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

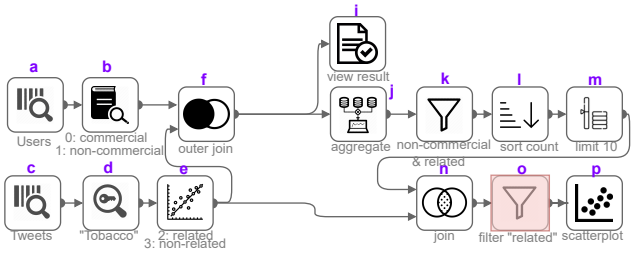
Data analytics using GUI-based workflows is an iterative process in which an analyst makes many iterations of changes to refine the workflow, generating a different version at each iteration. In many cases, the result of executing a workflow version is equivalent to a result of a prior executed version. 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 complexity of their operators can make existing equivalence verifiers (EV) 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 efficient solutions to generate these windows and verify the equivalence within each window. Our thorough experiments on real data sets show that Veer is able to not only verify workflows that cannot be supported by existing EVs, but also do the verification efficiently.

PVLDB Reference Format:
Sadeem Alsudais, Avinash Kumar, and Chen Li. Veer: Verifying Equivalence of Workflow Versions in Iterative Data Analytics . PVLDB, 14(1): XXX-XXX, 2020.
doi:XX.XX/XXX.XX

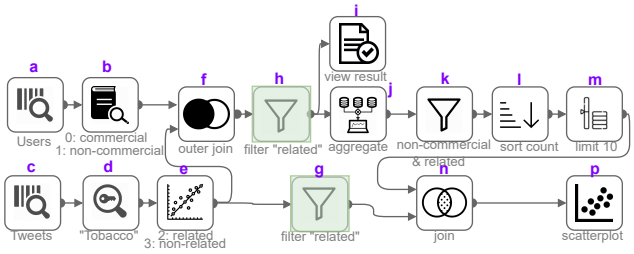
PVLDB Artifact Availability:
The source code, data, and/or other artifacts have been made available at https://github.com/sadeemsaleh/texera/tree/add-veer.

1 INTRODUCTION

Big data processing platforms, especially GUI-based systems, enable users to quickly construct complex analytical workflows [6, 14, 31]. 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 [17, 48]. 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 [5, 14, 28, 45, 47]. In many applications, these workflows have a significant amount of overlap and equivalence [5, 27, 51, 52]. For example, 45% of the daily jobs in Microsoft's analytics clusters have overlaps [27]. 27% of 9,486 workflows to detect fraud transactions from Ant Financial have overlaps, 6% of which is equivalent [51]. 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 [8, 31, 51]. Optimizing the performance of a workflow execution has been studied extensively in the literature [18, 38]. One optimizing technique is by leveraging the iterative nature of data analytics to reuse previously materialized results [17, 27].

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 [19]. Due to the nature of the overlap

Proceedings of the VLDB Endowment, Vol. 14, No. 1 ISSN 2150-8097.
doi:XX.XX/XXX.XX

and equivalence of consecutive versions, one line of works [3, 17] 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 [4, 6, 48]. For example, the top 8 workflows in Alteryx's workflows hub [7] 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 [43] 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 [4]. 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 [18, 48]. For example, 50% of the workflows belonging to the benchmarks that simulated real iterative tasks on video [48] and TPC-H [18] data had overlap. The refinements between the successive versions comprised of only a few changes over a particular part of the workflow.

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

Limitations of existing solutions. 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]. There have been many Equivalence Verifiers (EVs) proposed to verify the equivalence of two SQL queries [13, 51, 52]. 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, 46, 51, 52] 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.

Challenges and Contributions. 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 an existing EV as a black box.

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

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 [10]	232	34.81%
Knime workflows hub [29]	37	2.70%
Orange workflows [37]	32	0.00%
IMDB sample workload [25]	5	0.00%
TPC-DS benchmark [44]	99	2.02%
Texera workflows [43]	105	0.00%

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. While exploring this seemingly simple idea, we face several challenges in developing Veer: 1) How to capture the EVs restrictions? 2) How to deal with workflow versions with a single edit and do the verification efficiently? 3) How to deal with the case of workflow versions having multiple edits? We study these challenges and make the following contributions.

(1) We give an overview of the solution and formally define the "window" concept that is used in the equivalence verification algorithm (§ 3).
(2) 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" (§ 4).
(3) We study the general case where the two versions have multiple edits. We analyze the challenges of using overlapping windows. We then propose a solution based on the "decomposition" concept. We find an interesting hierarchical structure of the decompositions, and use this structure in a baseline solution (§ 5).
(4) We provide a number of optimizations in Veer$^+$ to improve the performance of the baseline algorithm (§ 6).
(5) 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 (§ 7).

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.

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 [32, 47]. 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,

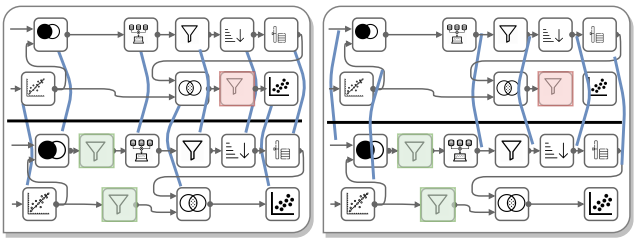
$$v_{j+1} = v_j \oplus \delta_j. \quad (1)$$

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.

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



(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.

2.2 Workflow's Execution and Results

The user submits an execution request E_j to run the workflow version v_j. The execution E_j produces *artifacts*, which are the results of each sink in the workflow version.

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

After the completion of the execution E_j for a version v_j, each sink s_{ji}'s result (or artifact) is saved as A_{ji}. Thus each execution E_j produces a set of saved results that correspond to each sink in the workflow $\mathbb{A}_j = \{A_{j1}, \ldots, A_{jn}\}$. In the running example, executing the workflow version v_1 produces two results corresponding to the sink operators s_i and s_p, which are saved as A_{ji} and A_{jp}, respectively.

Result equivalence of workflow versions. The execution request E_j 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 $\delta = \{c_1 \ldots c_n\}$ 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. The two versions are called equivalent if each pair of their sinks under the mapping is equivalent.

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 $\mathbf{Q_2}$, and returns TRUE when $\mathbf{Q_1} \equiv \mathbf{Q_2}$ under a specific table semantics [13, 15, 22, 46, 51, 52]. For instance, UDP [13] and Equitas [52] 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 [15] to decide its satisfiability. As a logical plan of a SQL query is a DAG, the problem of testing the equivalence of the sinks of two DAGs can be treated as verifying the equivalence of two SQL queries. An EV requires expressions to meet certain requirements (called "restrictions") in order to test their equivalence. We will discuss these restrictions in detail in Section 4.1.

Problem Statement. *Given an EV and two workflow versions P and Q with their mapping $\mathcal{M}$, find the set of equivalent pairs of sinks from the two versions.*

We first study the problem where the two versions have a single sink. We then generalize the solution to the case of multiple sinks.

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 EV's 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 EV's, making it usable for any EV of choice.

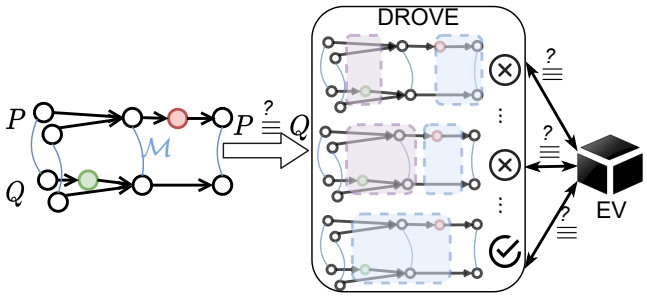


Figure 3: Overview of Veer. Given an EV and two versions with their mapping, Veer breaks (decomposes) the version pair into small windows, each of which satisfies the EV's restrictions. It finds different possible decompositions until it finds one with each of windows verified as equivalent by the EV.

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-DAG's $\omega(P)$ 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.

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 $C \subset \delta$. 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-DAG's in a window). We say the two sub-DAG's $\omega(P)$ 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-DAG's (i.e., those operators without ancestors in the sub-DAG's), each sink s of $\omega(P)$ and the corresponding $\mathcal{M}(s)$ in $\omega(Q)$ produce 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-DAG's $\omega'(v_1)$ 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 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-DAG's 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).

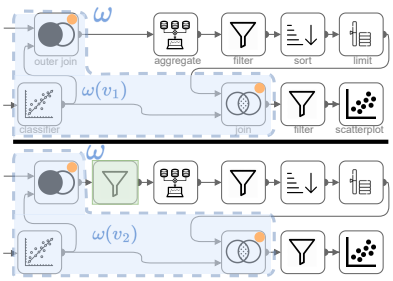


Figure 4: An example window ω and each sub-DAG of $\omega(v_1)$ and $\omega(v_2)$ contains two sinks (shown as "●").

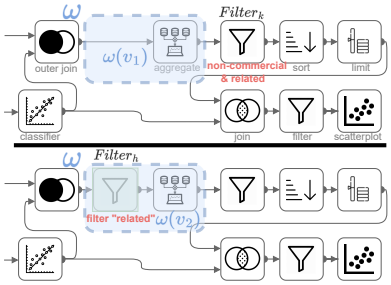


Figure 5: A covering window ω for adding $\mathsf{Filter_h}$.

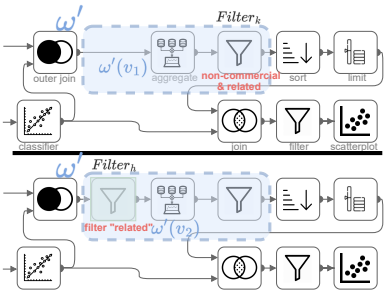


Figure 6: An example covering window ω' showing its pair of sub-DAGs are equivalent.

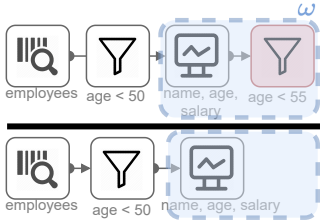


Figure 7: Two sub-DAG's in the window ω are not equivalent, as sub-DAG equivalence in Definition 3.3 does not consider constraints from the upstream operators. But the two complete workflow versions are indeed equivalent.

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.

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 with a single edit operation between them. If there is a covering window of the edit operation such that the sub-DAG's of the window are equivalent, then the version pair is equivalent.*

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 in the window to an EV. If the EV shows that all the sink pairs are equivalent, then the version pair is 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 $\mathsf{Filter_k}$ 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. 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 due to the modules used by the EV: For example, Equitas [52], Spes [51], and Spark Verifier [22] use an SMT solver [15] to determine if an FOL formula is satisfiable or not. SMT solver is not complete for determining the satisfiability of formulas when their predicates have non-linear conditions [9]. 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 a summary of the explicit and *derived* restrictions of the Equitas [52] to test the equivalence of two queries.

R1. The table semantics has to be set semantics.
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 4.2 (Valid window). 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 4.3. *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-DAG's in window ω are equivalent, then the sub-DAG's of ω' are also equivalent.*

Based on Lemma 4.3, 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 5.5.) Based on this observation, we introduce the following concept.

Definition 4.4 (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.

The change c may have more than one MCW, For example, suppose the EV is Equitas. Figure 8 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.

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.

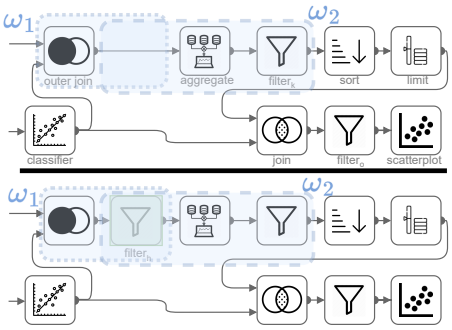


Figure 8: Two MCW ω_1 and ω_2 satisfying the restrictions of Equitas to cover the change of adding Filter_h to v_2.

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.

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 $\omega \leftarrow$ create an initial window to include the source and the corresponding target (operator/link) of the edit c
2 $\Omega = \{\omega\}$ // initialize a set for exploring widows
// using memoization, a window is explored only once
3 **while** Ω *is not empty* **do**
4 $\omega_i \leftarrow$ 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`

We use the example in Figure 9 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

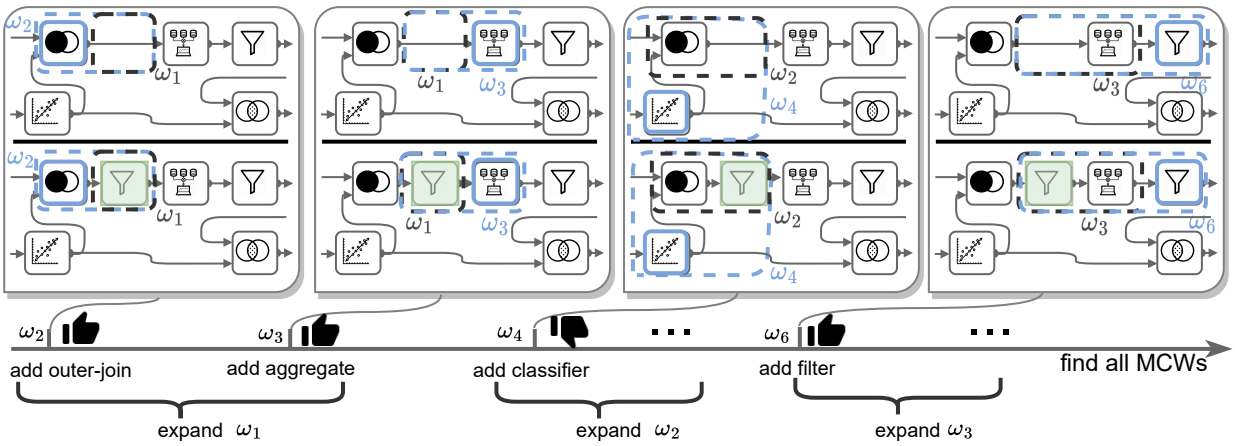


Figure 9: Example to illustrate the process of finding MCWs for the change of adding $\mathsf{Filter_h}$ to v_2.

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).

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.

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 10, is inspired from the NY Taxi dataset [35] to calculate the trip time based on the duration and starting time. Suppose the $\mathsf{Select_x}$ and $\mathsf{Select_z}$ operators are deleted from a version v_1 and $\mathsf{Select_y}$ operator is added to transform the workflow to version v_2. The example shows two overlapping windows ω and ω', each window is equivalent.

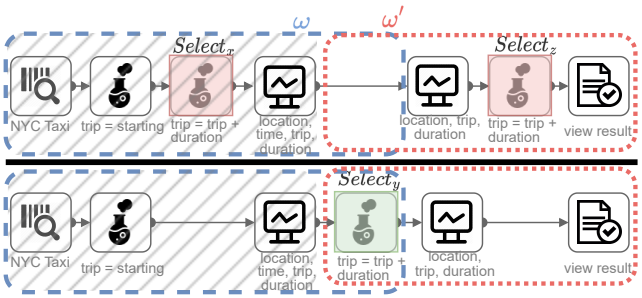


Figure 10: In this example, the blue window ω is equivalent and the red window ω' is also equivalent. But the version pair is not equivalent. The shaded gray area is the input to window ω'.

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 11 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. For simplicity, we only show covering windows of a decomposition in the figures throughout this section. Next, we show how to use a decomposition to verify the equivalence of the version pair by generalizing Lemma 4.1 as follows.

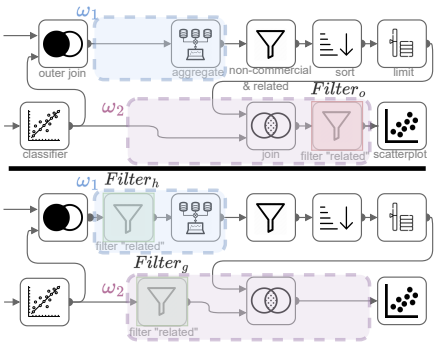


Figure 11: A decomposition θ with two windows ω_1 and ω_2 that cover the three edits.

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

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 12 shows an example of a decomposition θ' that contains the decomposition θ in Figure 11. 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.

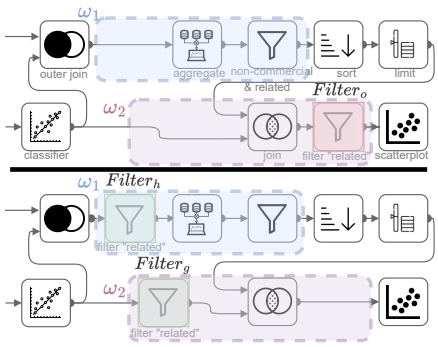


Figure 12: Example to show equivalent pair of sub-DAGs for every window in a decomposition θ'.

The following lemma, which is a generalization of Lemma 4.3, can help us prune the search space by ignoring decompositions that are properly contained by other decompositions.

Lemma 5.4. *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.*

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 13 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}.

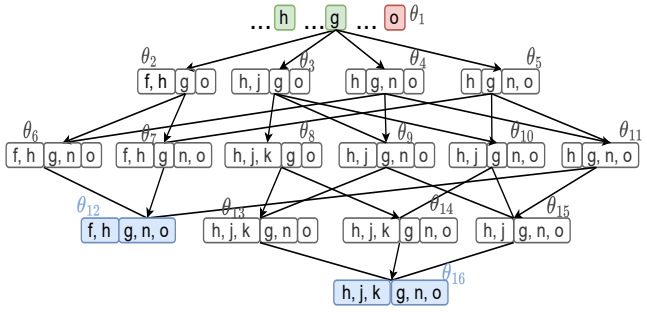


Figure 13: Hierarchy of different decompositions for the running example and the MD are highlighted in blue. We show the containment of covering windows and we omit the details of other possible containment combinations of non-covering windows.

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 13. We add the initial decomposition to a set of decompositions to be expanded (line 1). 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 8). If a window cannot be further expanded, then we mark the window as maximal to avoid checking it again (line 12). 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 15). If all of the windows are verified to be equivalent, we return `True` to indicate that the version pair is equivalent (line 16). 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 18). 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 20). This algorithm generalizes Algorithm 1 to handle cases of two versions with multiple edits.

Theorem 5.7. *(Correctness). Given a workflow version pair (P, Q), if* Veer *returns* TRUE, *then* $P \equiv Q$.

5.5 Improving the Completeness of Algorithm 2

Algorithm 2 follows a greedy approach to expanding each window. In line 11, if none of the neighbor windows of ω_j can be merged

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 $\theta \leftarrow$ decomposition with each operator as a window
2 $\Theta = \{\theta\}$ // initial set of decompositions
3 **while** Θ *is not empty* **do**
4 Remove a decomposition θ_i from Θ
5 **for** *every covering window* ω_j *(in* θ_i*) not marked* **do**
6 **for** *each neighbor* ω_k *of* ω_j **do**
7 **if** $\omega_k \cup \omega_j$ *is valid and not explored before* **then**
8 $\theta_i' \leftarrow \theta - \omega_k - \omega_j + \omega_k \cup \omega_j$
9 add θ_i' to Θ
10 **end**
11 **if** *none of the neighbor windows can be merged* **then**
12 mark ω_j
13 **end**
14 **if** *every covering* $\omega \in \theta_i$ *is marked* **then**
15 **if** γ *verifies each covering window in* θ_i *to be equivalent* **then**
16 **return** `True`
17 **if** θ_i *has only one window and* γ *verifies it not to be equivalent* **then**
18 **return** `False`
19 **end**
20 **return** `Unknown`

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 12), 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 7 by not checking if the merged window $\omega_j \cup \omega_k$ is valid or not. We also modify line 11 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.

6 VEER⁺: IMPROVING VERIFICATION PERFORMANCE

In this section, we develop three 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 6.1). We present a way to detect and prune decompositions that are not equivalent (Section 6.2). We then discuss how to rank the decompositions to efficiently explore their search space (Section 6.3).

6.1 Reducing Search Space Using Segmentations

The size of the decomposition structure in Figure 13 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 14. In this case, we can consider the covering windows of those never overlapping separately. Based on this observation, we introduce the following concepts.

Definition 6.1 (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.

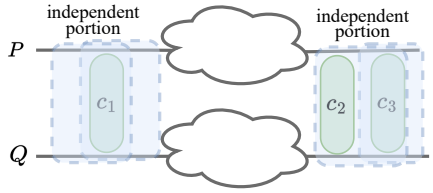


Figure 14: An example where any covering window of an edit operation c_1 never overlaps with a covering window of another edit operation c_2 or c_3.

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. *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.

Lemma 6.2. *For a version pair P and Q with a set of edit operations $\delta = \{c_1 \ldots c_n\}$ from P to Q, if every segment $\mathcal{S}$ in a segmentation ψ is equivalent, then 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`.

Figure 15 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

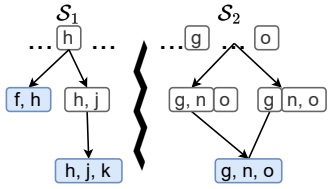


Figure 15: Two segments to reduce the decomposition-space of the running example.

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

```
1  ψ ← constructSegmentation (P, Q, M)
2  for every segment S_i ∈ ψ do
3  |   result_i ← Algorithm 2 (S_i)
4  |   if result_i is not True then
5  |   |   break
6  end
7  if every result_i is True then
8  |   return True
9  if result_i is False and there is only one segment including
     the entire version pair then
10 |   return False
11 return Unknown
```

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.

6.2 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 16 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.

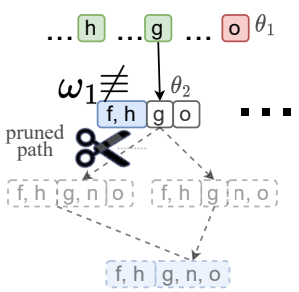


Figure 16: Example to show the pruned paths after verifying the maximal window highlighted in blue to be not equivalent.

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 12[3]. If the window is equivalent, we continue the search as before.

6.3 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 15 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 4) 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 13 has 11 unmerged windows, and the average number of operators in its covering

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

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 EXPERIMENTS

In this section we present an experimental evaluation of the proposed solutions.

7.1 Experimental Setup

Workflow workloads. We constructed six workflows on two datasets TPC-DS [44] and IMDB [24] as shown in Table 2. 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.

Table 2: Workloads used in the experiments.

Workflow#	Description	Type of operators	# of operators	# of links	# of generated pairs
$W1$	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 aggregates	12	12	3
$W6$	IMDB all movies of directors with certain criteria	2 replicates, 4 joins, 2 unnests	18	20	3

Edit operations. For each workflow, we constructed versions by performing edit operations. We used two types of edit operations.

(1) Calcite transformation rules [10] 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 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 2. For each pair of versions, one of the versions is always the original one.

(2) TPC-DS V2.1 [44] iterative edits for non-equivalent 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 [52] as the EV in Scala. We implemented Veer$^+$ by including the optimization techniques presented in Section 6. We evaluated the solution by comparing Veer and Veer$^+$. 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.

7.2 Verifying Two Versions with a Single Edit

We compared the performance of the baseline and Veer$^+$ for verifying equivalent pairs with a single edit using workflows $W1 - W6$. Every workflow in the experiment had one unique MCW, except workflows $W1$ and $W5$, each having two MCWs. Each MCW in $W1$ was equivalent. One MCW of $W5$ consisted of 5 operators and was not equivalent to its mapped sub-DAG. The other MCW of $W5$ had 8 operators and was equivalent to its mapped sub-DAG. The number of operators in the MCW of all the workflows in the experiment was between 5 and 24. We want to compare Veer and Veer$^+$ in terms of how many windows each approach explored and tested, and how much time each approach took to verify the equivalence.

Table 3 shows the result of the experiments. The number of windows explored by the baseline depends on the number of neighbors of the covering window every time it is expanded. Workflow $W4$ had many links because of the large number of replicate and join operators, which caused the number of explored windows to go up to 27,456. The baseline explored only 14 and 21 windows for workflows $W5$ and $W6$, respectively. This is due to the small size of the MCW in both cases (only 6 and 8). The baseline took only a few milliseconds to verify the equivalence of the pairs in every workflow except for workflows $W2$ and $W4$ as it took 2.100 seconds up to 31 minutes, respectively. The baseline tested two MCW's for verifying the pair of workflow $W5$. This means that it tested the non-equivalent MCW first.

In general, Veer$^+$ was not affected by the number of neighbors. The number of windows it explored was almost the same as the size of the MCW thanks to the ranking optimization, which caused the algorithm to explore fewer windows to reach a MCW. It explored only 23 windows for $W4$. Therefore, the time taken to check the equivalence was small. It took only 0.001 seconds for verifying $W1$, $W5$, and $W6$, and only 0.140 seconds for verifying $W4$. Unlike the baseline, Veer$^+$ tested only one MCW for $W5$. In general, the time it took both approaches to verify the equivalence was proportional to the number of windows explored. Both approaches took around the same time to call the EV, which took less than a second (between 0.020s and 0.800s).

7.3 Evaluating Veer$^+$ Optimizations

We used workflow $W3$ for evaluating the three optimization techniques discussed in Section 6. 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

Table 3: Analysis of verifying two versions with a single edit.

Workflow	# of MCWs	AVG. # of operators in MCW	# of explored windows		# of tested MCWs		Time of exploring windows		Time of calling EV	
			Veer	Veer$^+$	Veer	Veer$^+$	Veer	Veer$^+$	Veer	Veer$^+$
$W1$	2	12	152	11	1	1	0.590	0.001	0.800	0.800
$W2$	1	17	1,456	18	1	1	2.100	0.006	0.070	0.110
$W3$	1	13	378	13	1	1	0.420	0.004	0.130	0.080
$W4$	1	24	27,456	23	1	1	1,897.720	0.140	0.130	0.080
$W5$	2	6	14	8	2	1	0.020	0.001	0.100	0.030
$W6$	1	8	21	8	1	1	0.010	0.001	0.030	0.020

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 4 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 4: 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

7.4 Verifying Two Versions with Multiple Edits

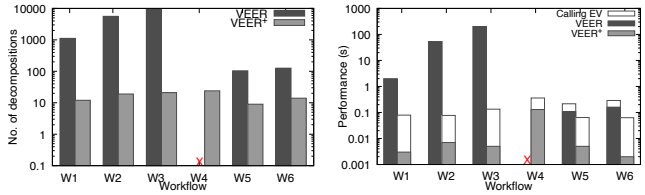
We compared the performance of the baseline and Veer$^+$. We want to know how much time each approach took to test the equivalence of the pair and how many decompositions each approach explored. We used workflows $W1-W6$ with two edits. We used one equivalent pair and one non-equivalent pair from each 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 11), 'is valid' (line 7), and 'merge' (line 8) in Algorithm 2 was negligible, thus we only report the overhead of calling the EV.

Performance for verifying equivalent pairs. Figure 17a 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 6.1 (using non-supported operators to divide the pair). We note that the overhead of constructing a segmentation using the second approach was negligible.

Figure 17b 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 non-equivalent 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.

Performance of verifying non-equivalent pairs. Figure 18a shows the number of decompositions explored by each approach. Since the pairs are not equivalent, both approaches almost exhaustively explored all of the possible decompositions, trying to find an equivalent one. Veer$^+$ explored fewer decompositions compared to the baseline when testing $W3$ and $W6$, 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.

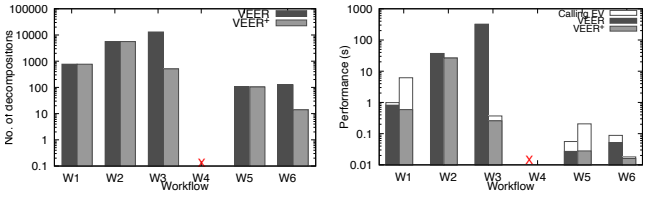
The result of the running time of each approach is shown in Figure 18b. Veer's performance when verifying non-equivalent pairs was the same as when verifying equivalent pairs because, in



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

Figure 17: 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.

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, and its performance was nearly identical to the baseline because it explored the same number of decompositions. We observe that for $W1$ and $W5$, Veer$^+$'s running time was even longer than the baseline due to the overhead of calling the EV up to 130 and 10 times, respectively, 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 non-equivalent decompositions. Veer$^+$'s performance on $W3$ and $W6$ 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 non-equivalent segment was tested first.



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

Figure 18: Comparison between the two algorithms for verifying non-equivalent pairs with two edits. An "×" sign means the algorithm was not able to finish within an hour.

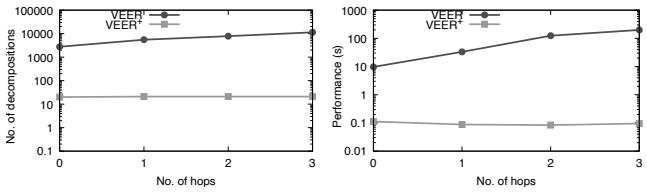
7.5 Effect of the Distance Between Edits

We evaluated the effect of the placement of changes on the performance of both approaches. We are particularly interested in how 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 19a 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 19b 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 19: Effect of the distance between changes (W2 with two edits)

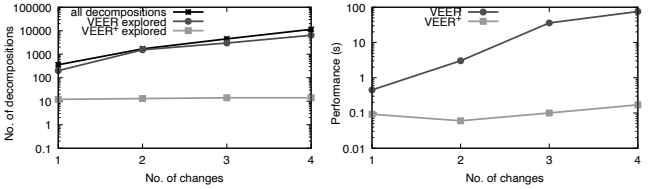
7.6 Effect of the Number of Changes

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 [48]. 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 20a 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 20b 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.



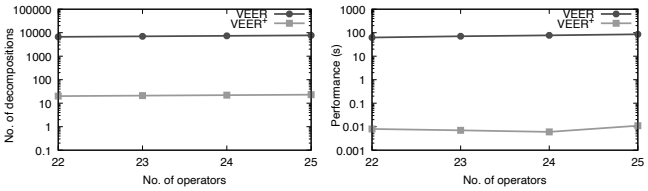
(a) # of explored decompositions. **(b) Running time.**

Figure 20: Effect of the number of changes (W1).

7.7 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 21a 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 21b 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 21: 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.

8 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, 11, 26, 41]. With the recent advancement of developing proof assists and solvers [15, 16], there have been new solutions [13, 51, 52]. UDP [13] and WeTune [46] use semirings to model the semantics of the pair and use a proof assist, such as Lean [16] to prove if the expressions are equivalent. Equitas [52] and Spes [51] 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 [15]. Other works also use an SMT solver to verify the equivalence of a pair of Spark jobs [22]. Our solution uses them as black boxes to verify the equivalence of a version pair.

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, 21, 28, 33, 45, 49]. The second includes solutions to track results of different versions of data processing workflows [5, 14, 32, 47].

Materialization reuse. There is a large body of work on answering data processing workflows using views [17, 18, 27, 39, 40]. Some solutions [19] focus on deciding which results to store to maximize future reuse. Other solutions [34, 50] 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 [20, 23, 30, 42] 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) [36, 39, 48], and do not generalize to finding reuse opportunities by finding equivalence of any pair of workflows.

9 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] Rana Alotaibi, Bogdan Cautis, Alin Deutsch, and Ioana Manolescu. 2021. HADAD: A Lightweight Approach for Optimizing Hybrid Complex Analytics Queries. In *SIGMOD '21: International Conference on Management of Data, Virtual Event, China, June 20-25, 2021*. ACM, 23–35. https://doi.org/10.1145/3448016.3457311

[5] 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

[6] Alteryx Website, https://www.alteryx.com/.

[7] Alteryx Weekly Challenge, https://community.alteryx.com/t5/Weekly-Challenge/bd-p/weeklychallenge.

[8] Apache Flink http://flink.apache.org.

[9] 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

[10] Calcite benchmark, https://github.com/uwdb/Cosette/tree/master/examples/calcite.

[11] 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

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

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

[16] 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

[17] 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

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

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

[20] 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

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

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

[23] 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

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

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

[26] 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

[27] 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

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

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

[30] 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

[31] 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

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

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

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

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

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

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

[38] 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

[39] 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

[40] 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

[41] 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

[42] 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

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

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

[45] 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*.

[46] 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

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

[48] 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

[49] 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*.

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

[51] 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

[52] 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).