# Raven: Accelerating Execution of Iterative Data Analytics by Reusing Results of Previous Equivalent Versions

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

## ABSTRACT

Using GUI-based workflows for data analysis is an iterative process. During each iteration, an analyst makes changes to the workflow to improve it, generating a new version each time. The results produced by executing these versions are materialized to help users refer to them in the future. In many cases, a new version of the workflow, when submitted for execution, produces a result equivalent to that of a previous one. Identifying such equivalence can save computational resources and time by reusing the materialized result. One way to optimize the performance of executing a new version is to compare the current version with a previous one and test if they produce the same results using a workflow version equivalence *verifier*. As the number of versions grows, this testing can become a computational bottleneck. In this paper, we present Raven, an optimization framework to accelerate the execution of a new version request by detecting and reusing the results of previous equivalent versions with the help of a version equivalence verifier. Raven *ranks* and prunes the set of prior versions to quickly identify those that may produce an equivalent result to the version execution request. Additionally, when the verifier performs computation to verify the equivalence of a version pair, there may be a significant overlap with previously tested version pairs. Raven identifies and avoids such repeated computations by extending the verifier to reuse previous knowledge of equivalence tests. We evaluated the effectiveness of Raven compared to baselines on real workflows and datasets.

## CCS CONCEPTS

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

## KEYWORDS

workflow version control, iterative data analysis, semantic optimization, workflow equivalence verification

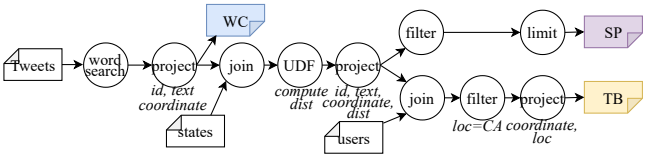
**ACM Reference Format:**
Sadeem Alsudais, Avinash Kumar, and Chen Li. 2023. Raven: Accelerating Execution of Iterative Data Analytics by Reusing Results of Previous Equivalent Versions. In *Workshop on Human-In-the-Loop Data Analytics (HILDA '23), June 18, 2023, Seattle, WA, USA.* ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/3597465.3605219

This work is licensed under a Creative Commons Attribution International 4.0 License.

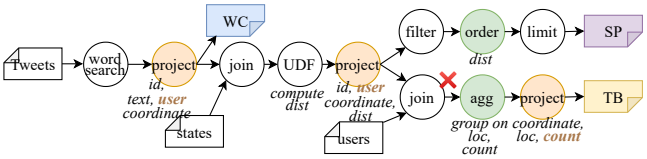
*HILDA '23, June 18, 2023, Seattle, WA, USA*
© 2023 Copyright held by the owner/author(s).
ACM ISBN 979-8-4007-0216-7/23/06.
https://doi.org/10.1145/3597465.3605219

## 1 INTRODUCTION

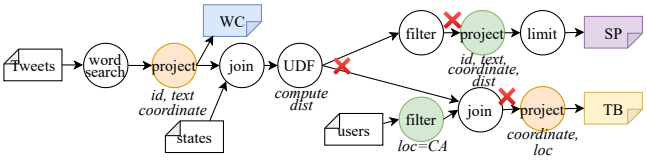
GUI-based big data processing workflow platforms are popular for efficiently processing and analyzing large volumes of data with user-friendly interfaces, making them accessible to individuals of varying technical expertise [16]. A dataflow is represented as a Directed Acyclic Graph (DAG), where each node corresponds to an operator that incorporates the processing logic, and the links represent the data flow between the operators. Operators without incoming edges retrieve data from various sources, such as datastores or files, while operators without outgoing edges serve as *sinks*, representing the final output of the task from its upstream operators.




**(a) Version 1: An initial construction of the workflow with a *word-cloud* result $WC$, a *scatterplot* result $SP$, and a *table* result $TB$.**




**(b) Version 2: A refined version of the workflow to get the count of users who posted tweets about the topic from different locations.**




**(c) Version 3: A refined version to optimize the performance by pushing the filter operator past the join operator.**

**Figure 1: Example of a workflow for analyzing tweets that discuss popular wildfires, and the workflow's evolution in three versions.** Orange operators are modified, green operators are added, and a red cross indicates a deleted operator.

When an analyst employs workflows for data analytics, she starts with a basic workflow and iteratively revises it based on the observed execution results as part of the iterative process of data analytics [8, 27]. She may edit the operators and links in the workflow during each iteration, producing a new *version* of the workflow. Figure 1 shows an overview of a workflow for analyzing tweets related to popular wildfires and the tweets' distance from the center of the wildfire. The workflow includes three sinks, each

producing different results based on the logic of its upstream operators. The example illustrates the workflow's evolution in three different versions.

**Motivation.** As an analyst iteratively refines a workflow, many versions can be created. For example, one deployment of Texera [19], a collaborative data processing system, recorded a total of 2,424 executions of different workflow versions for one workflow [3]. Tracking the versions of a workflow and the outcomes of their executions has gained interest recently [5, 26]. One observation in many applications is that versions of the same workflow frequently produce the same results [15, 29]. In other words, given an instance of input sources, the versions produce the same sink results. For instance, there is an overlap in 45% of the daily tasks performed by Microsoft's analytics clusters [15]. 27% of 9,486 workflows from Ant Financial to detect fraud transactions share common computation, and 6% of them is equivalent [29]. In the running example, the modifications applied to version 1b to transform it into version 1c resulted in each sink producing the same results as the corresponding sinks in version 1a.

The execution of a workflow can be time-consuming and resource intensive due to the large amounts of input data and the workflow's complex operators such as advanced machine learning techniques and User-defined functions (UDF) [16]. One way to save time and resources is to reuse the results of previously executed versions of the same workflow by identifying those that produce *equivalent* results. In particular, when a user submits a new version execution request, we want to compare the version with a prior executed version. If the two versions are equivalent, then the new version does not need to be executed, and we can reuse the materialized result of the prior one.

**Limitations of existing works.** To reuse previous results to answer a new execution request, a body of existing work [9, 15] relies on identifying the exact match between the workflow versions. One limitation of these works is that they cannot identify reuse opportunities from versions when their DAGs' have different structures, e.g., the workflow version 1a and version 1c in the running example are semantically equivalent, i.e., their sinks produce the same results, but their DAGs have different structures. Other works take a semantic approach by analyzing the workflows' predicates [22, 27] to identify redundant and overlapping tuples between multiple jobs. However, these works cannot identify the exact equivalence of the results of the two workflows. Thus we want to study the following:

Problem Statement. ***Given*** *a set of results from executions of previous versions of a workflow and an execution request of a new version,* ***find*** *a subset of prior versions, which include sinks that produce equivalent results to those in the execution request's version.*

**Our approach and challenges.** The problem of testing the equivalence of two queries has been studied for SQL [7], Spark programs [10] or workflow versions [23]. We can leverage these verifiers, and a naive solution to the "result reuse" problem stated above is to iteratively check every past version to see if it produces results equivalent to the new one by passing the pair to a verifier. While this approach is straightforward, it is not efficient when the number of versions increases, leading to many pairwise tests before finding an equivalent one. We want a framework that can rank and prune the set of previous versions based on their semantic equivalence or inequivalence compared to the new version's execution request. When the verifier checks to see if two workflow versions are equivalent, it does so by following an internal procedure. Since the tested pairs share similar structures, there may be a lot of computational overlap with previously-tested version pairs. To save computational resources and time, we want to identify and avoid such repeated computations. To address these challenges, we propose a novel framework called Raven, which accelerates the execution of a workflow version by detecting previously-stored semantically equivalent results from previous versions. We make the following **contributions** in this paper:

(1) We propose a framework and a novel technique to let the optimizer identify and reuse the materialized result of previous equivalent versions of a workflow using Veer (§3).
(2) We propose two approaches to ranking the versions and choosing those with a high rank to be tested for equivalence with a given execution request (§4.1).
(3) We extend Veer by adding optimization techniques that allow it to reuse computations (needed to do the verification) from historical equivalence tests (§4.2).
(4) We evaluate the execution speedup of Raven against a baseline on a real-world workload (§ 5).

## 1.1 Related Work

There is extensive research on reusing stored results to answer an execution request, as summarized in the following surveys [1, 6, 12].

**Exact expression matching.** Compared to general materialization reuse methods, exact pattern matching is a more specific and syntax-based approach, commonly used in systems with high workloads [15, 24, 28]. Raven differs as it employs an approach by identifying semantic equivalence and is not limited to exact DAG match.

**Reusing intermediate results.** Several works reuse intermediate results found in iterative pipelines [14, 17, 21]. Restore [9] caches map-reduce and intermediate jobs, while Recycler [20] uses a graph to recycle fine-grained partial query results. Nectar [11] caches subcomputations that are likely to be reused. Raven aims to identify previous equivalent versions with respect to the final results even when there are changes.

**Semantic reuse.** Prior works such as Eva [27], Acorn [22], and the work by LeFevre [18] proposed methods to semantically reuse previous results, using techniques such as UDF signatures and predicate overlap detection. These methods focus on detecting predicate equivalence and overlap rather than final result equivalence.

**Equivalence verification.** Some works verify the equivalence of SQL queries under certain assumptions [7, 30]. These solutions cannot reason about UDF semantics, making them unsuitable for detecting workflow version equivalence. Veer [23] addresses this limitation by verifying the equivalence of two workflow versions with UDFs, and Raven leverages it in its solution.

# 2 BACKGROUND

In this section, we give an overview of workflows and their edit operations, discuss equivalence verifiers, and show how a workflow

version equivalence verifier (Veer) uses these equivalence verifiers in its solution.

**Data processing workflows and their edit operations.** A workflow is a directed-acyclic graph (DAG) of operators, each with a computation function and properties. Source operators have no incoming links, while sink operators have no outgoing links and produce final results. Workflows may have multiple sources and sinks. Some of the sinks can have their results materialized as views.

A workflow undergoes many edits over time, resulting in different versions $[v_1, \ldots, v_n]$. The versions are created through a series of *edit operations*, including adding or deleting an operator or link, or modifying an operator's properties. These edit operations are combined to form a transformation that can be applied to a workflow version to create a new version. A *workflow edit mapping* ($\mathcal{M}$) aligns operators and links between different versions to produce a transformation from one version to the other. Unmapped operators and links are considered to be deleted or inserted accordingly.

**Equivalence verifier (EV).** An EV takes a pair of SQL queries and returns `True` when the pair produces the same result [7, 30] under a specific table semantics. Proving the equivalence of two SQL queries, in general, is undecidable [2], and an EV may require the pair to meet certain restrictions in order to test their equivalence.

**Workflow version equivalence verifier (Veer).** Proving the equivalence of two workflows can be challenging due to the semantic richness of their operators, which can include complex data processing tasks, such as UDFs or machine learning operations [8, 27]. Our recent study [23] introduces Veer, a workflow version equivalence verifier that leverages the changes between the pair to prove their equivalence using EVs as a black box.

Veer takes two workflow versions as input, a transformation that contains the edit operations converting one version to the other, and an EV. It uses the EV to verify the equivalence of the two versions by decomposing the pair into multiple portions called "windows," each of which includes local changes and satisfies the EV's restrictions. Each window is then provided to the EV to verify if the pair of portions in the window are equivalent. For simplicity, we refer to this step as "testing the equivalence of the window," as illustrated in Figure 2. In this way, Veer identifies which sinks in the two versions produce equivalent results. Next we give formal definitions of Veer's concepts.

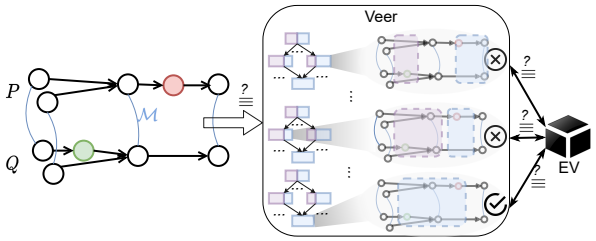



**Figure 2: Veer: a workflow version equivalence verifier.**

*Definition 2.1 (Window, covering window, and valid window [23]).* 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 *window*, denoted as $\omega$, 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 $\omega$ or both outside $\omega$. A *covering window*, denoted as $\omega_C$, is a window to cover a set of changes $C \subseteq \delta$. A window is *valid* w.r.t. an EV if it satisfies the EV's restrictions.

*Definition 2.2 (Equivalence of the two sub-DAGs in a window [23]).* The two sub-DAGs $\omega(P)$ and $\omega(Q)$ of a window $\omega$ are *equivalent*, denoted as "$\omega(P) \equiv \omega(Q)$," if they are equivalent as two stand-alone DAG's without considering the constraints from their upstream operators.

*Definition 2.3 (Decomposition [23]).* For a version pair $P$ and $Q$ with a set of edit operations $\delta = \{c_1 \ldots c_n\}$ from $P$ to $Q$, a *decomposition*, denoted as $\theta$, is a set of windows $\{\omega_1, \ldots, \omega_m\}$ such that:

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

## 3 RAVEN: OVERVIEW

Figure 3 presents an overview of the steps involved in the optimization lifecycle to accelerate the execution of a workflow version DAG by Raven. Given an execution request for the workflow version $v_n$ (called the "current version"), the optimizer searches for a "prior version" $v_p \in [v_1, \ldots, v_{n-1}]$, which has sinks equivalent to the corresponding sinks in $v_n$. It takes the following steps.

**Step 1. Ranking the prior versions.** Raven ranks the previous versions in the order of their likelihood of being equivalent to the current one. To do this, we propose two approaches. One uses the edit mapping between the pairwise of $v_n$ and every other prior version $v_p$. Another approach is to organize the versions of a workflow in a hierarchy and model the versions in a lightweight representation to speed up the traversal search of the prior versions. In this way, we avoid testing the equivalence with every past version. Raven chooses a prior version with the highest rank to test its equivalence with the current one.

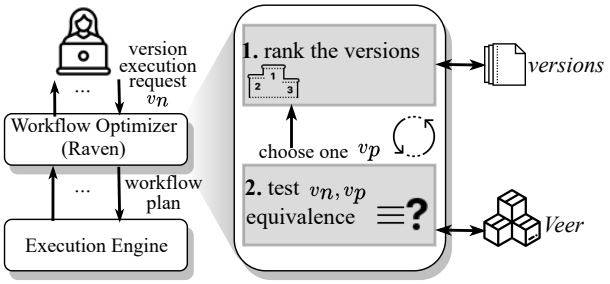



**Figure 3: Overview of Raven's framework.**

**Step 2: Testing the equivalence of the version pair.** Raven uses Veer to test the equivalence of the pair of versions. Veer adopts a procedure based on an edit mapping between the pair to return a set of equivalent sinks. When we invoke Veer multiple times to do the equivalence testing by passing multiple pairwise versions with a lot of commonalities in their structural DAG, some of the steps can be redundant. Raven extends Veer to avoid repeated computation by performing memoization and checkpointing.

We repeat the above steps till all the sinks in the current workflow version have been answered using prior versions, or there are no more previous versions left to check.

## 4 REUSE-AWARE OPTIMIZATION

In this section, we propose two ideas to optimize the performance of workflow version execution in Raven. Firstly, we discuss how Raven ranks versions to select the one with the highest score for equivalence testing with the current version (Section 4.1). Secondly, we highlight the issue where, even if the versions are ranked in the correct order, not all equivalent sinks may be found in the first chosen version, leading to additional pairs being pushed to Veer. This can result in repeated computations in Veer's internals. To address this, Raven extends Veer to reuse previous computations from other evaluations of previous pairs (Section 4.2).

### 4.1 Ranking Versions for Equivalence Check

*4.1.1 Ranking by using an edit mapping.* A naive way to rank the versions is to choose those with the fewest edits compared to the current version. The intuition is that with a smaller number of edits, Veer needs to do fewer decompositions, and we can get the answer faster. To find the edits for each pair of the current version and a prior one, we iterate over every prior version DAG and pass the pair to a Graph Edit Distance (GED) algorithm, which returns the set of edit operations needed to transform one graph to the other [4]. We then rank the versions based on the number of differences, giving a higher score to those with fewer edits.

The following example shows that using the minimum edit distance for ranking may not necessarily rank the equivalent version higher than an inequivalent one,

*Example 4.1.* Consider the following three versions:

$$v_1 = \{Project(all) \rightarrow Filter(age > 24) \rightarrow Aggr(\text{count by age})\}.$$

$$v_2 = \{Project(all) \rightarrow Aggr(\text{count by age})\}.$$

$$v_3 = \{Filter(age > 24) \rightarrow Project(all) \rightarrow Aggr(\text{count by age})\}.$$

Consider a mapping for transforming $v_1$ to $v_3$, which involves substituting Project in $v_1$ with Filter in $v_3$ and substituting Filter in $v_1$ with Project in $v_3$ yielding two edits. The mapping to transform $v_2$ to $v_3$ is done by adding a Filter operator, yielding a single edit operation. Given the ranking proposed above, the algorithm chooses $v_2$ as it has fewer differences with $v_3$ i.e., 1 compared to the differences between the pair $(v_1, v_3)$ i.e., 2. However, $v_2 \not\equiv v_3$ while $v_1 \equiv v_3$.

While this approach helps us quickly get an answer if a prior and the current version pair are equivalent or not, running the GED algorithm from scratch every time for every version pair can be computationally expensive due to its $NP$-hard complexity [4].

*4.1.2 Ranking by using a view representation.* The method presented earlier focuses on ranking versions but does not consider the actual stored results of sinks, i.e., views. To efficiently identify reusable views across different versions, we need a lightweight fingerprint representation that models the semantics of the sinks' results. We organize the sinks in a hierarchy to facilitate traversal for finding reusable views and avoid inspecting versions that include sinks that are guaranteed to be not equivalent to the execution request. We model the result of a sink as a tuple $(T, \vec{S}, \vec{O})$, where $T$ is a First-Order-Logic (FOL) formula indicating the existence of a tuple in the table, and $\vec{S}$ and $\vec{O}$ are the lists of fields in the table and the fields on which the table is ordered, respectively. By using $(\vec{S}, \vec{O})$, we can quickly identify and eliminate views that are not equivalent to the sinks in the execution request, without considering the complexity of determining a tuple's existence and its cardinality [7, 18] represented by $T$ in this paper.

**Representation construction.** To construct the view representation, we follow the same techniques in existing literature [7, 30] by using predefined transformations for each operator. Operators inherit the representation from their upstream/parent operator and update the fields based on their internal logic.

We leverage the knowledge of the changes made to the previous version and build the representation incrementally by propagating the difference starting from a changed operator closest to the source. This requires tracking and storing transformation results on every operator, not just in the sink. We can choose between constructing the representation from scratch or propagating the delta considering factors such as how far the changes are from the sinks and the size of the workflow.

**View organization in a V2-structure.** We organize the sinks in the versions in a hierarchy "V2", which stands for "versioned views". A node includes the view representation and includes physical pointers to where the sinks that have the same representation (not necessarily equivalent) are grouped. An edge between two nodes means the result of the child node is a subset of the result of the parent node (when ignoring the $T$ field). A subset result can be detected by running two tests, one for each field in the representation, as discussed below.

*Definition 4.2 (V2 Node Subsumption Test).* Given a node $v$ and a child node $u$, we say $u$ is a proper subset of node $v$, denoted as" $u \subset v$," when $\vec{O}_v$ is a subset of $\vec{O}_u$ and $\vec{S}_u$ is a subset of $\vec{S}_v$.

The intuition is that the set of projected columns in $v$ includes all of the elements in the set of projected columns in $u$, and the ordering fields in $v$ are more general than in $u$. The structure may have multiple root nodes. Figure 4 shows a sample V2-structure to organize the sinks in the running example. Each node has a physical pointer to the saved results of the sinks in this node.

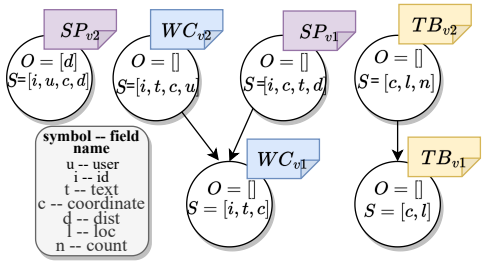



**Figure 4: A sample V2-structure to organize the saved results of sinks from the first two versions in the running example.**

**V2-structure traversal and maintenance.** We use the task of finding an equivalent view for the word cloud sink in $v_3$ of the running example to explain the traversal and maintenance of the hierarchy. Given a new workflow execution request $v_3$, we first construct the view representation of the word cloud sink, $\vec{S} = [i, t, c]$

and $\vec{O} = []$. After that, we traverse the hierarchy in a depth-first-search manner. Starting from a root node, we simultaneously run two tests, one to ask if the list $\vec{S}$ in the node *contains* the one in the current version, and the other is to ask if the list $\vec{O}$ in the word cloud sink *contains* the one in the node. Both tests must return `True`; otherwise, we stop traversing the children of that node.

In this example, one of the tests on the first node in Figure 4 returns `False`. Therefore, we continue the search by inspecting a sibling node. When both tests return `True`, we further test if both $(\vec{S}, \vec{O})$ in the node are the same as those in the current version. If the two representations are not the same, we expand the search to test the child nodes. In this example, both tests return `True` when testing the second node and their representations are not the same, so we consider the child nodes and follow the same procedure. In this example, the test on the child node shows that the two representations are the same. If the two representations are the same, we retrieve the physical pointers to the versions the node points to. We iterate through every version on the list and push it to Veer with the current version to test their equivalence until we find one that includes sinks equivalent to the current version. Additionally, we add a new pointer to point to the current version.

When all of the sibling nodes are traversed and none of them are expanded to test their child nodes, we insert a new node containing the current version's sink representation and a physical pointer to its result. We do the same for every sink in the version. The benefit of this lightweight representation resulted in pushing only one pair to Veer, instead of iterating over every past version.

## 4.2 Reusing Past Equivalence Tests in Windows

Recall that Veer breaks the versions into smaller windows and checks the equivalence of each window by passing it to an EV. Veer reported that the time taken to verify a window by the EV takes on average 87% of the total time [23] Veer takes. Some of these windows may have been checked in previous iterations when testing other pairs. Therefore, memoizing the results of previous windows' equivalence checks can help improve the performance.

In this section, we explore two methods for optimizing Veer's performance by extending it to reuse information about previously tested windows. The first method groups windows into *equivalence classes* and the second involves "chopping" a version pair into a smaller portion and excluding the chopped portion in the decomposition process based on the knowledge that it has already been verified in previous computations that it is equivalent.

*4.2.1 Grouping windows in equivalence classes.* We explain the details of extending Veer to group windows into equivalence classes. An equivalence class is a set of elements, each of which is a sub-DAG from a window that have been proven equivalent by Veer. When an analyst submits an execution request for a second version, Raven pushes the two versions to Veer. For each window, Veer checks if the sub-DAGs were seen before by checking a map, where the key is the sub-DAG and the value is the sub-DAG's equivalence class. The map check yields the following possible cases.

**1. None of the two sub-DAGs were tested before:** Veer pushes the window to the EV to test their equivalence. If the EV proves the two sub-DAGs in the window are equivalent, then Veer uses this knowledge to group them in the same equivalence class. The newly created equivalence class is assigned an identifying label. On the other hand, if the EV proves the two sub-DAGs are not equivalent, then each sub-DAG will be assigned a new equivalence class label.

**2. One sub-DAG only was tested before:** Veer pushes the pair to the EV. For the unseen sub-DAG, we assign it the same equivalence class as the other one if the EV proves the pair is equivalent. Otherwise, we give it a new equivalence class.

**3. Both sub-DAGs were tested before:** Veer checks if the pair is in the same equivalence class by checking the value of their equivalence class using their key. If so, it marks their equivalence, and there is no need to push the pair to the EV. Otherwise, every sub-DAG is in a different equivalence class. Then we check a *memoization matrix* as explained in Figure 6. If the two equivalence classes were checked before, it means they are not equivalent. Otherwise, we push the pair to the EV. If the EV says they are equivalent, we merge the two classes and update the sub-DAGs' pointers in the map to point to the newly merged class.

Finally, for the cases where a sub-DAG was never seen before, we insert a new entry in the map with the sub-DAG as the key and the value being a pointer to the sub-DAG's equivalence class. Figure 5 illustrates an example of three equivalent windows from four different versions. When modifying Veer to use equivalence classes, it only pushes the first two windows but not the third, thus it can save computation.

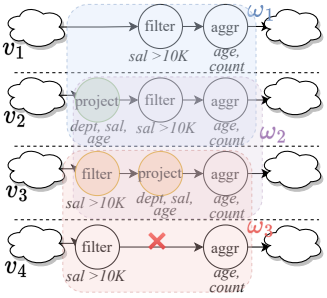



**Figure 5: Example to show three different windows belonging to the same equivalence class.**

| EC | 1 | 2 | 3 | ... |
|---|---|---|---|---|
| 1 | 1 | 0 | 0 | |
| 2 | 0 | 1 | 0 | |
| 3 | 0 | 0 | 1 | ... |

**Figure 6: A sample 2-D matrix for storing the equivalence tests between a pair of equivalence classes. A cell initially is 0 and is changed to 1 when the two classes are tested.**

*4.2.2 Checkpointing previous decompositions.* While the previous discussion solved the problem of avoiding repeated checks on the EV, we still need to do the decomposition from scratch. This repeated computation can be avoided, as shown in the example in Figure 7. Suppose we test the equivalence of the first two versions, and we know the two sub-DAGs in a window $\omega_1$ were equivalent. Suppose Raven ranks $v_1$ higher and selects this version first to test its equivalence to the current request $v_3$. Instead of computing the edit distance between the two versions $(v_1, v_3)$ from scratch, we can exploit the sequence of deltas recorded by the analysts when performing the edits. Knowing the accumulative edits allows us to identify the fact that the portion until the end of the window was proven equivalent in a previous test. Thus, we extend Veer to let it checkpoint and cut the portion that was tested before and only perform the decomposition on the parts after the checkpoint.

# 5 PRELIMINARY EXPERIMENTS

In this section, we report our experimental results of evaluating the effectiveness of Raven.

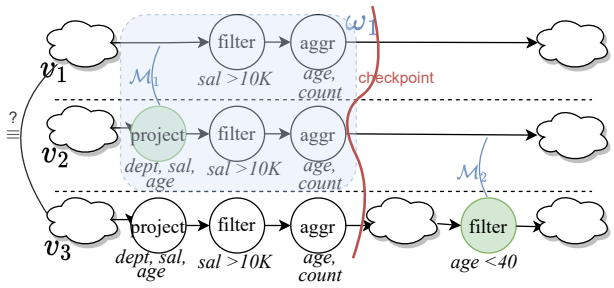



**Figure 7: Example to show reusing the knowledge of a previous equivalence test to place a marker and ignore a portion of the pair when performing a new equivalence test.**

## 5.1 Experimental Setup

**Real workload.** We analyzed a total of 179 real-world pipelines from a deployment of Texera [19]. Among the workflows, 81% had deterministic sources and operators, and we focused on these workflows. Among these workflows, 8% consisted of 8 operators, and another 8% had 12 operators. 76% of the workflows contained a UDF operator. Additionally, 33% of the workflows consisted of 3 different versions, while 19% had 35 versions. 58% of the versions had a single edit, while 22% had two edits. We also observed that the UDF operator was changed in 17% of the cases. From these workflows, we selected four as a representative subset excluding those with non-deterministic data sources. We created similar workflows, which are presented as $W1 \ldots W4$ in Table 1. We used IMDB [13] ($\approx 3GB$) and Twitter [25] ($\approx 0.5GB$) datasets. All versions included UDF operators. The average time it took to execute a version without reuse is 1.9 minutes.

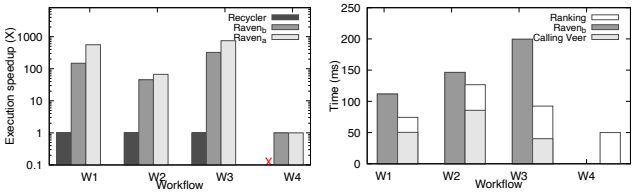
**Table 1: Workloads used in the experiments.**

| Work flow# | Description | # of operators | # of sinks | # of versions | % of equivalent sinks |
|---|---|---|---|---|---|
| $W1$ | IMDB ratio of non-original to original movie titles | 13 | 3 | 3 | 55 |
| $W2$ | IMDB all movies of directors with certain criteria | 26 | 3 | 3 | 55 |
| $W3$ | Tobacco Twitter analysis | 18 | 1 | 5 | 60 |
| $W4$ | Wildfire Twitter analysis | 12 | 3 | 12 | 16 |

**Implementation.** We evaluated our solution against the Recycler [20] baseline, which compares a workflow query DAG with previously executed workflow DAGs by examining their structures for equality. We extended Recycler to be able to match a few non-relational operators, whose semantics can be abstracted to simple APIs, such as $\langle A, F, K \rangle$ [18], where "A is the set of attributes, F is the set of filters previously applied to the input, and K is the current grouping of the input, which captures the keys of the data." We implemented a basic Raven, denoted $\text{Raven}_\text{b}$, which is a basic approach that iterates over past versions without ranking them and uses a verifier without enabling reusing previous tests. We implemented $\text{Raven}_\text{a}$, which is advanced Raven and included ranking past versions and reusing previous equivalence tests. We implemented the Veer [23] verifier and used Equitas [30] as its EV. We implemented the baseline and Raven using Java8 and Scala in Texera [19]. The system ran on a single node of a MacBook Pro running the MacOS Monterey operating system with a 2.2GHz Intel Core i7 CPU, 16GB DDR3 RAM, and a 256GB SSD.

## 5.2 Performance of Identifying Reuse and Execution Speedup

Figure 8 shows the results of evaluating the effectiveness of Raven in identifying semantic equivalence of workflows with UDF compared to the baseline. Recycler successfully identified 25% of the equivalent cases, while both $\text{Raven}_\text{b}$ and $\text{Raven}_\text{a}$ successfully identified 60% of the equivalent cases. Recycler failed to rewrite any of the workflow versions to reuse the identified equivalent results, yielding a speedup of 1. On the other hand, $\text{Raven}_\text{b}$ and $\text{Raven}_\text{a}$ were able to rewrite the workflows to reuse the results for 40% of the equivalent sinks, yielding a speedup of up to 322 using $\text{Raven}_\text{b}$ and 747 using $\text{Raven}_\text{a}$ for $W3$. The inability to rewrite a workflow version to reuse the identified equivalent sinks, in some cases, is due to the following: the workflow version DAG may include a sink that is not identified as equivalent, and its output depends on executing all of the operators in the DAG. To overcome this limitation, storing intermediate results could be a potential solution. Overall, $\text{Raven}_\text{a}$ outperformed $\text{Raven}_\text{b}$ by achieving a higher speedup, thanks to the utilization of ranking and reusing tests of other windows by grouping them in equivalence classes. None of the three approaches could reason about the semantics of $W4$ because $W4$ involved changes made to an ML model that were not supported by the approaches.




**(a) Execution speedup.** **(b) Time taken to identify reuse.**

**Figure 8: Effectiveness of** Raven**.** An "X" indicates the workflow was not supported by the solution.

Figure 8b shows the overhead of the three approaches. The time it took Recycler to match a DAG with previous DAGs was negligible due to the small size of historically seen queries, so we do not report its overhead in Figure 8b. $\text{Raven}_\text{b}$ and $\text{Raven}_\text{a}$ had more overhead than Recycler because they needed to invoke Veer multiple times. The overhead of $\text{Raven}_\text{a}$ is less than $\text{Raven}_\text{b}$ because it used the equivalence class concept and the ranking approach to optimize and reduce the time spent on Veer.

## 6 CONCLUSION

In this paper, we proposed Raven, a novel optimization technique that uses stored results from previously executed versions to answer a given version execution request after testing their equivalence. We showed how Raven uses an equivalence verifier in its modules. We proposed ranking the versions and utilizing previously conducted equivalence tests on workflow versions or portions of the versions to minimize redundant computations. We empirically evaluated the effectiveness of Raven, which achieved up to 747 times speedup, compared to other baselines.

## ACKNOWLEDGMENTS

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

# REFERENCES


[1] Serge Abiteboul and Oliver M. Duschka. 1998. Complexity of Answering Queries Using Materialized Views. In *Proceedings of the Seventeenth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems, June 1-3, 1998, Seattle, Washington, USA*, Alberto O. Mendelzon and Jan Paredaens (Eds.). ACM Press, 254–263. https://doi.org/10.1145/275487.275516

[2] Serge Abiteboul, Richard Hull, and Victor Vianu. 1995. *Foundations of Databases: The Logical Level* (1st ed.). Addison-Wesley Longman Publishing Co., Inc., USA.

[3] 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

[4] David B. Blumenthal, Nicolas Boria, Johann Gamper, Sébastien Bougleux, and Luc Brun. 2020. Comparing heuristics for graph edit distance computation. *VLDB J.* 29, 1 (2020), 419–458. https://doi.org/10.1007/s00778-019-00544-1

[5] 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*.

[6] Rada Chirkova, Chen Li, and Jia Li. 2006. Answering queries using materialized views with minimum size. *VLDB J.* 15, 3 (2006), 191–210.

[7] 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).

[8] 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

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

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

[11] Pradeep Kumar Gunda, Lenin Ravindranath, Chandramohan A. Thekkath, Yuan Yu, and Li Zhuang. 2010. Nectar: Automatic Management of Data and Computation in Datacenters. In *9th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2010, October 4-6, 2010, Vancouver, BC, Canada, Proceedings*, Remzi H. Arpaci-Dusseau and Brad Chen (Eds.). USENIX Association, 75–88. http://www.usenix.org/events/osdi10/tech/full_papers/Gunda.pdf

[12] 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

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

[14] Milena Ivanova, Martin L. Kersten, Niels J. Nes, and Romulo Goncalves. 2009. An architecture for recycling intermediates in a column-store. In *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2009, Providence, Rhode Island, USA, June 29 - July 2, 2009*, Ugur Çetintemel, Stanley B. Zdonik, Donald Kossmann, and Nesime Tatbul (Eds.). ACM, 309–320. https://doi.org/10.1145/1559845.1559879

[15] 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

[16] 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

[17] Mayuresh Kunjir, Brandon Fain, Kamesh Munagala, and Shivnath Babu. 2017. ROBUS: Fair Cache Allocation for Data-parallel Workloads. In *Proceedings of the 2017 ACM International Conference on Management of Data, SIGMOD Conference 2017, Chicago, IL, USA, May 14-19, 2017*, Semih Salihoglu, Wenchao Zhou, Rada Chirkova, Jun Yang, and Dan Suciu (Eds.). ACM, 219–234. https://doi.org/10.1145/3035918.3064018

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

[19] Xiaozhen Liu, Zuozhi Wang, Shengquan Ni, Sadeem Alsudais, Yicong Huang, Avinash Kumar, and Chen Li. 2022. Demonstration of Collaborative and Interactive Workflow-Based Data Analytics in Texera. *Proc. VLDB Endow.* 15, 12 (2022), 3738–3741. https://www.vldb.org/pvldb/vol15/p3738-liu.pdf

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

[21] 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

[22] 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

[23] Veer: Verifying Equivalence of Workflow Versions in Iterative Data Analytics (Technical Report). https://sadeemsaleh.github.io/Veer__Extended_.pdf.

[24] Yasin N. Silva, Per-Åke Larson, and Jingren Zhou. 2012. Exploiting Common Subexpressions for Cloud Query Processing. In *IEEE 28th International Conference on Data Engineering (ICDE 2012), Washington, DC, USA (Arlington, Virginia), 1-5 April, 2012*, Anastasios Kementsietsidis and Marcos Antonio Vaz Salles (Eds.). IEEE Computer Society, 1337–1348. https://doi.org/10.1109/ICDE.2012.106

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

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

[27] 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

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

[29] 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

[30] 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).