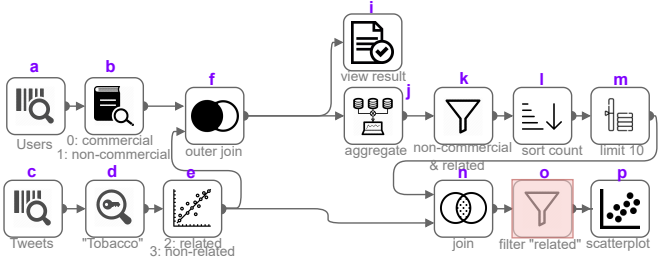
Veer: Verifying Equivalence of Workflow Versions in Iterative Data Analytics

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

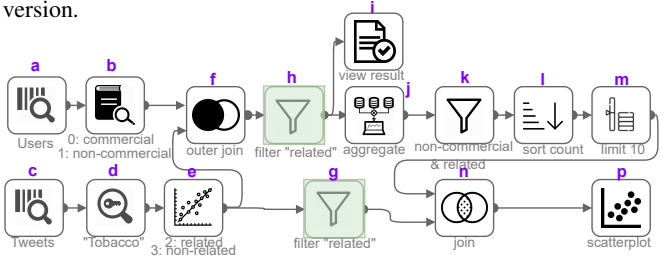
***Abstract*—Data analytics is an iterative process in which an analyst makes many iterations of changes to refine a workflow, generating a different version at each iteration. In many cases, the result of executing a workflow version is equivalent to the result of a prior one. Identifying such equivalence between the execution results of different workflow versions is important for optimizing the performance of a workflow by reusing results from a previous run. The size of the workflows and the complexity of their operators (e.g., UDF and ML models) often make existing equivalence verifiers (EVs) unable to solve the problem. In this paper, we present "Veer," which verifies the equivalence of two workflow version by leveraging the knowledge of the edits between the two versions. Veer divides the version pair into small parts, called *windows* and verifies the equivalence within each window using an existing EV as a black box. Our thorough experiments on real-world workflows show that Veer is able to not only verify the equivalence of workflows that cannot be verified by existing EVs but also do the verification efficiently.**

I. Introduction

Data-processing platforms enable users to quickly construct complex analytical workflows [1]–[3]. These workflows are refined in iterations, generating a new version at each iteration [4], [5]. 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 an analyst constructs the initial workflow version (a) and executes it, she refines the workflow to achieve the desired results. This change 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.

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

Motivation. There has been a growing interest recently in keeping track of these workflow versions and their execution results [3], [6], [7]. In many applications, these workflows have a significant amount of overlap and they can often produce the same results [6], [8]–[10]. For example, 45% of the daily jobs in Microsoft's analytics have overlap [9]. 27% of $9,486$ workflows from Ant Financial to detect fraud transactions share common computation, and 6% of them are equivalent [8]. In the running example, the edits applied on version (a) that lead to a new version (b) have no effect on the result of the 'p' sink. Identifying such equivalence between execution results of different workflow versions is important as 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 [2], [8]. Optimizing the performance of a workflow execution has been studied extensively in the literature [11], [12]. One optimizing technique is to leverage the iterative nature of data analytics to reuse previously computed results [4], [9].

Challenges and opportunities. We observe two unique traits of iterative data analytics. (*C1*) The workflows can be large and complex, with operators that are semantically rich [1], [5]. For example, the latest 8 workflows in the Alteryx's workflows hub [1] (at the time of accessing the website as detailed in [13]) have an average of 48 operators, with one of the workflows containing 102 operators. Some operators are user-defined functions (UDFs) that implement highly customized logic including machine learning techniques for analyzing data of different modalities such as text, images, audios, and videos [5]. For instance, the workflows in the running example contain two non-relational operators, namely Dictionary Matcher and Classifier. (*O1*) Adjacent versions of the same workflow tend to be similar, due to fine tuning [5], [12]. For example, 50% of the workflows belonging to the benchmarks that simulated real iterative tasks on video [5] and TPC-H [12] data are similar. The refinements between the successive versions comprised of only a few changes over a particular part of the workflow.

Limitations of existing solutions. Workflows include relational operators and UDFs [2]. Thus, we can view the problem of checking the equivalence of two workflow versions as a problem of checking the equivalence of two SQL queries. The latter is undecidable in general [14]. There have been many Equivalence Verifiers (EVs) proposed to verify the equivalence of two SQL queries [8], [10], [15]. 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 (*C1*). To investigate the limitations of the existing EVs, we analyzed the SQL queries and workflows from 6 workloads. First, we generated equivalent versions of the workflows by adding an empty filter operator. Then, we used EVs from the literature [8], [10], [15], [16] to test the equivalence of these two versions. Table I shows the average percentage of pairs that can be verified by the EVs is low. More details about all the workloads and the results of these EVs in the extended version [13].

TABLE I: 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 [17]	232	34.81%
Knime workflows hub [18]	37	2.70%
TPC-DS benchmark [19]	99	2.02%

Our approach. To solve the problem of verifying the equivalence of two complex workflow versions, we leverage the fact that the two workflow versions are almost identical except for a few local changes (*O1*). In this paper, we present Veer, a verifier to test the equivalence of two workflow versions. It addresses the aforementioned problem by utilizing existing EVs as a black box. In §III, we give an overview of the solution, which divides the workflow version pair into small parts, called "windows", and we push testing the equivalence of a window to the EV. This solution is simple yet highly effective in solving a challenging problem, making it easily applicable to a wide range of applications.

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

Contributions. (1) We formulate the problem of verifying the equivalence of two complex workflow versions in iterative data analytics. Veer is the first work that studies this problem by incorporating the knowledge of user edit operations into the solution (§II). (2) We give an overview of Veer and define the "window" concept that is used in the equivalence verification algorithm (§ III). (3) We first consider a single edit case. We analyze how the containment between two windows is related to their equivalence results, and use this analysis to derive the concept of "maximal covering window". We give insights on how to use EVs and the subtle cases of the EVs completeness (§IV). (4) We study the general case where the two versions have multiple edits. We analyze the challenges of using overlapping windows, and propose a solution based on the "decomposition" concept. We discuss the correctness and the completeness of Veer (§V). (5) We provide optimizations in Veer$^+$ to improve the performance of the baseline algorithm (§VI). (6) We report the results of a thorough experimental evaluation, which shows 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 (§VII).

II. Problem Formulation

Data processing workflow. We consider a data processing workflow W as a directed acyclic graph (DAG), where each vertex is an operator. Each operator contains a computation function, we call it a *property* such as a predicate condition, e.g., Price < 20. 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. A workflow may have multiple data source operators and multiple sink operators. The workflow in Figure 1 has two source operators "Tweets" and "Users" and two sink operators s_i and s_p.

A. Workflow Version Control

A workflow W has a list of versions $V_W = [v_1, \ldots, v_m]$ and each v_j is an immutable version of workflow W and is produced by a transformation from edit operations.

Definition II.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/removal of a new/existing link.*

A combination of these edits is a *transformation*, denoted as δ_j. In the running example, the analyst makes edits to revise the workflow version v_1. In particular, she (1) deletes the $\mathsf{Filter_o}$ operator; (2) adds a new $\mathsf{Filter_h}$ operator; (3) and adds a new $\mathsf{Filter_g}$ operator. These operations correspond to a transformation, δ_1, which results in a new version v_2.

Workflow edit mapping. Given a version pair (P, Q), there is a corresponding edit mapping $\mathcal{M}$ to transform 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 in P that are not mapped to any operators in Q are assumed to be deleted. Similarly, those operators in Q that are not mapped

onto by any operators in P are assumed to be inserted. More details about the relation between workflow edit operations and mapping in [13].

B. Result Equivalence of Workflow Versions

A user submits a request to execute a workflow version. The execution produces a *result* for each sink in the version.

Assumption. *Multiple executions of a workflow (or a portion of the workflow) will always produce the same results* [1].

The execution request for a 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 1, executing the workflow version v_2 produces a result of the sink s_p equivalent to the result of the corresponding sink in v_1. Notice however that the result of s_i in v_2 is not equivalent to the result of s_i in v_1.

Definition II.2 (Sink Equivalence and Version-Pair Equivalence)**.** *Consider two workflow versions P and Q with a set of edits $\delta = \{c_1 \dots c_n\}$ and a mapping $\mathcal{M}$ from P to Q. For each sink s of P and its 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 under the application's specified table semantics, which is further detailed in the extended version [13]. We say s is* inequivalent to $\mathcal{M}(s)$*, denoted as "$s \not\equiv \mathcal{M}(s)$," if there exists an instance of data sources of P and Q such that the two sinks produce different results. The two versions are called* equivalent*, denoted as "$P \equiv Q$", if each pair of their sinks under the mapping is equivalent. The two versions are called* inequivalent*, denoted as "$P \not\equiv Q$, if there is a pair of their sinks under the mapping that is inequivalent.*

Expressive power of workflows and SQL. Data-processing workflows may involve complex operations, such as user-defined functions (UDFs). We consider workflow DAGs that can be viewed as a class of SQL queries that do not contain recursion. In this paper, we focus on workflows with Union-Selection-Projection-Join-Aggregation (USPJA) and UDFs with deterministic functions under the application's specific table semantics (Set, Bag, or Ordered Bag). Thus, the problem of testing the equivalence of two workflow versions can be treated as testing the equivalence of two SQL queries without recursion.

C. Equivalence Verifiers (EVs)

An equivalence verifier (or "EV" for short) takes as an input a pair of SQL queries $\mathbf{Q_1}$ and $\mathbf{Q_2}$. An EV returns `True` when $\mathbf{Q_1} \equiv \mathbf{Q_2}$, `False` when $\mathbf{Q_1} \not\equiv \mathbf{Q_2}$, or `Unknown` when the EV cannot determine the equivalence of the pair under a specific table semantics [8], [10], [15], [16], [20], [21]. An EV requires two queries to meet certain requirements (called "restrictions") in order to test their equivalence. We will discuss these restrictions in detail in Section IV-B.

[1]This is valid in many real-world applications as detailed in (§ VII)

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

In this paper, we study the problem of testing the equivalence of two versions with a single sink and given a single EV. The solution can be easily generalized to the case of multiple EVs as discussed in the extended version [13], and to the case of multiple sinks as discussed in a followup work [22].

III. 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 III-A). We formally define the concepts of "window" and "covering window" (Section III-B).

A. Veer: Overview

To verify the equivalence of a pair of sinks in two workflow versions, Veer breaks the version pair into multiple windows, each of which includes local changes and satisfies the EV's restrictions. It then uses existing EVs as a black box to verify if the pair of portions of the workflow versions in the window is equivalent. Veer is agnostic to the underlying EVs, making it usable for any EV of choice.

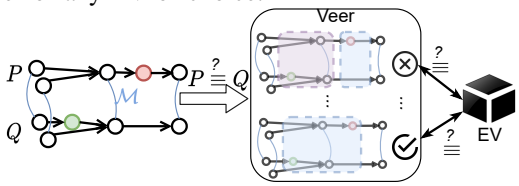


Fig. 2: Overview of Veer.

B. Windows and Covering Windows

Definition III.1 (Window)**.** *Consider two workflow versions P and Q with a set of edits $\delta = \{c_1 \dots c_n\}$ from P to Q for a corresponding mapping $\mathcal{M}$. 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 $\mathcal{M}$ must be either both in ω or both outside ω.*

The operators in the sub-DAGs $\omega(P)$ and $\omega(Q)$ without their descendants in the sub-DAGs are called the window's *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. The operators in the sub-DAGs $\omega(P)$ and $\omega(Q)$ without their ancestors in the sub-DAGs are called the window's *sources*. Figure 3 shows a window ω, where each sub-DAG includes the Classifier operator and two downstream operators Left-Outerjoin and Join, the two sinks of the sub-DAG. We omit portions of the workflows in all the figures throughput the paper for clarity.

Definition III.2 (Neighbor Window)**.** *Consider two workflow versions P and Q with a set of edits δ from P to Q and a corresponding mapping $\mathcal{M}$. We say two windows, w_1 and w_2, are neighbors if there exists a sink/source operator of the sub-DAGs in one of the windows that is a direct*

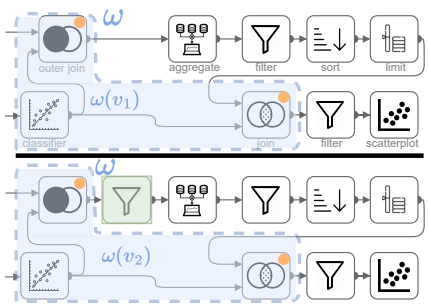


Fig. 3: A window ω (shown as "▭") and each sub-DAG of ω contains two sinks (shown as "●").

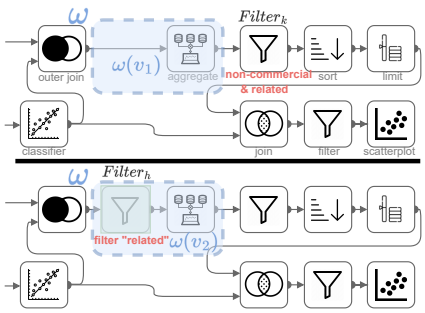


Fig. 4: A covering window ω for adding $\mathsf{Filter_h}$. v_1 is above the horizontal line and v_2 is below the line.

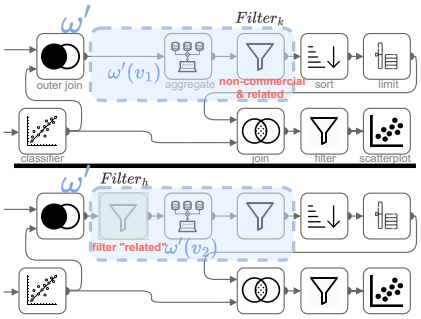


Fig. 5: A covering window ω' where its pair of sub-DAGs are equivalent.

upstream/downstream operator in the original DAG to a source/sink operator of the sub-DAGs in the other window.

Definition III.3 (Covering window)**.** *Consider two workflow versions P and Q with a set of edits $\delta = \{c_1 \dots c_n\}$ from P to Q and a corresponding mapping $\mathcal{M}$ from P to Q. A* covering window*, denoted as ω_C, is a window to cover a set of changes $C \subseteq \delta$. That is, the sub-DAG in P (respectively sub-DAG in Q) in the window includes operators/links of the edit operations in C.*

When the edit operations are clear in the context, we simply write ω to refer to a covering window. Figure 4 shows a covering window for the change of adding the $\mathsf{Filter_h}$ to v_2. The covering window includes the sub-DAG $\omega(v_1)$ of v_1 that contains the Aggregate operator. It also includes the sub-DAG $\omega(v_2)$ of v_2 that contains the $\mathsf{Filter_h}$ and Aggregate operators.

Definition III.4 (Equivalence of the two sub-DAGs in a window)**.** *The two sub-DAGs $\omega(P)$ and $\omega(Q)$ of a window ω are* equivalent*, denoted as "$\omega(P) \equiv \omega(Q)$," if they are equivalent as two stand-alone DAG's, i.e., without considering the constraints from their upstream operators. That is, for every instance of input sources in the sub-DAGs, each sink s of $\omega(P)$ and the corresponding $\mathcal{M}(s)$ in $\omega(Q)$ produces the same results. In this case, for simplicity, we say this window is equivalent.*

Figure 5 shows an example of a covering window ω', where its sub-DAGs $\omega'(v_1)$ and $\omega'(v_2)$ are equivalent. [13] shows an illustrative figure to explain the input sources of a window.

Definition III.5 (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'.*

For instance, the window ω in Figure 4 is contained in the window ω' in Figure 5.

IV. 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.

A. Verification Using a Covering Window

Lemma IV.1. *Consider a version pair with a single edit between them. If there is a covering window of the edit such that the sub-DAGs of the window are equivalent, then the version pair is equivalent.*

Due to space limitation, we refer interested readers to the extended version for all of the proofs [13]. Based on this lemma, we can verify the equivalence of a pair of versions as follows: We consider a covering window and check the equivalence of its sub-DAGs by passing each pair of sinks and the sink's ancestor operators in the window (to form a query pair) to an EV. To pass a pair of sub-DAGs to the EV, we need to complete it by attaching virtual sources and sinks. This step is vital for determining schema information. The sub-DAGs are then transformed into a representation understandable by the EV, e.g., logical DAG. If the EV shows that all the sink pairs are equivalent, then the two versions are equivalent.

A key question is how to find such a covering window. Notice that the two sub-DAGs in Figure 4 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 5), 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.

B. EV Restrictions and Valid Windows

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

Definition IV.1 (EV's restrictions)**.** Restrictions *of an EV are a set of conditions such that for each query pair if this pair satisfies these conditions, then the EV is able to determine the equivalence of the pair without giving* ``Unknown''.

There are two types of restrictions.

- Restrictions due to the EV's explicit assumptions: For example, UDP [15] supports reasoning of certain operators, e.g., Aggregate and SPJ, but not Sort.
- Restrictions that are *derived* due to the modules used by the EV: For example, Spes [8] uses an SMT solver [20] to determine if a FOL formula is satisfiable. SMT solver

is not complete for determining the satisfiability of formulas when their predicates have non-linear conditions [23]. Thus, these EVs require the predicate conditions in their expressions to to be linear.

The following is a sample of the explicit and derived restrictions of the Equitas [10] EV.

R1. Both queries should have the same number of Aggregate.

R2. If there is 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.

Providing an exhaustive list of all derived restrictions is challenging. A more comprehensive set of conditions allows Veer to push more windows to be verified by the given EV. A conservative approach can be adopted by limiting the restrictions, e.g., considering SPJ only. We relax the definition of EV restrictions in the extended version [13], discuss the consequences of relaxing the definition, and propose solutions.

Definition IV.2 (Valid window w.r.t an EV)**.** *We say a window is* valid *with respect to an EV if it satisfies the EV's restrictions.*

C. Maximal Covering Window (MCW)

A main question is how to find a valid covering window w.r.t 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. 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 IV.2. *Consider a version pair (P, Q) with a single edit c. Suppose a covering window ω of c is contained in another covering window ω'. If the sub-DAGs in window ω are equivalent, then the sub-DAGs of ω' are also equivalent.*

Based on Lemma IV.2, we can focus on covering windows that have as many operators as possible without violating the restrictions of the EV. If the EV shows that such a window is not equivalent, then none of its sub-windows can be equivalent.

Definition IV.3 (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 [10]. Figure 6 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 [10]. Similarly, the Aggregate operator on the right cannot be included in ω_1 because it violates restriction $R2$. 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 [10]. The MCW ω_2 is verified by Equitas [10] to be equivalent, whereas ω_1 is not. Notice that one equivalent covering window is enough to show the equivalence of the two versions.

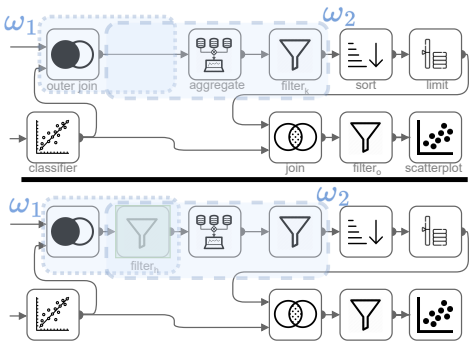


Fig. 6: Two MCW ω_1 and ω_2 satisfying the restrictions of Equitas to cover the change of adding Filter_h to v_2.

D. Finding MCWs to Verify Equivalence

We use the running example to explain the details of finding a MCW. An illustrative figure and a pseudocode are in the extended version [13]. The first step is to initialize the window to include the change only. In this example, window ω_1 includes sub-DAG $\omega_1(v_2)$ with Filter_h and its corresponding operator using the mapping in $\omega_1(v_1)$. Then we expand all the windows created so far, i.e., ω_1. To expand a 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 the window has not been explored yet and is valid. In this example, we create two windows ω_2 and ω_3 to include the operators Outer-join and Aggregate in each window. We add those valid windows in the traversal list to be further expanded in the following iterations. When none of the neighbors can be merged with a window to produce a valid one, we mark the window as maximal. A subtle case arises when adding a single neighbor yields an invalid window, but adding a combination of neighbors yields a valid window. This is discussed in Section V-E. We test the equivalence of this maximal window by calling the EV. If the EV says it is equivalent, the algorithm returns `True`. 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`. 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.

V. Two Versions with Multiple Edits

So far, we assumed there is a single edit to transform a workflow version to another. 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.

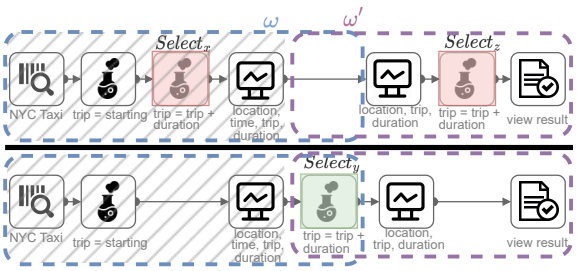


Fig. 7: Each of window ω and window ω' is equivalent. But the version pair is not. The shaded gray area is the input to window ω'.

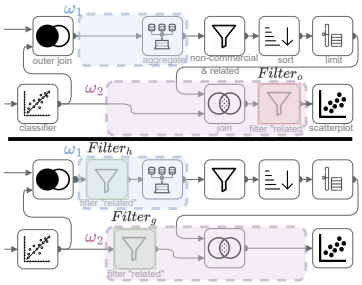


Fig. 8: A decomposition θ with two covering windows ω_1 and ω_2 that cover the three edits.

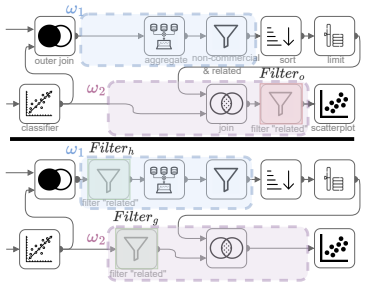


Fig. 9: Equivalent pair of sub-DAGs for every covering window in a decomposition θ'.

A. Can we use overlapping windows?

Consider the example in Figure 7, which is inspired from the NY Taxi dataset [24] 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 from version v_1 to version v_2. The example shows two overlapping windows ω and ω', each window is equivalent.

Definition V.1 (Overlapping windows)**.** *We say that two windows, ω_1 and ω_2, overlap if at least one operator is included in any of the sub-DAGs of both windows.*

We cannot say the version pair in the above example is equivalent. The reason is that for the pair of sub-DAGs in ω' to be equivalent, its input sources have to be the same (the shaded area in grey in the example). However, we cannot infer the equivalence of the shaded portion of the sub-DAG, because the pair includes a change and their equivalence is not explicitly tested. 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 will always be a one-to-one mapping of the two sub-DAGs. The solution in Section IV finds *any* window such that its sub-DAGs are equivalent, hence cannot be directly used to solve the case of verifying the equivalence of the version pair when there are multiple edits.

To overcome this, we require the covering windows to be disjoint. A naive solution is to decompose the version pair into all possible disjoint windows.

B. Version Pair Decomposition

Definition V.2 (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 8 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 [2]. Next, we show how to use a decomposition to verify the equivalence of the version pair by generalizing Lemma IV.1.

Lemma V.1. *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 every possible one, but the number can grow as the size of the workflow and the number of changes increase. The following concept helps us reduce the number of decompositions that need to be considered.

Definition V.3 (Decomposition containment)**.** *We say a decomposition θ is* contained *in another θ', denoted as $\theta \subseteq_d \theta'$, if for every window in θ, there exists a window in θ' that contains it.*

Figure 9 shows an example of a decomposition θ' that contains the decomposition θ in Figure 8. The following generalization of Lemma IV.2 can help us ignore decompositions that are properly contained by other decompositions.

Lemma V.2. *Consider a version pair P and Q with a set of edit operations δ 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.*

C. Maximal Decompositions (MD) w.r.t. an EV

Lemma V.2 shows that we can focus on finding a decomposition that contains 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.

Definition V.4 (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 V.5 (Maximal Decomposition (MD))**.** *We say a valid decomposition θ is* maximal *if no other valid decomposition θ' exists such that θ' properly contains θ.*

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

In order to find an MD, we leverage the fact that valid decompositions w.r.t to 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 is an MD as there are no other decompositions that contain it. 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. The branching factor depends on the number of changes, the neighbors of windows, and the EV's restrictions. Figure 10 shows the hierarchical relationships of the valid decompositions of the running example when the EV is Equitas [10]. The example shows two MD θ_{12} and θ_{16}.

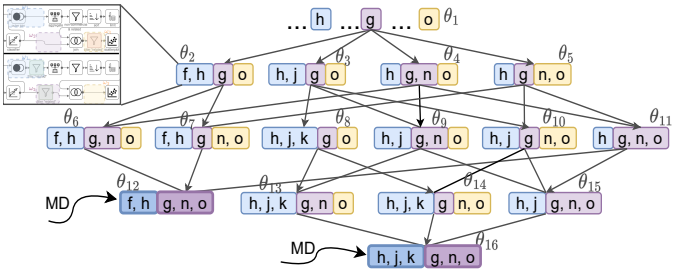


Fig. 10: Hierarchy of valid decompositions w.r.t an EV. Each letter corresponds to a pair of operators from the running example. We show the containment of covering windows and omit details of non-covering windows.

D. Finding an MD to Verify Equivalence (Baseline)

We use the example in Figure 10 to explain Algorithm 1. Line 1-2 returns `True` to indicate the pair is equivalent if there are no changes. Otherwise, we construct an initial decomposition, which includes each operator as a window (line 3). In each of the following iterations, we expand constructed decompositions. We merge windows if their union is valid (line 10). If a window cannot be further expanded, we mark the window as maximal (line 14). A subtle case arises when the merge of two windows yields an invalid window but the merge of a combination of more windows produces a valid window. We discuss this in Section V-E. If all of the windows in the decomposition are maximal, we mark the decomposition as maximal, and verify whether each covering window is equivalent by passing it to the given EV (line 17). If all of the windows are verified to be equivalent, we return `True` to indicate that the version pair is equivalent (line 18). If the decomposition includes a single window consisting of the entire version pair, and the EV decides that the window is not equivalent, then the algorithm returns `False` (line 20). Otherwise, we continue exploring other decompositions until there are no more decompositions to explore. In that case, we return `Unknown` to indicate that the equivalence of the version pair cannot be determined (line 22).

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

Algorithm 1: 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

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

E. Improving the Completeness of Algorithm 1

In general, the equivalence problem for two workflow versions is undecidable [14], [21] (reduced from First-order logic). So there is no verifier that is complete [25]. However, there are classes of queries that are decidable such as SPJ [8]. In this section, we show factors that affect the completeness of Algorithm 1 and propose ways to improve its completeness.

1) Window validity. In line 13, if none of the neighbor windows of ω_j can be merged with ω_j, we mark ω_j and stop expanding it.

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

$$P = \{Proj(all) \rightarrow Fltr(age > 24) \rightarrow Agg(count\ by\ age)\}.$$

$$Q = \{Agg(count\ by\ age) \rightarrow Fltr(age > 24) \rightarrow Proj(all)\}.$$

Consider a mapping from P to Q is swapping $Project$ with $Aggregate$ and vice-versa for both versions. Suppose the EV is Equitas *[10] and a covering window ω contains the $Project$ from P and its mapped $Aggregate$ from Q. Consider the window expansion procedure in Algorithm 1. If we add filter of both versions to the window, then the merged window is not valid because it violates Equitas's [10] restriction R1, thus the algorithm stops expanding the window. Had we*

continued expanding the window till the end, the final window with three operators is valid.

Using this final window, we can see that the two versions are equivalent, but the algorithm missed this opportunity. A main reason is that the Equitas [10] EV does not have the following property.

Definition V.6 (EV's Restriction Monotonicity)**.** *We say an EV is* restriction monotonic *if for each invalid window ω of a version pair, every window containing ω is also invalid.*

Intuitively, for an EV that has this property such as Spes [8], when the algorithm marks the window ω_j (line 14), this window must be maximal. If the EV does not have this property such as Equitas [10], we can improve the completeness of the algorithm as follows. We modify line 9 by ignoring if the merged window $\omega_j \cup \omega_k$ is valid or not. To avoid expanding the window indefinitely, we devise a procedure for each EV that can test if a window is maximal by reasoning the EV's restrictions.

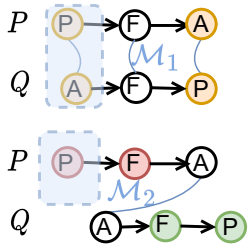


Fig. 11: Example of two edit mappings, one satisfying EV restrictions, the other not.

2) Different edit mappings. Consider two different edit mappings, $\mathcal{M}_1$ and $\mathcal{M}_2$, for the version pair in Example V-E, as shown in Figure 11. Let us assume the given EV is Equitas [10]. If we follow the baseline Algorithm 1, mapping $\mathcal{M}_1$ results in a decomposition that violates Equitas's **R1** restriction. On the other hand, mapping $\mathcal{M}_2$ satisfies the restrictions. This example shows that different edit mappings can lead to different decompositions.

One way to address this issue is to enumerate all possible edit mappings [26] and perform the decomposition search by calling Algorithm 1 for each edit mapping.

Theorem V.2. *(Completeness). Given two workflow versions P and Q, and an EV γ with its restrictions,* Veer *is complete for determining the equivalence of P and Q if the pair satisfies the EV's restrictions.*

VI. Veer$^+$: Improving the Verification Performance

In this section, we discuss four optimization techniques.

A. Reducing Search Space Using Segmentations

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. In this case, we can consider the covering windows of those never overlapping separately.

Definition VI.1 (Segment and segmentation)**.** *Consider a version pair (P, Q) with a set of edits $\delta = \{c_1, \ldots, c_n\}$ for the 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.*

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 that are not supported by the given EV. For example, a Sort operator in the running example cannot be supported by Equitas [10]. 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.

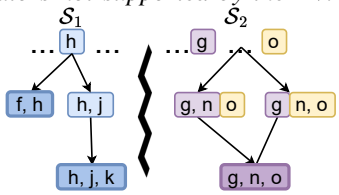


Fig. 12: Two segments to reduce the decomposition-space of the running example.

Figure 12 shows the segments of the running example when using Equitas [10] as the EV. Using the second approach for computing a segmentation, we know Equitas [10] does not support the Sort operator, so we divide the version pair into two segments. The first $\mathcal{S}_1$ includes those operators before Sort, and the second $\mathcal{S}_2$ includes those operators after Sort.

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 is to check the equivalence of the two sub-DAGs in a segment.

Lemma VI.1. *For a version pair (P, Q) with a set of edit operations δ from P to Q, if every segment $\mathcal{S}$ in a segmentation ψ is equivalent, then the version pair is equivalent.*

We first construct a segmentation. For each segment we find if its pair is equivalent by calling Algorithm 1. 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 of a single segment consisting of the entire version pair and Algorithm 1 returns `False`, then we return `False`.

The example in Figure 12 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).

B. Pruning Decompositions

Another way to improve the performance is to prune decompositions that would not be verified equivalent even if they are further expanded. For instance, Figure 13 shows

part of the decomposition hierarchy of the running example. Notice that the first window, $\omega_1(f,h)$ in θ_2, cannot be further expanded and is marked "maximal" but the decomposition can still be further expanded by the other two windows. After expanding the other windows and reaching a maximal decomposition, we realize that the decomposition is not equivalent because one of its windows, ω_1, is not equivalent.

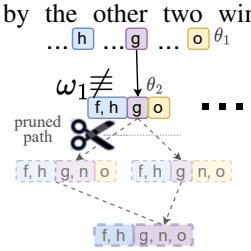


Fig. 13: Pruned paths after verifying the MCW ω_1 is not equivalent.

Based on this observation, if one of the windows in a decomposition becomes maximal, we can immediately test its equivalence to terminate the traversal of the decompositions that contain the inequivalent maximal window.

C. Ranking-Based Search

Ranking segments within a segmentation. We discussed that we need to verify those segments in a segmentation, and if any segment is not equivalent, then there is no need for verifying the other ones. 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. An example scoring function is $\mathcal{F}(S) = m_S + n_S$, where m_S is the number of operators in a segment $\mathcal{S}$ and n_S is its number of changes. A segment should be ranked higher if it has fewer changes, as it leads to a smaller number of decompositions. Similarly, if the number of operators in a segment is smaller, then the number of decompositions is also smaller.

Ranking decompositions within a segment. For each segment, we use Algorithm 1 to explore its decompositions. The algorithm needs an order (line 6) to explore the decompositions. The order, if not chosen properly, can lead to exploring many decompositions before finding an equivalent one. We can optimize the performance by ranking the decompositions. 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 windows in the decomposition. A decomposition is ranked higher if it is closer to reaching an MD. 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.

D. Identifying Inequivalent Pairs Efficiently

The approach discussed so far attempts to find a decomposition in which all of its windows are verified to be equivalent. However, in cases where the version pair is inequivalent, such a decomposition does not exist, and the search framework would continue to look for one to no avail. Testing the equivalence of maximal decompositions (by pushing it to an EV) incurs an overhead due to the EV's reasoning about the semantics of the window. Thus, we want to avoid sending a window to the EV if we can quickly determine beforehand that the version pair is not equivalent.

To achieve this quick identification of inequivalence between two workflow versions, we create a lightweight representation that allows us to partially reason about the semantics of the version pair. Our approach relies on a symbolic representation similar to the existing works [10], [27], denoted as $(\vec{S}, \vec{O})$. In this representation, $\vec{S}$ and $\vec{O}$ are lists that represent the projected fields in the result table and the fields on which the result table is sorted, respectively. To construct the representation, we follow the techniques [10], [15] by using predefined transformations for each operator. In this way, if the list of projected columns (based on $\vec{S}$) of the sink of version P is different from those of the sink in version Q, we can quickly know the two sinks in the versions do not produce the same results because their schemas are different. We can apply the same check to the sorted columns.

VII. Experiments

A. Experimental Setup

Synthetic workload. We constructed workflows $W1-W4$ on TPC-DS [19] dataset. 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 operator.

Real-world workload. We analyzed a total of 179 real-world pipelines from an open-source system. Among the workflows, 81% had deterministic sources and operators, and we focused our analysis on these workflows. Among the analyzed workflows, 58% of the versions had a single edit, while 22% had two edits. We also observed that the UDF operator was changed in 17% of the cases, followed by the Projection operator (6% of the time) and the Filter operator (6% of the time). From this set of workflows, we selected four as a representative subset ($W5 \ldots W8$) and we used IMDB [28] and Twitter [29] datasets. Table II shows a sample of the workflow (full details in the extended version [13]).

TABLE II: Workloads used in the experiments.

Work flow#	Description	Type of operators	# of operators	# of links	# of equivalent version pairs / total pairs
$W1$	TPC-DS $Q40$	4 joins and 1 aggregate	17	16	4/5
$W5$	IMDB ratio of non-original to original movies	1 replicate, 2 joins, 2 aggregate	12	12	2/3
$W8$	Wildfire Twitter analysis	1 join, 1 UDF	13	12	2/3

Edit operations. Table II shows the number of version pairs for each workflow, where one version of the pair is always the original workflow and the other is produced by performing edit operations on the original version. For each real-world workflow, we used the edits performed by the users. For each synthetic workflow, we constructed versions by performing edit operations. We used two types of edit operations.

(1) Calcite transformation rules [17] 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. 'testPushProject-PastFilter' is an example that produces more than a single change. We used a variation of a different number of edits and their placements as we explain in each experiment.

(2) TPC-DS V2.1 [19] iterative edits for inequivalent pairs: These edits are common for the early stages of the iterative analytics, so they may produce a version that is *not equivalent* to the first version. An example edit is changing the aggregate function. 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 any two edit operations from Calcite.

Implementation. We implemented the baseline (Veer) and an optimized version (Veer+) in Java8. We implemented Equitas [10] as the EV in Scala. We evaluated the solution by comparing Veer and Veer+ against a state of the art verifier (Spes [8]), known for its proficiency in verifying query equivalence compared to other solutions. 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.

B. Comparisons with Other EVs

To our best knowledge, Veer is the first technique to verify the equivalence of complex workflows. To evaluate its performance, we compared Veer and Veer+ against Spes [8]. We chose one equivalent pair and one inequivalent pair of versions with two edits from each workflow. Among the 8 workflows examined, Spes [8] failed to verify the equivalence and inequivalence of any of the pairs, because all of the workflow versions included operators not supported by Spes [8]. In contrast, Veer and Veer+ successfully verified the equivalence of 50% ($W1 \dots W3, W5$) and 75% ($W1 \dots W6$), respectively, of the equivalent pairs as reported in details in Section VII-D. Both Veer and Veer+ did not verify the equivalence of the equivalent pair in $W7$ because none of the constructed decompositions were verified as equivalent by the EV. Veer and Veer+ did not verify the equivalent pair of $W8$ because the change to its versions was made on a UDF operator, resulting in the absence of a valid window that satisfies the EV's restrictions used. Veer+ was able to use the heuristic discussed in Section VI-D to detect the inequivalence of about 50% of the inequivalent pairs ($W5 \dots W8$). We note that Veer and Veer+ can be made more powerful if we employ an EV that can reason the semantics of a UDF operator.

TABLE III: Comparison evaluation of Veer and Veer+ against Spes [8].

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

C. Evaluating Veer+ Optimizations

We used an equivalent version pair from workflow $W3$ for evaluating the first three optimization techniques discussed in Section VI. We used three edit operations: one edit after the Union operator (which is not supported by Equitas [10]) and two edits (pushFilterPastJoin) before the Union.

Table IV shows that the worst performance was 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. 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. 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 0.17 seconds for exploring the decompositions and 0.22 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 IV: Enabling optimizations ($W3$ with three edits). "S" indicates segmentation, "P" for pruning, and "R" for ranking. A ✓means the optimization was enabled and an × disabled.

S	P	R	# of decom-positions explored	Exploration time (s)	Calling EV (s)	Total time (s)
×	×	×	19,656	1,629	0.22	1,629
×	✓	×	3,614	111	0.15	111
✓	×	×	430	0.51	0.18	0.69
✓	✓	✓	21	0.20	0.31	0.51

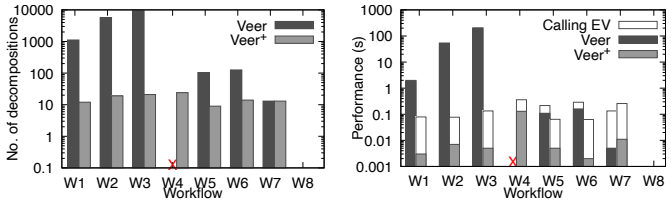
D. Veer and Veer+ on Verifying Two Versions

The workflows in the experiment had one segment, except $W3$, $W5$, $W6$ had two segments. The overhead of 'is valid', 'merge', and 'is maximal' (lines 9, 10, 13 respectively) in Algorithm 1 was negligible. So we only report the overhead of calling the EV.

Performance for verifying equivalent pairs. Figure 14a 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 $W4$ in less than an hour, due to the large number of neighboring windows. 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, while we used the second approach of finding a segmentation as we discussed in Section VI-A. For workflow $W7$, the size of the windows in a decomposition were small because the windows violated the restrictions of the used EV. Therefore,

the "expanding decompositions" step stopped early. Veer and Veer+ detected that the change on $W8$ was done on a non-supported operator (UDF) by the chosen EV (Equitas [10]), thus the decomposition was not expanded to explore other ones and the algorithm terminated without verifying its equivalence.

Figure 14b shows the running time for each approach to verify the equivalence. The baseline took 2 seconds to verify the equivalence of $W1$, and 2 minutes for verifying $W3$. Veer+, on the other hand, had a running time of a sub-second in verifying the equivalence of all of the workflows. Veer+ tested 9 MCWs for a chance of pruning inequivalent decompositions when verifying $W6$. This caused the running time for verifying $W6$ to increase due to the overhead of calling the EV. In general, the overhead of calling the EV was about the same for both approaches. In particular, it took an average of 0.04 and 0.10 seconds for both the baseline and Veer+, respectively, to call the EV.

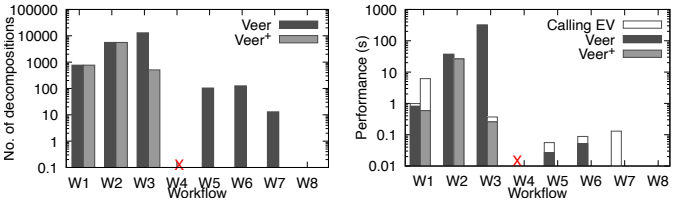


(a) # of explored decompositions. (b) Running time.

Fig. 14: Comparison between Veer and Veer+ for verifying equivalent pairs with two edits. Overhead of calling EV by Veer is not visible due to the logscale.

Performance of verifying inequivalent pairs. Figure 15a shows the number of decompositions explored by each approach. Veer exhaustively explored all of the possible decompositions, trying to find an equivalent one. Veer+ explored fewer decompositions compared to the baseline when testing $W3$, thanks to the segmentation. Both approaches were not able to finish testing $W4$ within one hour because of the large number of possible neighboring windows. Veer+ was able to quickly detect the inequivalence of the pairs of workflows $W5 \ldots W8$ thanks to the partial symbolic representation.

Veer's performance when verifying inequivalent pairs was the same as when verifying equivalent pairs because, in both cases, it explored the same number of decompositions. On the other hand, Veer+'s running time was longer than when the pairs were equivalent for workflows $W1 \ldots W4$. We observe that for $W1$, Veer+'s running time was even longer than the baseline due to the overhead of calling the EV up to 130, compared to only 4 times for the baseline (shown in Figure 15b). Veer+ called the EV more due to continuously testing MCWs for a chance of pruning inequivalent decompositions. Veer+'s performance on $W3$ was better than the baseline. The reason is that there were two segments, and each segment had a single change. We note that Veer+ tested the equivalence of both segments, even though there could have been a chance of early termination if the inequivalent segment was tested first. The time it took Veer+ to verify the inequivalence of the pairs in workflows $W5 \ldots W8$ was negligible. The heuristic approach was not effective in detecting the inequivalence of the TPC-DS workflows $W1 \ldots W4$. This limitation arises from the technique's reliance on identifying differences in the *final* projected columns, which remained the same across all versions of these workflows.



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

Fig. 15: Comparison between Veer and Veer+ for verifying inequivalent pairs with two edits. An "×" sign means the algorithm was not able to finish within an hour.

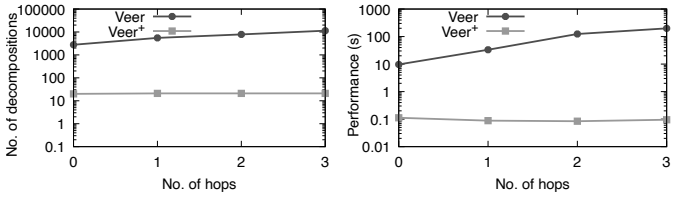
E. Effect of the Distance Between Edits

We evaluated the effect of the placement of changes. We used an equivalent version pair from $W2$ 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.

Figure 16a 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. Veer+'s number of explored decompositions remained at 21, thanks to the ranking optimization, as once a covering window included a neighboring window, it became larger than the other covering window and would be explored first until both windows merge.

Figure 16b shows the performance 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 the type of changed operators. 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 test the pair's equivalence, was negligible because the exploration stops after detecting the initial covering window as it is 'invalid'.



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

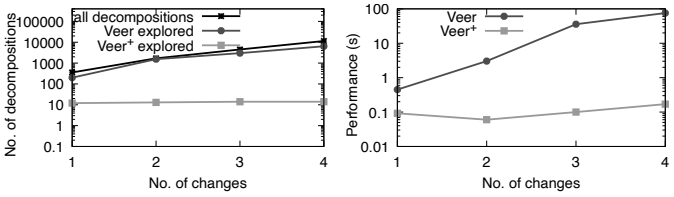
Fig. 16: Effect of the distance between changes (on $W2$)

F. Effect of the Number of Changes

We used an equivalent pair of W_1 to evaluate the effect of the number of changes on the number of decompositions and the time taken to verify a version pair.

Figure 17a 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 that was identified as equivalent. 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, thanks to the ranking optimization.

Figure 17b shows the baseline had a performance of 0.42 seconds when there was a single change, up to 75 seconds for four changes. Veer$^+$, on the other hand, maintained a sub-second performance with an average of 0.1 seconds.

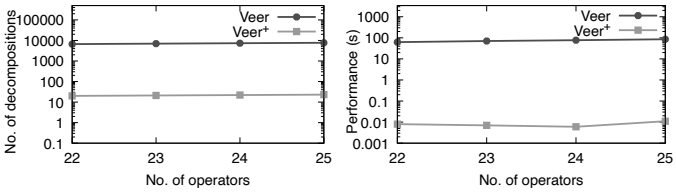


(a) # of explored decompositions. (b) Running time.

Fig. 17: Effect of the number of changes (on $W1$).

G. Effect of the Number of Operators

Varying the number of supported operators. Figure 18a shows the baseline explored $6,650$ and $7,700$ decompositions when there were 22 and 25 operators, respectively. 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. We observed that the performance of Veer was negatively affected (from a minute up to 1.4 minutes) due to the addition of possible decompositions from these operators' neighbors while the performance of Veer$^+$ remained in a sub-second as shown in Figure 18b.



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

Fig. 18: Effect of the number of operators (on $W2$).

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

H. Limitations

Veer could verify complex workflow version pairs that other verifiers were unable to verify. However, given the undecidability of the problem of determining the equivalence of two workflow versions, there are cases where Veer fails to verify due to the following reasons:

Determinism and context: Veer focuses on a small portion of the workflow pair (windows) and ignores the context. It treats the input to the small windows as any instance of input sources. In some cases, these windows may not be equivalent, but the entire pair is equivalent. Moreover, Veer assumes that the input sources are not changing and that the operator functions are deterministic across different versions.

Dependence on EV: Veer is a general-purpose framework that internally depends on existing EVs. When changes between versions are performed on operators such as UDF, then Veer would need to rely on an EV that can reason about the semantics of a UDF, which remains understudied. Moreover, striking a balance between maximizing the number of windows pushed to the EV for verification and ensuring thorough coverage remains a challenge when defining the EV's restrictions. We address these limitations in a followup work.

VIII. Related Works

Equivalence verification. With the recent advancement of developing proof assists and solvers [30], there have been new solutions [8], [10], [15]. UDP [15] and WeTune [16] use semirings to model the semantics of the pair and use a proof assist to prove the equivalence of two expressions. Equitas [10] and Spes [8] model the semantics of a pair into a FOL formula and push the formula to be solved by an SMT solver [20]. Other works also use an SMT solver to verify the equivalence of a pair of Spark jobs [21]. Our solution uses them as black boxes to verify the equivalence of a version pair.

Tracking workflow executions. There are tools that 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 [31], [32]. The second includes solutions to track results of different versions of data processing workflows [6], [33], [34].

Materialization reuse. Some solutions [35], [36] focus on identifying reuse opportunities by relying on finding an exact match of the workflow's DAG. On the other hand, some works [37], [38] reason the semantics of the query to identify reuse that are not limited to structural matching. However, these solutions are applicable to a specific class of functions, such as UDF [5], [39], [40], and do not generalize to finding reuse by finding equivalence of any pair of workflows.

IX. Conclusion

We presented "Veer," which leverages the similarity between two workflow versions to verify their equivalence. We presented a "window" concept to leverage the existing solutions for verifying the equivalence. 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 an experimental study to show the high efficiency and effectiveness of the solution on a real workload.

Acknowledgment

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

References

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

[2] A. Kumar, Z. Wang, S. Ni, and C. Li, "Amber: A debuggable dataflow system based on the actor model," *Proc. VLDB Endow.*, vol. 13, no. 5, pp. 740–753, 2020.

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

[4] B. Derakhshan, A. R. Mahdiraji, Z. Kaoudi, T. Rabl, and V. Markl, "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*, pp. 1962–1976, ACM, 2022.

[5] Z. Xu, G. T. Kakkar, J. Arulraj, and U. Ramachandran, "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* (Z. Ives, A. Bonifati, and A. E. Abbadi, eds.), pp. 602–616, ACM, 2022.

[6] S. Alsudais, "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* (Z. Bao and T. K. Sellis, eds.), vol. 3186 of *CEUR Workshop Proceedings*, CEUR-WS.org, 2022.

[7] S. Woodman, H. Hiden, P. Watson, and P. Missier, "Achieving reproducibility by combining provenance with service and workflow versioning," in *WORKS'11*, 2011.

[8] Q. Zhou, J. Arulraj, S. B. Navathe, W. Harris, and J. Wu, "SPES: A symbolic approach to proving query equivalence under bag semantics," pp. 2735–2748, 2022.

[9] A. Jindal, K. Karanasos, S. Rao, and H. Patel, "Selecting subexpressions to materialize at datacenter scale," *Proc. VLDB Endow.*, vol. 11, no. 7, pp. 800–812, 2018.

[10] Q. Zhou, J. Arulraj, S. B. Navathe, W. Harris, and D. Xu, "Automated verification of query equivalence using satisfiability modulo theories," *VLDB'19*, 2019.

[11] L. L. Perez and C. M. Jermaine, "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* (I. F. Cruz, E. Ferrari, Y. Tao, E. Bertino, and G. Trajcevski, eds.), pp. 520–531, IEEE Computer Society, 2014.

[12] K. Dursun, C. Binnig, U. Çetintemel, and T. Kraska, "Revisiting reuse in main memory database systems," in *SIGMOD'17*, 2017.

[13] S. Alsudais, A. Kumar, and C. Li, "Veer: Verifying equivalence of workflow versions in iterative data analytics," *arXiv preprint arXiv:2309.13762*, 2023.

[14] S. Abiteboul, R. Hull, and V. Vianu, *Foundations of Databases: The Logical Level*. USA: Addison-Wesley Longman Publishing Co., Inc., 1st ed., 1995.

[15] S. Chu, B. Murphy, J. Roesch, A. Cheung, and D. Suciu, "Axiomatic foundations and algorithms for deciding semantic equivalences of SQL queries," *VLDB'18*, 2018.

[16] Z. Wang, Z. Zhou, Y. Yang, H. Ding, G. Hu, D. Ding, C. Tang, H. Chen, and J. Li, "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*, pp. 94–107, ACM, 2022.

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

[18] "Knime workflows website."

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

[20] L. M. de Moura and N. S. Bjørner, "Z3: an efficient SMT solver," in *TACAS'08*, 2008.

[21] S. Grossman, S. Cohen, S. Itzhaky, N. Rinetzky, and M. Sagiv, "Verifying equivalence of spark programs," in *CAV'17*, 2017.

[22] S. Alsudais, A. Kumar, and C. Li, "Raven: Accelerating execution of iterative data analytics by reusing results of previous equivalent versions," in *Proceedings of the Workshop on Human-In-the-Loop Data Analytics, HILDA 2023, Seattle, WA, USA, 18 June 2023*, pp. 3:1–3:7, ACM, 2023.

[23] C. Borralleras, D. Larraz, E. Rodríguez-Carbonell, A. Oliveras, and A. Rubio, "Incomplete SMT techniques for solving non-linear formulas over the integers," *ACM Trans. Comput. Log.*, vol. 20, no. 4, pp. 25:1–25:36, 2019.

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

[25] S. Chu, C. Wang, K. Weitz, and A. Cheung, "Cosette: An automated prover for SQL," in *8th Biennial Conference on Innovative Data Systems Research, CIDR 2017, Chaminade, CA, USA, January 8-11, 2017, Online Proceedings*, www.cidrdb.org, 2017.

[26] K. Riesen, S. Emmenegger, and H. Bunke, "A novel software toolkit for graph edit distance computation," in *Graph-Based Representations in Pattern Recognition - 9th IAPR-TC-15 International Workshop, GbRPR 2013, Vienna, Austria, May 15-17, 2013. Proceedings* (W. G. Kropatsch, N. M. Artner, Y. Haxhimusa, and X. Jiang, eds.), vol. 7877 of *Lecture Notes in Computer Science*, pp. 142–151, Springer, 2013.

[27] J. LeFevre, J. Sankaranarayanan, H. Hacigümüs, J. Tatemura, N. Polyzotis, and M. J. Carey, "Opportunistic physical design for big data analytics," in *International Conference on Management of Data, SIGMOD 2014, Snowbird, UT, USA, June 22-27, 2014* (C. E. Dyreson, F. Li, and M. T. Özsu, eds.), pp. 851–862, ACM, 2014.

[28] "Imdb datasets website."

[29] "Twitter api v1.1."

[30] L. M. de Moura, S. Kong, J. Avigad, F. van Doorn, and J. von Raumer, "The lean theorem prover (system description)," in *Automated Deduction - CADE-25 - 25th International Conference on Automated Deduction, Berlin, Germany, August 1-7, 2015, Proceedings* (A. P. Felty and A. Middeldorp, eds.), vol. 9195 of *Lecture Notes in Computer Science*, pp. 378–388, Springer, 2015.

[31] A. Chen, A. Chow, A. Davidson, A. DCunha, A. Ghodsi, S. A. Hong, A. Konwinski, C. Mewald, S. Murching, T. Nykodym, P. Ogilvie, M. Parkhe, A. Singh, F. Xie, M. Zaharia, R. Zang, J. Zheng, and C. Zumar, "Developments in mlflow: A system to accelerate the machine learning lifecycle," in *DEEM@SIGMOD'20*, 2020.

[32] G. Gharibi, V. Walunj, R. Alanazi, S. Rella, and Y. Lee, "Automated management of deep learning experiments," in *DEEM@SIGMOD'19*, 2019.

[33] J. F. Pimentel, L. Murta, V. Braganholo, and J. Freire, "noworkflow: a tool for collecting, analyzing, and managing provenance from python scripts," *VLDB*, 2017.

[34] S. P. Callahan, J. Freire, E. Santos, C. E. Scheidegger, C. T. Silva, and H. T. Vo, "Managing the evolution of dataflows with vistrails," in *Proceedings of the 22nd International Conference on Data Engineering Workshops, ICDE 2006, 3-7 April 2006, Atlanta, GA, USA*, p. 71, IEEE Computer Society, 2006.

[35] F. Nagel, P. A. Boncz, and S. Viglas, "Recycling in pipelined query evaluation," in *ICDE'13*, 2013.

[36] J. Zhou, P. Larson, J. C. Freytag, and W. Lehner, "Efficient exploitation of similar subexpressions for query processing," in *SIGMOD'07*, 2007.

[37] A. Chaudhary, S. Zeuch, V. Markl, and J. Karimov, "Incremental stream query merging," in *Proceedings 26th International Conference on Extending Database Technology, EDBT 2023, Ioannina, Greece, March 28-31, 2023*, pp. 604–617, OpenProceedings.org, 2023.

[38] J. Kossmann, T. Papenbrock, and F. Naumann, "Data dependencies for query optimization: a survey," *VLDB J.*, vol. 31, no. 1, pp. 1–22, 2022.

[39] L. Ramjit, M. Interlandi, E. Wu, and R. Netravali, "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*, pp. 206–219, ACM, 2019.

[40] "Optimizing apache spark udfs website."