Drove: Tracking Execution Results of Workflows on Large Data

Sadeem Alsudais

Supervised by Chen Li
Department of Computer Science, UC Irvine, CA 92697, USA

Abstract
Data analytics using workflows is an iterative process, in which an analyst makes many iterations of changes, such as additions, deletions, and alterations of operators and their links. In many cases, the analyst wants to compare these workflow versions and their execution results to help in deciding the next iterations of changes. Moreover, the analyst needs to know which versions produced undesired results to avoid refining the workflow in those versions. To enable the analyst to get an overview of the workflow versions and their results, we introduce Drove, a framework that manages the end-to-end lifecycle of constructing, refining, and executing workflows on large data sets and provides a dashboard to monitor these execution results. In many cases, the result of an execution is the same as the result of a prior execution. Identifying such equivalence between the execution results of different workflow versions is important for two reasons. First, it can help us reduce the storage cost of the results by storing equivalent results only once. Second, stored results of early executions can be reused for future executions with the same results. Existing tools that track such executions are geared towards small-scale data and lack the means to reuse existing results in future executions. In Drove, we reason the semantic equivalence of the workflow versions to reduce the storage space and reuse the materialized results.

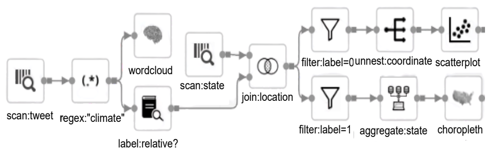
Keywords
workflow version control, workflow reproduciblity, semantic workflow equivalence verification

1. Introduction

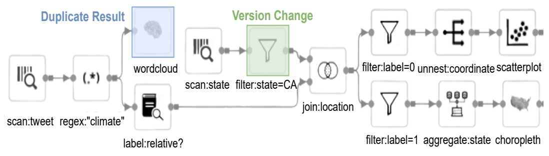
Data-processing workflows are extensively used by analysts to extract and analyze data over large volumes. Texera is an open source system we have been developing in the past years that provides a GUI-based interface for users to construct a workflow as a DAG of operators, refine and fine-tune the workflow, execute it, and examine the final results [1]. The users may perform multiple iterations of refinement, execution, and examination before producing the final version of the workflow [2, 3]. A refinement of the workflow creates a new version. Tracking different versions of a workflow and its produced results is a growing area of interest [4, 5, 6, 7, 8, 9]. Due to the iterative process in data analytics, one would be interested in looking at the past execution results to get answers for the following questions.

Q1. *Which workflow versions generated these results?*
Q2. *How did the differences between two versions affect their results?*

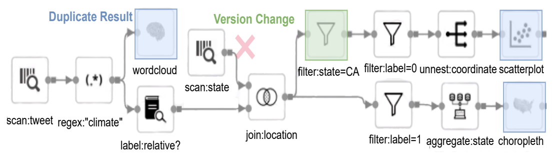
Motivation. Figure 1 illustrates an example of an analysis workflow that evolved into three versions. In the example, a data analyst is interested in the tweets whose



(a) **Version a: initial construction.**



(b) **Version b: after adding a filter operator (highlighted in green). The operator highlighted in blue has the same results of the corresponding operator in version a.**



(c) **Version c: after deleting the filter operator and adding it after the join operator. The two operators highlighted in blue have the same results of the corresponding operators in version a and version b.**

Figure 1: Multiple versions of a workflow for tweet analysis and their duplicate results of different executions.

Proceedings of the VLDB 2022 PhD Workshop, September 5, 2022. Sydney, Australia.
salsudai@uci.edu (S. Alsudais)
0000-0003-3928-690X (S. Alsudais)

content contains the keyword `climate`. In the first version shown in Figure 1a, she wants to look at the most discussed topics in a "wordcloud" visualization. She also uses an operator to further label the tweets as related to climate or not, using labels "related" and "unrelated". She wants to visualize the "related" tweets as a choropleth map that aggregates the tweet count by each state. She wants to visualize those "unrelated" tweets as a scatterplot by their location. After running this version, she notices that the most associated topic is `wildfire` from the wordcloud operator. Since wildfires are common in California, she decides to look at the spatial distribution of the tweets in this state. Hence, she refines the workflow to version b by adding a filter in the California region, as shown in Figure 1b. After she runs this version, she notices that adding the region filter caused the choropleth map to highlight only California. So she decides to do the filter for the scatterplot result only, and generates version c in Figure 1c. Now, she wants to compare the scatterplots of version a and version c to examine the density of "unrelated" tweets in California versus the entire US.

This example shows the importance of providing a dashboard for managing the different workflow versions and their execution results. In this work, we seek to provide such a dashboard to guide the analytics tasks.

Our solution. One way is to store each workflow version and its execution results [2, 3]. Its main limitation is that there can be many versions and executions of a workflow, and these results can be large. Thus this approach can consume a lot of storage space [10]. For example, in one deployment of Texera, it recorded 2,039 executions from 91 different workflows in 75 days. Storing all these execution results required a lot of space.

To address this challenge, we want to develop a solution that leverages the fact many of these results can be equivalent due to the iterative analytic process [3, 6], as illustrated in Figure 1. We assume the input relations between the workflow versions to be the same and determinism of the operators. We present Drove, a holistic approach to managing the end-to-end lifecycle of orchestrating, refining, and executing workflows and examining their corresponding results. It is developed in Texera to provide the means for the user to conduct the analytics and efficiently store and *reuse* the versions and results.

Related work. Existing solutions for workflows [11, 10] rely on identifying the exact match of the entire DAG or a sub-DAG of a workflow to reuse materialized results. These solutions cannot solve the case where two workflows are semantically equivalent but have different structures. The solution in [12] verifies the semantic equivalence of two Spark workflow jobs, and it supports a limited number of operators such as aggregation. It cannot verify the equivalence of the workflows in Figure 1. Another large body of work is about checking the semantic equivalence of two SQL queries, such as UDP [13], Equitas [14], and Spes [15]. One may want to solve our problem by treating a workflow as a SQL query, possibly with UDF functions, then using these solutions. Unfortunately, these solutions have certain restrictions on the type of supported operators, such as relational operators and a restrictive class of user-defined function (UDF) operators. They cannot support those operators that are common in workflows, such as labeling and unnest in the running example.

Our goal is to develop Drove to overcome these limitations. In particular, it should find semantic equivalence of workflows even if they have different structures, and support a variety of operators including relational operators and other types such as UDFs.

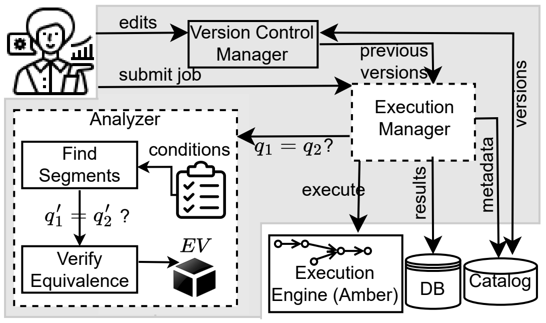


Figure 2: Drove's Tracking Executions Modules.

2. Drove **Overview**

Figure 2 depicts an overview of Drove. A user formulates a data-processing workflow through the UI of Texera. A workflow is a directed-acyclic graph (DAG) $G = (V, E)$, where vertices are operators and edges represent the direction of the data flow. A workflow can have multiple *sink* operators, each of which produces its own results. For each sink operator, we can view the sub-DAG consisting of its ancestors as a query that produces the results in this sink. Figure 3 shows the three sub-DAGs corresponding to the three sink operators in the running example (version a).

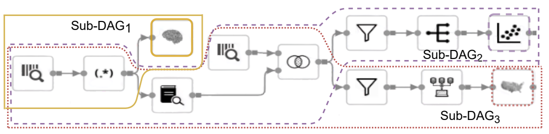


Figure 3: Sink operators and their corresponding sub-DAGs.

Drove uses a catalog to store the following information: (1) workflows; (2) their versions and the differences between two adjacent versions; (3) metadata about workflow executions, such as version, the start and end times, states, etc. We also store the sink-operator results of each execution in a database. To reduce the storage space, the system allows users to delete the results of some of the executions. The Version Control Manager is in charge of the workflow versions and their creation, storage, and retrieval. When a workflow with a version v_n is modified, a new version is created that includes those changes, i.e., $v_{n+1} = v_n + \Delta$. The changes Δ can be any combination of DAG operations such as addition of a new operator, deletion of an existing operator, substitution of an operator, property editing of an operator, and addition or deletion of a link. For the remaining of this discussion, when we refer to a version, it is a version that has been executed, because we are interested in their results. For each executed version, we keep at most one execution result. In the case where a version is executed multiple times, we only store the result of the first execution, and reuse for the later executions of the same version. A workflow submitted by the user is executed by the backend engine called Amber [16]. The Analyzer checks the semantic equivalence of DAGs. More details about those modules will be explained shortly.

2.1. Answering Workflows Using Materialized Results

When the user submits a version of the workflow, the Execution Manager tries to see if one of the previous execution results can be used to answer the workflow. For instance, when the user submits the version v_c in the running example, we try to find any reusable results from the executions of v_a and v_b. In particular, consider the wordcloud sink w_c operator in v_c and its corresponding sink operator w_b in v_b. Notice that the sub-DAG of w_c and the sub-DAG of w_b have the same structure. Such structural similarities can be identified using existing techniques [11]. In this case, we can use the results of w_b as the result of w_c.

Now consider the scatterplot sink s_c operator in v_c and its corresponding sink operator s_b in v_b. The sub-DAG of s_c and the sub-DAG of s_b have different structures. We push the pair of these two sub-DAGs to the Analyzer to verify their equivalence. Based on the positive answer from the Analyzer, the Execution Manager decides to reuse the results of s_b to answer s_c.

Next we consider the choropleth sink operator h_c in v_c and its corresponding sink operator h_b in v_b. The sub-DAG of h_c and the sub-DAG of h_b have different structures. We push the pair of these two sub-DAGs to the Analyzer to verify their equivalence. Since the answer from the Analyzer is negative, the Execution Manager cannot reuse the results of h_b to answer h_c. In this case, it looks for an earlier version, v_a. It compares the sub-DAG of h_a with its corresponding one from v_a. The Analyzer gives a positive answer this time, so we can reuse the result of h_a to answer h_c.

In general, for every sink in the execution request of a workflow, we do the following. For each previous version and the corresponding sink operator, the Execution Manager first identifies if the structure of the sub-DAGs are similar, otherwise it contacts the Analyzer to verify their semantic equivalence. This process terminates when the Analyzer confirms an equivalence between the current sink and the sink of one of the earlier versions.

Open Problems. The number of versions for a single workflow can be large. For instance, in a Texera production system, some workflows can have more than 80 executed versions. Checking the semantic equivalence between a sink operator with those of all the previous versions can have a high overhead. One interesting question that we are exploring is deciding when to stop comparing the version of the execution request with prior ones instead of comparing with all the previous versions without finding any positive semantic equivalent match. We plan to devise an objective function to decide on the fly when to stop the comparison considering a few factors, such as the degree of differences between the versions, the size of the processed data, the expense of the execution job and other indicators. Another direction is to avoid unnecessary equivalence verification by identifying the structural similarities of the sub-DAGs. For example, before calling the Analyzer to verify the equivalence of the sub-DAGs of h_c and h_b, we first verify the structural similarity of the sub-DAG of h_c with an earlier sub-DAG that has the same structure, i.e. h_a. To avoid storing all the different structures of all workflow versions, we only keep the structure of the sub-DAG that created the materialized result first.

2.2. Verifying equivalence between two DAGs

When the Analyzer receives a pair of DAGs, it needs to verify their semantic equivalence. We view each DAG a query, so essentially we want to check the equivalence of two queries. Notice that checking equivalence of two queries is undecidable in general [17]. In the literature there are solutions for queries with certain constraints, such as UDP [13], Equitas [14], and Spes [15]. Table 1 describes the conditions that need to be satisfied to use Equitas and Spes to verify set equivalence. In our system we want to support different kinds of semantics, such as set semantic, bag semantic, and list semantic, depending on the needs of the user.

The Analyzer incorporates one of these solutions as a module called "Equivalence Verifier" (*EV*). To use this

Table 1
Constraints the query pair should satisfy to use Equitas(♣) and Spes(◇) for *set* equivalence verification.

Condition/Operator	SPJ	Outer Join	Agg (count, sum, avg)	Agg (max, min)	Union
Predicate conditions have to be linear	♣ ◇	♣ ◇	♣ ◇	♣ ◇	◇
The pair should have exactly 0 or 1 of the operator types		♣	♣	♣	
Table is not scanned more than once			♣		
Input must be SPJ			♣		
The pair is isomorphic		◇	◇	◇	◇
Grouping columns must be the same			◇	◇	

module, we need to ensure the queries passed to it meet its constraints. When the pair of DAGs violate the constraints, e.g., inclusion of Unnest and Label in the running example, the Analyzer cannot pass the pair to the *EV*. Notice that the two DAGs are isomorphic to each other except those places with changes. We exploit this isomorphic mapping to break each DAG into smaller "segments". We can reduce the problem of evaluating the equivalence of entire DAGs to the problem of verifying segment-wise equivalence. The segments start from the left most changes and end at the right most changes following the topological ordering of the DAG's. For example, Figure 4 shows the pair of DAGs of the scatterplot sink operator in versions v_c and v_b. The minimum segments are highlighted to show the inclusion of all the differences between the two versions, i.e., addition of Filter before Join and removal of Filter after Join in v_b, and removal of Filter before Join and addition of Filter after Join in v_c.

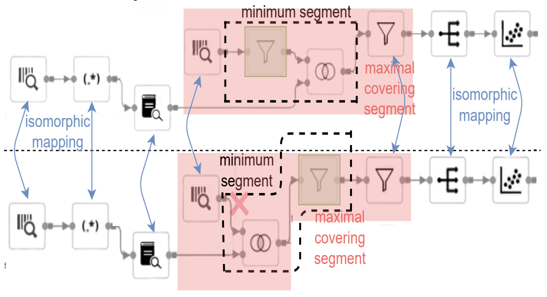


Figure 4: Minimum segments and maximal covering segments for the scatterplot sinks in v_b and v_c.

Once the minimum segment in a version is identified, the Analyzer expands it in both directions to include as many operators as possible while they still satisfy the constraints of the *EV*. In other words, the Analyzer aims to find a maximal covering segment that satisfies the constraints of the *EV*. In the running example, to find a maximal covering segment for each minimum segment, we include Scan on the left and Filter on the right. The inclusion of these operators does not violate the constraints of the *EV*. Adding Label on the left or Unnest on the right will violate the constraints. In general, there can be more than one maximal covering segment.

Given two workflow DAGs, an *EV*, and a type of equivalence such as set equivalence, bag equivalence, or list equivalence, we compute a pair of maximal segments that cover all the changes between these two DAGs and satisfy the constraints of the *EV*. After that, the Analyzer passes the two segments as two queries to the *EV* to verify their equivalence.

2.3. Extending Equivalence Verifiers to Include Non-relational Operators

It is possible that the minimum segments that contain the changes do not satisfy the constraints to use an existing *EV*s. For instance, in the running example, a change on the Label operator will result in the operator being included in the minimum segment. The equivalence of the two operators in the two versions cannot be verified by the existing *EV*'s because it is a non-relational operator. In this case, the ANALYZER cannot verify equivalence of the pair. We plan to extend the existing *EV*s to overcome these limitations.

We define the result of a sink operator as a list of tuples with a specific schema and a possible order. To reason the semantics of a segment, we represent its results as $\langle Tuple \rangle$ and $\langle List \rangle$. $\langle Tuple \rangle$ represents a single tuple on the list. $\langle List \rangle$ represents the entire result list and contains information about how many times a tuple exists in the list and the order of the list. $\langle Tuple \rangle$ and $\langle List \rangle$ can be represented using the elements T, S, C, and O as follows.

$$\langle Tuple \rangle \qquad T ::= FOL \quad \text{and} \quad S ::= columns$$

$$\langle List \rangle \qquad C ::= SPNF \quad \text{and} \quad O ::= constraint$$

T is a first-order-logic (FOL) formula that indicates if an input tuple exists in the output result or not. S represents the set of columns, i.e., the schema of the tuple in the output result. C indicates the cardinality of a tuple in the entire relation and is represented in a sum-product normal form (SPNF). O contains the columns the list is ordered in. This result representation allows us to only use $\langle Tuple \rangle$ elements when we need the set semantics. We abstract the operators to show their impact on each part of the representation in Table 2. We construct the representation at each operator using its own logic based on its properties. Finally, to verify if two segments are equivalent, we ask an SMT solver [18] to verify whether $segment_1 \neq segment_2$ using their representation is satisfiable or not.

3. Conclusion and future work

In this work, we introduced Drove, a framework that manages the end-to-end lifecycle of the execution result

Table 2
Impact of an operator on element of the ⟨*Tuple*⟩ and ⟨*List*⟩ representation.

Operator Representation		Order By	Label	Unnest	Replicate
Tuple	T		✓	✓	
	S		✓		
List	C				✓
	O	✓			

of a data-processing workflow. Drove includes in its core a few modules to achieve the tracking of the results. We presented a few optimizations to reuse previously-stored results by reasoning the semantics of workflow versions that produced the results. We showed a unique technique to decompose a complex workflow DAG to smaller segments that include the version changes to verify their equivalence. We also proposed a technique to capture the semantics of non-relational operators using a lightweight representation.

We plan to enhance the framework by studying the following topics. (1) We plan to extend the current prototype to highlight fine-grain differences between a pair of results. We also plan to include a high-level snapshot of the different versions and their results to give the user an overview of the different runs. (2) One challenge in the framework is to decide the number of versions to compare the current version with in order to maximize the chance of reusing earlier results. In the future, we plan to use a cost-based process to decide the number. (3) We plan to extend the verification to include containment relation so that we further reduce the storage and maximize reuse opportunities. We need a way to identify a delta query to be run on existing results to answer the contained version request. (4) We plan to study the degree of similarity between workflow versions so that we can quickly identify those with the highest similarity with the current version.

Acknowledgments

This work is supported by a graduate fellowship from KSU and partially supported by the National Science Foundation under the awards III 1745673 and III 2107150 and the Orange County Health Care Agency.

References

[1] Z. Wang, A. Kumar, S. Ni, C. Li, Demonstration of interactive runtime debugging of distributed dataflows in texera, VLDB 13 (2020).
[2] H. Miao, A. Deshpande, Provdb: Provenance-enabled lifecycle management of collaborative data analysis workflows, IEEE Data Eng. Bull. (2018).
[3] S. Woodman, H. Hiden, P. Watson, P. Missier, Achieving reproducibility by combining provenance with service and workflow versioning, in: WORKS'11, 2011.
[4] Y. Zhang, F. Xu, E. Frise, S. Wu, B. Yu, W. Xu, Datalab: a version data management and analytics system, in: BIGDSE@ICSE'16, 2016.
[5] 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, C. Zumar, Developments in mlflow: A system to accelerate the machine learning lifecycle, in: DEEM@SIGMOD'20, 2020.
[6] M. Vartak, H. Subramanyam, W. Lee, S. Viswanathan, S. Husnoo, S. Madden, M. Zaharia, Modeldb: a system for machine learning model management, in: HILDA@SIGMOD'16, 2016.
[7] Klaus Greff, Aaron Klein, Martin Chovanec, Frank Hutter, Jürgen Schmidhuber, The Sacred Infrastructure for Computational Research, in: SciPy'17, 2017.
[8] G. Gharibi, V. Walunj, R. Alanazi, S. Rella, Y. Lee, Automated management of deep learning experiments, in: DEEM@SIGMOD'19, 2019.
[9] H. Miao, A. Li, L. S. Davis, A. Deshpande, Towards unified data and lifecycle management for deep learning, in: ICDE'17, 2017.
[10] I. Elghandour, A. Aboulnaga, Restore: Reusing results of mapreduce jobs, VLDB'12 (2012).
[11] F. Nagel, P. A. Boncz, S. Viglas, Recycling in pipelined query evaluation, in: ICDE'13, 2013.
[12] S. Grossman, S. Cohen, S. Itzhaky, N. Rinetzky, M. Sagiv, Verifying equivalence of spark programs, in: CAV'17, 2017.
[13] S. Chu, B. Murphy, J. Roesch, A. Cheung, D. Suciu, Axiomatic foundations and algorithms for deciding semantic equivalences of SQL queries, VLDB'18 (2018).
[14] Q. Zhou, J. Arulraj, S. B. Navathe, W. Harris, D. Xu, Automated verification of query equivalence using satisfiability modulo theories, VLDB'19 (2019).
[15] Q. Zhou, J. Arulraj, S. B. Navathe, W. Harris, J. Wu, SPES: A two-stage query equivalence verifier, CoRR'20 (2020).
[16] A. Kumar, Z. Wang, S. Ni, C. Li, Amber: A debuggable dataflow system based on the actor model, VLDB 13 (2020).
[17] A. Mostowski, Impossibility of an algorithm for the decision problem in finite classes, Journal of Symbolic Logic 15 (1950).
[18] L. M. de Moura, N. S. Bjørner, Z3: an efficient SMT solver, in: TACAS'08, 2008.