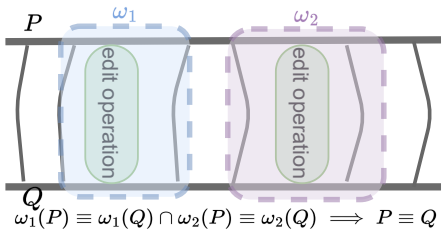


Figure 13: Using multiple covering windows on multiple edits to check the equivalence of two versions.

A natural question is how to find a decomposition where each of its windows is equivalent. We could exhaustively consider all the possible decompositions, but the number can grow exponentially as the size of the workflow and the number of changes increase. The

following "decomposition containment" concept, defined shortly, helps us reduce the number of decompositions that need to be considered.

Definition 5.3 (Decomposition containment). We say a decomposition θ is *contained* in another decomposition θ', denoted as $\theta \subseteq_d \theta'$, if every window in θ, there exists a window in θ' that contains it.

Figure 14 shows an example of a decomposition θ' that contains the decomposition θ in Figure 12. We can see that in general, if a decomposition θ is contained in another decomposition θ', then each window in θ' is a concatenation of one or multiple windows in θ.

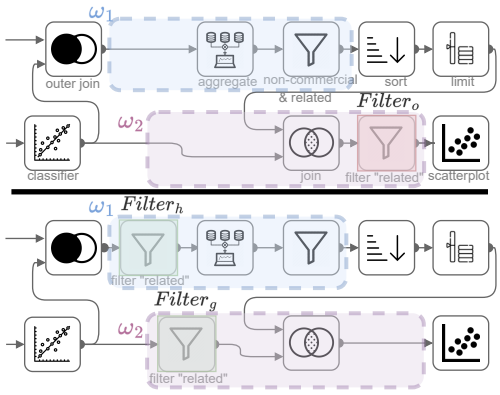


Figure 14: Example to show equivalent pair of sub-DAGs of every covering window in a decomposition θ'.

The following lemma, which is a generalization of Lemma 4.4, can help us prune the search space by ignoring decompositions that are properly contained by other decompositions.

Lemma 5.4. *(Corresponding to Lemma 4.4) Consider a version pair P and Q with a set of edit operations $\delta = \{c_1 \dots c_n\}$ from P to Q. Suppose a decomposition θ is contained in another decomposition θ'. If each window in θ is equivalent, then each window in θ' is also equivalent.*

Proof. Suppose each window in a decomposition θ is equivalent and the decomposition is contained in another decomposition θ'. Based on the Definition of decomposition containment 5.2, we know that each window in θ is contained in a window in θ'. According to Lemma 4.4, if a window is equivalent then a window that contains it is also equivalent. We can deduce that every window in θ' is equivalent, therefore the version pair is equivalent as per Lemma 5.2. □

5.3 Maximal Decompositions w.r.t. an EV

Lemma 5.4 shows that we can safely find decomposition that contain other ones to verify the equivalence of the version pair. At the same time, we cannot increase each window arbitrarily, since the equivalence of each window needs to be verified by the EV, and the window needs to satisfy the restrictions of the EV. Thus we want decompositions that are as containing as possible while each window is still valid. We formally define the following concepts.

Definition 5.5 (Valid Decomposition). We say a decomposition θ is *valid* with respect to an EV if each of its covering windows is valid with respect to the EV.

Definition 5.6 (Maximal Decomposition (MD)). We say a valid decomposition θ is *maximal* if no other valid decomposition θ' exists such that θ' properly contains θ.

The decompositions w.r.t an EV form a unique graph structure, where each decomposition is a node. It has a single root corresponding to the decomposition that includes every operator as a separate window. A downward edge indicates a "contained-in" relationship. A decomposition can be contained in more than one decomposition. Each leaf node at the bottom of the hierarchy is an MD as there are no other decompositions that contain it and the hierarchy may not be balanced. If the entire version pair satisfies the EV's restrictions, then the hierarchy becomes a lattice structure with a single leaf MD being the entire version pair. Each branching factor depends on the number of changes, the number of operators, and the EV's restrictions. Figure 15 shows the hierarchical relationships of the valid decompositions of the running example when the EV is Equitas. The example shows two MD θ_{12} and θ_{16}.

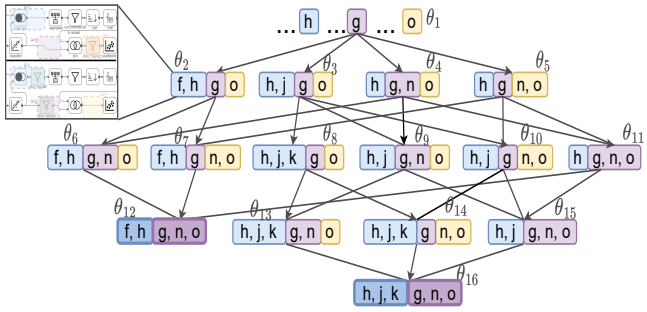


Figure 15: Hierarchy of valid decompositions w.r.t an EV. Each letter corresponds to a pair of operators from the running example. We show the containment of covering windows and we omit details of containment of non-covering windows.

5.4 Finding a Maximal Decomposition to Verify Equivalence (A Baseline Approach)

Now we present an algorithm for finding maximal decompositions shown in Algorithm 2. We will explain it using the example in Figure 15. We return `True` to indicate the pair is equivalent if there are no changes and the two versions exactly match (Line 1-2). Otherwise, we add the initial decomposition, which includes each operator as a window, to a set of decompositions to be expanded (line 3). In each of the following iterations, we remove a decomposition from the set, and iteratively expand its windows. To expand a window, we follow the procedure as in Algorithm 1 to expand its neighbors. The only difference is that the neighbors in this case are windows, and we merge windows if their union is valid (line 10). If a window cannot be further expanded, then we mark the window as maximal to avoid checking it again (line 14). If all of the windows in the decomposition are maximal, we mark the decomposition as

maximal, and verify whether each covering window is equivalent by passing it to the given EV (line 17). If all of the windows are verified to be equivalent, we return `True` to indicate that the version pair is equivalent (line 18). If in the decomposition there is only a single window, which includes the entire version pair, and the EV decides that the window is not equivalent, then the algorithm returns `False` (line 20). Otherwise, we continue exploring other decompositions until there are no more decompositions to explore. In that case, we return `Unknown` to indicate that the equivalence of the version pair cannot be determined (line 22). This algorithm generalizes Algorithm 1 to handle cases of two versions with multiple edits.

Algorithm 2: Verifying the equivalence of a workflow version pair with one or multiple edits (Baseline)

Input: A version pair (P, Q); A set of edit operations δ and a mapping $\mathcal{M}$ from P to Q; An EV γ
Output: A version pair equivalence flag EQ

```
// A True value indicates the pair is equivalent, a False value indicates the pair is not equivalent, and an Unknown value indicates the pair cannot be verified
1  if δ is empty then
2      return True
3  θ ← decomposition with each operator as a window
4  Θ = {θ} // initial set of decompositions
5  while Θ is not empty do
6      Remove a decomposition θ_i from Θ
7      for every covering window ω_j (in θ_i) not marked do
8          for each neighbor ω_k of ω_j do
9              if ω_k ∪ ω_j is valid and not explored before then
10                 θ'_i ← θ − ω_k − ω_j + ω_k ∪ ω_j
11                 add θ'_i to Θ
12         end
13         if none of the neighbor windows can be merged then
14             mark ω_j
15     end
16     if every covering ω ∈ θ_i is marked then
17         if γ verifies each covering window in θ_i to be equivalent then
18             return True
19         if θ_i has only one window and γ verifies it not to be equivalent then
20             return False
21 end
22 return Unknown
```

Theorem 5.7. *(Correctness). Given a workflow version pair (P, Q), an edit mapping, and a sound EV, 1) if* Veer *returns* `True`*, then* $P \equiv Q$*, and 2) if* Veer *returns* `False`*, then* $P \not\equiv Q$*.*

Proof. 1) Suppose $P \not\equiv Q$. According to definition 2.2, this means that for a given input sources $\mathbb{D}$, there is a tuple t that exists in the sink of P but does not exist in the sink of Q. Following Assumption 2.2 that multiple runs of a workflow produce the same result, we can infer that there must be a set of edit operations $\delta = \{c_1, \ldots, c_n\}$ to transform P to Q that caused the sink of P to contain the tuple t but does not exist in the sink of Q. Veer must find a valid maximal decomposition θ following Algorithm 2. There are four cases the procedure terminates and returns the result:

`CASE1`: The set of edits is empty and this is not the case as inferred above that there exists a change.

`CASE2`: The set of maximal decompositions is empty, because none of the decompositions satisfies the EV's restrictions or none of the decompositions were verified equivalent. In this case, Veer returns `Unknown`.

`CASE3`: There is a decomposition that is verified to be equivalent by a correct EV, which according to Lemma 5.2, implies that the version pair is equivalent given the assumptions in our setting. However, this is not the case because we assumed $P \not\equiv Q$.

`CASE4`: There is a single window in the decomposition and it is verified by the EV to be not equivalent, when the EV can verify the inequivalence of the pair, in this case Veer returns `False`.

In all cases, Veer did not return `True`, by contraposition, this proves that $P \equiv Q$.

2) We follow the same approach as above to prove the second case. □

5.5 Improving the Completeness of Algorithm 2

In general, the equivalence problem for two workflow versions is undecidable [1, 23] (reduced from First-order logic). So there is no verifier that is complete [14]. However, there are classes of queries that are decidable such as SPJ [55]. In this section, we show factors that affect the completeness of Algorithm 2 and propose ways to improve its completeness.

1) Window validity. In line 13 of Algorithm 2, if none of the neighbor windows of ω_j can be merged with ω_j to become a valid window, we mark ω_j and stop expanding it, hoping it might be a maximal window. The following example shows that this approach could miss some opportunity to find the equivalence of two versions.

Example 1. *Consider the following two workflow versions:*

$$P = \{Project(all) \rightarrow Filter(age > 24) \rightarrow Aggregate(count\ by\ age)\}.$$

$$Q = \{Aggregate(count\ by\ age) \rightarrow Filter(age > 24) \rightarrow Project(all)\}.$$

Consider the following mapping from P to Q: substituting Project in P with Aggregate in Q and substituting Aggregate in P with Project in Q. Suppose the EV is Equitas *and a covering window ω contains the Project from P and its mapped operator Aggregate from Q. Consider the window expansion procedure in Algorithm 2. If we add filter operator of both versions to the window, then the merged window is not valid. The reason is that it violates Equitas's restriction R5 in Section 4.2, i.e., both DAGs should have the same number of Aggregate operators. The algorithm thus stops expanding the window. However, if we continue expanding the window till the end, the final window with three operators is still valid.*

Using this final window, we can see that the two versions are equivalent, but the algorithm missed this opportunity. This example shows that even though the algorithm is correct in terms of claiming the equivalence of two versions, it may miss opportunities to verify

their equivalence. A main reason is that the Equitas EV does not have the following property.

Definition 5.8 (EV's Restriction Monotonicity). We say an EV is *restriction monotonic* if for each version pair P and Q, for each invalid window ω, each containing window of ω is also invalid.

Intuitively, for an EV with this property, if a window is not valid (e.g., it violates the EV's restrictions), we cannot make it valid by expanding the window. For an EV that has this property such as Spes, when the algorithm marks the window ω_j (line 14), this window must be maximal. Thus further expanding the window will not generate another valid window, and the algorithm will not miss this type of opportunity to verify the equivalence.

If the EV does not have this property such as Equitas, we can improve the completeness of the algorithm as follows. We modify line 9 by not checking if the merged window $\omega_j \cup \omega_k$ is valid or not. We also modify line 13 to test if the window ω_j is maximal with respect to the EV. This step is necessary in order to be able to terminate the expansion of a window. We assume there is a procedure for each EV that can test if a window is maximal by reasoning the EV's restrictions.

2) Different edit mappings.

Consider two different edit mappings, $\mathcal{M}_1$ and $\mathcal{M}_2$, for the version pair in Example 5.5, as shown in Figure 16. Let us assume the given EV is Equitas. If we follow the baseline Algorithm 2, mapping $\mathcal{M}_1$ results in a decomposition that violates Equitas's **R2** restriction. On the other hand, mapping $\mathcal{M}_2$ satisfies the restrictions and allows the EV to test their equivalence. This example shows that different edit mappings can lead to different decompositions. Notably, the edit distance of the first mapping is 2, while the edit distance of the second one is 4. This result shows that a minimum-distance edit mapping does not always produce the best decomposition to show the equivalence.

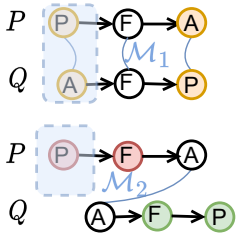


Figure 16: An example of two edit mappings, where one leads to a decomposition that satisfies an EV's restrictions, while the other does not.

One way to address this issue is to enumerate all possible edit mappings [42] and perform the decomposition search by calling Algorithm 2 for each edit mapping. If the changes between the versions are tracked, then the corresponding mapping of the changes can be treated as the first considered edit mapping before enumerating all other edit mappings.

6 COMPLETENESS OF VEER

In this section, we first discuss the completeness of Veer and the dependence on the completeness of its internal components (§6.1). Then we show examples of using different EVs to illustrate the restrictions that a workflow version pair needs to satisfy for Veer to be complete for verifying the pair's equivalence and formally prove its completeness (§6.2).

6.1 Veer's Completeness Dependency on Internal Components

For any pair of workflow versions, if the pair is equivalent and there is a valid decomposition w.r.t. to a given EV where each of its covering windows is verified as equivalent by the given EV, Veer returns `True`. Recall that Veer considers all possible edit mappings and explores all possible valid decompositions for each mapping. If there is a valid decomposition under a mapping, Veer guarantees to find it. For any pair of workflow versions, if the pair is inequivalent and there exists a valid decomposition that includes a single window consisting of the entire version pair, and this window is verified as inequivalent by the EV, then Veer returns `False`. For simplicity, throughout this section, in both cases we say there is a valid decomposition whose equivalence is determined by the given EV. In both cases, Veer does not return `Unknown`. Note that the completeness of Veer relies on the completeness of the given EV. If the EV is incomplete and returns `Unknown` to all possible valid decompositions generated by Veer, accordingly Veer returns `Unknown`.

The completeness of modern EVs [13, 23, 50, 55, 56] depends on the internal components used. For instance, most EVs [13, 23, 50, 55, 56] model queries as expressions, such as FOL formulas, and utilize a solver, e.g., SMT, to determine the satisfiability of formulas. SMT solvers [16] are complete for testing the satisfiability of linear formulas [8]. Therefore, EVs that use SMT solvers in their internal verification procedure are complete for verifying the equivalence of two queries (workflows or SQL) when the two queries include only linear conditions in their predicates. Likewise, Veer is complete for verifying the equivalence of two workflow versions that satisfy the EV's restrictions. Figure 17 illustrates the internal components Veer uses and how these components contribute to Veer's overall completeness.

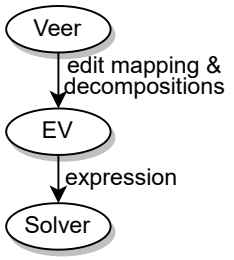


Figure 17: Components related to the equivalence verification process.

6.2 Restrictions of Some EVs and Veer's Completeness

We use the following examples on three EVs (summarized in Table 3) to explain Veer's completeness process. Suppose a given EV is Spes [55]. Spes determines the equivalence of two queries under the "Bag" table semantics. Spes is complete for determining the equivalence of two queries that satisfy the following restrictions [55]: 1) the two queries should contain only SPJ operators; 2) the selection predicates in every query should not include non-linear conditions.

Lemma 6.1. *Given two SPJ workflow versions (P, Q),* Spes *as the EV, and* Spes' *restrictions, if the two versions satisfy* Spes' *restrictions, then* Veer *can determine the equivalence of the pair. That is 1) if the two versions are equivalent, then* Veer *returns* `True`*, 2) and if the pair is not equivalent then* Veer *returns* `False`*.*

Proof. We prove this theorem with the method of contradiction.

Table 3: Example EVs and their restrictions along with how Veer is complete for verifying a version pair that satisfy the EV's restrictions.

EV	EV's restrictions	Explanation of why Veer is complete
Spes [55]	1. the pair should not include other than Select-Project-Join (SPJ.) 2. queries should have only linear predicate conditions.	Veer finds all possible windows that satisfy the EV's restrictions, and in this example, the given pair satisfies the restrictions, so Veer can find it.
UDP [13]	1. the pair should not include other than Union-SPJ (USPJ.) 2. the two versions must have a bijective mapping.	Veer finds all possible windows that satisfy the EV's restrictions under all possible mappings, and in this example, Veer finds a bijective mapping if there exists one.
Spark Verifier [23]	1. the pair should not include other than SPJ-Aggregate (SPJA.) 2. there should not be more than one aggregate operator in each version such that the aggregate is without grouping and outputs a primitive, e.g., MAX and MIN.	Veer finds all possible windows that satisfy the EV's restrictions, and in this example, the given pair satisfies the restrictions, so Veer finds it.

1) Assume the two versions P and Q are equivalent. Assume Veer does not return `True`. This means that Veer returns any of the following two values:

a) `False`. According to Algorithm 2, Veer returns `False` when there is a decomposition with a single window of the entire version pair and verified inequivalent by the EV. In this case, $P \not\equiv Q$, which contradicts the assumption.

b) `Unknown`. Veer calls Algorithm 2 multiple times on all possible mappings. Algorithm 2 returns `Unknown` when all valid decompositions w.r.t. the given EV (Spes) are explored and there was no valid decomposition that each of its covering windows was verified equivalent. This contradicts the given constraint that the pair satisfies Spes's restrictions, and Veer finds all possible valid windows and a window that includes the entire version pair is one of the possible valid windows.

Given these two cases, we prove by contradiction that Veer can verify the equivalence of P and Q.

2) We follow the same approach as above to prove the second case.

□

Theorem 6.2. *(Completeness.)* Veer *is complete for determining the equivalence of two workflow versions (P, Q) if the pair satisfies the restrictions of a given EV.*

Proof. Suppose the two versions satisfy the restrictions of the given EV. Since Veer considers all possible mappings and all possible decompositions that satisfy the EV's restrictions, it will find a decomposition with a single window that includes the entire pair because the given pair satisfies the EV's restriction. According to Definition 4.2, the EV is able to determine the equivalence of a pair if the pair satisfies the EV's restriction and the EV is sound. Veer returns the equivalence result from the EV. □

7 VEER⁺: IMPROVING VERIFICATION PERFORMANCE

In this section, we develop four techniques to improve the performance of the baseline algorithm for verifying the equivalence of two workflow versions. We show how to reduce the search space of the decompositions by dividing the version pair into segments (Section 7.1). We present a way to detect and prune decompositions that are not equivalent (Section 7.2). We also discuss how to rank the decompositions to efficiently explore their search space (Section 7.3). Lastly, we propose a way to efficiently identify the inequivalence of two workflow versions (Section 7.4).

7.1 Reducing Search Space Using Segmentations

The size of the decomposition structure in Figure 15 depends on a few factors, such as the number of operators in the workflow, the number of changes between the two versions, and the EV's restrictions. When the number of operators increases, the size of the possible decompositions increases. Thus we want to reduce the search space to improve the performance of the algorithm.

The purpose of enumerating the decompositions is to find all possible cuts of the version pair to verify their equivalence. In some cases a covering window of one edit operation will never overlap with a covering of another edit operation, as shown in Figure 18. In this case, we can consider the covering windows of those never overlapping separately. Based on this observation, we introduce the following concepts.

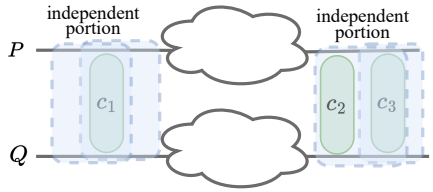


Figure 18: An example where any covering window of an edit operation c_1 never overlaps with a covering window of another edit operation c_2 or c_3.

Definition 7.1 (Segment and segmentation). Consider two workflow versions P and Q with a set of edits $\delta = \{c_1, \ldots, c_n\}$ from P to Q and a corresponding mapping $\mathcal{M}$ from P to Q. A *segment* $\mathcal{S}$ is a window of P and Q under the mapping $\mathcal{M}$. A *segmentation* ψ is a set of disjoint segments, such that they contain all the edits in δ, and there is no valid covering window that includes operators from two different segments.

A version pair may have more than one segmentation. For example, consider a version pair with a single edit. One segmentation has

a single segment, which includes the entire version pair. Another segmentation includes a segment that was constructed by finding the union of MCWs of the edit.

Computing a segmentation. We present two ways to compute a segmentation. *1) Using unions of MCWs:* For each edit $c_i \in \delta$, we compute all its MCWs, and take their union, denoted as window U_i. We iteratively check for each window U_i if it overlaps with any other window U_j, and if so, we merge them. We repeat this step until no window overlaps with other windows. Each remaining window becomes a segment and this forms a segmentation. Notice that a segment may not satisfy the restrictions of the given EV. *2) Using operators not supported by the EV:* We identify the operators not supported by the given EV. For example, a Sort operator cannot be supported by Equitas. Then we mark these operators as the boundaries of segments. The window between two such operators forms a segment. Compared to the second approach, the first one produces fine-grained segments, but is computationally more expensive.

Using a segmentation to verify the equivalence of the version pair. As there is no valid covering window spanning over two segments, we can divide the problem of checking the equivalence of P and Q into sub-problems, where each sub-problem is to check the equivalence of the two sub-DAGs in a segment. Then to prove the equivalence of a version pair, each segment in a segmentation needs to be equivalent. A segment is equivalent, if there is any decomposition such that every covering window in the decomposition is equivalent. We can organize the components of the version pair verification problem as an AND/OR tree as shown in the Figure 19.

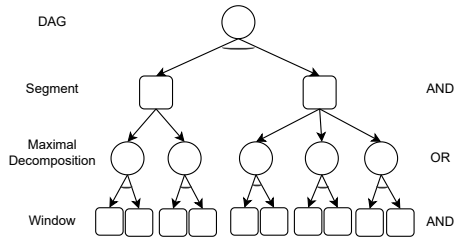


Figure 19: A sample abstract AND/OR tree to organize the components of the version pair verification problem.

Lemma 7.2. *For a version pair P and Q with a set of edit operations $\delta = \{c_1 \dots c_n\}$ from P to Q, if every segment $\mathcal{S}$ in a segmentation ψ is equivalent, then the version pair is equivalent.*

Proof. Suppose every segment S_i in a segmentation ψ is equivalent. Since according to Definition 7.1 a segment is a window and each change is covered in all of the segments in a segmentation, then we can infer that any part of the version pair that is not in a segment is structurally identical. Following the same procedure of the proof for Lemma 5.2, we can say that the result is either from a structurally identical pair of sub-DAGs or from a segment, which is said to be equivalent. We can deduce that the version pair is equivalent. □

Algorithm 3 shows how to use a segmentation to check the equivalence of two versions. We first construct a segmentation. For each segment we find if its pair is equivalent by calling Algorithm 2. If any segment is not equivalent, we can terminate the procedure early. We repeat this step until all of the segments are verified equivalent and we return `True`. Otherwise we return `Unknown`. For the case where there is a single segment consisting of the entire version pair and Algorithm 2 returns `False`, the algorithm returns `False`.

Algorithm 3: Using segments to verify the equivalence

Input: A version pair (P, Q); A set of edit operations δ and a mapping $\mathcal{M}$ from P to Q; An EV γ
Output: A version pair equivalence flag EQ

// A `True` value indicates the pair is equivalent, a `False` value to indicate the version pair is not equivalent, and an `Unknown` value indicates the pair cannot be verified

```
ψ ← constructSegmentation (P, Q, M)
for every segment S_i ∈ ψ do
    result_i ← Algorithm 2 (S_i, δ, M, γ)
    if result_i is not True then
        break
end
if every result_i is True then
    return True
if result_i is False and there is only one segment including
  the entire version pair then
    return False
return Unknown
```

Figure 20 shows the segments of the running example when using Equitas as the EV. Using the second approach for computing a segmentation, we know Equitas does not support the Sort operator, so we divide the version pair into two segments. The first one $\mathcal{S}_1$ includes those operators before Sort, and the second one $\mathcal{S}_2$ includes those operators after the Sort. The example shows the benefit of using segments to reduce the decomposition-space to a total of 8 (the sum of number of decompositions in every segment) compared to 16 (the number of all possible combinations of decompositions across segments) when we do not use segments.

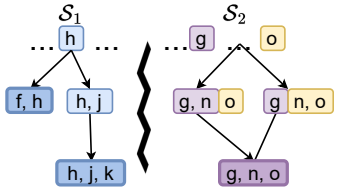


Figure 20: Two segments to reduce the decomposition-space of the running example.

7.2 Pruning Stale Decompositions

Another way to improve the performance is to prune *stale* decompositions, i.e., those that would not be verified equivalent even if they are further expanded. For instance, Figure 21 shows part of the decomposition hierarchy of the running example. Consider the decomposition θ_2. Notice that the first window, $\omega_1(f, h)$, cannot be further expanded and is marked "maximal" but the decomposition can still be further expanded by the other two windows, thus the

decomposition is not maximized. After expanding the other windows and reaching a maximal decomposition, we realize that the decomposition is not equivalent because one of its windows, e.g., ω_1, is not equivalent.

Based on this observation, if one of the windows in a decomposition becomes maximal, we can immediately test its equivalence. If it is not equivalent, we can terminate the traversal of the decompositions after this one. To do this optimization, we modify Algorithm 2 to test the equivalence of a maximal window after Line 14[7]. If the window is equivalent, we continue the search as before.

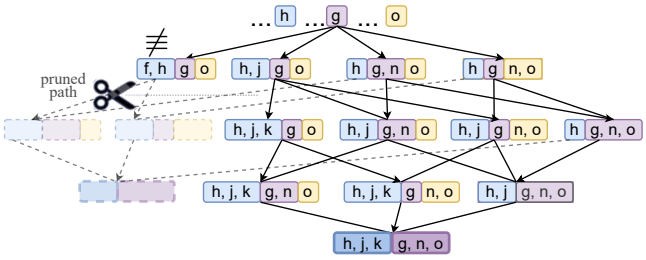


Figure 21: Example to show the pruned paths after verifying the maximal window highlighted in blue to be not equivalent.

7.3 Ranking-Based Search

Ranking segments within a segmentation. Algorithm 3 needs an order to verify those segments in a segmentation one by one. If any segment is not equivalent, then there is no need for verifying the other segments. We want to rank the segments such that we first evaluate the smallest one to get a quick answer for a possibility of early termination. We consider different signals of a segment S to compute its score. Various signals and ranking functions can be used. An example scoring function is $\mathcal{F}(S) = m_S + n_S$, where m_S is its number of operators and n_S is its number of changes. A segment should be ranked higher if it has fewer changes. The reason is that fewer changes lead to a smaller number of decompositions, and consequently, testing the segment's equivalence takes less time. Similarly, if a segment's number of operators is smaller, then the number of decompositions is also smaller and would produce the result faster.

For instance, the numbers of operators in $\mathcal{S}_1$ and $\mathcal{S}_2$ in Figure 20 are 4 and 3, respectively. Their numbers of changes are 1 and 2, respectively. The ranking score for both segments is the total of both metrics, which is 5. Then any of the two segments can be explored first, and indeed the example shows that the number of decompositions in both segments is the same.

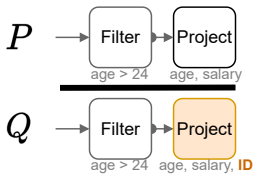
Ranking decompositions within a segment. For each segment, we use Algorithm 2 to explore its decompositions. The algorithm needs an order (line 6) to explore the decompositions. The order, if not chosen properly, can lead to exploring many decompositions before finding an equivalent one. We can optimize the performance by ranking the decompositions and performing a best-first search exploration. Again, various signals and ranking functions can be used to rank a decomposition. An example ranking function for a decomposition d is $\mathcal{G}(d) = o_d - w_d$, where o_d is the average number of operators in its covering windows, and w_d is the number of its unmerged windows (those windows that include a single operator and are not merged with a covering window). A decomposition is ranked higher if it is closer to reaching an MD for a chance of finding an equivalent one. Intuitively, if the number of operators in every covering window is large, then it may be closer to reaching an MD. Similarly, if there are only a few remaining unmerged windows, then the decomposition may be close to reaching its maximality.

For instance, decomposition θ_3 in Figure 15 has 11 unmerged windows, and the average number of operators in its covering windows is 1. While θ_6 has 10 unmerged windows, and the average number of operators in its covering windows is 2. Using the example ranking function, the score of θ_3 is $1 - 11 = -10$ and the score of θ_6 is $2 - 10 = -8$. Thus, θ_6 is ranked higher, and it is indeed closer to reaching an MD.

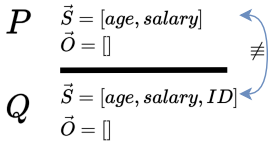
[7]We can test the equivalence of the other windows for early termination.

7.4 Identifying Inequivalent Pairs Efficiently

In this section, we use an example to show how to quickly detect the inequivalence between two workflow versions using a symbolic representation to represent partial information of the result of the sink of each version. Consider the case where two workflow versions P and Q are inequivalent, as shown in Figure 22a. The approach discussed so far attempts to find a decomposition in which all its windows are verified equivalent. However, in cases where the version pair is inequivalent, as in this example, such a decomposition does not exist, and the search framework would continue to look for one unsuccessfully. Moreover, detecting the inequivalence of the pair happens if there exists a decomposition that includes the entire version pair and satisfies the given EV's restrictions. Although the cost of maximizing the window and testing if it is valid could be low, testing the equivalence of maximal decompositions by pushing it to the EV incurs an overhead due to the EV's reasoning about the semantics of the window. Thus, we want to avoid sending a window to the EV if we can quickly determine beforehand that the version pair is not equivalent.



(a) A sample of two inequivalent workflow versions.



(b) A partial symbolic representation of two versions showing the projected columns are different.

Figure 22: Example of two inequivalent workflow versions and their partial symbolic representation.

To quickly identify the inequivalence between two workflow versions, our approach is to create a lightweight representation that allows us to partially reason about the semantics of the version pair. This approach relies on a symbolic representation similar to other existing methods [33, 56], denoted as $(\vec{S}, \vec{O})$. In this representation, $\vec{S}$ and $\vec{O}$ are lists that represent the projected columns in the table

and the columns on which the result table is sorted, respectively. To construct the representation, we follow the same techniques in existing literature [13, 56] by using predefined transformations for each operator. Operators inherit the representation from their upstream/parent operator and update the fields based on their internal logic. In this way, if the list of projected columns (based on $\vec{S}$) of version P is different from those in Q, as in Figure 22b, we can know the two versions do not produce the same results. We can apply the same check to the sorted columns.

8 EXTENSIONS

In this section, we discuss relaxing the definition of EV's restrictions then discuss the consequences of relaxing the definition and propose extending Algorithm 2 to handle incomplete EVs and handle multiple given EVs.

Relaxing EV's restrictions. Recall an EV's restrictions are conditions that a query pair must satisfy to guarantee the EV's completeness for determining the equivalence of the query pair. This definition of restrictions limits the opportunity to cover more query pairs. Thus, we relax the definition of EV's restrictions as follows.

Definition 8.1 (Relaxed EV's restrictions). Restrictions of an EV are a set of conditions such that for each query pair if this pair satisfies these conditions, then the EV can attempt to determine the equivalence of the pair.

However, relaxing the definition of an EV's restriction may not guarantee the completeness of the EV and may introduce the following implication.

Handling greedy window verification when an EV is incomplete. In Line 16 of Algorithm 2 we push testing the equivalence of a decomposition to the given EV only when the decomposition is marked maximal. The following example shows that this approach could miss some opportunity to find the equivalence of two versions, because the EV is not able to verify the equivalence of the two sub-DAGs in the maximal window.

Example 2. *Consider two workflow versions P and Q with a single edit c based on a given mapping $\mathcal{M}$. Let γ be a given EV. Suppose a covering window ω_c satisfies the restrictions of γ, and the EV is able to verify the equivalence of the two sub-DAGs in ω_c. According to Algorithm 2, we do not check the equivalence of this window if it is not marked maximal. Let ω' be the only MCW that contains ω. Following Line 16 in Algorithm 2, if a window is maximal (ω' in this example) we push testing its equivalence to the EV. However, suppose in this case, the EV returns* Unknown*, because EVs are mostly incomplete [14] for verifying two general relational expressions. Since there is no other MCW to test,* Veer *accordingly returns* Unknown *for verifying the equivalence of the sub-DAGs in ω'. However, if we pushed testing the equivalence of the smaller window ω_c, then* Veer *would have been able to verify the equivalence of the pair.*

This example highlights the significance of verifying the equivalence of sub-DAGs within smaller windows before expanding to larger windows. The challenge arises when an EV can verify the equivalence of a small window but fails to do so for a larger one. To address this, we modify Line 16 in Algorithm 2 to check the equivalence of smaller windows by backtracking when the maximal window is not verified. This modification ensures that we do have more opportunities to verify the equivalence of the version pair. Note that this approach may introduce a computational overhead due to the repeated checking of each window and not just the maximal ones.

Using multiple EVs. As mentioned earlier, the problem of verifying the equivalence of two relational expressions is undecidable [1]. Thus, any given EV would have limitations and is incomplete for solving the problem of deciding the equivalence of two queries. To harness the capabilities of different EVs, we extend Veer to take in a set of EVs and their associated restrictions. We do not modify Algorithm 2. However, we extend the 'isValid' function in Line 9 to encode a window is valid w.r.t which EV so that when we verify the equivalence of the sub-DAGs in the window in Line 17, we call the corresponding EV the window satisfies.

9 EXPERIMENTS

In this section, we present an experimental evaluation of the proposed solutions. We want to answer the following questions: Can our solution verify the equivalence of versions in a real-world pipelines workload? How does our solution perform compared to other verifiers? How the optimization techniques help in the performance? What are the parameters that affect the performance?

9.1 Experimental Setup

Synthetic workload. We constructed four workflows $W1 - W4$ on TPC-DS [47] dataset as shown in Table 4. For example, workflow $W1$'s first version was constructed based on TPC-DS Q40, which contains 17 operators including an outer join and an aggregate operators. Workflow $W2$'s first version was constructed based on TPC-DS Q18, which contains 20 operators. We omit details of other operators included in the workflows such as Unnest, UDF, and Sort as these do not affect the performance of the experimental result as we explain in each experiment.

Real workload. We analyzed a total of 179 real-world pipelines from Texera [46]. Among the workflows, 81% had deterministic sources and operators, and we focused our analysis on these workflows. Among the analyzed workflows, 8% consisted primarily of 8 operators, and another 8 had 12 operators. Additionally, 33% of the workflows contained 3 different versions, while 19% had 35 versions. 58% of the versions had a single edit, while 22% had two edits. We also observed that the UDF operator was changed in 17% of the cases, followed by the Projection operator (6% of the time) and the Filter operator (6% of the time). From this set of workflows, we selected four as a representative subset, which is presented as $W5 \ldots W8$ in Table 4 and we used IMDB [25] and Twitter [48] datasets.

Edit operations. For each real-world workflow, we used the edits performed by the users. For each synthetic workflow, we constructed versions by performing edit operations. We used two types of edit operations.

(1) Calcite transformation rules [9] for equivalent pairs: These edits are common for rewriting and optimizing workflows, so these edits would produce a version that is *equivalent* to the first version. For example, 'testEmptyProject' is a single edit of adding an empty projection operator. In addition, 'testPushProjectPastFilter' and 'testPushFilterPastAgg' are two example edits that produce more

Table 4: Workloads used in the experiments.

Work flow#	Description	Type of operators	# of operators	# of links	# of versions
$W1$	TPC-DS $Q40$	4 joins and 1 aggregate operators	17	16	5
$W2$	TPC-DS $Q18$	5 joins and 1 aggregate operators	20	20	9
$W3$	TPC-DS $Q71$	1 replicate, 1 union, 5 joins, and 1 aggregate operators	23	23	4
$W4$	TPC-DS $Q33$	3 replicates, 1 union, 9 joins, and 4 aggregates operators	28	34	3
$W5$	IMDB ratio of non-original to original movie titles	1 replicate, 2 joins, 2 aggregate operators	12	12	3
$W6$	IMDB all movies of directors with certain criteria	2 replicates, 4 joins, 2 unnest operators	18	20	3
$W7$	Tobacco Twitter analysis	1 outer join, 1 aggregate, classifier	14	13	3
$W8$	Wildfire Twitter analysis	1 join, 1 UDF	13	12	3

than a single change, in particular, one for deleting an operator and another is for pushing it past other operator. We used a variation of different numbers of edits, different placements of the edits, etc., for each experiment. Thus, we have numbers of pairs as shown in Table 4. For each pair of versions, one of the versions is always the original one.

(2) TPC-DS V2.1 [47] iterative edits for inequivalent pairs: These edits are common for exploratory and iterative analytics, so they may produce a version that is *not equivalent* to the first version. Example edits are adding a new filtering condition or changing the aggregate function as in TPC-DS queries. We constructed one version for each workflow using two edit operations from this type of transformations to test our solution when the version pair is not equivalent.

We randomized the edits and their placements in the workflow DAG, such that it is a valid edit. Unless otherwise stated, we used two edit operations from Calcite in all of the experiments.

Implementation. We implemented the baseline (Veer) and an optimized version (Veer$^+$) in Java8 and Scala. We implemented Equitas [56] as the EV in Scala. We implemented Veer$^+$ by including the optimization techniques presented in Section 7. We evaluated the solution by comparing Veer and Veer$^+$ against a state of the art verifier (Spes). We ran the experiments on a MacBook Pro running the MacOS Monterey operating system with a 2.2GHz Intel Core i7 CPU, 16GB DDR3 RAM, and 256GB SSD. Every experiment was run three times.

9.2 Comparisons with Other EVs

To our best knowledge, Veer is the first technique to verify the equivalence of complex workflows. To evaluate its performance, we compared Veer and Veer$^+$ against Spes, known for its proficiency in verifying query equivalence compared to other solutions. We chose one equivalent pair and one inequivalent pair of versions with two edits from each workflow. Among the 8 workflows examined, Spes failed to verify the equivalence and inequivalence of any of the pairs, because all of the workflow versions included operators not supported by Spes. In contrast, Veer and Veer$^+$ successfully verified the equivalence of 62% ($W1, W2, W3, W5$) and 78% ($W1 \ldots W6$), respectively, of the equivalent pairs. Both Veer and Veer did not verify the equivalence of the versions in $W7$ because none of the constructed decompositions were verified as equivalent by the EV. Moreover, Veer and Veer did not verify the equivalent pairs of $W8$ because the change to its versions was made on a UDF operator, resulting in the absence of a valid window that satisfies the EV's restrictions used in our experiments. Veer$^+$ was able to detect the inequivalence (using the heuristic discussed in Section 7.4) of about 50% of the inequivalent pairs ($W5 \ldots W8$). We note that Veer and Veer$^+$ can be made more powerful if we employ an EV that can reason the semantics of a UDF operator. Table 5 summarizes the evaluation of the compared techniques.

Table 5: Comparison evaluation of Veer and Veer$^+$ against Spes.

Verifier	% of proved equivalent pairs	Avg. time (s)	% of proved inequivalent pairs	Avg. time (s)
Spes	0.0	NA	0.0	NA
Veer	62.5	32.1	0.0	44.5
Veer$^+$	75.0	0.1	50.0	4.1

9.3 Evaluating Veer$^+$ Optimizations

We used workflow $W3$ for evaluating the first three optimization techniques discussed in Section 7. We used three edit operations: one edit was after the Union operator (which is not supported by Equitas) and two edits (pushFilterPastJoin) were before the Union. We used the baseline to verify the equivalence of the pairs, and we tried different combinations of enabling the optimization techniques. We want to know the effect of these optimization techniques on the performance of verifying the equivalence.

Table 6 shows the result of the experiments. The worst performance was the baseline itself when all of the optimization techniques were disabled, resulting in a total of 19,656 decompositions explored in 27 minutes. When only "pruning" was enabled, it was slower than all of the other combinations of enabling the techniques because it tested 108 MCWs for possibility of pruning them. Its performance was better than the baseline thanks to the early termination, where it resulted in 3,614 explored decompositions in 111 seconds. When "segmentation" was enabled, there were only two segments, and the total number of explored decompositions was lower. In particular, when we combined "segmentation" and "ranking", one of the segments had 8 explored decompositions while the other had 13. If "segmentation" was enabled without "ranking", then the total number of explored decompositions was 430, which was only 2% of the number of explored decompositions when "segmentation" was not enabled. The time it took to construct the segmentation was negligible. When "ranking" was enabled, the number of decompositions explored was around 21. It took an average of 0.04 seconds for exploring the decompositions and 0.40 for testing the equivalence by calling the EV. Since the performance of enabling all of the optimization techniques was the best, in the remaining experiments we enabled all of them for Veer$^+$.

Table 6: Result of enabling optimizations (W3 with three edits). "S" indicates segmentation, "P" indicates pruning, and "R" indicates ranking. A ✓means the optimization was enabled, a × means the optimization was disabled.The results are sorted based on the worst performance.

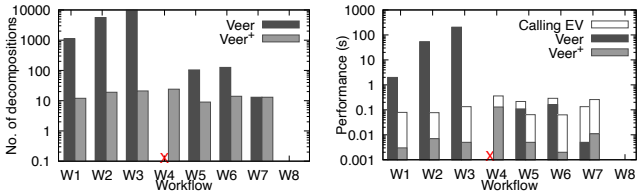
S	P	R	# of decompositions explored	Exploration (s)	Calling EV (s)	Total time (s)
×	×	×	19,656	1,629	0.22	1,629
×	✓	×	3,614	111	0.15	111
✓	✓	×	430	0.82	0.20	1.02
✓	×	×	430	0.51	0.18	0.69
×	✓	✓	20	0.39	0.12	0.52
✓	✓	✓	21	0.20	0.31	0.51
×	×	✓	20	0.07	0.23	0.30
✓	×	✓	21	0.03	0.21	0.24

9.4 Verifying Two Versions with Multiple Edits

We compared the performance of the baseline and Veer$^+$. We want to know how much time each approach took to test the equivalence of the pair and how many decompositions each approach explored. We used workflows $W1-W8$ with two edits. We used one equivalent pair and one inequivalent pair from each workflow to evaluate the performance in these two cases. Most workflows in the experiment had one segment, except workflows $W3$, $W5$, and $W6$, each of which has two segments. The overhead for each of the following steps, 'is maximal' (line 13), 'is valid' (line 9), and 'merge' (line 10) in Algorithm 2 was negligible, thus we only report the overhead of calling the EV.

Performance for verifying equivalent pairs. Figure 23a shows the number of decompositions explored by each approach. In general, the baseline explored more decompositions, with an average of 3,354 compared to Veer$^+$'s average of 16, which is less than 1% of the baseline. The baseline was not able to finish testing the equivalence of $W3$ in less than an hour. The reason is because of the large number of neighboring windows that were caused by a large number of links in the workflow. Veer$^+$ was able to find a segmentation for $W3$ and $W6$. It was unable to discover a valid segmentation for $W5$ because all of its operators are supported by the EV, but we used the second approach of finding a segmentation as we discussed in Section 7.1. We note that the overhead of constructing a segmentation using the second approach was negligible. For workflow $W7$, the size of the windows in a decomposition were small because the windows violated the restrictions of the used EV. Therefore, the "expanding decompositions" step stopped early and thus the search space (accordingly the running time) was small for both approaches. For workflow $W8$, both Veer and Veer$^+$ detected that the change was done on a non-supported operator (UDF) by the chosen EV (Equitas), thus the decomposition was not expanded to explore other ones and the algorithm terminated without verifying its equivalence.

Figure 23b shows the running time for each approach to verify the equivalence. The baseline took 2 seconds to verify the equivalence of $W1$, and 2 minutes for verifying $W3$. Veer$^+$, on the other hand, had a running time of a sub-second in verifying the equivalence of all of the workflows. Veer$^+$ tested 9 MCWs for a chance of pruning inequivalent decompositions when verifying $W6$. This caused the running time for verifying $W6$ to increase due to the overhead of calling the EV. In general, the overhead of calling the EV was about the same for both approaches. In particular, it took an average of 0.04 and 0.10 seconds for both the baseline and Veer$^+$, respectively, to call the EV.



(a) # of explored decompositions. **(b) Running time.**

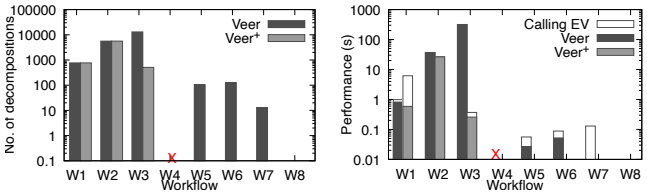
Figure 23: Comparison between the two algorithms for verifying equivalent pairs with two edits. An "×" means the algorithm was not able to finish running within one hour.

Performance of verifying inequivalent pairs. Figure 24a shows the number of decompositions explored by each approach. Since the pairs are not equivalent, Veer almost exhaustively explored all of the possible decompositions, trying to find an equivalent one. Veer$^+$ explored fewer decompositions compared to the baseline when testing $W3$, thanks to the segmentation optimization. Both approaches were not able to finish testing $W4$ within one hour because of the large number of possible neighboring windows. Veer$^+$ was able to quickly detect the inequivalence of the pairs of workflows $W5 \ldots W8$ thanks to the partial symbolic representation discussed in Scetion 7.4, resulting in Veer$^+$ not exploring any decompositions for these workflows.

The result of the running time of each approach is shown in Figure 24b. Veer's performance when verifying inequivalent pairs was the same as when verifying equivalent pairs because, in both cases, it explored the same number of decompositions. On the other hand, Veer$^+$'s running time was longer than when the pairs were equivalent for workflows $W1 \ldots W4$. We observe that for $W1$, Veer$^+$'s running time was even longer than the baseline due to the overhead of calling the EV up to 130, compared to only 4 times for the baseline. Veer$^+$ called the EV more as it tried to continuously test MCWs when exploring a decomposition for a chance of pruning inequivalent decompositions. Veer$^+$'s performance on $W3$ was better than the baseline. The reason is that there were two segments, and each segment had a single change. We note that Veer$^+$ tested the equivalence of both segments, even though there could have been a chance of early termination if the inequivalent segment was tested first. The time it took Veer$^+$ to verify the inequivalence of the pairs in workflows $W5 \ldots W8$ was negligible. The heuristic approach was not effective in detecting the inequivalence of the TPC-DS workflows $W1 \ldots W4$. This limitation arises from the technique's reliance on identifying differences in the *final* projected columns, which remained the same across all versions of these workflows (due to the aggregation operator), with most changes occurring in the filtering conditions.

9.5 Effect of the Distance Between Edits

We evaluated the effect of the placement of changes on the performance of both approaches. We are particularly interested in how



(a) # of explored decompositions. **(b) Running time.**

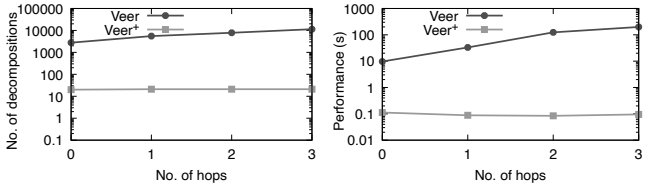
Figure 24: Comparison between the two algorithms for verifying inequivalent pairs with two edits. An "×" sign means the algorithm was not able to finish within an hour.

many decompositions would be explored and how long each approach would take if the changes were far apart or close together in the version DAG. We used $W2$ for the experiment with two edits. We use the 'number of hops' to indicate how far apart the changes were from each other. A 0 indicates that they were next to each other, and a 3 indicates that they were separated by three operators between them. For a fair comparison, the operators that were separating the changes were one-to-one operators, i.e., operators with one input and one output links.

Figure 25a shows the number of decompositions explored by each approach. The baseline's number of decompositions increased from 2,770 to 11,375 as the number of hops increased. This is because it took longer for the two covering windows, one for each edit, to merge into a single one. Before the two covering windows merge, each one produces more decompositions to explore due to merging with its own neighbors. Veer$^+$'s number of explored decompositions remained the same at 21 thanks to the ranking optimization, as once one covering window includes a neighboring window, its size is larger than the other covering window and would be explored first until both covering windows merge.

Figure 25b shows the time each approach took to verify the equivalence of a pair. The performance of each approach was proportional to the number of explored decompositions. The baseline took between 9.7 seconds and 3 minutes, while Veer$^+$'s performance remained in the sub-second range (0.095 seconds).

Effect of type of changed operators. We note that when any of the changes were on an unsupported operator by the EV, then both Veer and Veer$^+$ were not able to verify their equivalence. We also note that the running time to prove the pair's equivalence, was negligible because the exploration stops after detecting an 'invalid' covering window.



(a) # of explored decompositions. **(b) Running time.**

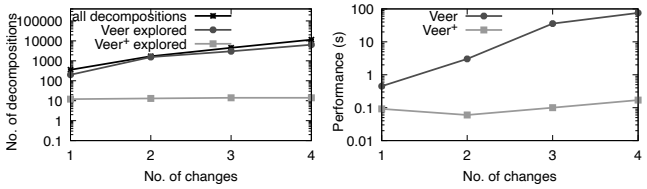
Figure 25: Effect of the distance between changes (W2 with two edits)

9.6 Effect of the Number of Changes

In iterative data analytics, when the task is exploratory, there can be many changes between two consecutive versions. Once the analytical task is formulated, there are typically only minimal changes to refine some parameters [52]. We want to evaluate the effect of the number of changes on the number of decompositions and the time each approach takes to verify a version pair. The number of changes, intuitively, increases the number of initial covering windows, and consequently, the possible different combinations of merging with neighboring windows increases. We used $W1$ in the experiment.

Figure 26a shows the number of decompositions explored by each approach and the total number of "valid" decompositions.The latter increased from 356 to 11,448 as we increased the number of changes from 1 to 4. The baseline explored almost all those decompositions, with an average of 67% of the total decompositions, in order to reach a maximal one. Veer$^+$'s number of explored decompositions, on the other hand, was not affected by the increase in the number of changes and remained the same at around 14. The ranking optimization caused a larger window to be explored first, which sped up the merging of the separate covering windows, those that include the changes.

Figure 26b shows the time taken by each approach to verify the equivalence of a pair. Both approaches' time was proportional to the number of explored decompositions. The baseline showed a performance of around 0.42 seconds when there was a single change, up to slightly more than a minute at 75 seconds when there were four changes. Veer$^+$, on the other hand, maintained a sub-second performance with an average of 0.1 seconds.



(a) # of explored decompositions. **(b) Running time.**

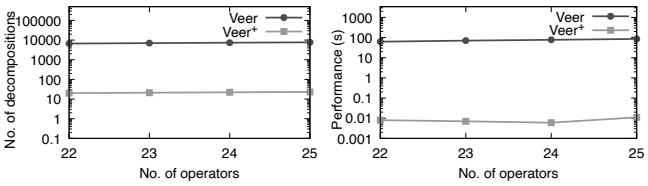
Figure 26: Effect of the number of changes (W1).

9.7 Effect of the Number of Operators

We evaluated the effect of the number of operators. We used $W2$ with two edits and varied the number of operators from 22 to 25. We varied the number of operators in two different ways. One was varying the number of operators by including only those supported by the EV. These operators may be included in the covering windows, thus their neighbors would be considered during the decomposition exploration. The other type was varying the number of non-supported operators, as their inclusion in the workflow DAG would not affect the performance of the algorithms.

Varying the number of supported operators. Figure 27a shows the number of explored decompositions. The baseline explored 6,650 decompositions when there were 22 operators, and 7,700 decompositions when there were 25 operators. Veer$^+$ had a linear increase in the number of explored decompositions from 21 to 24

when we increased the number of operators from 22 to 25. Figure 27b shows the results. We observed that the performance of Veer was negatively affected due to the addition of possible decompositions from these operators' neighbors while the performance of Veer$^+$ remained the same. In particular, Veer verified the pair from a minute up to 1.4 minutes, while Veer$^+$ verified the pair in a sub-second.



(a) # of explored decompositions. **(b) Running time.**

Figure 27: Effect of the number of operators (W2 with two edits).

Varying the number of unsupported operators. Both the baseline and Veer$^+$ were not affected by the increase in the number of unsupported operators as these operators were not included in the covering windows.

10 RELATED WORKS

Equivalence verification. There are many studies to solve the problem of verifying the equivalence of two SQL queries under certain assumptions. These solutions were applicable to a small class of SQL queries, such as conjunctive queries [2, 10, 27, 44]. With the recent advancement of developing proof assists and solvers [16, 17], there have been new solutions [13, 55, 56]. UDP [13] and WeTune's verifier [50] use semirings to model the semantics of the pair and use a proof assist, such as Lean [17] to prove if the expressions are equivalent. These two works support reasoning semantics of two queries with integrity constraints. Equitas [56] and Spes [55] model the semantics of the pair into a First-Order Logic (FOL) formula and push the formula to be solved by a solver such as SMT [16]. These two works support queries with three-valued variables. Other works also use an SMT solver to verify the equivalence of a pair of Spark jobs [23]. Our solution uses them as black boxes to verify the equivalence of a version pair. The work in [11] finds a weighted edit distance based on the semantic equivalence of two queries to grade students queries.

Tracking workflow executions. There has been an increasing interest in enabling the reproducibility of data analytics pipelines. These tools track the evolution and versioning of datasets, models, and results. At a high level they can be classified as two categories. The first includes those that track experiment results of different versions of ML models and the corresponding hyper-parameters [12, 22, 29, 35, 49, 53]. The second includes solutions to track results of different versions of data processing workflows [4, 15, 34, 51].

Materialization reuse. There is a large body of work on answering data processing workflows using views [18, 19, 28, 41, 43]. Some solutions [20] focus on deciding which results to store to maximize future reuse. Other solutions [36, 54] focus on identifying materialization reuse opportunities by relying on finding an exact match of the workflow's DAG. On the other hand, semantic query optimization works [21, 24, 31, 45] reason the semantics of the query to identify reuse opportunities that are not limited to structural matching. However, these solutions are applicable to a specific class of functions, such as user defined function (UDF) [38, 41, 52], and do not generalize to finding reuse opportunities by finding equivalence of any pair of workflows.

11 CONCLUSION

In this paper, we studied the problem of verifying the equivalence of two workflow versions. We presented a solution called "Veer," which leverages the fact that two workflow versions can be very similar except for a few changes. We analyzed the restrictions of existing EVs and presented a concept called a "window" to leverage the existing solutions for verifying the equivalence. We proposed a solution using the windows to verify the equivalence of a version pair with a single edit. We discussed the challenges of verifying a version pair with multiple edits and proposed a baseline algorithm. We proposed optimization techniques to speed up the performance of the baseline. We conducted a thorough experimental study and showed the high efficiency and effectiveness of the solution.

ACKNOWLEDGMENTS

This work is supported by a graduate fellowship from King Saud University and was supported by NSF award III 2107150.

REFERENCES

[1] Serge Abiteboul, Richard Hull, and Victor Vianu. 1995. *Foundations of Databases: The Logical Level* (1st ed.). Addison-Wesley Longman Publishing Co., Inc., USA.
[2] Foto N. Afrati, Chen Li, and Prasenjit Mitra. 2004. On Containment of Conjunctive Queries with Arithmetic Comparisons. In *EDBT*. 459–476.
[3] Yanif Ahmad, Oliver Kennedy, Christoph Koch, and Milos Nikolic. 2012. DBToaster: Higher-order Delta Processing for Dynamic, Frequently Fresh Views. *Proc. VLDB Endow.* 5, 10 (2012), 968–979. https://doi.org/10.14778/2336664.2336670
[4] Sadeem Alsudais. 2022. Drove: Tracking Execution Results of Workflows on Large Data. In *Proceedings of the VLDB 2022 PhD Workshop co-located with the 48th International Conference on Very Large Databases (VLDB 2022), Sydney, Australia, September 5, 2022 (CEUR Workshop Proceedings)*, Zhifeng Bao and Timos K. Sellis (Eds.), Vol. 3186. CEUR-WS.org. http://ceur-ws.org/Vol-3186/paper_10.pdf
[5] Alteryx Website, https://www.alteryx.com/.
[6] Alteryx Weekly Challenge, https://community.alteryx.com/t5/Weekly-Challenge/bd-p/weeklychallenge.
[7] Apache Flink http://flink.apache.org.
[8] Cristina Borralleras, Daniel Larraz, Enric Rodríguez-Carbonell, Albert Oliveras, and Albert Rubio. 2019. Incomplete SMT Techniques for Solving Non-Linear Formulas over the Integers. *ACM Trans. Comput. Log.* 20, 4 (2019), 25:1–25:36. https://doi.org/10.1145/3340923
[9] Calcite benchmark, https://github.com/uwdb/Cosette/tree/master/examples/calcite.
[10] Ashok K. Chandra and Philip M. Merlin. 1977. Optimal Implementation of Conjunctive Queries in Relational Data Bases. In *Proceedings of the 9th Annual ACM Symposium on Theory of Computing, May 4-6, 1977, Boulder, Colorado, USA*, John E. Hopcroft, Emily P. Friedman, and Michael A. Harrison (Eds.). ACM, 77–90. https://doi.org/10.1145/800105.803397
[11] Bikash Chandra and S. Sudarshan. 2022. Automated Grading of SQL Queries. *IEEE Data Eng. Bull.* 45, 3 (2022), 17–28. http://sites.computer.org/debull/A22sept/p17.pdf
[12] Andrew Chen, Andy Chow, Aaron Davidson, Arjun DCunha, Ali Ghodsi, Sue Ann Hong, Andy Konwinski, Clemens Mewald, Siddharth Murching, Tomas Nykodym, Paul Ogilvie, Mani Parkhe, Avesh Singh, Fen Xie, Matei Zaharia, Richard Zang, Juntai Zheng, and Corey Zumar. 2020. Developments in MLflow: A System to Accelerate the Machine Learning Lifecycle. In *DEEM@SIGMOD'20*.

[13] Shumo Chu, Brendan Murphy, Jared Roesch, Alvin Cheung, and Dan Suciu. 2018. Axiomatic Foundations and Algorithms for Deciding Semantic Equivalences of SQL Queries. *VLDB'18* (2018).

[14] Shumo Chu, Chenglong Wang, Konstantin Weitz, and Alvin Cheung. 2017. Cosette: An Automated Prover for SQL. In *8th Biennial Conference on Innovative Data Systems Research, CIDR 2017, Chaminade, CA, USA, January 8-11, 2017, Online Proceedings*. www.cidrdb.org. http://cidrdb.org/cidr2017/papers/p51-chu-cidr17.pdf

[15] Databricks Data Science Website, https://www.databricks.com/product/data-science.

[16] Leonardo Mendonça de Moura and Nikolaj S. Bjørner. 2008. Z3: An Efficient SMT Solver. In *TACAS'08*.

[17] Leonardo Mendonça de Moura, Soonho Kong, Jeremy Avigad, Floris van Doorn, and Jakob von Raumer. 2015. The Lean Theorem Prover (System Description). In *Automated Deduction - CADE-25 - 25th International Conference on Automated Deduction, Berlin, Germany, August 1-7, 2015, Proceedings (Lecture Notes in Computer Science)*, Amy P. Felty and Aart Middeldorp (Eds.), Vol. 9195. Springer, 378–388. https://doi.org/10.1007/978-3-319-21401-6_26

[18] Behrouz Derakhshan, Alireza Rezaei Mahdiraji, Zoi Kaoudi, Tilmann Rabl, and Volker Markl. 2022. Materialization and Reuse Optimizations for Production Data Science Pipelines. In *SIGMOD '22: International Conference on Management of Data, Philadelphia, PA, USA, June 12 - 17, 2022*. ACM, 1962–1976. https://doi.org/10.1145/3514221.3526186

[19] Kayhan Dursun, Carsten Binnig, Ugur Çetintemel, and Tim Kraska. 2017. Revisiting Reuse in Main Memory Database Systems. In *SIGMOD'17*.

[20] Iman Elghandour and Ashraf Aboulnaga. 2012. ReStore: Reusing Results of MapReduce Jobs. *VLDB'12* (2012).

[21] Ronald Fagin, Phokion G. Kolaitis, Renée J. Miller, and Lucian Popa. 2005. Data exchange: semantics and query answering. *Theor. Comput. Sci.* 336, 1 (2005), 89–124. https://doi.org/10.1016/j.tcs.2004.10.033

[22] Gharib Gharibi, Vijay Walunj, Rakan Alanazi, Sirisha Rella, and Yugyung Lee. 2019. Automated Management of Deep Learning Experiments. In *DEEM@SIGMOD'19*.

[23] Shelly Grossman, Sara Cohen, Shachar Itzhaky, Noam Rinetzky, and Mooly Sagiv. 2017. Verifying Equivalence of Spark Programs. In *CAV'17*.

[24] Alon Y. Halevy. 2001. Answering Queries Using Views: A Survey. *The VLDB Journal* 10, 4 (Dec. 2001), 270–294. https://doi.org/10.1007/s007780100054

[25] IMDB Datasets Website. https://www.imdb.com/interfaces/

[26] IMDB Workload Website. https://github.com/juanmanubens/SQL-Advanced-Queries/blob/master/imdb.sql

[27] T. S. Jayram, Phokion G. Kolaitis, and Erik Vee. 2006. The containment problem for REAL conjunctive queries with inequalities. In *Proceedings of the Twenty-Fifth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems, June 26-28, 2006, Chicago, Illinois, USA*, Stijn Vansummeren (Ed.). ACM, 80–89. https://doi.org/10.1145/1142351.1142363

[28] Alekh Jindal, Konstantinos Karanasos, Sriram Rao, and Hiren Patel. 2018. Selecting Subexpressions to Materialize at Datacenter Scale. *Proc. VLDB Endow.* 11, 7 (2018), 800–812. https://doi.org/10.14778/3192965.3192971

[29] Klaus Greff, Aaron Klein, Martin Chovanec, Frank Hutter, and Jürgen Schmidhuber. 2017. The Sacred Infrastructure for Computational Research. In *SciPy'17*.

[30] Knime Workflows Website. https://hub.knime.com/search?type=Workflow&sort=maxKudos

[31] Jan Kossmann, Thorsten Papenbrock, and Felix Naumann. 2022. Data dependencies for query optimization: a survey. *VLDB J.* 31, 1 (2022), 1–22. https://doi.org/10.1007/s00778-021-00676-3

[32] Avinash Kumar, Zuozhi Wang, Shengquan Ni, and Chen Li. 2020. Amber: A Debuggable Dataflow System Based on the Actor Model. *Proc. VLDB Endow.* 13, 5 (2020), 740–753. https://doi.org/10.14778/3377369.3377381

[33] Jeff LeFevre, Jagan Sankaranarayanan, Hakan Hacigümüs, Jun'ichi Tatemura, Neoklis Polyzotis, and Michael J. Carey. 2014. Opportunistic physical design for big data analytics. In *International Conference on Management of Data, SIGMOD 2014, Snowbird, UT, USA, June 22-27, 2014*, Curtis E. Dyreson, Feifei Li, and M. Tamer Özsu (Eds.). ACM, 851–862. https://doi.org/10.1145/2588555.2610512

[34] Hui Miao and Amol Deshpande. 2018. ProvDB: Provenance-enabled Lifecycle Management of Collaborative Data Analysis Workflows. *IEEE Data Eng. Bull.* (2018).

[35] Hui Miao, Ang Li, Larry S. Davis, and Amol Deshpande. 2017. Towards Unified Data and Lifecycle Management for Deep Learning. In *ICDE'17*.

[36] Fabian Nagel, Peter A. Boncz, and Stratis Viglas. 2013. Recycling in pipelined query evaluation. In *ICDE'13*.

[37] https://www1.nyc.gov/site/tlc/about/tlc-trip-record-data.page.

[38] Optimizing Apache Spark UDFs Website. https://www.databricks.com/session_eu20/optimizing-apache-spark-udfs

[39] Orange Data Mining Workflows. https://orangedatamining.com/workflows/

[40] Luis Leopoldo Perez and Christopher M. Jermaine. 2014. History-aware query optimization with materialized intermediate views. In *IEEE 30th International Conference on Data Engineering, Chicago, ICDE 2014, IL, USA, March 31 - April 4, 2014*, Isabel F. Cruz, Elena Ferrari, Yufei Tao, Elisa Bertino, and Goce Trajcevski (Eds.). IEEE Computer Society, 520–531. https://doi.org/10.1109/ICDE.2014.6816678

[41] Lana Ramjit, Matteo Interlandi, Eugene Wu, and Ravi Netravali. 2019. Acorn: Aggressive Result Caching in Distributed Data Processing Frameworks. In *Proceedings of the ACM Symposium on Cloud Computing, SoCC 2019, Santa Cruz, CA, USA, November 20-23, 2019*. ACM, 206–219. https://doi.org/10.1145/3357223.3362702

[42] Kaspar Riesen, Sandro Emmenegger, and Horst Bunke. 2013. A Novel Software Toolkit for Graph Edit Distance Computation. In *Graph-Based Representations in Pattern Recognition - 9th IAPR-TC-15 International Workshop, GbRPR 2013, Vienna, Austria, May 15-17, 2013. Proceedings (Lecture Notes in Computer Science)*, Walter G. Kropatsch, Nicole M. Artner, Yll Haxhimusa, and Xiaoyi Jiang (Eds.), Vol. 7877. Springer, 142–151. https://doi.org/10.1007/978-3-642-38221-5_15

[43] Abhishek Roy, Alekh Jindal, Priyanka Gomatam, Xiating Ouyang, Ashit Gosalia, Nishkam Ravi, Swinky Mann, and Prakhar Jain. 2021. SparkCruise: Workload Optimization in Managed Spark Clusters at Microsoft. *Proc. VLDB Endow.* 14, 12 (2021), 3122–3134. https://doi.org/10.14778/3476311.3476388

[44] Yehoshua Sagiv and Mihalis Yannakakis. 1980. Equivalences Among Relational Expressions with the Union and Difference Operators. *J. ACM* 27, 4 (1980), 633–655. https://doi.org/10.1145/322217.322221

[45] Michael Schmidt, Michael Meier, and Georg Lausen. 2010. Foundations of SPARQL query optimization. In *Database Theory - ICDT 2010, 13th International Conference, Lausanne, Switzerland, March 23-25, 2010, Proceedings (ACM International Conference Proceeding Series)*, Luc Segoufin (Ed.). ACM, 4–33. https://doi.org/10.1145/1804669.1804675

[46] Texera Website, https://github.com/Texera/texera.

[47] TPC-DS http://www.tpc.org/tpcds/.

[48] Twitter API v1.1. https://developer.twitter.com/en/docs/twitter-api/v1/tweets/filter-realtime/overview

[49] Manasi Vartak, Harihar Subramanyam, Wei-En Lee, Srinidhi Viswanathan, Saadiyah Husnoo, Samuel Madden, and Matei Zaharia. 2016. ModelDB: a system for machine learning model management. In *HILDA@SIGMOD'16*.

[50] Zhaoguo Wang, Zhou Zhou, Yicun Yang, Haoran Ding, Gansen Hu, Ding Ding, Chuzhe Tang, Haibo Chen, and Jinyang Li. 2022. WeTune: Automatic Discovery and Verification of Query Rewrite Rules. In *SIGMOD '22: International Conference on Management of Data, Philadelphia, PA, USA, June 12 - 17, 2022*. ACM, 94–107. https://doi.org/10.1145/3514221.3526125

[51] Simon Woodman, Hugo Hiden, Paul Watson, and Paolo Missier. 2011. Achieving reproducibility by combining provenance with service and workflow versioning. In *WORKS'11*.

[52] Zhuangdi Xu, Gaurav Tarlok Kakkar, Joy Arulraj, and Umakishore Ramachandran. 2022. EVA: A Symbolic Approach to Accelerating Exploratory Video Analytics with Materialized Views. In *SIGMOD '22: International Conference on Management of Data, Philadelphia, PA, USA, June 12 - 17, 2022*, Zachary Ives, Angela Bonifati, and Amr El Abbadi (Eds.). ACM, 602–616. https://doi.org/10.1145/3514221.3526142

[53] Yang Zhang, Fangzhou Xu, Erwin Frise, Siqi Wu, Bin Yu, and Wei Xu. 2016. DataLab: a version data management and analytics system. In *BIGDSE@ICSE'16*.

[54] Jingren Zhou, Per-Åke Larson, Johann Christoph Freytag, and Wolfgang Lehner. 2007. Efficient exploitation of similar subexpressions for query processing. In *SIGMOD'07*.

[55] Qi Zhou, Joy Arulraj, Shamkant B. Navathe, William Harris, and Jinpeng Wu. 2022. SPES: A Symbolic Approach to Proving Query Equivalence Under Bag Semantics. (2022), 2735–2748. https://doi.org/10.1109/ICDE53745.2022.00250

[56] Qi Zhou, Joy Arulraj, Shamkant B. Navathe, William Harris, and Dong Xu. 2019. Automated Verification of Query Equivalence Using Satisfiability Modulo Theories. *VLDB'19* (2019).