UNIVERSITY OF CALIFORNIA,
IRVINE

Improving Iterative Analytics in GUI-Based Data-Processing Systems with Visualization, Version Control, and Result Reuse

DISSERTATION

submitted in partial satisfaction of the requirements
for the degree of

DOCTOR OF PHILOSOPHY

in Computer Science

by

Sadeem Alsudais

Dissertation Committee:
Professor Chen Li, Chair
Professor Michael J. Carey
Assistant Professor Faisal Nawab

2023

DEDICATION

To the loving memory of my brother Abdullah, without whom this PhD journey would not have even begun. To my parents, who have always been supportive and understanding. I hope this dissertation serves as a tribute to their sacrifices, especially during the period when they needed me the most.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

ACKNOWLEDGMENTS

I could not have undertaken this Ph.D. journey without the help of my advisor, Professor Chen Li. I am immensely grateful for the lessons I have learned by observing his unwavering commitment to meeting deadlines and his tireless efforts to ensure the success of our projects. Through his example, I have understood the true meaning of perseverance and hard work. I am grateful for his prompt responses to my emails, his accessibility, and his willingness to engage in discussions. His "open-door" policy has fostered an environment of collaboration and innovation where ideas could flow freely. His mentorship has enabled me to tackle and formulate practical research questions in complex systems. Professor Chen has consistently provided me with opportunities to connect with other researchers and has always encouraged my participation in conferences.

I would also like to express my deepest gratitude to the members of the committee (Professor Michael J. Carey and Professor Faisal Nawab) for their valuable suggestions and guidance, which have been instrumental in improving my research work. I would also like to thank Professor Sharad Mehrotra, Professor Nalini Venkatasubramanian, and Professor Shuang Zhao for their warm welcome and help. Their willingness to assist me has been truly commendable.

This endeavor would not have been possible without the generous financial support from King Saud University. The work reported in this dissertation has been supported in parts by NSF awards III 1745673 and III 2107150. I would also like to thank my mentor, Dr. Rebecca Taft, for her guidance throughout the last year of my Ph.D. journey.

I would like to thank Dr. Qiushi Bai for his help in research discussions during the early years of my Ph.D. and acknowledge the Texera team (Dr. Avinash Kumar, Dr. Zuozhi Wang, Shengquan Ni, Yicong Huang, and Xiaozhen Liu) for their collaborations and contributions.

Lastly, I thank Dr. Thomas Hutter and Noura Alomar for providing feedback to improve my writing. I want to thank Dr. Avinash Kumar, Yicong Huang, and Xiaozhen Liu for always being there to help me on many different occasions. I thank my friends Norah Aljammaz and Arwa Alnabit for always providing me with positive energy and listening to my concerns. I want to thank my family members, especially my sisters Manal and Mona and my niece Ghada Abahussain for their constant encouragement and kind words. I want to thank my beloved cat, Blanco, who has given me company and the emotional strength to embark on any difficulty I may have encountered.

The second chapter of this dissertation is a reprint of the material as it appears in "GSViz: progressive visualization of geospatial influences in social networks." In Proceedings of (SIGSPATIAL '22). The co-authors listed in this publication are Qiushi Bai, Shuang Zhao, and Chen Li. The fifth chapter of this dissertation is a reprint of the material as it appears in "Raven: Accelerating Execution of Iterative Data Analytics by Reusing Results of Previous Equivalent Versions." In Proceedings of (HILDA '23). The co-authors listed in this publication are Avinash Kumar and Chen Li.

VITA

Sadeem Alsudais

EDUCATION

Doctor of Philosophy in Computer Science **2023**
University of California, Irvine *Irvine, CA*

Masters of Science in Computer Science **2016**
University of Southern California *Los Angeles, CA*

Bachelors of Science in Information Technology **2011**
King Saud University *Riyadh, Saudi Arabia*

SELECTED PUBLICATIONS

Raven: Accelerating Execution of Iterative Data Analytics by Reusing Results of Previous Equivalent Versions **2023**
Human-In-the-Loop Data Analytics (HILDA@SIGMOD)

GSViz: Progressive Visualization of Geospatial Influences in Social Networks **2022**
Proceedings of the 30th International Conference on Advances in Geographic Information Systems (SIGSPATIAL)

Drove: Tracking Execution Results of Workflows on Large Data **2022**
Proceedings of the VLDB 2022 PhD Workshop

ABSTRACT OF THE DISSERTATION

Improving Iterative Analytics in GUI-Based Data-Processing Systems with Visualization, Version Control, and Result Reuse

By

Sadeem Alsudais

Doctor of Philosophy in Computer Science

University of California, Irvine, 2023

Professor Chen Li, Chair

GUI-based data processing systems simplify and accelerate data tasks with a user-friendly interface, eliminating the need for extensive coding skills. This accessibility allows analysts to easily design, modify, and execute workflows with intuitive drag-and-drop operations and visual representations. Incorporating *visualization* operators into data processing systems to represent the processed result enables analysts to quickly gain insights, understand patterns, and make informed decisions from complex data. As analysts observe the results, they may uncover new trends, leading to further questions or hypotheses that require modifications and edits to the workflow. Each change to the workflow generates a new *version*. Given the iterative nature of data analytics, modifying workflows is a common practice. The results produced from executing these versions are materialized, enabling users to refer back to them to reproduce and replicate past experiments, ensuring the validity of reported outcomes. While striving for improved results, in many cases, the results of new iterations are *equivalent* to those of previous runs. Given the significant time required to execute analytical tasks on large datasets, it becomes imperative to reduce redundant computations by *reusing* previously-stored results. Hence, it is crucial to identify and verify the equivalence of results across different runs.

This dissertation is driven by these pressing needs to enhance iterative data analytics within GUI-based data processing systems by integrating visualization, version control, and result reuse. The dissertation is structured into four main parts.

The first part addresses the challenge of incrementally visualizing large spatial networks while minimizing visual clutter. To tackle this issue, we introduce GSViz, a general-purpose middleware-based solution consisting of two modules, namely edge-aware vertex clustering and incremental edge bundling to effectively visualize large spatial networks.

The second part focuses on the development of Drove, a framework designed to track changes in workflows, environment dependencies, workflow executions, and the generated results. By utilizing Drove, researchers and analysts can gain valuable insights into the evolution of workflows and understand the impact of modifications on the final outcomes.

In the third part, we present Veer, an algorithm for verifying the equivalence of two complex workflow versions. Additionally, we present a series of optimization techniques to improve the performance of the baseline algorithm.

Lastly, we introduce Raven, an optimization framework that ranks the previously executed workflow versions then tests their equivalence compared to a new workflow version execution request. By reusing the results generated from these versions, Raven minimizes redundant computations and significantly enhances performance when handling new workflow execution requests. Raven retrieves the previous versions from Drove and pushes testing their equivalence to Veer.

Chapter 1

Introduction

1.1 Motivation

The advent of online transactions, social media interactions, sensor-equipped devices, and internet-connected systems has led to a large volume of data. This wealth of data presents a tremendous opportunity for extracting valuable insights and driving innovation. Data-processing systems such as Apache Spark [14] and Apache Flink [13] offer infrastructures to process and analyze vast amounts of data. These tools often require specialized coding skills and expertise, which can pose a challenge for domain scientists with limited coding proficiency. To address this issue, GUI-based data-processing systems such as Alteryx [11], Knime [81], Einblick [130], and Texera [138] have emerged. These tools offer a user-friendly interface that simplifies and accelerates data tasks on large amounts of data, eliminating the need for extensive coding skills and making them accessible to a wide range of users [138, 11, 42]. This accessibility empowers analysts to design, modify, and execute data-processing workflows through intuitive drag-and-drop operations.

A data workflow is modeled as a Directed Acyclic Graph (DAG), where each node corre-

sponds 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, and operators without outgoing edges serve as sinks, representing the final output of the task from its upstream operators. Figure 1.1 shows an example of a data-processing workflow for analyzing people's perceptions of Tweets related to "climate change" and the propagation of these tweets through *replies* and *retweets*. The workflow includes three sink operators, each producing a different result.

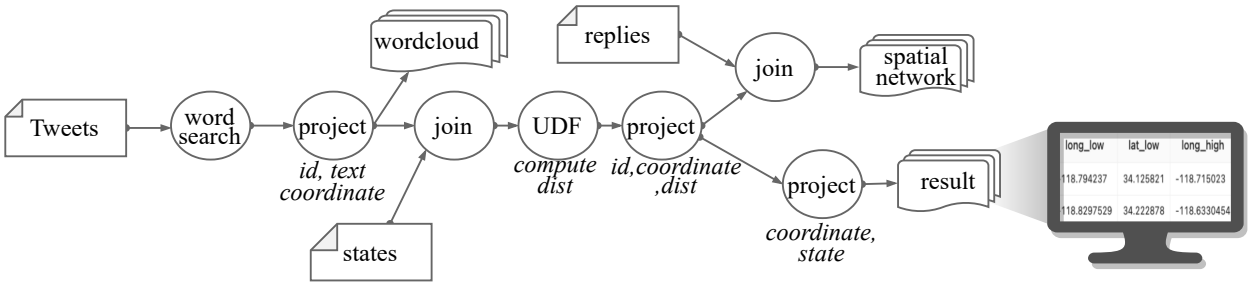


Figure 1.1: Example workflow to analyze tweets about "climate change," including three sink operators to show different final results, with a visual representation of the tabular result.

In the realm of data analytics, an essential aspect is an iterative refinement through a "trial-and-error" approach to enhancing a workflow [151]. Many existing data-processing engines primarily focus on interactivity and collaboration, falling short of fully providing the means to improve the user experience of constructing and refining these workflows iteratively within a GUI-based environment.

In this dissertation, we enhance iterative analytics in GUI-based data-processing workflow systems with visualization, version control, and semantic execution optimization.

Importance of visualization. With the growing volumes of data, it becomes increasingly challenging for analysts to comprehend the final result when it is presented with a large number of tuples in a tabular format, as depicted in the visual representation of the Result sink operator in Figure 1.1. By utilizing data-processing systems with *visualization* operators

to represent the processed tuples, analysts can quickly gain insights, discover patterns, and make time-critical decisions from complex data that may not be immediately apparent in a tabular representation [8]. Modern GUI-based data-processing systems equip their platforms with visualization operators, allowing for both data processing and visualization during the analytical task [130, 109, 11, 81, 138, 119]. User-facing data analytic systems often adopt a progressive execution paradigm to ensure responsive analytics and engage the analysts' focus by enabling users to observe intermediate results during the execution instead of waiting for prolonged durations until the end of the execution [130, 84, 13, 42]. Thus, these platforms provide native support to handle incremental computation to update the visual results [130]. Some of these techniques focus on specific types of visualizations, such as bar charts, while leaving other types unexplored, such as spatial networks [10], even though many applications analyze and visualize spatial networks [90, 91, 61, 147, 17, 166]. In the first part of the dissertation, we want to answer the following question:

How can we efficiently visualize a large spatial network progressively?

Importance of tracking workflow versions. As analysts observe the visual result after executing the workflow, they may uncover new trends that lead to additional questions requiring edits to the workflow to gain deeper insights. Making workflow edits is typical in data analytics due to its iterative nature [154]. In the running example (Figure 1.1), an analyst notices that the most discussed keyword in the visual representation of the Wordcloud sink is "wildfire." Consequently, the analyst makes changes to the workflow, resulting in the creation of a second version (shown in Figure 1.2) of the workflow to observe tweets in California, based on the assumption that most wildfire cases happen in this region.

Tracking workflow changes and materializing the corresponding results generated during the execution of the workflow version is crucial for reproducibility, enabling users to replicate and verify the results from previous runs [9]. There has been an increased interest in track-

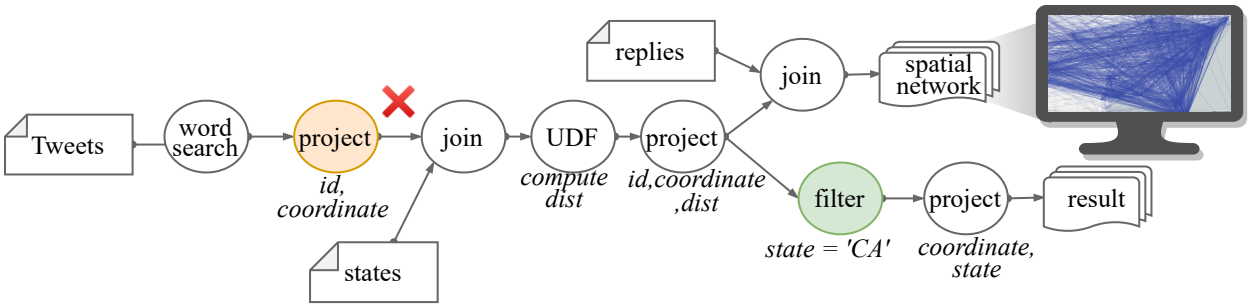


Figure 1.2: A refined version of the workflow to inspect tweets in "California," with a visual representation of the spatial network result. Orange operators are modified, green operators are added, and a red cross indicates a deleted operator.

ing executions and results of data-processing tasks [98, 143, 95]. Many of these solutions aim towards machine learning (ML)-based analytics, which is frequently constructed using a programming interface such as Jupyter Notebook [77]. These solutions use Git-like techniques to track versions of code, ML models, hyperparameters, and package dependencies. Git-like approaches are not directly applicable to GUI-based workflow systems. A reason is that constructing and refining workflows is an ad hoc process [123]. For example, many usability changes (e.g., changing the position of an operator or annotating and commenting on the workflow canvas) do not have any direct implications on the execution of the workflow. Since these non-semantic changes do not affect the results, there is no need to track them to ensure reproducibility. We need a technique that captures changes at a higher level of abstraction, such as modifications to the workflow structure, dependencies on external libraries, or changes to the data flow between operators. In the second part of the dissertation, we address the following problem:

> *How can a workflow and its associated resources, including package dependencies and results, be modeled to facilitate efficient and transparent tracking?*

Workflow equivalence verification. As analysts make changes to achieve better results, in many cases, these iterations produce outcomes that are *equivalent* to those of previous runs [164]. For instance, the changes made to transform the initial workflow in Figure 1.1 to

the refined one in Figure 1.2 have no effect on the spatial network result, i.e., the two spatial network results are the same in both versions. Testing the equivalence of two workflows can be viewed as testing the equivalence of two SQL queries, which is undecidable [3]. Some equivalence verifiers (EV) [164] solve the problem under certain assumptions, such as ignoring Union, Order by, and user-defined-function (UDF) operators. Workflows are complex and rich in semantics, which can make them violate these assumptions in many cases [9]. For instance, most EVs support only 2% of the TPC-DS workload [140], as detailed in Chapter 4. Moreover, these EVs can not verify the equivalence of the workflow versions in the running example, as they include complex operators, e.g., UDF. We need a verifier that can test the equivalence of two workflows, i.e., it can verify if they produce the same results given an instance of input sources. In the third part of the dissertation, we solve the following problem:

> *Given two similar versions of a workflow, can we verify if they produce the same results?*

Workflow execution optimization. Since executing analytical tasks on large datasets can take a significant amount of time [84], we aim to reduce redundant computation by *reusing* previously-stored results and identifying the equivalence of a new execution request with a previously executed version. There is a large body of work on optimizing the execution of complex workflows by matching a workflow DAG or sub-DAG with previously executed ones [49]. Adhering to a rigid approach of exact structure matching can overlook the potential to identify semantic equivalence. In the running example, although the two versions do not have the same structure, the sinks for producing the spatial network in both versions produce equivalent results. Therefore, it is essential to consider alternative methods that go beyond strict structural comparisons to recognize and utilize such equivalences. Other solutions adopt a *semantic* approach to identifying reuse but are limited to a class of optimizations, such as reusing shared predicates [154]. Such solutions identify overlapping tuples between

the results across different executions of the workflows and do not focus on reusing the entire result. We want a way to identify reuse (of equivalent results) as in the running example. In the final part of the dissertation, our objective is to address the following problem:

> *How can we efficiently identify a previously executed version of a workflow that is equivalent to the latest version to reuse its results to answer the execution request of the latest version?*

In this dissertation, we discuss how we address the above four needs in Texera [138], a cloud-based collaborative data-processing system, as an example. The solutions presented in this dissertation are applicable to general GUI-based data-processing systems.

1.2 Technical Contributions

Typical GUI-based data-processing systems follow a three-tier architecture. This architecture consists of a frontend responsible for handling processing task requests, a web server that handles high-level services, such as parsing user requests and workflow DAG optimizations, and an execution engine that manages the execution of the workflow DAG, whether on a single node or across multiple nodes [130, 84, 42, 154]. Figure 1.3 summarizes our technical contributions and provides an overview of the components related to this dissertation and how these components communicate. We propose a holistic approach to managing the end-to-end lifecycle of orchestrating, refining, and executing workflows, examining their results, and reusing the results to optimize the performance of pipelines in data-processing systems. We also tackle the problems discussed above that arise in iterative data analytics settings.

GSViz: Progressive Visualization of Geospatial Influences in Social Networks. In our first contribution, we examine a specific use case involving the visualization of a

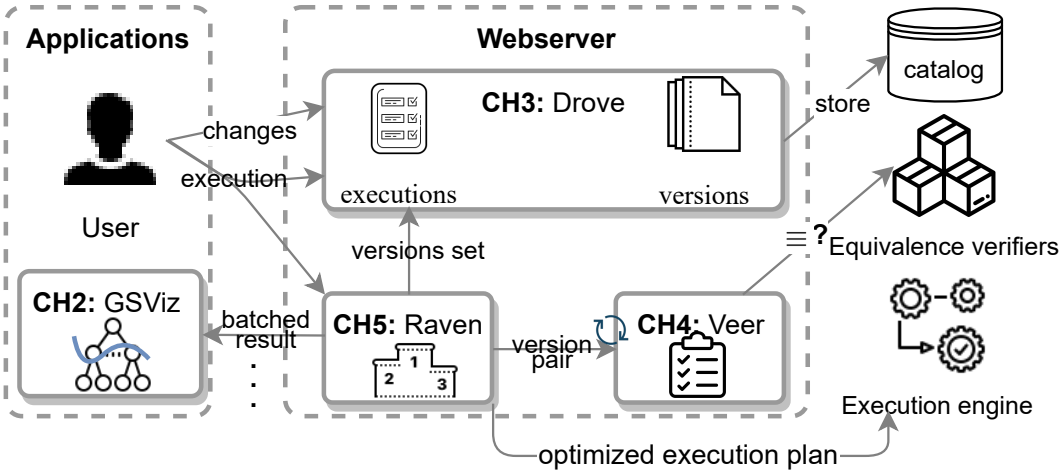


Figure 1.3: Overview of the scope of this dissertation.

large geosocial network. One naive approach to visualizing such a spatial network, where edges are progressively provided, is to append those edges in each batch. As the number of network edges increases, the resulting visualization becomes visually cluttered, as illustrated in Figure 1.2. We have observed that while the problem originated from visualizing workflow results in a data-processing system, it is a general issue that extends beyond such systems in the context of data analytics. We recognize the need to address this problem in a broader context, and propose a middleware-based solution called GSViz.

Our approach proposes two techniques within GSViz to reduce visual clutter. Firstly, we present an edge-aware clustering algorithm capable of clustering spatial network vertices to minimize edges between clusters in an incremental manner. Secondly, we introduce a novel data structure called the "PRB-tree," which organizes edges from previous batches to facilitate faster compatibility checks and edge bundling with new edges. Moreover, we show how the two techniques, i.e., vertex clustering and edge bundling, can be integrated and solve the technical challenges associated with the integration, such as updating the visual result of edges from previous batches. Finally, we show how GSViz supports zooming and panning, and use a hierarchical structure that supports zooming to further reduce the visual clutter by proposing a novel tree-cut approach.

Drove**: Tracking Data-Processing Workflow Versions and Executions to Facilitate Reproducibility.** To enable the versioning of data-processing workflows, we propose the Drove framework. It adopts a strategy where the metadata information of a workflow is stored based on its latest version. Whenever a user refines a workflow, its metadata details are updated with the most recent snapshot. Simultaneously, the edit operations performed on the workflow are stored as a patch, which can be applied when the user desires to view or revert to a previous version. This lightweight approach to storing deltas effectively manages workflow versioning without requiring explicit user commits for each version.

This approach also presents challenges that Drove addresses. One challenge involves determining the grouping of edit operations into a single commit. Another challenge arises as the number of refinements increases, making it computationally expensive to list all historical changes. Additionally, checking out an old version becomes costly due to the large number of patches required to reconstruct it. To tackle these challenges, Drove proposes a periodic checkpointing approach based on predefined rules. Finally, Drove ensures the reproducibility of executions by storing metadata information related to these executions.

Veer**: Verifying Equivalence of Workflow Versions in Iterative Data Analytics.** Given the undecidability of testing the equivalence of two workflows, any verifier developed to address this problem will inherently possess limitations and incompleteness. Our objective is to create a versatile solution that maximizes completeness by leveraging the capabilities of existing EVs. This approach ensures that our solution remains adaptable and flexible, even with the emergence of new EVs in the future. To achieve this, we propose Veer, a workflow version equivalence verifier that incorporates knowledge of the edit operations between the two versions in the decision-making process.

The methodology employed in Veer involves dividing and decomposing the version pair into smaller portions, each satisfying the assumptions of an existing EV. These portions are

then verified for equivalence using the EV as a black box. There are a few challenges in Veer, including how to capture the assumptions of the EVs and maximize completeness. We also identify properties that allow us to prune the considered decompositions, striking a balance between exploring all potential decompositions and avoiding unnecessary computational overhead.

Raven**: Accelerating Execution of Iterative Data Analytics by Reusing Results of Previous Equivalent Versions.** One way to enhance the performance of executing a new version is to check if it produces the same results as a previous one, which can be accomplished using Veer. This process can become computationally extensive as the number of versions increases. To reduce this computational overhead, we propose Raven, an optimization framework that ranks versions based on the semantics of their sinks, prioritizing those with sinks similar to the latest version. To avoid recomputing decompositions into windows for every new pair submitted to Veer, Raven extends Veer to consider the knowledge of edits, decompositions, and equivalences from previous computations by introducing equivalence classes.

1.3 Dissertation Outline

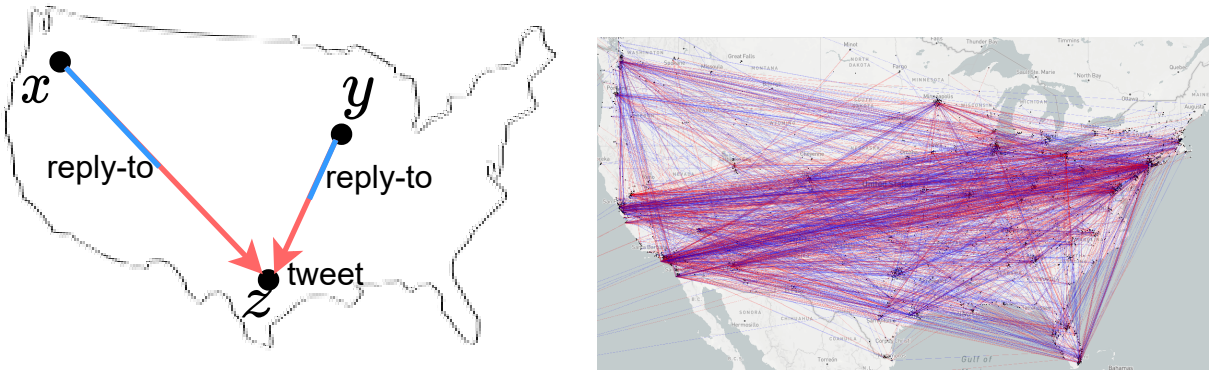
The rest of the dissertation is organized as follows: Chapter 2 presents the GSViz middleware. Chapter 3 discusses the Drove framework. Chapter 4 details the Veer algorithm. Chapter 5 presents the Raven optimization framework. Finally, we conclude the dissertation and discuss open problems and future directions in Chapter 6.

Chapter 2

GSViz: Progressive Visualization of Geospatial Influences in Social Networks

2.1 Introduction

With the prevalent use of social media, it is becoming increasingly important to understand binary relations between entities, such as users or their online posts. Example relationships are "follows" between users and "retweets" between social-network posts. As many entities are location-based, naturally we want to analyze the geospatial relationships between these entities. In fact, analytics of geospatial relationships on social networks can be used in applications such as viral marketing (word-of-mouth information transfer between socially connected users) [166] or personalized location-based recommendations using social relationships [17]. Many recent efforts study the influence of geospatial relationships among entities on social networks [157, 90, 91, 61, 36, 147].



(a) Example "replied-to" relationships from a tweet (blue) to the original tweet (red).

(b) A visual clutter of a network with many tweets and replied-to relationships.

Figure 2.1: A geo-social network of interactions between tweets containing keyword `vaccine`.

Motivation. As an example, it has been observed that Covid-19 vaccine hesitancy is influenced by many factors such as geographic interactions [20] and social media interactions [116]. Analyzing tweets is a useful way to understand how vaccine information propagates across different regions. Figure 2.1a shows a sample network, where a node is a tweet about vaccines, and an edge between two tweets is an interaction between them, i.e., "retweet" or "reply-to." For instance, the edge from node x to node z means that a user in Seattle (x) replied to a tweet posted by a user in Austin (z).

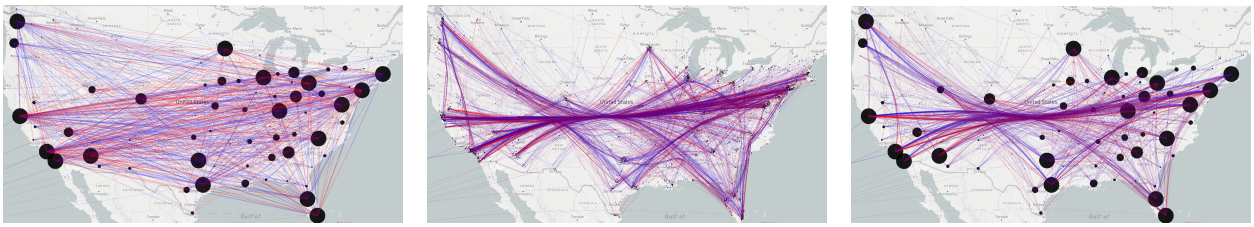
Visualization is a powerful and efficient tool to help analysts gain quick insights from data [8]. In this chapter, we study how to visualize *geo-social networks*, i.e., geospatial relationships on a social network, to help domain experts that need this type of data analytics. We consider the common setting where the data is stored in a database system. As social media data has semantically rich attributes such as temporal, spatial, and textual attributes, we are particularly interested in the case where a user submits a visualization request with ad hoc conditions on the attributes. For instance, a user wants to visualize the subnetwork of tweets containing keywords such as `Covid`, `Pfizer`, or `Moderna`.

Challenges. Due to the ever increasing data size, visualizing large geo-social networks faces

the following computational challenges:

C.1 Visualization requests with ad hoc conditions can be computationally expensive in terms of query execution in the database, network transfer, and frontend rendering. This negatively affects the responsiveness and consequently the user experience [52, 132, 40].

C.2 Visualizing full networks without any simplification can produce results that are visually too cluttered. For instance, Figure 2.1b shows a cluttered result that is very difficult for the user to interpret.



(a) Result of edge-aware vertex clustering (§2.4) **(b) Result of progressive edge bundling (§2.5)** **(c) Results of applying both techniques (§2.6)**

Figure 2.2: Sample results of applying our techniques to support progressive visualization to simplify the network in Figure 2.1b.

To address these challenges, we develop a novel middleware-based system called "GSViz", which stands for "Geo-social network visualizer." To overcome **C1**, we leverage *progressive* computation by slicing a long-running query into multiple mini-queries, each of which has an additional slicing predicate on an attribute. In this way, each mini-query can be executed efficiently, and the results are returned in batches [75]. Similar to streaming videos, users are willing to wait until the end of a long running query as long as there are bursts of frequent updates "for keeping the user's attention focused" [52] and to help users "lose their sense of time" [132].

We address **C2** by *clustering* network vertices and *bundling* the edges. The problem of clustering spatial points has been studied extensively (e.g., [158, 111]). These solutions cluster the points without considering the impact of edges between them. On the other hand,

graph-based solutions [1, 18, 139, 15, 126] do consider edges. However, they typically focus on optimizing the overall layout of a graph, e.g., by minimizing edge crossings. In contrast to vertices in general graphs, vertices in geo-social networks have *geo-location*. Thus, solutions on graphs are not directly applicable in our case. We solve this problem by developing a new algorithm that clusters geo-social network vertices in an edge-aware fashion (Figure 2.2a). Notice that even though vertex clustering could produce clusters with locations slightly different from the original spatial locations of points, the approximate results are still very valuable to users.

Additionally, we simplify a network using edge bundling that merges edges with similar directions and lengths (Figure 2.2b). Previous edge-bundling techniques [68, 64] are not *incremental* and can take a long time, e.g., many seconds or even minutes, to handle large input networks. To solve this problem, we develop a new technique to incrementally bundle the network edges that arrive in batches as the results of mini-queries. We show that these two techniques can be integrated to further simplify the network (Figure 2.2c). Note that if the network structure changes across batches, then there should be a visible animation to show those changes [69]. Smoothing the visual transition between the batches can be done using frontend techniques when rendering the network and is out of the scope of this chapter.

In this chapter we make the following **contributions**:

1. Introducing the architecture of GSViz and describing the details of its components needed to answer a visualization request with ad hoc conditions progressively (§2.3).
2. Developing a new technique that clusters network vertices in an incremental and edge-aware fashion (§2.4).
3. Presenting an efficient technique to bundle network edges progressively by leveraging a novel structure called PEB-tree (§2.5).
4. Integrating the two techniques, addressing related challenges, and supporting zooming

and panning. Moreover, we leverage the hierarchical structure that supports zooming to further reduce the visual clutter using a novel tree-cut approach (§2.6).

5. Conducting an experimental evaluation, including a user study, on real data sets (§2.7). The results show that, compared to previous methods, the proposed system offers better performance without compromising the quality of visualization results.

2.2 Related Work

Big graph visualization systems. Some studies visualize the *geo-social network* as an **O**rigin-**D**estination relation between spatial points [25, 60, 26]. The Gephi and Tulip systems [18, 15] load graph data into memory and process it offline, allowing interactive online filtering and exploration. These systems simplify the graphs and reduce visual clutter by performing graph clustering and layout algorithms. Tulip additionally uses the Winding Roads algorithm [86] to bundle the edges. GraphVizdb [21], CGV [139], and ASK-GraphView [1] systems focus on graph retrieval from a database and allow interactive exploration on the retrieved graph. Tableau [133] is a commercial tool to visualize data from local files and remote databases. All these techniques are complementary to ours as they mostly focus on visualizing graphs where the vertices have no predetermined spatial locations. Further, these techniques do not allow progressive processing of user queries.

Graph clustering and trajectory clustering. Most graph clustering techniques [79, 153] focus on discovering communities in a graph such as SCAN [153], which exhausts pairwise similarities between all adjacent vertices to detect clusters. pSCAN [30], SCAN-XP [134], anySCAN [96], and index-based SCAN [148] try to optimize SCAN's performance. Other graph clustering techniques [33, 155, 165] focus on partitioning a heterogeneous graph such as attributed graphs. Some works focus on reducing the number of edges between the clusters by optimizing modularity [22] or by local graph sparsification [126]. While these

solutions reduce the number of edges between the clusters, they rely on moving the vertices to different clusters freely, hence not applicable to spatial networks. Trajectory clustering [152, 97, 149], on the other hand, focuses on clustering trajectories in a spatial network. PIG [144] optimizes the performance of clustering trajectories into k representatives by utilizing k-paths technique that combines map matching and representation of trajectories. TraClass [87] generates a hierarchy of features by dividing the trajectory data into region-based and trajectory-based features. CACT [66] discovers frequent movement routes over time by considering temporal information on the sub-trajectories as many other studies [145, 135]. Sub-trajectory clustering [112, 5] mostly focus on grouping part of the trajectories with other similar sub-trajectories. TRACLUS [88], for example, uses a two-phase approach to partition the trajectories into sub-lines then group the similar ones. While these techniques do cluster edges in a spatial network, they are not applicable to GSViz, where the focus is on origin-destination spatial networks.

Visualization of spatial points. We present the works under two classes. The first includes visualization of spatial points such as VAS [110], Kyrix-s [136], Nanocubes [92], ImMens [94], Tabula [156], SOS [58], ForeCache [19], and HadoopViz [48] . The second includes techniques to progressively cluster spatial points. IncrementalDBSCAN [50] clusters spatial points by inserting each incoming point into a pre-existing nearby cluster or forming its own cluster if it is an outlier. BIRCH [158] uses a tree structure to group spatial points into clusters based on Euclidean distance. GRIN [32] groups points in a hierarchical structure and uses the gravity theory to decide the position a new point should be inserted into the hierarchy. These methods do not consider edges between points.

Edge bundling. Cost-based edge-bundling techniques [64, 129, 104] use a spatial metric to measure the closeness of the edges and move them closer. These techniques produce results with a high visual quality but can be very slow when handling a large graph with many edges. Moreover, these methods bundle edges implicitly by deforming them, while

the total number of edges rendered on the screen remains unchanged. Geometry-based edge bundling [63, 41, 161] uses geometric approaches such as Delaunay triangulation or grid-based techniques to decide which edges should be grouped. Image-based algorithms [69, 137, 67] use Gaussian filtering to measure edge densities. Although these implementations could be applicable in GSViz's setting, they do not support incremental computation.

Progressive visualization. A section of prior works [78, 115, 75] focus on progressive computation from the database perspective. Whereas other solutions [141, 53] push the progressive computation to later stages in the visualization lifecycle such as rendering the results. There are works that are more general such as sampling techniques [117, 101] to give an initial result using a sample, and gradually improve the sample result by decreasing the error margin. These solutions are complementary to GSViz, e.g., we can use a progressive query slicer, renderer, or sampler. GSViz's focus is on how to incrementally cluster the vertices of a network while considering the edges between them and bundle the edges of a geo-social network.

2.3 GSViz System Overview

In this section, we explain the problem setting using an example, then describe the overall architecture of GSViz and explain the lifecycle of a visualization request in the system.

Problem formulation. Consider a typical three-tier architecture comprised of: (i) a frontend that submits visualization requests; (ii) a backend database that stores geo-social networks in tables; and (iii) a middleware layer that translates visualization requests to database queries and forwards results to the frontend. To represent geo-social networks, we assume there is a table T, where every record is an edge connecting two geo-located points that may be associated with additional information such as text or timestamp. Table 2.1 shows

such an example, where individual data points are tweets and a directed edge represents a reply-to relationship between an original tweet and a reply tweet.

Table 2.1: Sample network data with tweets and their replies

from-id	**from-date**	**from-coordinate**	...	**to-id**	**to-date**	**to-coordinate**	...
667057004570	2020-08-19	(-74.0266, 40.6839)	...	669057256558	2020-08-17	(-73.9625, 40.5417)	...
669228452424	2020-08-11	(-122.4221, 37.7700)	...	667131783385	2020-08-11	(-70.3463, 43.6405)	...
669057004984	2020-08-03	(-111.9217, 40.5933)	...	669225335465	2020-08-01	(-71.1915, 42.2277)	...

The frontend layer allows a user to submit ad hoc visualization requests with arbitrary filtering conditions on spatial, textual, and temporal attributes. The following is an example query Q that requests for tweets and their replies posted during August 2020 and contain the keyword `vaccine`:

`Original Query` Q

SELECT `from-coordinate, to-coordinate`

FROM `tweet-replies`

WHERE `to_tsvector(from-text)@@to_tsquery('vaccine')`

AND `from-date between` '2020-08-01' **and** '2020-08-20'

AND `from-coordinate` box '((-124.4, 36.5),(-70.1, 45.0))';

For a large table T, these visualization requests can be computationally expensive as handling them requires querying the database, transferring the results via the network, and performing frontend rendering. To allow timely feedback to the user, it is desired to have the middleware slice the original query Q into multiple *mini-queries*, each of which has an additional predicate on an attribute (which we call a *slicing predicate*). In the running example, we use "from-date" as the slicing attribute.

`A mini-query` Q_i

SELECT `from-coordinate, to-coordinate`

FROM `tweet-replies`

WHERE `to_tsvector(from-text)@@to_tsquery('vaccine')`

AND from-date between '2020-08-01' and '2020-08-05'

AND `from-coordinate` box '((-124.4, 36.5),(-70.1, 45.0))';

The small date range in a mini-query makes it more selective, and an index on the slicing predicate helps fast retrieval of the result, a subset of the requested network called a *sub-network*. Moreover, the small date range in the mini-query captures the dynamic changes on the network over time. In the running example, the mini-query Q_i includes an additional predicate (in blue) to yield the subnetwork containing the keyword `vaccine` in the first 5 days in August 2020. The main objective of the middleware is to quickly process and visualize a subnetwork, in addition to those given by previous mini-queries, in a progressive fashion while minimizing visual clutter.

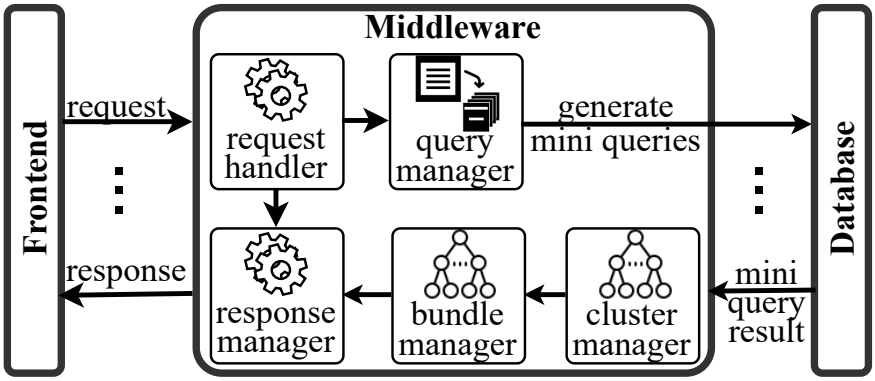


Figure 2.3: GSViz **system architecture.**

System architecture. We introduce a new system called GSViz that adopts the aforementioned three-tier architecture with a focus on the middleware layer for generating mini-queries then simplifying its results to visualize the network in a user-friendly fashion. Figure 2.3 depicts the system's architecture. Its QUERY MANAGER component answers long-running queries progressively. To reduce visual clutter, the middleware has a CLUSTER MANAGER and a BUNDLE MANAGER to incrementally cluster spatial points of a geo-social network and

bundle edges, respectively.

The lifecycle of a visualization starts when a user submits a request through the frontend interface. The middleware begins with slicing the original query into multiple mini-queries by using solutions from the literature [75] then forwards them to the backend database one by one. Every mini-query request and its response form a batch. Whenever the middleware receives the result of a mini-query from the database, it takes the following steps to process the result of the batch: (i) Cluster the vertices of the subnetwork using an edge-aware spatial point clustering (§2.4), which produces *super edges* between the clusters; (ii) Bundle the super edges between the clusters in a progressive fashion; (iii) Forward the simplified subnetwork to the frontend to render. We call an edge between two clusters a "super edge" because it represents many edges between individual points across the two clusters. The result of clustering the vertices and bundling the super edges of a subnetwork is called a *simplified* subnetwork.

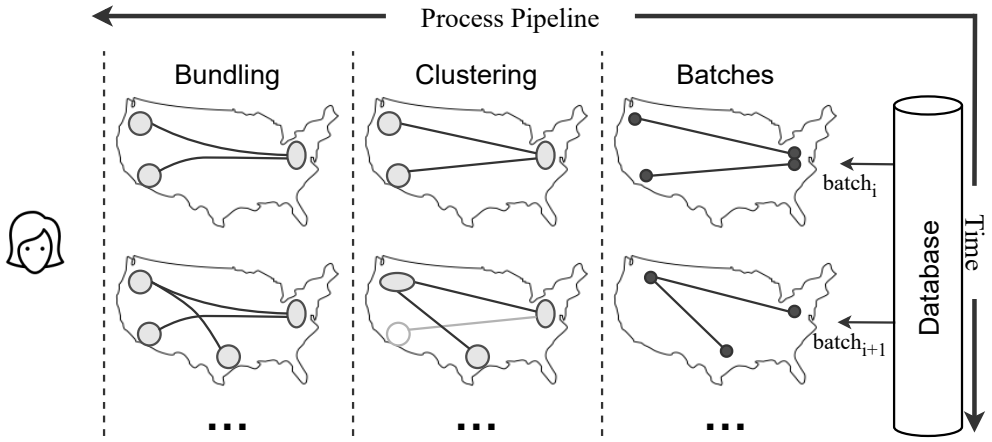


Figure 2.4: Processing subnetworks in two batches in GSViz.

Figure 2.4 shows an example of processing and accumulating the result of two subnetworks from two batches, by clustering the vertices first as we describe in details in the next section, then bundling the edges. Each batch is processed entirely before handling a subsequent batch.

2.4 Incremental Edge-Aware Clustering of Geo-social Network Vertices

A key operation performed by GSViz is incremental spatial clustering of geo-social network vertices. This merge of nearby points and their associated edges reduces visual clutter of visualizing large networks. In this section, we first revisit a widely used point-clustering technique in §2.4.1. Building upon this technique, we introduce in §2.4.2 an edge-aware clustering algorithm. Lastly, we discuss in §2.4.3 performance optimizations of this algorithm.

2.4.1 Incremental Clustering of Network Vertices

We build the vertex clustering operation based on a widely used spatial point clustering algorithm called supercluster [39]. At a high level, supercluster takes as input a set of points, which map to vertices of network edges, and clusters the vertices iteratively. When a new edge arrives, the algorithms first performs a range search for each vertex of the edge over centers of existing clusters. The center of the cluster is the average of its points. If there are no existing clusters, then a point forms its own cluster. The radius ρ of these searches is determined empirically based on several factors such as the field of view and screen resolution. Note that the range search can return multiple candidate clusters for each vertex. To determine which cluster a vertex should be inserted into, supercluster takes a greedy approach by selecting the cluster whose center lies closest to the vertex. Although this simple method works adequately for clustering spatial points, it neglects network edges, thus can lead to a large number of super edges after clustering.

Figure 2.5 illustrates such an example. For simplicity, we represent an edge e as two vertices (l, r) to denote the left and right vertex respectively. Upon receiving a new edge (l, r), supercluster performs a range search around l and r, as shown in Figure 2.5-i. Then, points

l and r are inserted in their nearest clusters, i.e., a and b respectively. This merge causes the resulting network to contain a super edge (a, b), as demonstrated in Figure 2.5-ii. In this example, since there already exists a super edge (c, d), inserting l into cluster c and r into d is better, as shown in Figure 2.5-iii. This example demonstrates that, to produce high-quality visualizations for geo-social networks, the point clustering algorithm needs to be *edge-aware*. That is, it needs to consider the information of existing super edges between the clusters during the clustering process. We present our solution to this problem next.

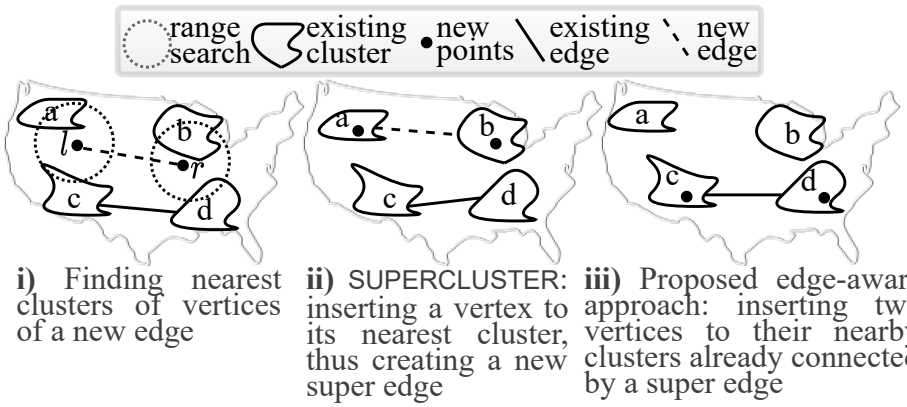


Figure 2.5: Incrementally clustering a new edge (l, r)**.** For simplicity, we omit the edge directions and cluster centers.

2.4.2 Achieving Edge-Awareness

Algorithm 2.1 details the steps of incrementally clustering vertices of a new batch of edges E. For each edge $e = (l, r)$ in E, our goal is to insert both of its vertices into nearby clusters while minimizing the number of super edges. To this end, we start with checking if the distance between the two vertices is within ρ. If so, we merge them into one point m and insert it into its nearest cluster by performing a nearest neighbor search. The goal of this step is to filter the edges that are too short (lines 2–5). If the distance is larger than ρ, we find clusters closest to each each vertex by performing a nearest neighbor search (lines 7–8). For each vertex, if it does not have any nearby clusters, we create a new cluster for it and

Algorithm 2.1: Clustering vertices in a batch of edges E

Input: A new batch of edges E; a set of existing clusters C; and a range radius ρ
Output: Updated set of clusters C

```
1  foreach edge e = (l, r) in E do
2      if distance(l,r) ≤ ρ then
3          merge them to a point m;
4          insert m to its nearest cluster;
5          continue;
6      end
7      C_l = C.rangeSearch(l, ρ) ;                    // find near clusters
8      C_r = C.rangeSearch(r, ρ);
9      if C_l is ∅ then
10         create a cluster c_l for l and add c_l to C_l;
11     end
12     if C_r is ∅ then
13         create a cluster c_r for r and add c_r to C_r;
14     end
15     if ∃c_l ∈ C_l, ∃c_r ∈ C_r with a super edge (c_l, c_r) then
16         c_l.insert(l); c_r.insert(r);
17     else
18         insert l to its nearest cluster;
19         insert r to its nearest cluster;
20         create a super edge between the two clusters;
21     end
22 end
23 return C;
```

insert the new cluster to the corresponding set (lines 9–13). We check if there exists a pair of candidate clusters that already has a super edge connecting them (line 15). If a pair exists, the edge vertices are inserted into these clusters. Revisiting the running example in Figure 2.5-iii, using this technique, point l will be inserted into cluster c and point r into d. If such a pair is not found, the two vertices are inserted into their nearest clusters (lines 18–19). Then we create a super edge to connect the two clusters (line 20). One may ask about the trade-off between merging vertices of edges with existing clusters and the accuracy of the clustering. The range radius of the search is bounded by ρ so a vertex does not merge with a far-off cluster. Moreover, we propose an objective function in future work to find the optimal balance between efficiency, reducing visual clutter, and accuracy.

Increasing the chance of including more candidate clusters. Noticeably, we get the benefit of edge-aware point clustering when the following two conditions are satisfied: (i) both vertices have candidate clusters in the range radius; and (ii) there exists a super edge between a pair of candidate clusters from each vertex. If one or both vertices do not have candidate clusters in the range, we increase the radius ρ gradually to include more candidate clusters. One would be interested in increasing the range search drastically to include as many candidate clusters as possible to insert the new edge into an existing super edge, resulting in a reduction in the visual clutter. However, this action might end up inserting an outlier vertex to a very far cluster, causing misleading information in the resulting visualization. To avoid this, we set a threshold on the maximum value of ρ.

Edge-aware merging of clusters. The edge-aware clustering discussed so far is utilized on newly added edges and affects the decision of inserting both vertices into existing clusters or creating new clusters. Recall that our motivation for clustering the points is to reduce the clutter by reducing the number of super edges. We take advantage of the greedy approach of this clustering algorithm to further reduce the number of super edges by merging two super edges into one. We check if two super edges can be merged whenever a new point is inserted into a cluster.

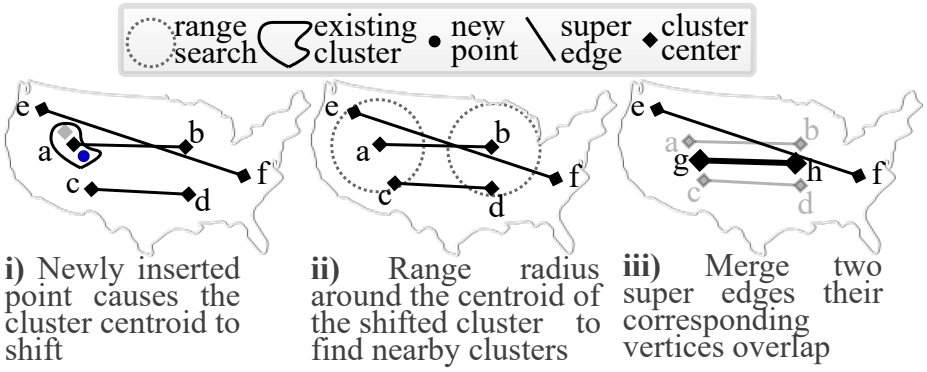


Figure 2.6: Progressive merging of super edges. We omitted the cluster shape for notational simplicity.

Figure 2.6 demonstrates the merge operation due to the insertion of a new point into cluster a. When the new point is inserted into cluster a, the cluster's center may shift due to the new addition, as shown in Figure 2.6-i. Over time, this shift may cause the cluster's center to be within the range radius ρ of a nearby cluster, such as clusters e and c around a as shown in Figure 2.6-ii. We say two clusters are "overlapping" if their centers are within ρ. We take this opportunity to merge cluster a with one of its neighboring clusters to further reduce the clutter. However, if we merge the two clusters without considering the super edges connected to them, we may not solve the problem of reducing the number of super edges. For example, if we merge the clusters e and a, the number of super edges remains the same. To solve this problem, we add one more condition to the merge operation: two clusters can be merged only if their corresponding other vertices connected to the clusters are also overlapping. Using this approach, clusters a and c are merged into a larger set cluster g, where its center is the weighted average of the centers of the two clusters. Similarly, the other clusters connected to g are also merged, i.e., merge b and d into a cluster h, as shown in Figure 2.6-iii. Notice that clusters b and d were not merged prior to the addition of the new point as the condition was not satisfied.

2.4.3 Improving Computational Efficiency

Computational challenges. The computational complexity of the edge-aware clustering algorithm is higher than that of traditional methods due to the need of examining connectivity between candidate clusters. Specifically, traditional point clustering algorithms such as supercluster always pick the nearest cluster, leading to a complexity of $O(dN)$, where d is the number of candidate clusters for a point n in a batch and N is the number of points in the batch, i.e., $N = 2E$, where E is the size of a subnetwork in the batch. The complexity of the edge-aware clustering is $O(d^2N)$, as we need to find the adjacency between every pair of candidate clusters from each corresponding vertex to check if they are connected. This high

complexity negatively affects the visualization performance. To address the performance issue, we propose a grid-based technique to quickly find a nearby super edge within the range radius that is not necessarily the nearest.

Grid-based acceleration. We first divide the space into a grid, where the size of a cell is determined by ρ. Then we build an in-memory hash table to store the super edges. A key in the hash table consists of a pair of cell IDs and the value is a set of edges whose vertices are in the corresponding cells. Whenever a new super edge is formed between two clusters, we insert it to the hash table. Figure 2.7 shows how we use the hash table to insert an edge $e = (l, r)$ into an existing super edge. We first identify the grid cells within the search radius from l and from r. Consider the cell-ID pair $[\langle x_2, y_2\rangle, \langle x_3, y_6\rangle]$ in the hash table that includes a set of existing super edges in the cells. We choose any edge that both of its vertices lie within the search radius. i.e., edge e_1. We insert the points l and r to the corresponding clusters represented as the vertices of the edge e_1. The complexity of this approach is reduced to $O(E)$.

Super Edge Hash Table

1st cell ID	2nd cell ID	super edges
(x_2,y_2)	(x_3,y_6)	e1, e2
(x_4,y_6)	(x_2,y_8)	e3

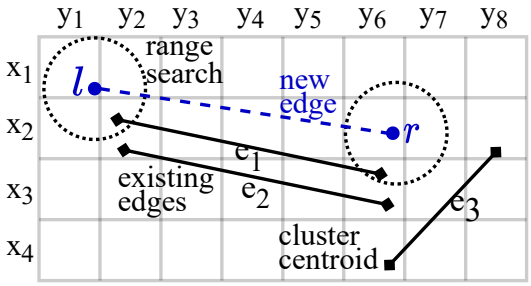


Figure 2.7: Using a grid to speed up edge-aware clustering.

2.5 Incremental Edge Bundling for Network Simplification

In this section, we first describe the problem of network simplification using edge bundling (§2.5.1). Then, we present a new technique to enable efficient and incremental bundling of network edges (§2.5.2 and §2.5.3). For simplicity, we assume vertices of an edge in the network do not change in later batches and will relax this assumption in the next section.

2.5.1 Problem Specification

Given a geo-social network, we consider the problem of visually simplifying the network while preserving as much information as possible. This has been typically achieved via *edge bundling*—a process that deforms network edges so that nearby ones share similar shapes, allowing the screen space to be used more efficiently. We utilize the widely adopted *force-directed edge bundling* (FDEB) [64].

Force-directed edge bundling. We now provide a brief recap of the FDEB algorithm, starting with the following key definitions:

Definition 2.1. *For two edges e_1 and e_2, their* compatibility measure*, denoted $C_e(e_1, e_2)$, is a value computed using their angle, length, position, and visibility [64].*

Definition 2.2. *We say two edges e_1 and e_2 are* compatible *if $C_e(e_1, e_2) \geq \delta$ for a given constant threshold δ.*

Given an edge a and a set of edges $S = \{b_1, \ldots, b_n\}$ compatible with a, FDEB deforms the edge a based on S as follows. Assuming that the edge is represented as a polyline $a = (a_0, a_1, \ldots, a_m)$ with a_0 and a_m being the two fixed endpoints, the remaining points

$a_1, \ldots, a_{m-1}$ are called the edge's *control points*. To deform a network edge a, FDEB applies two types of forces—*spring* and *electrostatic*—to its control points. The spring force exists between two adjacent points within the same edge. That is, each control point a_i (with $0 < i < m$) received spring forces $F_s(a_i, a_{i-1})$ and $F_s(a_i, a_{i+1})$ that are determined, respectively, by the positions of a_{i-1} and a_{i+1} [64]. The electrostatic force, on the other hand, is between control points from different compatible edges. Specifically, let $b_j = (b_{j,0}, \ldots, b_{j,m})$ be an edge from S. Then, for each $0 < i < m$, the electrostatic force acting on a_i by $b_{j,i}$ is

$$F_e(a_i, b_{j,i}) = \frac{\overrightarrow{a_i\, b_{j,i}}}{\|a_i - b_{j,i}\|}, \tag{2.1}$$

where $\overrightarrow{a_i\, b_{j,i}}$ denotes the unit vector pointing from a_i to $b_{j,i}$. To avoid singularities, F_e is set to zero when $\|a_i - b_{j,i}\|$ is below a predetermined threshold.

In this way, the net force acting on a_i (given S) is

$$F(a_i,\ S) = F_s(a_i, a_{i-1}) + F_s(a_i, a_{i+1}) + \sum_{b_j \in S} F_e(a_i, b_{j,i}). \tag{2.2}$$

All forces F, F_s, and F_e in the equation are two-dimensional vectors. By computing the net force acting on each control point a_i, we move these points accordingly, as shown in Figure 2.8. We call the entire process "bundling edge a using the set of edges S."

Given a set of edges $E = \{e_1, \ldots, e_n\}$, the FDEB algorithm bundles these edges as follows. For each $e_i \in E$, the algorithm first computes $S_i \subseteq E \setminus \{e_i\}$ comprised of edges compatible with e_i, and then uses S_i to bundle the edge e_i using the aforementioned process. Lastly, the bundling of each edge e_i, using the compatible ones, is repeated for a predetermined number of iterations.

Computational challenges. To adopt FDEB in GSViz, a main challenge we need to overcome is its high computational cost. Specifically, to enable progressive visualization, we need to efficiently bundle edges of subnetworks that arrive in batches. A naïve solution is to bundle the new edges using existing subnetworks. This, however, is inefficient as computing the compatible set S_i for each new edge e_i in the batch requires examining all the edges received so far. Moreover, computing electrostatic forces using Eq. (2.1) requires examining all compatible edges from earlier batches. This clearly cannot meet the *responsiveness* requirement in interactive visualization. In the rest of this section, we present a novel technique for efficient *incremental* edge bundling. Notice that the spring forces are computed within each edge locally, so they can be computed with a low overhead. Thus we mainly focus on improving the performance of ❶ finding the compatible edges for each new edge and ❷ the computation of electrostatic forces.

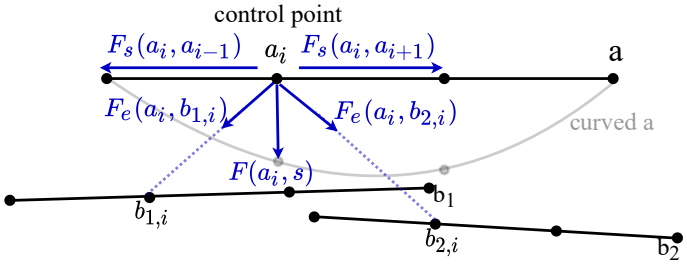


Figure 2.8: Dragging a control point on edge a using spring and electrostatic forces of compatible edges b_1 and b_2.

2.5.2 PEB-Tree

At the core of our technique is a new hierarchical data structure—which we call the "PEB-tree" —that represents a set of edges E. ("PEB" stands for *Progressive Edge Bundling*.) As shown in Figure 2.9, each leaf of the PEB-tree corresponds to a "raw" edge from E. Each internal node stores: (i) an edge as a representative of the raw edges in the descendants of this node; and (ii) a weight ω that is the number of those raw edges. Edges of the same parent siblings are initially compatible with each other, and the parent's edge is the weighted average edge computed using the child edges. Each vertex of the parent edge is a weighted

average of the corresponding vertices of the child edges. Figure 2.9 shows the details of computing the vertex l_p of the weighted edge in the node n_p, where (x_p, y_p) is the weighted average of vertices l_1 and l_2 from the child nodes n_1 and n_2. The tree has a pseudo root as the edges of its children are not required to be compatible with each other.

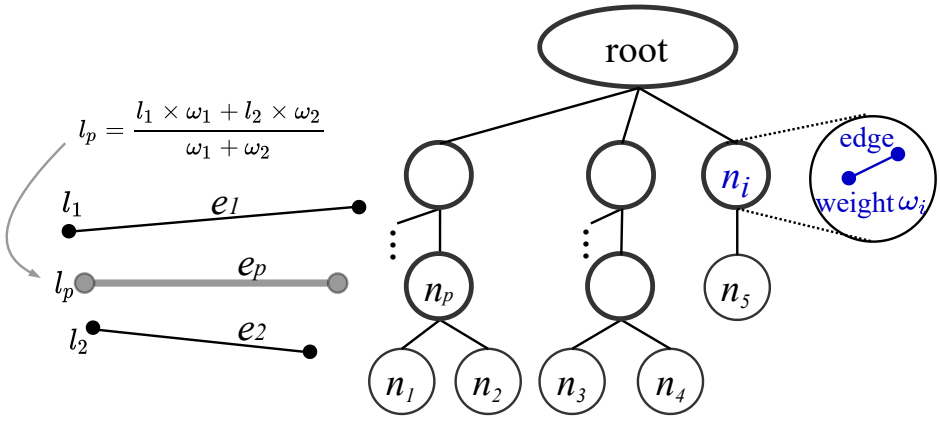


Figure 2.9: A PEB-tree for a set of edges.

Tree construction for the first batch. We construct a PEB-tree for the first batch in progressive visualization as follows. Initially, we create a leaf node for each edge in this batch with a weight of 1. We group these edges such that all the edges in each group are compatible with each other. For each group, we construct a parent node that (i) stores an edge given by the weighted average of all edges stored in its children; and (ii) has a weight that equals the sum of the weights of its children. This process is repeated by grouping the nodes without a parent until all edges in those nodes are mutually incompatible. For each remaining leaf node that is incompatible with any other node, we construct a replica as its parent to represent a group that includes the edge of the leaf, so that future compatible edges could be merged into this group. Lastly, we construct a pseudo root as the parent of these remaining edges.

2.5.3 Incremental Edge Bundling Using PEB-tree

We now describe how to use the PEB-tree to efficiently and incrementally bundle edges. Algorithm 2.2 details the steps of incrementally maintaining the tree when bundling a new batch of edges E. For every edge $e \in E$, we create a leaf node n with unit weight (lines 1–2). To find a place on the tree to insert this new node, we traverse the PEB-tree top-down, and use the compatibility value between each tree node and e to guide the traversal. We find a deepest, i.e., farthest from the root, compatible non-leaf node n'. Recall that this node stores an edge that is a representative of all raw edges in leaves that are descendants of this node. We use the same method as the one introduced in [64] to compute the compatibility value between a raw edge and a weighted edge (line 3). If there exists such an n' node, we add n as a child of n' (line 7) and then traverse upward to the root to adjust the edges and their weights on the way. As we already traversed the tree to node n', we can efficiently update the edges and weights along the path to the root. Notice that it is possible that the new edge e is not compatible with every child edge of node n'. In this case, after making n' the parent of n, the children of n' will no longer be mutually compatible. If we want to keep this all-pair-compatible property, we could partition the children into groups such that each group still has this property. A main downside of this approach is that there will be too many groups, and a new node can cause cascading effect on the tree, which can be computationally expensive. We could relax this property for each group of children of the same parent node. On the other hand, if n' does not exist, we create a new parent node n' for the new leaf n, which is a replica of the new edge, then add n' as a new child under the root (line 5). The replica parent represents a group containing this singular edge to allow future compatible edges to be merged into. Note that the time complexity of traversing and maintaining the PEB-tree depends on its depth and branching factor, which can be controlled using heuristics for efficient traversal.

After inserting all the edges of the new batch E into the PEB-tree, we use the new tree to

Algorithm 2.2: Incremental maintenance of PEB-tree and edge bundling

Input: A PEB-tree T of edges of previous batches; and a new batch of edges E;
Output: Updated T and a set of curved edges for E.

```
   // Update T
1  foreach edge e in E do
2  |   n = create a new node (e,1);
   |   // get lowest non-leaf node with an edge compatible with e
3  |   n' = T.traverse(n);
4  |   if n' is not found then
5  |   |   n' = create a new parent node for n;
6  |   end
7  |   n'.insert(n) ;                          // insert n as a child of n'
8  end
   // Bundle edges in E
9  S = ∅;
10 foreach edge e in E do
11 |   let n'(e', ω) be the corresponding parent node of e;
12 |   ê = Edge-Curving(e, e', ω) ;             // drag e towards e'
13 |   S.add(ê) ;
14 end
15 return (T, S) ;
```

bundle these edges. For each edge $e \in E$, we use its corresponding parent node n' on the tree to bundle e using the two types of forces in the FDEB algorithm (lines 10–12). In order to produce a similar edge bundling result of using all the child nodes under n' by using only their parent's weighted edge e', we redefine the electrostatic force on a control point a_i to include the weight information as follows:

$$F_e\big(e_i, n'(e'_i, \omega')\big) = \frac{\omega'}{\|e_i - e'_i\|}\overrightarrow{e_i e'_i}. \tag{2.3}$$

Deforming the new edge e using only the weighted average parent n' offers a better performance compared with using all raw edges from the leaf nodes under n'. We note that the DEB algorithm [129] also considers edge weights in its revised electrostatic force function. However, their approach does not use a single edge to represent multiple edges, thus does not solve the efficiency issue. Lastly, the resulting curved edge, which contains the information

of the new location of the control points, is sent to the frontend to be visualized while the edge in the tree remains the same as before the deformation (lines 13–15).

2.5.4 Sending Bundled Results to the Frontend

After incrementally bundling the edges in a new batch, the middleware needs to send the curved edges to the frontend. One way to do so is to store the results of all the batches up to now in the middleware, and send the accumulated results to the frontend to render from scratch after every batch. This approach has a high overhead in terms of data transfer and rendering cost, and the overhead further increases as more batches are processed. A more efficient way is to send the results of each batch to the frontend, i.e., curved edges in the new batch. Notice that a curved edge is a set of control points.

After the frontend receives the bundled results of each batch, it needs to render them on top of previous results. There are mainly two ways (i) The frontend represents the graph as an object, appends the new curved edges to the object, and re-renders the object from scratch. (ii) The frontend has multiple layers, creates a new layer for the new curved edges, and renders the new layer only. The second approach is more efficient. Moreover, as we will see in §2.6, it makes updating previously rendered results more efficient.

2.6 Integrating Vertex Clustering and Edge Bundling

So far we developed two progressive network-simplification techniques: one for clustering the vertices in a new batch, and one for bundling the new edges. In this section, we study how to integrate them in GSViz and address related challenges.

We integrate the two techniques in two steps. For the subnetwork in the first batch B_1, the

middleware first uses the edge-aware clustering algorithm in §2.4.2 to group these vertices and generate a set of super edges between the clusters, where a super edge connects the centers of two clusters. It then uses the technique in §2.5.2 to bundle these super edges. Finally, it sends the results, including the clusters and the curved super edges, to the frontend to display. For each new batch B_i, the middleware repeats the aforementioned steps. For simplicity, we denote a super edge as e, and the curved super edge after bundling as $\hat{e}$ throughout this section.

One problem in integrating these two techniques for the batch B_i is the effect of the vertex clustering on those existing super edges computed on earlier batches $B_1, \ldots, B_{i-1}$. There are two cases where these existing super edges can be affected.

2.6.1 Updating Edges Affected by Clustering

Figure 2.6 shows the first case. When we merge two existing clusters, their connected super edges will change. In the figure, after we merge clusters a and c to cluster g and merge clusters b and d to h, we should also delete super edges (a, b) and (c, d), and add a new super edge (g, h). ii) Figure 2.10 illustrates the second case. After adding a new vertex p in B_i to an existing cluster c, the center of c shifts. As a result, the super edge e connected to this center also shifts. In both cases, the changes to the existing super edges should be reflected in both the PEB-tree and the displayed results on the frontend.

Updating PEB-tree. The changes to existing super edges in both cases include edge deletions and edge insertions. In §2.5.3 we already discussed how to insert edges to the tree. To delete an existing edge from the tree,

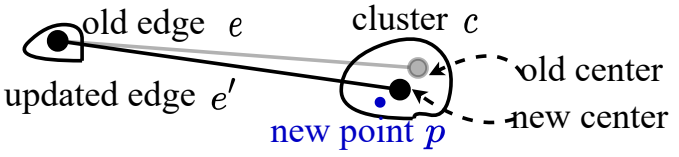


Figure 2.10: Adding a new point in an existing cluster causes the cluster and its related super edge to shift.

we locate the leaf node that represents

the old edge, and delete it from the tree. We store a pointer for each edge to its leaf node. Then, we propagate this deletion upwards and for each node on the path from the leaf to the root, we adjust its weight and edge. For instance, consider the example in Figure 2.11. Suppose e_1 is a super edge before batch B_i. After the vertex-clustering step for the new batch B_i, edge e_1 shifts to e_1'. We delete e_1 from the PEB-tree and insert the new edge e_1' in the tree. If we were to directly update the edge e_1 to the new edge e_1', then the new edge may not be compatible with its siblings. To address this concern, we first delete e_1, then insert e_1' by using the compatibility score as discussed in §2.5 to traverse the PEB-tree. Thus e_1' is still compatible with its new siblings. After we handle the updates of existing super edges, we start progressively inserting the newly generated super edges in the batch, e.g., e_2 in the running example.

Updating visualization results. As these updated super edges are already displayed to the user in the frontend, we also need to "hide" the outdated ones when changes occur. Consider the two approaches to rendering results in the frontend. Approach (i) that re-renders the new results from scratch is not appealing due to its low performance. For approach (ii) that renders new results as a new layer, we still need to identify the layers of those affected edges in order to delete these layers. To know which super edge belongs to which layer, the middleware stores for each super edge its batch number. We use the batch number to identify which layer the frontend has to replace. When updating those affected super edges, the middleware does not need to rebundle other edges. For those affected super edges, the middleware identifies their batches, then sends these batches and notifies the frontend to delete and redraw those corresponding layers. In the running example in Figure 2.11, after batch B_{i-1}, the hash map includes $\langle \mathbf{B_{i-1}} : \hat{e_1} \ldots \rangle$, since the super edge e_1 belongs to batch B_{i-1}. After the vertex-clustering step for batch B_i, the edge e_1 shifts. The middleware identifies its batch B_{i-1}, and notifies the frontend to delete and redraw the

layer for this batch. To reduce the overhead of redrawing multiple layers because of frequent updates in super edges whenever a center of the cluster is shifted, we can optionally relax the updates on existing super edges to be performed only when a vertex of a super edge is located outside the boundary of its corresponding cluster.

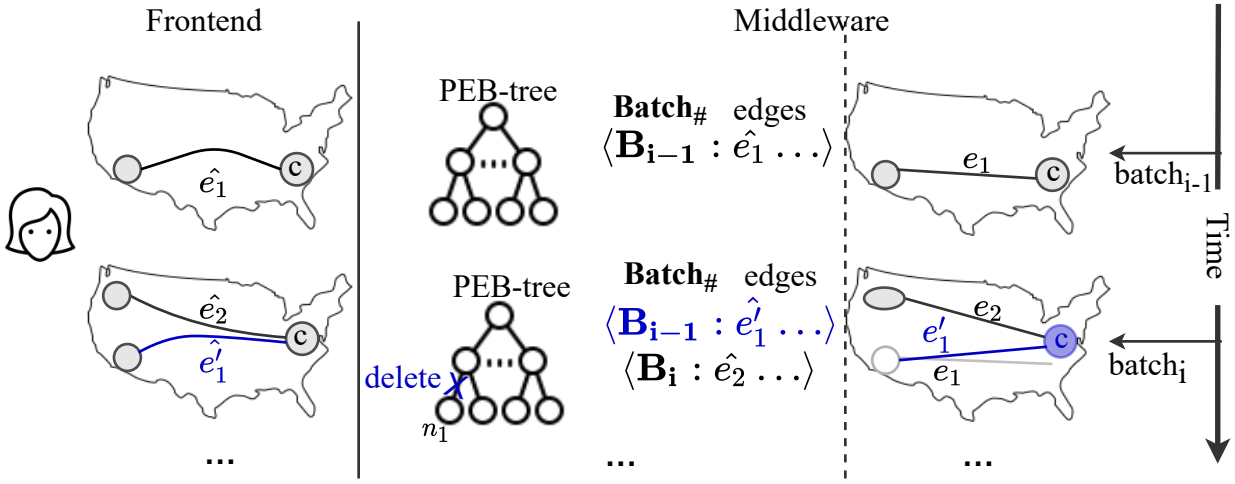


Figure 2.11: Maintaining updated edges affected by vertex clustering in batch B_i.

2.6.2 Supporting Zooming and Panning

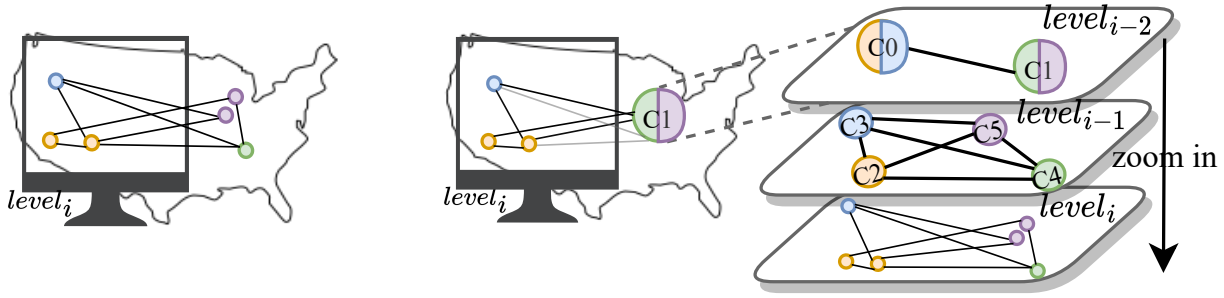
So far we discussed vertex clustering and edge bundling at one zoom level, where the result of both steps is a set of clusters and super edges between the clusters on a queried region. To support efficient zooming and panning operations, GSViz repeats the process of vertex clustering and edge bundling per batch at multiple zoom levels [62] in the background. GSViz maintains a PEB-tree at every level such that the leaves of the PEB-tree at each level represent the super edges between the clusters at that level. If a user wants to zoom or pan on the map, GSViz instantaneously retrieves the computed subnetwork for the particular region from the corresponding level. To reduce the overhead of redundantly finding candidate clusters using the range search at all levels for a particular vertex of an edge during clustering, GSViz maintains a tree data structure to connect the clusters at lower levels with

the corresponding aggregated clusters at upper levels.

2.6.2.1 Reducing Network Edges with a Tree-cut

We now discuss how we leverage the hierarchical structure, which supports zooming, to further reduce the number of edges and, consequently, the visual clutter. Let us consider an abstract view of the visual result of a network after applying the edge-aware clustering algorithm for a particular zoom level, as depicted in Figure 2.12a. In data analytics, users are typically interested in a specific region of the result before shifting their focus to other parts of the data [46]. In the provided example, the user's viewport approximately covers the western side of the US map. By knowing the direction and density of the remaining edges, the analyst can make informed decisions regarding which region to explore next. In this instance, the analyst might be interested in examining the eastern part of the map, as most of the edges are directed towards that region rather than the north or south. With this observation in mind, displaying only a few edges towards the eastern side of the map would suffice to provide a comprehensive overview of the network and guide the analyst accordingly. Notably, the number of edges visible on the screen in this example is 7. It is essential to recall that the primary objective of visual decluttering is to minimize the number of edges displayed on the screen.

We identify a technique that effectively reduces the number of edges by leveraging the hierarchical structure of the clusters. This approach involves selecting network clusters from the zoom level, which the user wishes to inspect within the viewport. At the same time, we consider the network clusters from higher zoom levels that represent abstracted and aggregated points outside the viewport. Revisiting the earlier example, we select the clusters inside the user's viewport from zoom level i while selecting those outside the screen to be from zoom level $i-2$ (namely, cluster c_1), as shown in Figure 2.12b. By employing this method, we can notably observe that the number of edges visible to the user has decreased from 7 to 5. Note



(a) Visual result of a clustered network.

(b) Visual result of a decluttered clustered network with clusters from different zoom levels.

Figure 2.12: Abstract example to show usage of clusters from different zoom levels to reduce the number of edges.

that this reduction in the number of edges is particularly significant when dealing with large networks, as depicted in Figure 2.15, where the number of the network edges reduced from 3,651 to 390. This technique primarily revolves around a careful selection of the network clusters to minimize the number of visible edges. We formally model this approach as a tree-cut problem, as we will detail in the subsequent section.

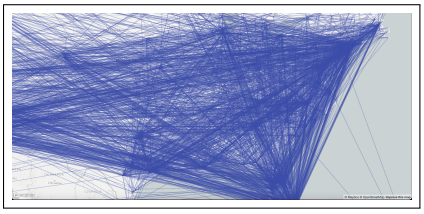


Figure 2.13: A geo-social network of tweets containing the keyword "vaccine" with 3,651 edges.

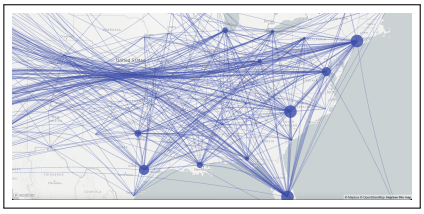


Figure 2.14: Visual result of the network after applying GSViz's techniques resulting in 473 edges.

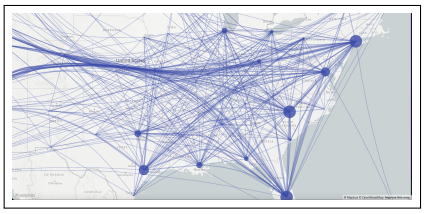


Figure 2.15: Visual result after applying GSViz's "tree-cut" approach to reduce the number of edges to 390.

Choice of clusters. A primary question we face is which clusters to choose. For example, why did we replace the points outside the viewport in the abstract network example presented in Figure 2.12b with clusters from $level_{i-2}$ instead of those in $level_{i-1}$? The objective is to select network clusters from the highest (abstracted) possible levels, as long as they do not conflict with those within the viewport. In Figure 2.16, we provide a sample hierarchy of

clusters to explain the definition of a conflict between the clusters. Each node in the hierarchy represents a cluster, and the links indicate the parent relation from the upper zoom level. We omit the details of the network edges between the clusters, in the figure, for simplicity. In the case of cluster 'a', it shares the same parent as the clusters within the screen's viewport, which means we cannot choose its parent as the aggregated cluster. Likewise, with cluster 'b', since its parent serves as an ancestor of the clusters within the viewport, we cannot select its parent.

Process of selecting clusters. When the user zooms into a specific region on the map or pans over a particular location, we determine which clusters and associated edges to display. Starting from the zoom level of the viewport, we select all of the network clusters within the viewport. Next, for each cluster outside the viewport at the same level, we traverse in a bottom-up manner to choose its ancestors. For each cluster in the current zoom level, we examine its highest ancestor that can be chosen. We perform a test to verify if the ancestor is also an ancestor of the clusters within the viewport. Once we include an ancestor in the set of chosen clusters, we cut off the path to all of its descendants and remove sibling clusters to avoid redundant computations in the future.

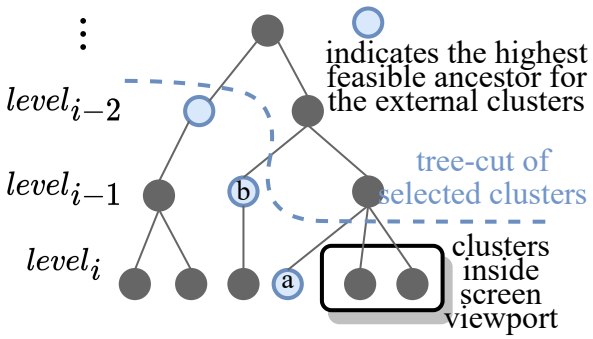


Figure 2.16: Sample clusters hierarchy and the highest ancestors of choice (highlighted in blue) for those clusters outside the screen's viewport.

Notice that this process implies that the choice of clusters, and consequently the edge bundles, are only computed and known after the user pans or zooms a particular region on the map, unlike the previous approach we discussed in Section 2.6.2. We trade-off between the two discussed approaches to achieve efficiency or aesthetic of the visual representation.

2.7 Experiments

In this section, we report an experimental evaluation of GSViz[1] using real datasets to answer the following questions. (1) How does edge-aware clustering perform (§2.7.2)? (2) How does the incremental edge bundling using PEB-tree perform (§2.7.3)? (3) How does GSViz perform when integrating the techniques and how does it compare to similar systems (§2.7.4)? (4) How is the quality of the final visual result perceived by users (§2.7.5)?.

2.7.1 Experiment Setting

We used three real geo-social network datasets as shown in Table 2.2. Gowalla [36] represents users' geotagged check-ins to places and their social friendship between early 2009 and late 2010. Foursquare [125] represents a social network between geo-tagged users collected from late 2011 till early 2012 in the US. We generated a random timestamp for every tuple and used it to specify a slicing predicate to query the data progressively. Twitter includes tweets and their replies collected from late 2015 until February 2021.

Table 2.2: Datasets.

Dataset	Content	Vertex #	Edge #	Size (GB)
Gowalla	Users' checkins and their social relation	99,563	913,660	0.12
Foursquare	Users' location and their social relation	28,419	7,176,141	2.5
Twitter	Interaction between Twitter users by replies	33,677,670	20,023,731	8.6

We developed GSViz in Java. Additionally, to evaluate the developed algorithm of Progressive Edge-aware Clustering (PEAC) in §2.4.2, we implemented a greedy incremental version of Supercluster [62] called "Hierarchical Greedy Clustering" (HGC) in the middleware as "Baseline". Similarly, we implemented non-incremental FDEB [64] as the baseline to evaluate the Incremental Edge Bundling (PEB) in §2.5.2. We used two approaches for slicing a query into multiple mini-queries using the time predicate. The first one is called fixed-interval,

[1] GSViz is available on Github (`https://github.com/sadeemsaleh/gsviz`)

which slices a query into equi-size time intervals. The other strategy is called DRUM [75], which slices a query into dynamic range intervals using a linear regression model to maintain the same running time from the database for each mini-query. Unless otherwise stated, the rhythm in DRUM was set to 500ms.

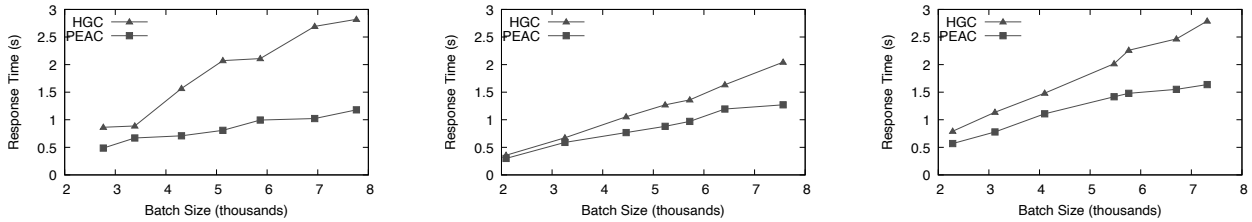
We ran the experiments on a 64-bit JVM on the Ubuntu 14 operating system on a machine with an Intel Xeon CPU, 98 GB of RAM, and a 2-TB hard disk. The data was stored in PostgreSQL 11.3 on the same machine. We built a B-tree index on the time attribute on which we specified the slicing predicate. Additionally, we built an inverted index on the text attribute on Twitter. We used keywords with different selectivity values to filter Twitter's "text". The Foursquare and Gowalla datasets did not have a text attribute, so we used the user-ID to specify range conditions. Each reported result is the average of three runs. We used a query that resulted in around $100K$ edges in total unless otherwise stated. To evaluate the quality of visualization on different zoom levels, we used levels ranging from 3 that showed north and central America to 7 that showed details of a US city. Experiments in (§2.7.4) and (§2.7.5) are only done on the largest dataset Twitter.

2.7.2 Progressive Vertex Clustering

Effect of batch size on clustering performance. We evaluated the performance of PEAC against the baseline HGC. We measured the effect of varying the batch size on the performance of clustering vertices of a subnetwork in a new batch. We used the DRUM approach for slicing the query to keep a constant running time and similar batch sizes. We took the average of the batch size over all the batches from three runs. We varied the rhythm in DRUM between 1 second to 3 seconds.

Figure 2.17 shows the average response time of all batches for one batch size. Both HGC and PEAC had a sub-second response time when the batch had around $2K$ edges. As the batch

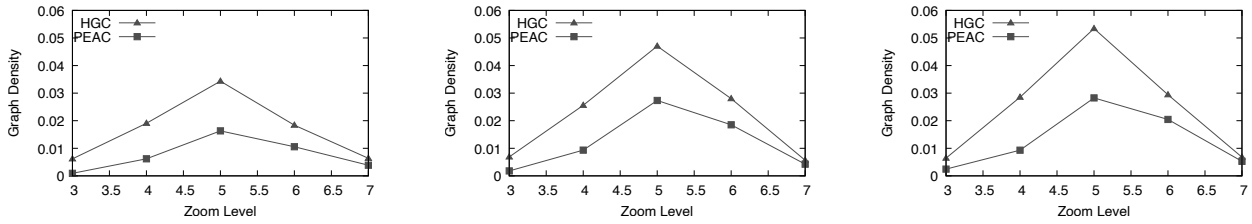
size increased to $7K$ edges, HGC's response time increased to 2.8 seconds while PEAC's time was within 1.6 seconds. The reason PEAC's response time increased at a slower rate compared to HGC was due to the benefit of applying the grid-based technique discussed in §2.4.3 on PEAC to cluster the edges in the batch. Hence its performance was proportional to the batch size only, whereas HGC's performance was additionally affected by the neighboring clusters per vertex.



(a) Gowalla. (b) Foursquare. (c) Twitter.

Figure 2.17: Time of vertex clustering per batch (fixed batch size using DRUM).

Effect of edge-aware clustering on graph density. We evaluated PEAC's reduction on the number of super edges compared to HGC. We used graph density [103], which is measured as the number of edges over the number of possible edges between the vertices in the graph. In our setting, we used the number of super edges that resulted from the clustering over the number of raw edges, i.e., $\frac{\#\text{ of super edges}}{\#\text{ of raw edges}}$. Figure 2.18 shows the results of the network's density for different zoom levels. For the Foursquare dataset, on zoom level 4, HGC resulted in $2,558$ super edges, while PEAC significantly reduced the number to as low as 934. The graph density of PEAC was more noticeable at zoom levels 4, 5, and 6. At zoom level 3, the range radius ρ was large and it resulted in aggregating the network to include only a few clusters, which led to only a few super edges connecting them in both HGC and PEAC. Zoom level 7 had only a few clusters due to the small number of vertices in the small area of a city. As a result, both HGC and PEAC had few super edges.



(a) Gowalla. **(b)** Foursquare. **(c)** Twitter.

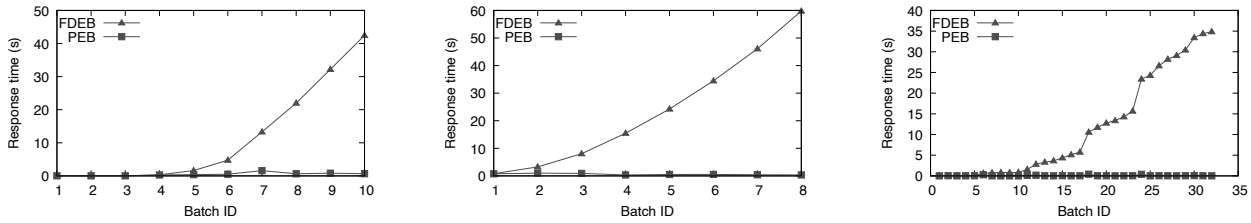
Figure 2.18: Graph density on different zoom levels (fixed batch size using DRUM**).**

2.7.3 Progressive Edge Bundling

We evaluated the performance of the non-incremental baseline FDEB and our incremental approach PEB. Since FDEB runs the bundling algorithm in each batch on the accumulated result of previous batches, we made sure the result of a batch is the same across different runs. In order to do that, we used the fixed-interval slicing approach. As there is no common way to quantitatively measure the edge-bundling quality [128, 105, 160], we measured the quality of the visual result of the entire network in (§2.7.5).

Bundling performance in different batches. Since the performance of FDEB was very slow, we chose a query that resulted in a small network with $10k$ edges in total. Figure 2.19 shows the performance results for different batches, each with range of two months. Since the time domain of the Twitter dataset was larger than the other two datasets, its number of batches used to slice the time range was higher. The results show that, not surprisingly, FDEB's performance decreased in later batches, resulting in response time as high as a minute. On the other hand, PEB had an average response time within 200ms on every batch. The Foursquare dataset had more edges compatible to each other, which resulted in a lower performance for FDEB (up to a minute) compared to the performance on Twitter (half a minute). PEB was not affected by this issue since the technique only uses one edge to bundle another edge. PEB showed a small spike in the early batches, of up to a second, due to the special case of constructing the PEB-tree in the first batch using the baseline

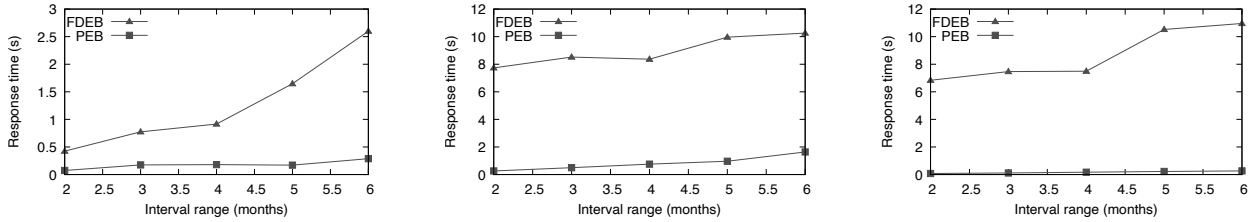
FDEB. PEB had a spike of a second on the 7^{th} batch using Gowalla dataset because that batch contained most of the edges of the query.



(a) Gowalla. (b) Foursquare. (c) Twitter.

Figure 2.19: Time of edge bundling for different batches.

Effect of interval size on bundling performance. We evaluated the effect of varying the interval range size on the performance of FDEB and PEB. As the slicing interval increased, the number of edges per batch increased as well. We used a query that generated around $3K$ edges in total. Figure 2.20 shows the average response time of edge bundling per batch using FDEB and PEB. When the range was two months in both Twitter and Foursquare, FDEB's response time was about 7.3 seconds, whereas PEB's response time was about 100 ms. When the slicing interval was 6 months, FDEB's response time increased to more than 10 seconds, while PEB's response time was only 1.6 seconds. FDEB's performance was better on Gowalla than its performance on Twitter and Foursquare datasets because most the edges were not compatible. We note that the response time of PEB on all datasets was mostly affected by the first batch when we used the baseline to construct the PEB-tree.

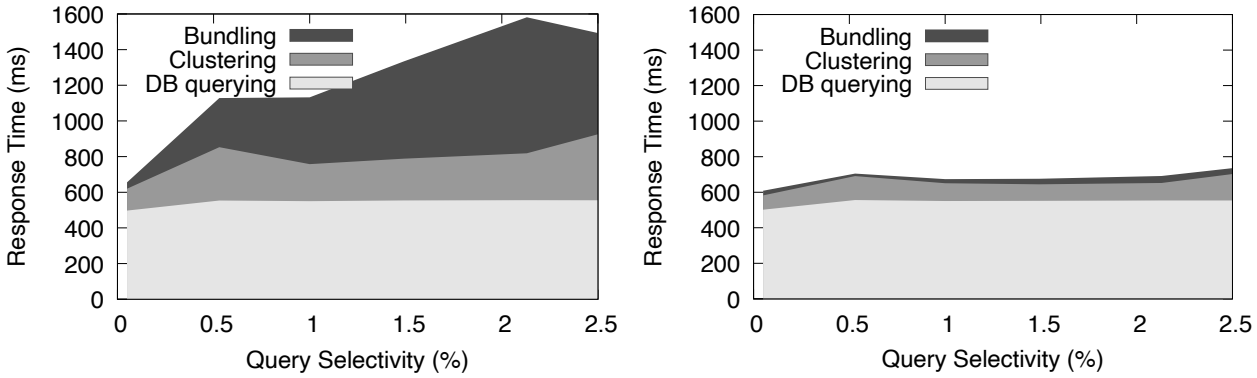


(a) Gowalla. (b) Foursquare. (c) Twitter.

Figure 2.20: Bundling time per batch for different slicing intervals.

2.7.4 Integrating Both Techniques

We evaluated the performance of integrating both vertex clustering and edge bundling during the whole lifecycle of a visualization request in GSViz, including querying the database, clustering vertices, and bundling edges. We considered a baseline approach that used HGC for vertex clustering and FDEB for edge bundling. We then considered GSViz's approach that used PEAC for vertex clustering and PEB for edge bundling. We used the Twitter dataset and varied the keyword selectivity from 0.05% to 2.5%, resulting in a network that consisted of 10K to 500K raw edges. The number of super edges after clustering ranged from 1K to 3K. Figure 2.21 shows the average response time per batch for different keyword selectivity values. GSViz had a stable response time of 600 ms per batch regardless of the network size because the batch size was almost the same for each mini-query. The baseline's performance, on the other hand, increased from 655 ms to 1.6 seconds when the network size increased. This increase was because FDEB re-bundled the network edges from scratch for each batch.

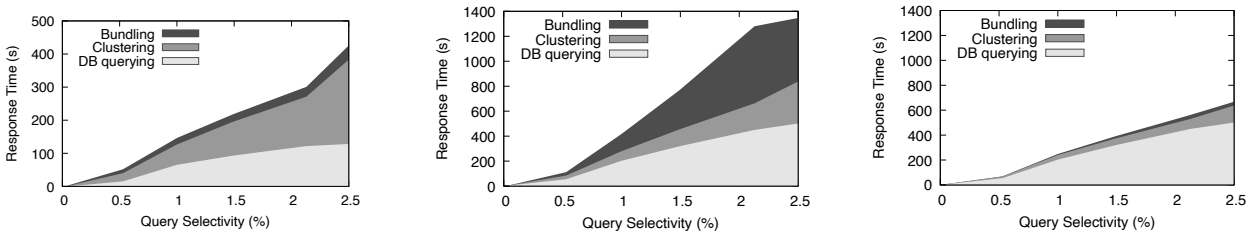


(a) Baseline. **(b) GSViz.**

Figure 2.21: Average response time per batch of all steps.

Total visualization time. To show the total time of visualizing the network across all the steps, we collected the total time it took to issue a single query to the database and process the data in a single batch. We call this method *non-progressive*. Figure 2.22a shows the performance for keyword conditions with different selectivity values. As the network size increased, the total time increased mostly to query the database and to cluster the vertices

up to 7 minutes, where the user waits in the dark not knowing if the request was successful or not. Conversely, Figures 2.22b and 2.22c show the total response time of all the *batches* when the computation is progressive. As the network size increased, the total time increased due to the increase in the number of batches. GSViz's total time was usually half the time of the baseline to visualize the entire network. The baseline took about 23 minutes to query the database, cluster the vertices, and bundle the super edges on a network of 500K edges. GSViz took 10 minutes to show the same network.

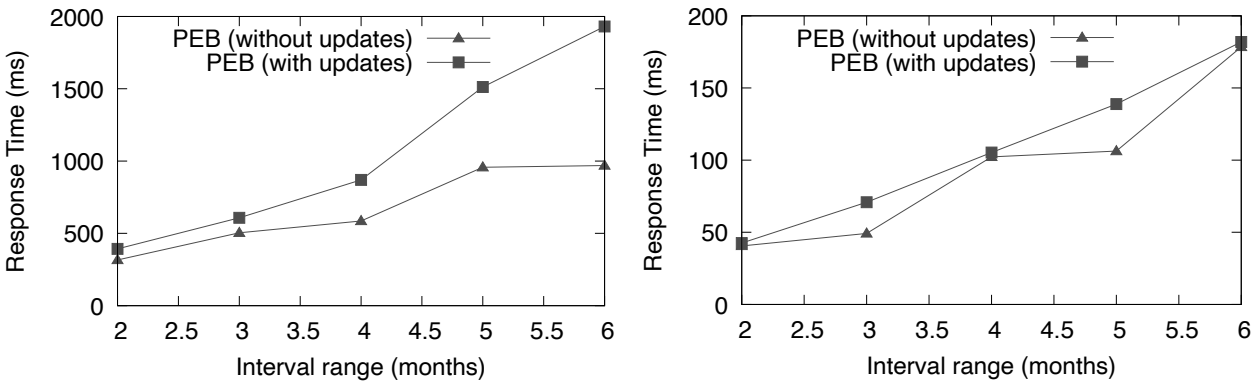


(a) non-progressive. **(b) Baseline.** **(c) GSViz.**

Figure 2.22: Total response time of all the steps.

Effect of updating clusters on bundling performance. We compared the performance of updating the super edges with the performance of not updating them. To make sure the batches are the same across different runs, we used the fixed-interval slicing method. Figure 2.23 shows the performance on different interval sizes. Foursquare's total date range was a few months. When the interval range was 6 months, the number of batches was three. As a result, the size of the first batch contained many raw edges and their vertices were grouped to many clusters. In the second and third batches, almost all the existing clusters were shifted due to insertions of new points, and the resulting response time was 1.9 seconds. The Twitter experiment achieved a time that ranged from 50 ms to 200 ms when the interval range increased from 2 months to 6 months, respectively. The results show that the overhead of updating existing super edges was low.

Comparison with existing systems. We compared GSViz with two existing popular graph visualization systems, namely Tableau [133] (version 2021.2) and Tulip [7] (version



(a) Foursquare. **(b)** Twitter.

Figure 2.23: Average time per batch to update super edges for different intervals.

5.5.1). We chose Tableau due to its capability of doing middleware-based visualizations. We chose Tulip since it supports edge bundling. As these two systems could not be installed on Ubuntu 14 OS, we used a machine that supported all three systems. It had an Intel Core i5, 8GB RAM, and a 500GB hard disk, running MacOS 10.15.7 and PostgreSQL 12.5. Tableau and Tulip were not open-source, hence we used a stopwatch to measure the end-to-end performance of all three systems to visualize the network. We used database logging to measure the database query time. Tulip was an in-memory solution, and the largest network it could load had more than $900K$ nodes and $500K$ edges with a file size of $63MB$. It filtered the tweets using a keyword that resulted in a network of $10K$ edges, and this step took 11.55 seconds. It took additional 74.4 seconds to do edge bundling.

Tableau and GSViz support middleware-based visualization using a database, so we used the Twitter dataset. We filtered the network on a keyword condition that resulted in more than $200K$ edges. Tableau visualized the network on a map without any simplification. It took 19.49 seconds, including 15.81 seconds for querying the database. While its results were retrieved efficiently, the user had to wait for a long time before seeing any results. Moreover, the network clutter significantly hindered the user experience. On the contrary, GSViz retrieved the results progressively in 39 batches, each within 500 ms. The total time was 71.00 seconds, including 54.52 seconds for all the mini-queries, and 16.48 seconds for the

steps of vertex clustering and edge bundling.

2.7.5 A User Study

We conducted a study to evaluate the user experience in GSViz. We mainly considered two methods: ❶ a baseline method using *non-incremental* HGC and FDEB to show its best visual quality, and ❷ GSViz using incremental PEAC and PEB. The goal of the user study is to answer the following question: "How do the two methods differ in terms of visualization quality?"

We invited 29 users, and each spent about 15 minutes to complete it. We generated 12 different sets from variations of 3 network sizes using different keywords at 4 zoom levels. The size of the network ranged from 10K to 100K raw edges. The zoom level ranged from an overview of North and Central America to a level of a few states in the US. Each set had 3 different methods, resulting in a total of 36 images. We first showed the result of the original network as is. We then showed the visual result yielded from the baseline and GSViz presented anonymously to the participants. To make the comparison fair for the baseline, we fixed the number of clusters and asked questions independently.

We used two metrics to measure the visualization quality:

1. *Readability* [105], which indicates how easy it is to read the visualized network. To measure the readability, we asked the participants to subjectively answer a question for each simplified network: *"Q: Rate how cluttered you think the network is."* . The answer is a rating score of 1 (very cluttered) up to 5 (very sparse).
2. *Task faithfulness* [105], which indicates how accurate the visualization of the simplified network is to correctly perform tasks. To measure the faithfulness, we asked the participants to answer analytical multiple-choice questions, each of which had 4 choices with

only one correct answer. A score of 0 means the network is unfaithful and a score of 1 indicates a very faithful network [105]. All of the questions had the following template: *"Q: Which of the points, highlighted with green boxes, has more original tweets compared to reply-to tweets?"*

Figure 2.24 shows sample images given to the users including labels to indicate the randomly chosen clusters in the questions.

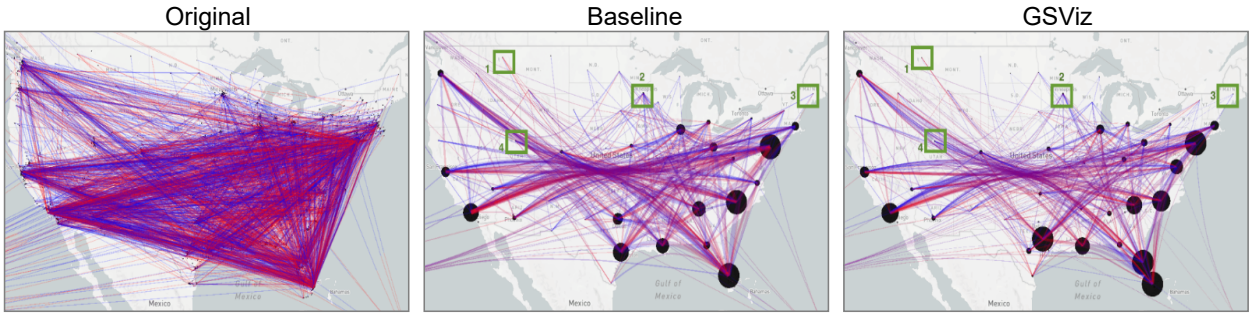


Figure 2.24: Example network visualizations in the user study at one zoom level.

Table 2.3: User study results. The reported numbers for the visualization quality are represented as "A—B," where "A" is the average score given by all the participants and "B" is the standard deviation. Compared to the baseline, GSViz was much more efficient and had comparable visualization quality.

Network Size	Time per batch (ms)		Visualization Quality			
			Readability		Faithfulness	
	baseline	GSViz	baseline	GSViz	baseline	GSViz
10K	648.70	525.20	2.59—0.11	3.28—0.26	0.82—0.11	0.82—0.09
50K	2,603.54	635.42	2.15—0.17	2.85—0.13	0.53—0.25	0.63—0.33
100K	10,165.61	733.67	2.09—0.23	2.48—0.21	0.75—0.32	0.79—0.36

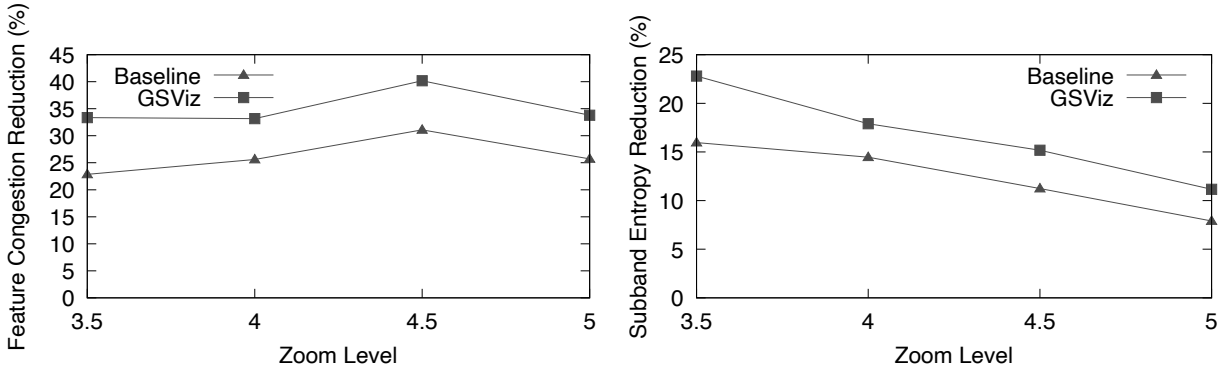
The results are shown in Table 2.3. The average response time using Baseline was 4,473ms while GSViz was 631ms showing much higher performance. As the network size increased, the time difference also increased. This increase was more noticeable in the baseline. The average readability rating of Baseline was 2.3, which means the network was perceived as cluttered. GSViz's average readability rating was 2.9, which indicates that the network was perceived as not cluttered nor sparse. The average faithfulness score of Baseline was 0.70, it

means that the network was faithful. GSViz had a better average faithfulness score of 0.74. The user study showed that, compared to the Baseline, GSViz had much higher performance and produced visualization with comparable quality.

2.7.6 Reduction of Visual Clutter.

We report the visual quality of the two methods used in the user study across the different zoom levels. We used two common metrics to measure the visual display clutter, namely "feature congestion" and "subband entropy" [121].

Figure 2.25 shows the percentage reduction on the visual clutter score using both metrics. The higher the percentage, means the reduction was more. Figure 2.25a shows that the baseline and GSViz on average reduced the clutter score by 26% and 35%, respectively. Figure 2.25b shows that the baseline reduced the subband entropy clutter score by 15% at zoom level 3.5 (which showed the entire US), and 7% at zoom level 5 (which showed cities). GSViz reduced the score by 22% and 11%, respectively. We observe that both methods reduced the visual clutter score compared to the original graph, and GSViz achieved a better reduction. This finding was consistent with the readability result in the user study.



(a) Feature Congestion. **(b) Subband Entropy.**

Figure 2.25: Reduction of visual clutter score compared to the original network.

2.8 Conclusion

In this chapter, we presented GSViz, a system to enable progressive visualization of geo-social networks. We first demonstrated how to improve incremental spatial clustering to make it edge-aware to reduce visual clutter. We also studied supporting incremental edge bundling by storing previous edges as nodes in a novel tree index called PEB-tree to optimize the traversal and processing of bundling edges. Moreover, we discussed the integration of the two techniques and solved new challenges. Lastly, we conducted an extensive evaluation of GSViz compared to baseline algorithms for spatial clustering and edge bundling. The experiments also included a user study to evaluate the quality of the produced visualization. The results showed that the techniques can not only support efficient, responsive visualization of networks progressively but also produce high-quality simplified network visualization.

Chapter 3

Drove: Tracking Data-Processing Versions and Executions to Facilitate Reproducibility

3.1 Introduction

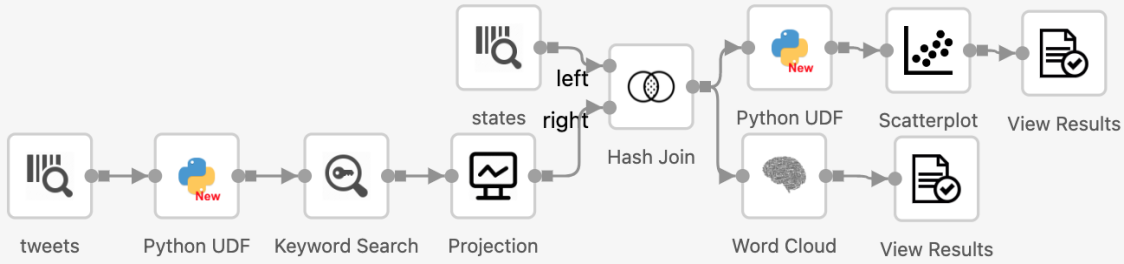
Data analytics is an iterative process. An analyst constructs a workflow, executes it to observe results, and refines it to achieve better results or gain deeper insights. As the number of iterations grows, information about the past executions is valuable, and storing such information can be helpful in many ways [123, 9]. First, the analyst can use it to avoid repeating the mistakes made in the previous executions and decide the next iteration of changes. Second, the past executions can be shared with other analysts to *repeat* and *reproduce* a particular execution of choice. Repeatability means performing a new execution by understanding the details of how a previous execution was conducted [123]. Reproducibility means performing an execution by re-running a script, which is expected to lead to the same results.

As an example, suppose an analyst is interested in analyzing tweets containing the keyword "`climate`." Figure 3.1 shows the workflow created by the analyst to achieve the task. After running the initial workflow (3.1a), the analyst notices that "`fire`" is the most frequent keyword in the output of the Wordcloud operator. Based on this observation, she revises the workflow to a new version (3.1b), to inspect the tweets that are from California, based on the assumption that California has many of the wildfires. To her surprise, the number of tweets from California was not large. Then she refines the workflow again, and constructs a new version (3.1c) to compute the count of climate-related tweets from each state. Surprisingly, the number of tweets from California in the result of version 3.1c was different from what she remembered seeing in the result of version 3.1b.

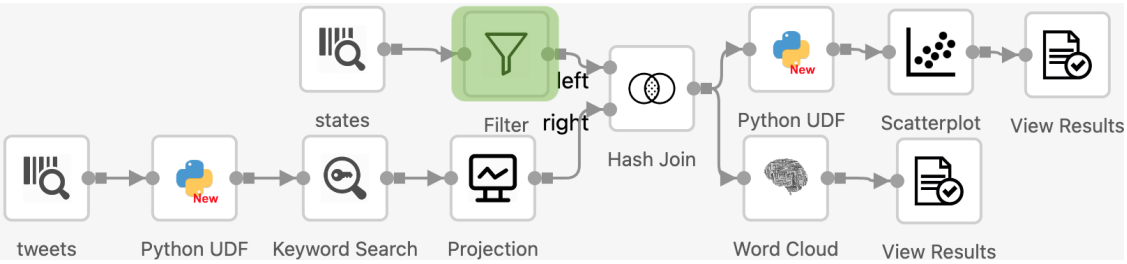
Importance of tracking executions. A natural question is what could have caused the difference between the number of California tweets in the results of versions b and c. There can be various reasons for unexpected results in data analytics, especially in a collaborative environment. Following are a few possible reasons.

- Case 1 (change in the runtime environment): Between versions b and c, a new deployment of the system was done to use a new csv-parsing library in the CSV Scan operator. The new operator parsed some tweets erroneously, leading to a different number of California tweets.
- Case 2 (change in the workflow): A collaborator changed the search predicate between versions b and c.
- Case 3 (lost previous version's results): The analyst may not remember the result of version b.

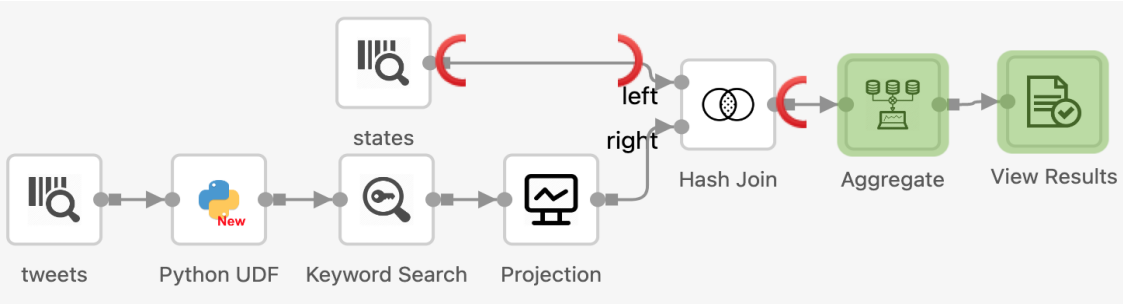
The difference between the results of the two executions raises doubts about the integrity of the analysis. A way to avoid doubt and maintain integrity is to keep track of these executions and all the associated factors that affect them in a ledger.



(a) Version a: Initial workflow to analyze tweets containing "`climate`".



(b) Version b: Adding a filter operator (highlighted in green) to filter the tweets from California.



(c) Version c: Removing the filter and adding an aggregate operator to show the count of tweets from each state. The red arches indicate the start and end of the removed operators.

Figure 3.1: Multiple versions of a workflow for tweet analysis.

There has been an increased interest in tracking executions and results of data-processing tasks [98, 143, 95]. Many of these solutions are geared towards Machine Learning (ML)-based analytics, which is often carried out using a programming interface, such as Jupyter Notebook. They use Git-like techniques to track versions of the code, ML model, hyper-parameters, and package dependencies. This is not directly applicable to GUI-based workflow systems. A reason is that constructing and refining workflows is an ad-hoc process [123], wich makes tracking its evolution difficult. In addition, workflow execution depends on many aspects, such as the input data sources, the workflow version, or the runtime dependencies [123]. Moreover, many usability changes (e.g., changing the position of an operator,

or annotating and commenting on the workflow canvas) do not have any direct implications on the execution of the workflow.

In this chapter, we showcase how we can automatically track runtime environment changes (Section 3.4), workflow versions (Section 3.5), and executions (Section 3.6) to facilitate reproducibility in Texera. We will also show how a user can do basic bookkeeping of executions using an execution dashboard as shown in Figure 3.2, which provides various functionalities, such as viewing, renaming, sorting, or filtering the executions. The dashboard also provides charts to help summarize the statistics of the executions.

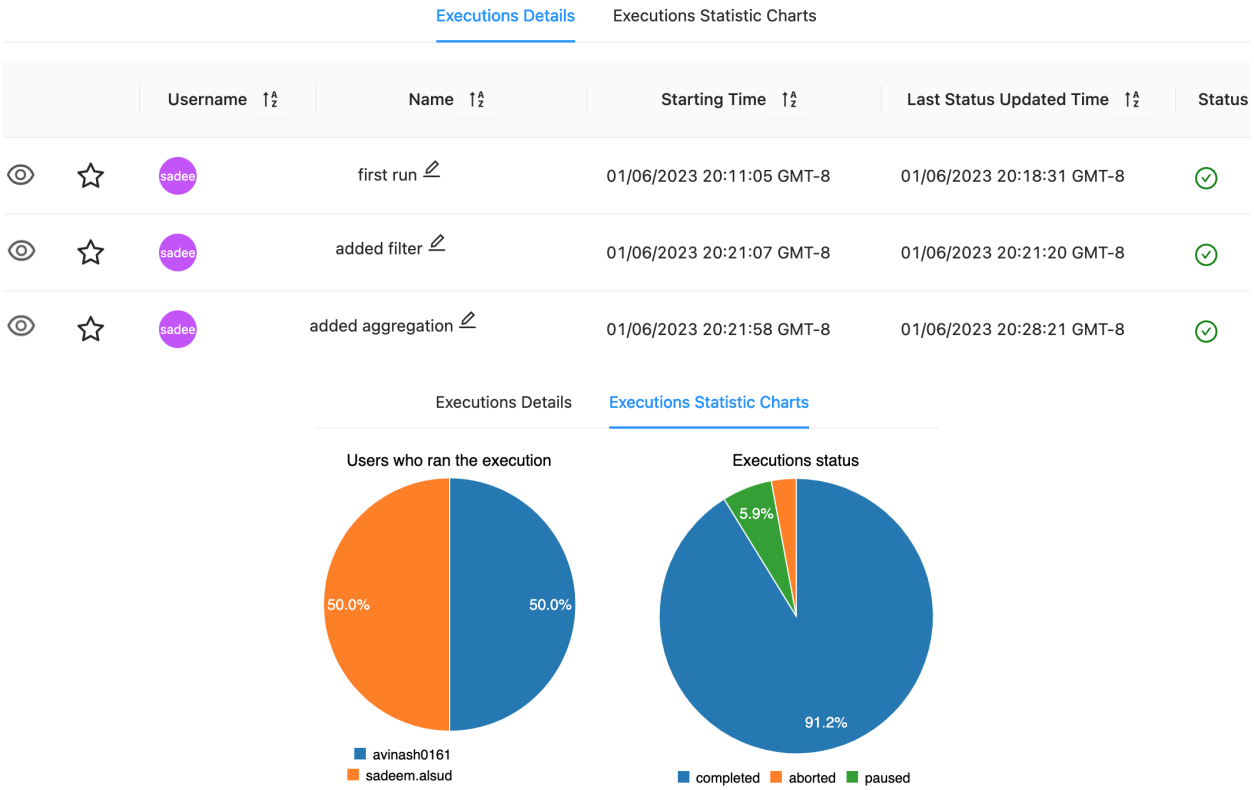


Figure 3.2: Interface of the executions dashboard in Texera. Due to space considerations, we do not show the rest of the table, which includes each execution's workflow version number, sample results, deletion button, etc.

3.2 Related Work

There has been an increasing interest in enabling the reproducibility of data analytics pipelines. These tools track the evolution and versioning of datasets, models, and results. At a high level they can be classified as two categories. The first includes those that track experiment results of different versions of ML models and the corresponding hyper-parameters [159, 31, 143, 80, 54, 100]. ML-based solutions [143, 95] for tracking executions are suited for tech-savvy analysts as these solutions require them to install packages and use API calls in the analytical task to enable tracking the executions, thus they are not suitable for users who use GUI-based workflow analytical systems. The second category includes solutions to track results of different versions of data-processing workflows [99, 150, 42, 9]. These workflow-based solutions [150, 98] track the result of executions using provenance, and they do not track the version of the runtime environment. ReproZip [34] is a more general tool used for reproducing an experiment in Linux. This tool requires users to be able to know Linux commands. Moreover, it only generates the final result of an experiment and does not show the historical sequence of constructing and executing a workflow.

Table 3.1: A comparison of related tools and systems that track evolution of data analytic tasks

System	**Task Formulation**	**Usage of Enabling Tracking**	**Tracking**			
			Version	*Experiment*	*Environment*	*Result*
ModelDB	Notebook	Python package calls	Yes	Yes	No	Yes
ProvDB	Notebook	Unix commands or Python package calls	Yes	Yes	No	Yes
Reprozip	Code files	Linux commands	No	Yes	No	Yes
MLcask	Notebook	Script run	Yes	No	No	No
Sacred	Notebook	Python package calls	Yes	Yes	No	No
MlFlow	Notebook	Python package calls	Yes	Yes	No	Yes
Texera	**GUI-based workflows**	**Automatic and transparent**	**Yes**	**Yes**	**Yes**	**Yes**

Compared to these efforts, Drove is the first open-source tool that tracks the evolution of data-processing workflow versions, runtime environment dependencies, and executions of the

different versions. It does the tracking *automatically* and *transparently*. We say automatically to indicate that once the tracking is enabled, all the necessary resources are tracked without any manual intervention from the user. We say transparently to indicate that the tracking is happening behind the scene and the user does need to know about the details of what resources need to be tracked. Table 3.1 summarizes the differences between Drove and existing solutions.

3.3 Drove: Overview

Figure 3.3 depicts an overview of the modules that track workflow versions and executions. The Environment Version Manager manages the tracking of the *engine* version, i.e., code-related changes to the source code of the system, e.g., changes to the frontend and the engine. It also tracks *operator* versions, i.e., changes in the logic of operators, such as adding a capability to the Project operator to rename the projected columns. Information about the environment versions is stored in a catalog. The environment related versions are computed once after deployment and stored in a catalog. The Workflow Version Manager manages the tracking of the workflow versions, each of which is stored as an entry in a "versions" table in the catalog. This module is also responsible for retrieving and displaying a list of past versions of a particular workflow on the GUI. It manages various operations related to a version, such as checking-out a version, reverting to a previous version, comparing and highlighting the differences between two versions. The Execution Manager tracks every execution

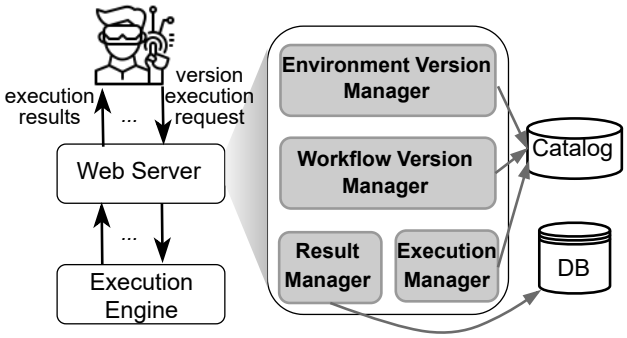


Figure 3.3: Overview of tracking workflow versions and executions.

of a workflow during the workflow's lifecycle. It stores the information about every execution in a table in the catalog. Each execution entry contains metadata information such as: 1) the workflow version number, 2) the engine version number, 3) the user who submitted the execution, 4) the starting time of the execution, 5) the duration of the execution, and 6) pointers to the artifacts, such as the final results and the counts of input and output tuples for each operator. Additionally, Texera supports interaction during the execution of a workflow such as pausing, resuming, debugging, and aborting a workflow during its execution. Hence the workflow execution entry contains the status of the execution and is updated periodically. The Result Manager manages the artifacts of executions including storage of the results, and the statistics of each operator during its execution.

3.4 Version Control of the Runtime Environment

We use the Git hash ID of each commit of the system's source code to track the version of the runtime environment. During each deployment, the system takes the following steps to capture runtime environment changes. First, it checks the latest Git commit ID of the code used in the deployment and compares it against the previously persisted one in the catalog. If the ID is different, we update the version stored in the catalog to reflect the latest one. We use the JGit API [47] to get the Git log, which shows the history of the changes. The second step is to check the operators affected by the latest commit. We do that by verifying if the commit includes any changes on the files related to any of the operators. In the codebase of the system, every operator is implemented in a separate file with the same name as the operator. If the logic of an operator is changed as part of the latest commit, we update the version of the operator in the catalog to reflect the latest commit ID. The user can view the version of every operator in its properties, as highlighted in Figure 3.4.

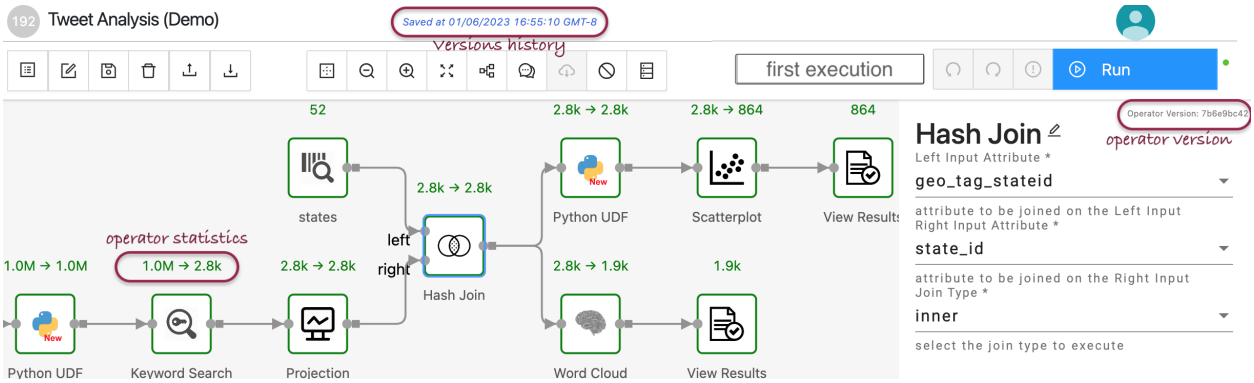


Figure 3.4: Example to show the components associated with a workflow that need to be tracked. Due to space considerations, we omit the parts of the window that show the results of the workflow execution and the engine version.

3.5 Version Control of a Workflow

A workflow undergoes a series of refinements as part of the iterative and exploratory process of data analytics. These refinements can be categorized as follows:

- **Semantic workflow changes:** which include addition of a new operator or a link, removal of an existing operator or a link, modification of the properties of an operator (e.g., changing the predicate condition of a Select operator), and alteration on the input data source (e.g., choosing a different data source as an input). Tracking the changes within a particular data source, e.g., stream data sources or versioned data stores is out of the scope of this chapter.
- **Non-semantic workflow changes:** which include addition, modification, or removal of a user comment on the editor pane, modification of the position of an operator or a link on the editor pane.
- **Operator version changes:** which include runtime environment changes that directly affect the workflow, e.g., changes in the logic of an operator.

Given a workflow version v_i, one or more of the above changes cause it to evolve into a new version v_{i+1}. Since the second category does not make any semantic difference in the

workflow but a user would like to persist their changes, we annotate those changes to indicate that they are non-semantic changes.

In contrast to writing code, designing a workflow through the intuitive process of dragging and dropping operators offers a user-friendly experience that involves numerous ad-hoc steps [123]. To alleviate the burden on users and avoid overwhelming them with the task of manually deciding when to commit a version, Drove takes an approach that automatically and transparently tracks all user changes. This automated tracking frees users from the cumbersome process of manual tracking. However, a question arises: How does Drove determine what group of changes correspond to a new version? We will address this question in the following discussion.

3.5.1 Tracking Workflow Versions

Now we discuss two events that trigger creating and persisting of a new workflow version.

1. **User changes:** A user can perform one or multiple edits from the first two categories mentioned above. Since we want the tracking to be seamless and transparent to the users to lessen their burden of explicitly saving versions, our approach follows a time-based approach of periodically checking for the user's modifications (e.g., every 300 ms) and persisting the updated workflow version if any modifications are found.
2. **Operator version changes:** When the user loads a workflow after a new deployment, we check the runtime environment versions table to detect any change in the versions of the operators being used in the workflow. In case of a change, the properties panel of the operator is updated to reflect the new operator version (Figure 3.4) and a new version of the workflow is persisted.

Version storage. We store the metadata information and the version information of the

workflows in two tables in the catalog. Each version entry contains a foreign key of the workflow it belongs to. One way to store every newly committed version is to store it as-is in a separate entry. This is straightforward but can quickly explode the storage space because there can be many versions and each version is a complex workflow. The approach we take is to store a light-weight delta or the difference between a pair of consequent versions. This approach is similar to code version control and data version control (DVC) [107]. We leverage existing tools to efficiently compute the difference between two workflow versions. Since we model a workflow as a JSON document, similar to many other workflow systems [150], we use the JSON diff standard RFC-6902 [73] to compute a patch. This patch contains the operations that can be applied to a version to transform it to the other version. Listing 3.5.1 shows a sample of the JSON file corresponding to the first version from the running example. Listing 3.5.1 shows a sample of the patch file for adding the Filter operator to the first version to transform it to the second version in the running example. Note that the patch includes the operation of removing the filter opposite to the actual operation of adding the filter. The reason is to be able to construct the previous version by applying the patch to the latest version as we will explain in details next.

Listing 3.1: A sample JSON to represent the first workflow version in the running example.

```
{
   "operators":[
      {"operatorID":"CSVFileScan-operator-...2a0",
      "operatorType":"CSVFileScan",
      "operatorVersion":"c14...503",
      "operatorProperties":{
      "customDelimiter":",",
      "fileName":"states", ...},
      "inputPorts":[ ],
```

```
10        "outputPorts":[{"portID":"output-0", ...}],
11        ...},
12        {"operatorID":"HashJoin-operator-...7d7",
13           "operatorType":"HashJoin",
14           "operatorVersion":"7f2...94b",
15           "operatorProperties":{
16              "joinType":"inner",
17              "buildAttributeName":"geo_tag_stateid",
18              "probeAttributeName":"state_id"
19           },
20           "inputPorts":[
21              {"portID":"input-0"}
22              {"portID":"input-1"}
23           ],
24           "outputPorts":[
25           {"portID":"output-0"}
26           ],
27         ...},
28     ...],
29    "operatorPositions":{
30        "CSVFileScan-operator-...2a0":
31        {"x":420,
32        "y":85},
33        ...},
34    "links":[
35        {"linkID":"d8c...315",
36        "source":{
```

```
37        "operatorID":"CSVFileScan-operator-...2a0",
38        "portID":"output-0"},
39        "target":{
40        "operatorID":"HashJoin-operator-...7d7",
41        "portID":"input-0"}
42        }, ... ],
43    "groups":[ ],
44    "breakpoints":{ },
45    "commentBoxes":[ ]
46 }
```

Listing 3.2: A sample JSON to represent the patch of adding a Filter operator to the first version.

```
1  [
2      {"op":"add","path":"/links/1","value":
3          {"linkID":"d8c...315",
4              "source":{"operatorID":"CSVFileScan-operator-...2a0
                   ","portID":"output-0"},
5              "target":{"operatorID":"HashJoin-operator-...7d7","
                   portID":"input-0"}}
6      },
7      {"op":"remove","path":"/operators/11"},
8      {"op":"remove","path":"/operators/11/operatorProperties/
          predicates"},
9      {"op":"remove","path":"/operatorPositions/Filter-operator
          -...299"},
10     {"op":"remove","path":"/links/9"},
```

```
11      {"op":"remove","path":"/links/9"}
12  ]
```

Whenever a workflow changes, its content is updated in the workflow table, and a new entry that only contains the edits applied to the workflow is added to the version table. The version entry includes a flag to indicate whether it includes a semantic change or not. It is worth noting that since Texera is a collaborative cloud-based system, the collaboration module [93] handles the conflict resolution when multiple users make changes to the workflow at the same time and is out of the scope of this chapter. Moreover, a version is immutable, i.e., it cannot be modified once created. If the user wants to revert a workflow version to a previous one, all the patches after the particular version need to be applied in order to create a new version, which is a copy of the version being reverted to.

3.5.2 Retrieval of a Particular Workflow Version

After multiple iterations of refining a workflow, the user can revisit and check the previous versions to help refine subsequent ones. Figure 3.5 shows the interface for showing the list of versions preceding the one displayed on the canvas (i.e., the latest version). The user can select any version to inspect from the list. To display the selected version, we apply all the patches from the latest version up to the selected one to construct the workflow of that version.

In some cases, the user wishes to inspect a version that is far from the latest one, leading the system to apply a considerable number of patches, especially because there can be many versions given the fact that we track all of the small edits to eradicate the burden of asking users to commit a version.

Depending on the number of patches to be applied, it may take a considerable amount of

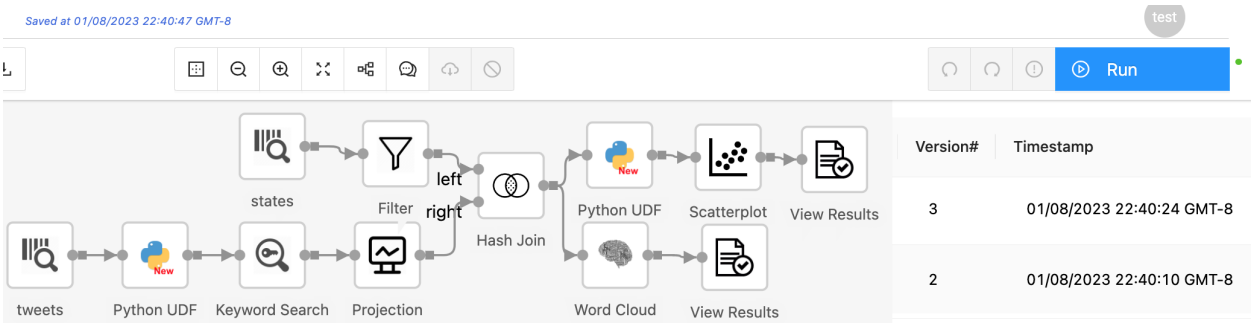


Figure 3.5: Example interface showing a list of versions in a reverse chronological order after a user adds a Filter operator to the workflow.

time. To optimize the process, we maintain the version information of the workflows in two tables: a table of *fine-grained* versions and a table of *coarse-grained* versions. The former table contains entries of all the versions of the workflows. Periodically, we checkpoint a group of fine-grained versions of a workflow to form a coarse-grained one, which is a snapshot of the workflow version at that point, by applying the patches in the group. Every entry in the fine-grained version table also has a pointer to the coarse-grained version it belongs to. We apply the technique in OrpheusDB [65] to decide when to perform a checkpoint. Now, when the user selects a previous version, we find the nearest coarse-grained version succeeding the selected one. Then, starting from the coarse-grained version, we apply all the fine-grained patches up to the selected version to construct the workflow. The two tables are used to optimize the retrieval of a particular version by reducing the number of patches to be applied. It becomes more efficient to compute the patches from a smaller subset from a checkpoint instead of applying patches all the way to the beginning. Figure 3.6 shows an example of three workflow versions to illustrate how the two version tables are maintained.

3.5.3 Highlighting Changes between two Workflow Versions

When a user clicks on a particular version from the list of previous versions, we display the previous version and highlight its *semantic* differences compared to the latest one, as

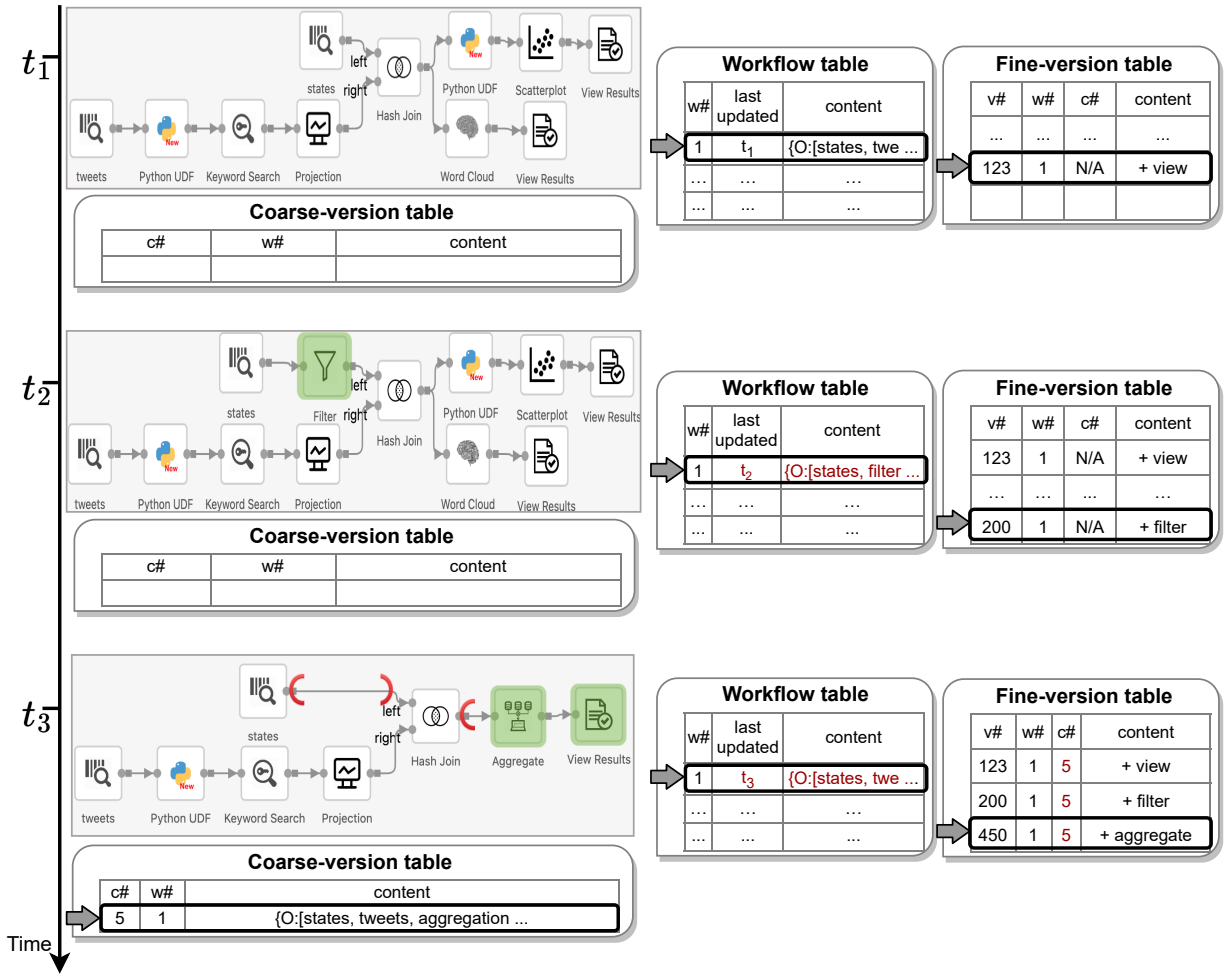


Figure 3.6: Example of a workflow evolving into three versions and the content of each "Version Table" to maintain the relation between these versions.

shown in Figure 3.7. Unlike Git, highlighting the workflow version changes is not as simple as highlighting the content of code. We need to differentiate between the resources and their changes, for example, a change in the properties of an operator. We use three colors to highlight the differences. A green highlight on an operator indicates that the operator has to be added to transform the latest to the selected version, an orange highlight on an operator indicates that the operator properties are modified, and red arches on either ends of a link indicate that one or more operators have to be removed. We omit highlighting the *non-semantic* changes, as too many highlights can overwhelm the user and hinder their ability to grasp important changes.

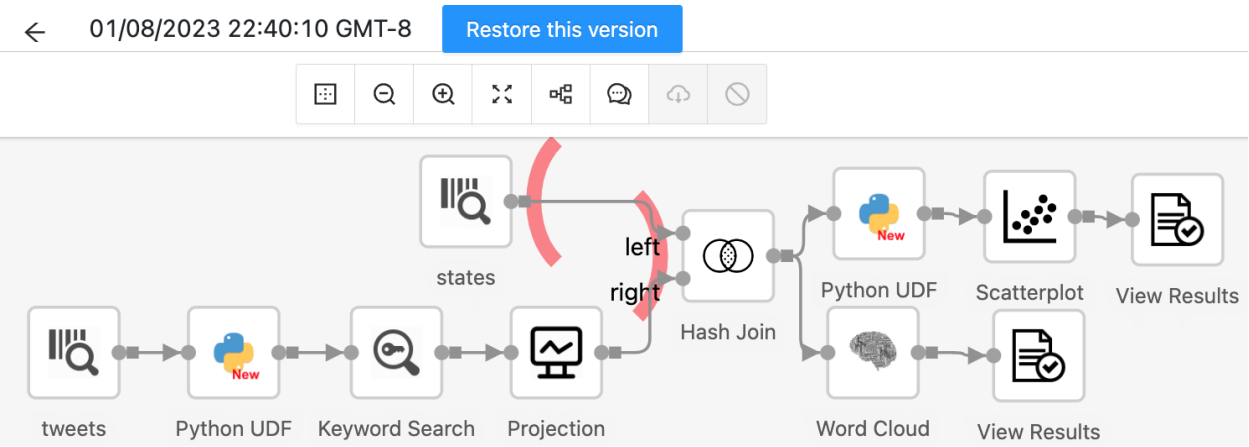


Figure 3.7: An interface showing that a deletion of an operator/s between the red arches transforms the latest workflow version to the selected version. In this example, deleting a Filter between the Source and Join operators is needed to transform the displayed historical version to the latest version.

3.6 Workflow Execution Manager

When a user submits a new request to execute a workflow version, we generate a new entry for the execution and capture its metadata information, as previously discussed. This execution entry encompasses the details of the most recent *semantic* version of the workflow. We specifically associate the execution with the latest semantic version, which is annotated,

as it is only the semantic changes that impact the corresponding results. Additionally, we store each operator's statistics including input and output count of tuples. We also store the results of each sink operator in the workflow. Since storing all the results of all the executions can be costly, we can use Raven, as we will explain in Section 5, to answer the execution request using previously stored results, if possible, and increase the reference point of the results to indicate the number of workflow versions that have (and use) the same results. In our implementation, we periodically clean and delete cached results to accommodate enough storage to store new ones.

Retrieving a particular execution. When the user wants to inspect a particular execution, its details such as the engine version, each operator's version, the workflow version, and the results are retrieved. The system displays the version of the workflow used in the execution and highlights the differences compared to the latest version of the workflow. We also retrieve the statistic counts of processed tuples for each operator in the execution. After the user views the details of the execution, they can repeat the execution by manually replicating the specifications of the execution. Alternatively, the user may wish to *reproduce* the execution. To do so, they can click on the "Run" button as shown in Figure 3.4. Note that if the current engine version or any of the operator's versions are different because of a new deployment, the system notifies the user to indicate the presence of runtime environment changes that may affect the results. We assume that an execution has been completed or aborted before it can be repeated or reproduced. If the experiment is not completed or aborted, we need to handle serializing the states of the execution and this is a future work and out of the scope of this chapter.

After we retrieve the details of an execution, a user may want to export the execution into a file. To do so, we retrieve the details of the execution the same as explained above, we create a new workflow by applying the patches and create a single version, in our current implementation, we only export a sample of the result, we leave storing and exporting the

entire result to a future work. Moreover, a user may wish to delete an execution entry. To do so, we also delete the artifacts associated with it –operator statistics and workflow results. Recall that some executions reuse other executions results. Therefore, we only delete the artifact if the reference count of pointers from executions is 0.

3.7 Conclusion

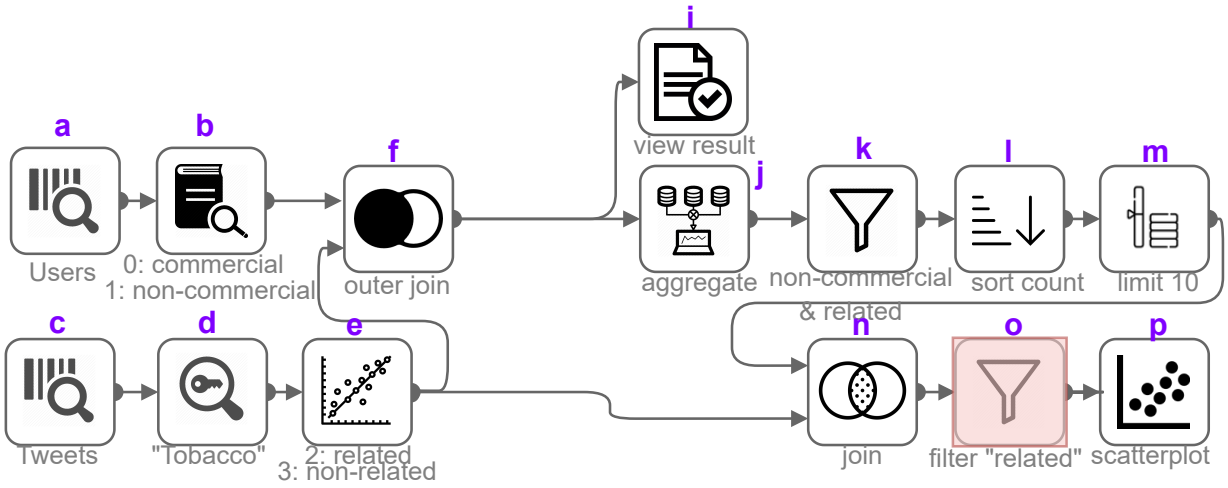
Tracking experiments is an important feature that will greatly impact the way analysts share their result. In this chapter, we showed how we support workflow version control and bookkeeping of workflow executions. We demonstrated our implemented dashboard for tracking the versions and executions.

Chapter 4

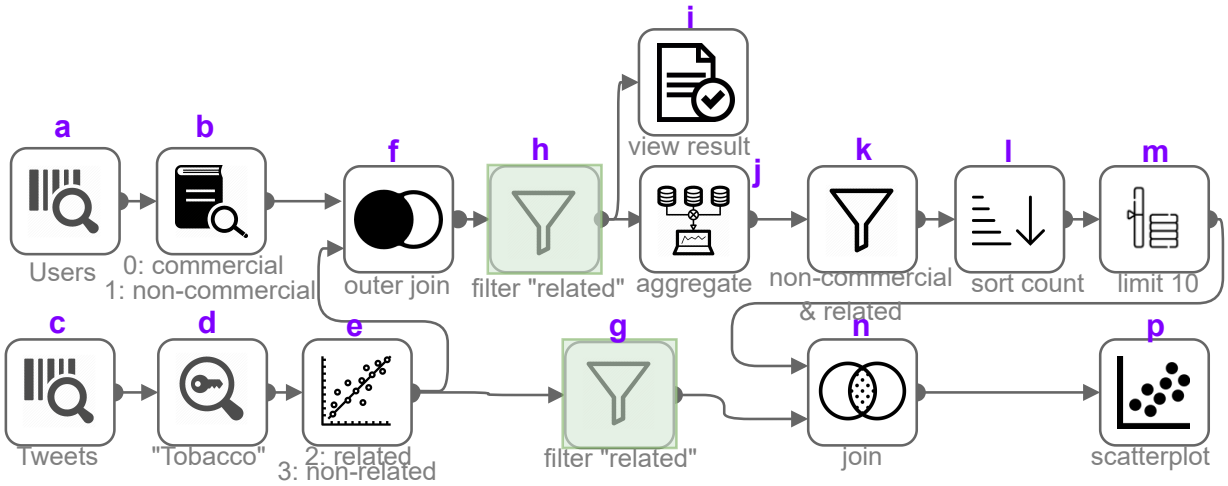
Veer: Verifying Equivalence of Workflow Versions in Iterative Data Analytics

4.1 Introduction

Big data-processing platforms, especially GUI-based systems, enable users to quickly construct complex analytical workflows [11, 84, 42]. These workflows are refined in iterations, generating a new version at each iteration, before a final workflow is constructed, due to the nature of exploratory and iterative data analytics [45, 154]. For example, Figure 4.1 shows a workflow for finding the relevant Tweets by the top k non-commercial influencers based on their tweeting rate on a specific topic. After the analyst constructs the initial workflow version (a) and executes it, she refines the workflow to achieve the desired results. This yields the following edit operations highlighted in the figure, 1) deleting the filter 'o' operator, 2) adding the filter 'g' operator, and 3) adding the filter 'h' operator.



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



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

Figure 4.1: Example workflow and its evolution in two versions.

There has been a growing interest recently in keeping track of these workflow versions and their execution results [80, 143, 42, 9, 150]. In many applications, these workflows have a significant amount of overlap and equivalence [163, 76, 164, 9]. For example, 45% of the daily jobs in Microsoft's analytics clusters have overlaps [76]. 27% of 9,486 workflows to detect fraud transactions from Ant Financial have overlaps, 6% of which are equivalent [163]. In the running example, the edits applied on version (a) that led to a new version (b) had no effect on the result of the sink labeled 'p'. Identifying such equivalence between the execution results of different workflow versions is important. The following are two example use cases.

Use case 1: Optimizing workflow execution. Workflows can take a long time to run due to the size of the data and their computational complexity, especially when they have advanced machine learning operations [163, 84, 13]. Optimizing the performance of a workflow execution has been studied extensively in the literature [113, 46]. One optimizing technique is by leveraging the iterative nature of data analytics to reuse previously materialized results [45, 76].

Use case 2: Reducing storage space. The execution of a workflow may produce a large number of results and storing the output of all generated jobs is impractical [49]. Due to the nature of the overlap and equivalence of consecutive versions, one line of works [6, 45] periodically performs a view de-duplication to remove duplicate stored results. Identifying the equivalence between the workflow versions can be used to avoid storing duplicate results and helps in avoiding periodic clean-up of duplicate results.

These use cases show the need for effective and efficient solutions to decide the equivalence of two workflow versions. We observe the following two unique traits of these GUI-based iterative workflows. (*T1*) these workflows can be large and complex, with operators that are semantically rich [11, 154, 45]. For example, the top 8 workflows in Alteryx's workflows hub [12] had an average of 29 operators, with one of the workflows containing 102 operators, and comprised of mostly non-relational operators. Real workflows in Texera [138] had an

average size of 23 operators, and most of them had visualization and UDF operators. Some operators are user-defined functions (UDF) that implement highly customized logic including machine learning techniques for analyzing data of different modalities such as text, images, audios, and videos [154]. For instance, the workflows in the running example contain two non-relational operators, namely a Dictionary Matcher and a Classifier. (*T2*) Those adjacent versions of the same workflow tend to be similar, especially during the phase where the developer is refining the workflow to do fine tuning [46, 154]. For example, 50% of the workflows belonging to the benchmarks that simulated real iterative tasks on video [154] and TPC-H [46] data had overlap. The refinements between the successive versions comprised of only a few changes over a particular part of the workflow. Thus, we want to study the following:

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

Limitations of existing solutions. Workflows include relational operators and UDFs [84]. Thus, we can view the problem of checking the equivalence of two workflow versions as the problem of checking the equivalence of two SQL queries. The latter is undecidable in general [3] (based on the reduction from First-order logic). There have been many Equivalence Verifiers (EVs) proposed to verify the equivalence of two SQL queries [37, 164, 163]. These EVs have *restrictions* on the type of operators they can support, and mainly focus on relational operators such as SPJ, aggregation, and union. They cannot support many semantically rich operators common in workflows, such as dictionary matching and classifier operators in the running example, and other operators such as unnest and sentiment analyzer. To investigate their limitations, we analyzed the SQL queries and workflows from 6 workloads, and generated an equivalent version by adding an empty filter operator. Then, we used EVs from the literature [37, 164, 163, 146] to test the equivalence of these two versions. Table 4.1 shows the average percentage of pairs for each workload that can be verified by

these EVs, which is very low.

Table 4.1: Limitations of existing EVs to verify equivalence of workflow versions from real workloads.

Workload	# of pairs	AVG. % of pairs supported by existing EVs
Calcite benchmark [27]	232	34.81%
Knime workflows hub [82]	37	2.70%
Orange workflows [109]	32	0.00%
IMDB sample workload [71]	5	0.00%
TPC-DS benchmark [140]	99	2.02%
Texera workflows [138]	105	0.00%

Our Approach. To solve the problem of verifying the equivalence of two workflow versions, we leverage the fact that the two workflow versions are almost identical except for a few local changes (*T2*). In this chapter, we present Veer[1], a verifier to test the equivalence of two workflow versions. It addresses the aforementioned problem by utilizing existing EVs as a black box. In §4.4, we give an overview of the solution, which divides the workflow version pair into small parts, called "windows", so that each window satisfies the EV's restrictions in order to push testing the equivalence of a window to the EV. Our approach is simple yet highly effective in solving a challenging problem, making it easily applicable to a wide range of applications.

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

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

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

1. We formulate the problem of verifying the equivalence of two complex workflow versions in iterative data analytics. To the best of our knowledge, Veer is the first work that studies this problem by incorporating the knowledge of user edit operations into the solution (§4.3).
2. We give an overview of the solution and formally define the "window" concept that is used in the equivalence verification algorithm (§ 4.4).
3. We first consider the case where there is a single edit. We analyze how the containment between two windows is related to their equivalence results, and use this analysis to derive the concept of "maximal covering window". We provide complexity analysis (§4.5).
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 our algorithm (§4.6).
5. We provide a number of optimizations in Veer$^+$ to improve the performance of the baseline algorithm (§ 4.8).
6. We report the results of a thorough experimental evaluation of the proposed solutions. The experiments show that the proposed solution is not only able to verify workflows that cannot be verified by existing EVs, but also able to do the verification efficiently (§ 4.10).

4.2 Related Work

Equivalence verification. There are many studies to solve the problem of verifying the equivalence of two SQL queries under certain assumptions. These solutions were applicable to a small class of SQL queries, such as conjunctive queries [28, 4, 124, 74]. With the recent advancement of developing proof assists and solvers [43, 44], there have been new solutions [37, 164, 163] leveraging these solvers. UDP [37] and WeTune's verifier [146] use semirings to model the semantics of the pair and use a proof assist, such as Lean [44] to prove if the expressions are equivalent. These two works support reasoning semantics of two queries with integrity constraints. Equitas [164] and Spes [163] model the semantics of the pair into a First-Order Logic (FOL) formula and push the formula to be solved by a solver such as SMT [43]. These two works support queries with three-valued variables. Other works also use an SMT solver to verify the equivalence of a pair of Spark jobs [56]. The work in [29] finds a weighted edit distance based on the semantic equivalence of two queries to grade students queries. Veer uses these verifiers as black boxes to verify the equivalence of a version pair.

Tracking workflow executions. There has been an increasing interest in enabling the reproducibility of data analytics pipelines. These tools track the evolution and versioning of datasets, models, and results. At a high level they can be classified as two categories. The first includes those that track experiment results of different versions of ML models and the corresponding hyper-parameters [159, 31, 143, 80, 54, 100]. The second includes solutions to track results of different versions of data-processing workflows [99, 150, 42, 9]. These solutions are motivations for Veer.

Materialization reuse. There is a large body of work on answering data-processing workflows using views [122, 118, 46, 45, 76]. Some solutions [49] focus on deciding which results to store to maximize future reuse. Other solutions [102, 162] focus on identifying material-

ization reuse opportunities by relying on finding an exact match of the workflow's DAG. On the other hand, semantic query optimization works [59, 51, 127, 83] reason about the semantics of the query to identify reuse opportunities that are not limited to structural matching. However, these solutions are applicable to a specific class of functions, such as user defined function (UDF) [118, 154, 108], and do not generalize to finding reuse opportunities by finding equivalence of any pair of workflows.

4.3 Problem Formulation

In this section, we use an example workflow to describe the setting. We also formally define the problem of verifying equivalence of two workflow versions. Table 4.2 shows a summary of the notations used in this section.

Table 4.2: Notations used for a single workflow.

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

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

denoted as $\mathbb{D}_W = \{D_1, \ldots, D_l\}$ and multiple sink operators denoted as $\mathbb{S}_W = \{s_1, \ldots, s_n\}$.

For example, consider a workflow in Figure 4.1a. It has two source operators "Tweets" and "Users" and two sink operators s_i and s_p to show a tabular result and a scatterplot visualization, respectively. The OuterJoin operator has two outgoing links to push its processed tuples to the downstream Aggregate and Sink operators. The Filter operator's properties include the boolean selection predicate.

4.3.1 Workflow Version Control

A workflow W undergoes many edits from the time it was first constructed as part of the iterative process of data analytics [99, 150]. A workflow W has a list of versions $V_W = [v_1, \ldots, v_m]$ along a timeline in which the workflow changes. Each v_j is an immutable version of workflow W in one time point following version v_{j-1}, and contains a number of edit operations to transform v_{j-1} to v_j.

Definition 4.1 (Workflow edit operation)**.** *We consider the following edit operations on a workflow:*

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

A combination of these edit operations is a *transformation*, denoted as δ_j. The operation of

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

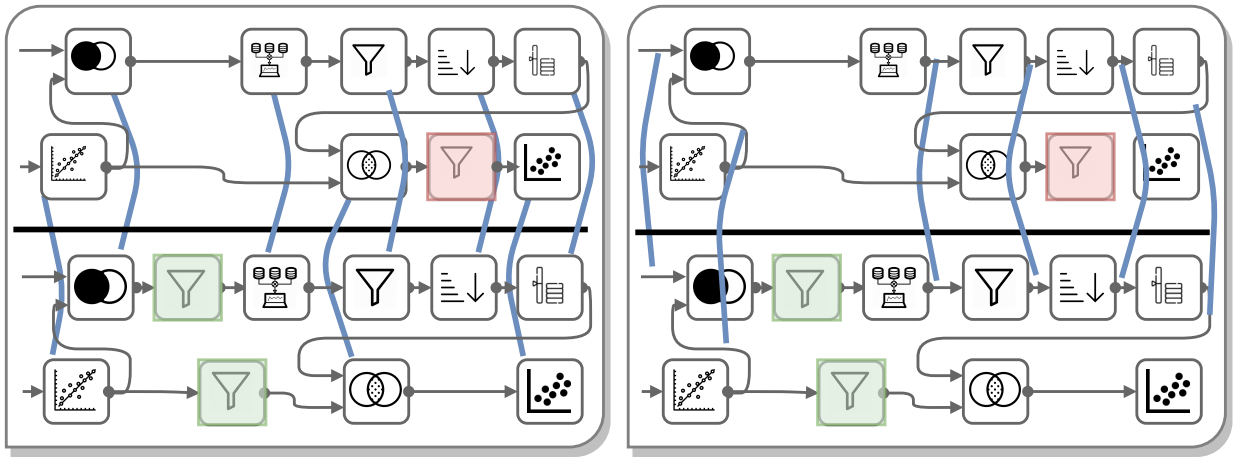
applying the transformation δ_j to a workflow version v_j is denoted as $\oplus$, which produces a new version v_{j+1}. Formally,

$$v_{j+1} \quad = \quad v_j \quad \oplus \quad \delta_j. \tag{4.1}$$

In the running example, the analyst makes edits to revise the workflow version v_1 in Figure 4.1a. 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 along with the necessary link changes to add those operators correspond to a transformation, δ_1 and applying it on v_1 will result in a new version v_2, illustrated in Figure 4.1b.

Workflow edit mapping. Given a pair of versions (P, Q) and an edit mapping $\mathcal{M}$, there is a corresponding transformation from P to Q, which aligns every operator in P to at most one operator in Q. Each operator in Q is mapped onto by at most one operator in P. A link between two operators in P maps to a link between the corresponding operators in Q. Those operators and links in P that are not mapped to any operators and links in Q are assumed to be deleted. Similarly, those operators and links in Q that are not mapped onto by any operators and links in P are assumed to be inserted. This mapping is similar to the one discussed in Chapter 3.

Figure 4.2 shows an example edit mapping between the two versions v_1 and v_2 in the running example. As $\mathsf{Filter_y}$ from v_1 is deleted, the operator is not mapped to any operator in v_2.



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

Figure 4.2: Example of an edit mapping between version v_1 and v_2. Portions of the workflows are omitted for clarity.

4.3.2 Workflow's Execution and Results

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

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

Result equivalence of workflow versions. The execution request for the version v_j may produce a sink result equivalent to the corresponding sink of a previous executed version v_{j-k}, where $k < j$. For example, in Figure 4.1b, executing the workflow version v_2 produces a result of the scatterplot sink s_2 equivalent to the result of the corresponding scatterplot of v_1. In particular, v_2's edit is pushing down the Filter operator and the scatterplot result remains the same. Notice however that the result of s_i in v_2 is not equivalent to the result of s_i in v_1 because of the addition of the new Filter_h operator. Now, we formally define "sink equivalence."

Definition 4.2 (Sink Equivalence and Version-Pair Equivalence)**.** *Consider two workflow*

[3] This assumption is valid in many real-world applications as we detail in the experiment Section 4.10

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

In this chapter, we first study the problem where each of the two versions has a single sink. We generalize the solution to the case of multiple sinks in Chapter 5.

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

4.3.3 Equivalence Verifiers (EVs)

An equivalence verifier (or "EV" for short) takes as an input a pair of SQL queries $\mathbf{Q_1}$ and $\mathbf{Q_2}$. An EV returns `True` when $\mathbf{Q_1} \equiv \mathbf{Q_2}$, `False` when $\mathbf{Q_1} \not\equiv \mathbf{Q_2}$, or `Unknown` when the EV cannot determine the equivalence of the pair under a specific table semantics [146, 37, 164, 163, 43, 56]. For instance, UDP [37] and Equitas [164] are two EVs. The former uses U-expressions to model a query while the latter uses a symbolic representation. Both EVs internally convert the expressions to a first-order-logic (FOL) formula and then push the formula to a solver such as an SMT solver [43] to decide its satisfiability. An EV requires two

queries to meet certain requirements (called "restrictions") in order to test their equivalence. We will discuss these restrictions in detail in Section 4.5.2.

Problem Statement. *Given an EV and two workflow versions P and Q with their mapping $\mathcal{M}$, test the equivalence of the two versions.*

4.4 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 4.4.1). We formally define the concepts of "window" and "covering window" (Section 4.4.2).

4.4.1 Veer: Overview

To verify the equivalence of a pair of sinks in two workflow versions, Veer leverages the fact that the two versions are mostly identical except for a few places with edit operations. It uses existing EVs as a black box. Given an EV, our approach is to break the version pair into multiple "windows," each of which includes local changes and satisfies the EV's restrictions to verify if the pair of portions of the workflow versions in the window is equivalent, as illustrated in Figure 1.3. We consider different semantics of equivalence between two tuple collections, including sets, bags, and lists, depending on the application of the workflow and the given EV. Veer is agnostic to the underlying EVs, making it usable for any EV of choice.

Next we define concepts used in this approach.

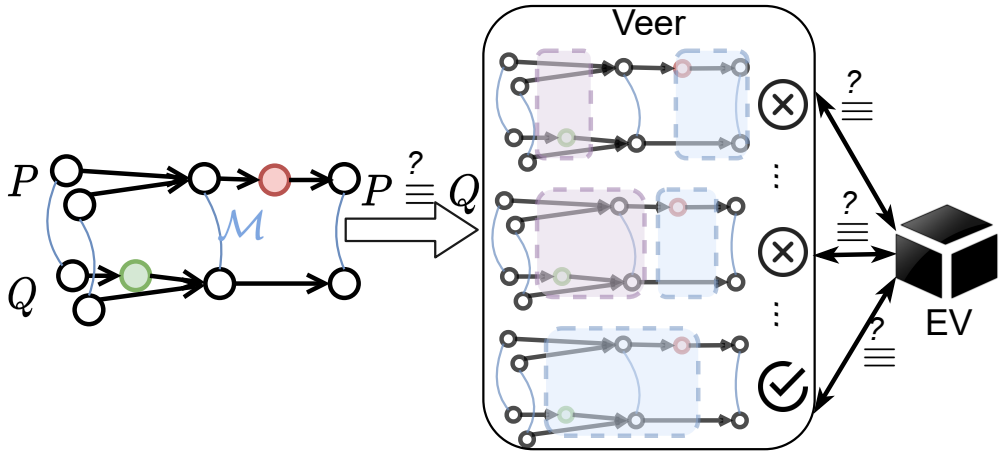


Figure 4.3: Overview of Veer. Given an EV and two versions with their mapping, Veer breaks (decomposes) the version pair into small windows, each of which satisfies the EV's restrictions. It finds different possible decompositions until it finds one with each of windows verified as equivalent by the EV.

4.4.2 Windows and Covering Windows

Definition 4.3 (Window)**.** *Consider two workflow versions P and Q with a set of edits $\delta = \{c_1 \dots c_n\}$ from P to Q and a corresponding mapping $\mathcal{M}$ from P to Q. A* window*, denoted as ω, is a pair of sub-DAGs $\omega(P)$ and $\omega(Q)$, where $\omega(P)$ (respectively $\omega(Q)$) is a connected induced sub-DAG of P (respectively Q). Each pair of operators/links under the mapping $\mathcal{M}$ should be either both in ω or both outside ω.*

The operators in the sub-DAGs $\omega(P)$ and $\omega(Q)$ without outgoing links are called their *sinks*. Recall that we assume each workflow has a single sink. However, the sub-DAG $\omega(P)$ and $\omega(Q)$ may have more than one sink. This can happen, for example, when the window contains a Replicate operator. A *neighbor* of a window is either an operator before a source operator of the window or an operator after a sink of the window. Figure 4.4 shows a window ω, where each sub-DAG includes the Classifier operator and two downstream operators Left-Outerjoin and Join, which are two sinks of the sub-DAG.

Definition 4.4 (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

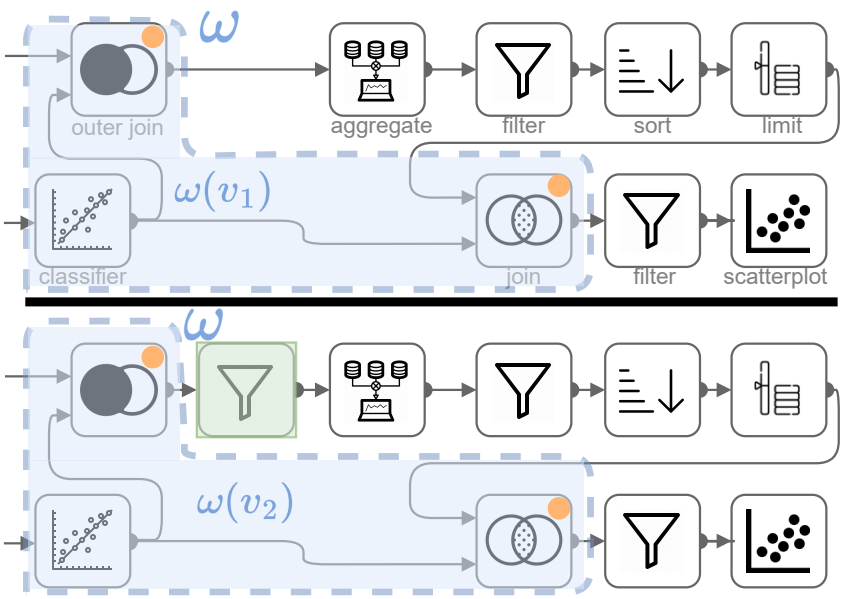


Figure 4.4: An example window ω and each sub-DAG of $\omega(v_1)$ and $\omega(v_2)$ contains two sinks (shown as an Orange circle).

window, *denoted as* ω_C, *is a window to cover a set of changes* $C \subseteq \delta$. *That is, the sub-DAG in* P *(respectively sub-DAG in* Q*) in the window includes the sources if any (respectively targets, if any) operators/links of the edit operations in* C.

When the edit operations are clear in the context, we will simply write ω to refer to a covering window. Figure 4.5 shows a covering window for the change of adding the operator $\mathsf{Filter_h}$ to v_2. The covering window includes the sub-DAG $\omega(v_1)$ of v_1 and contains the Aggregate operator. It also includes the sub-DAG $\omega(v_2)$ of v_2 and contains the $\mathsf{Filter_h}$ and Aggregate operators.

Definition 4.5 (Equivalence of the two sub-DAGs in a window)**.** *We say the two sub-DAGs* $\omega(P)$ *and* $\omega(Q)$ *of a window* ω *are* equivalent, *denoted as "*$\omega(P) \equiv \omega(Q)$*," if they are equivalent as two stand-alone DAG's, i.e., without considering the constraints from their upstream operators. That is, for every instance of source operators in the sub-DAGs (i.e., those operators without ancestors in the sub-DAGs), each sink* s *of* $\omega(P)$ *and the corresponding* $\mathcal{M}(s)$ *in* $\omega(Q)$ *produces the same results. In this case, for simplicity, we say this window is equivalent.*

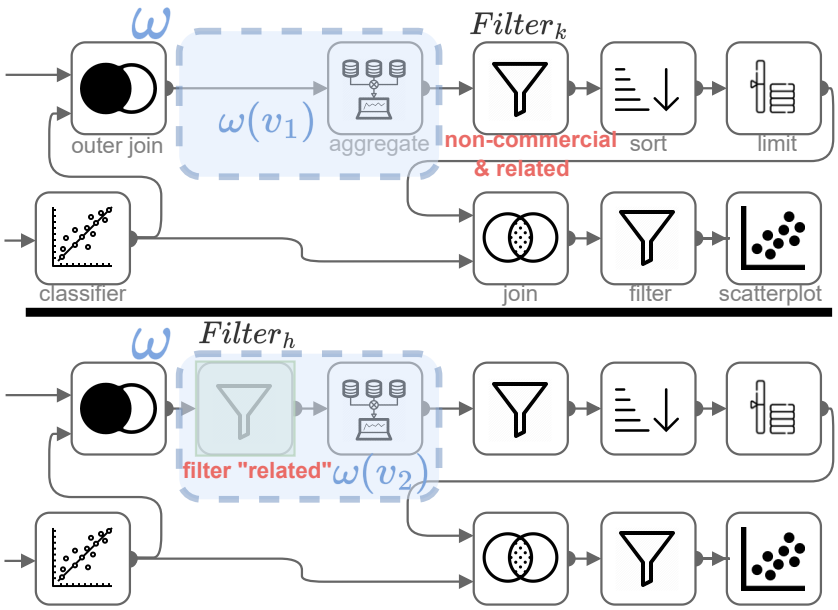


Figure 4.5: A covering window ω for adding $\mathsf{Filter_h}$.

Figure 4.6 shows an example of a covering window ω', where its sub-DAGs $\omega'(v_1)$ and $\omega'(v_2)$ are equivalent.

Notice that for each sub-DAG in the window ω, the results of its upstream operators are the input to the sub-DAG. The equivalence definition considers all instances of the sources of the sub-DAG, without considering the constraints on its input data as the results of upstream operators. For instance, consider the two workflow versions in Figure 4.7. The two sub-DAGs of the shown window ω are clearly not equivalent as two general workflows, as the top sub-DAG has a filter operator, while the bottom one does not. However, if we consider the constraints of the input data from the upstream operators, the sub-DAGs in ω are indeed equivalent, because each of them has an upstream filter operator with a predict $age < 50$, making the predicate $age < 55$ redundant. We use this definition of sub-DAG equivalence despite the above observation, because we treat the sub-DAGs in a window as a pair of stand-alone workflow DAGs to pass to the EV for verification (see Section 4.5.1).

Definition 4.6 (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*

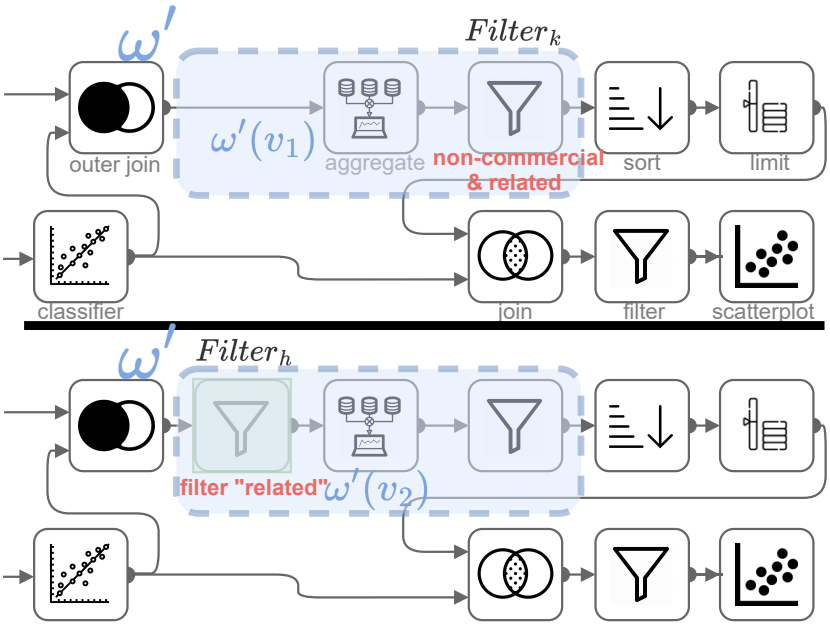


Figure 4.6: An example covering window ω' showing its pair of sub-DAGs are equivalent.

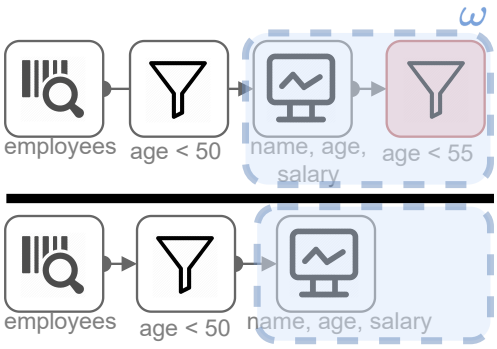


Figure 4.7: Two sub-DAGs in the window ω are not equivalent, as sub-DAG equivalence in Definition 4.5 does not consider constraints from the upstream operators. But the two complete workflow versions are indeed equivalent.

in ω'. In this case, we call ω a sub-window *of ω', and ω' a* super-window *of ω.*

For instance, the window ω in Figure 4.5 is contained in the window ω' in Figure 4.6.

4.5 Two Versions with a Single Edit

In this section, we study how to verify the equivalence of two workflow versions P and Q with a single change c of the corresponding mapping $\mathcal{M}$ from P to Q. We leverage a given EV γ to verify the equivalence of two queries. We discuss how to use the EV to verify the equivalence of the version pair in a window (Section 4.5.1), and discuss the EV's restrictions (Section 4.5.2). We present a concept called "maximal covering window", which helps in improving the performance of verifying the equivalence (Section 4.5.3), and develop a method to find maximal covering windows to verify the equivalence of the two versions (Section 4.5.4).

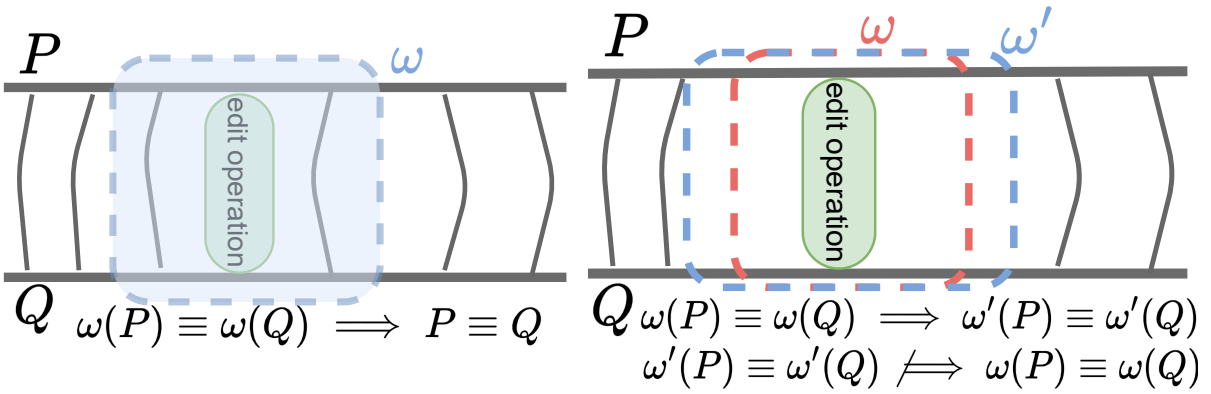
4.5.1 Verification Using a Covering Window

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

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

Proof. Suppose $\omega(P) \equiv \omega(Q)$. From the definition of a covering window, every operator in one sub-DAG of the window ω has its corresponding mapped operator in the other sub-DAG of the window, and the change c is included in the window. This means that the sub-DAGs of P and Q that precede the window ω are isomorphic (structurally identical) and

the sub-DAGs of P and Q that follow the window are isomorphic as shown in Figure 4.8a. Following the assumption that multiple runs of a workflow produce the same result, this infers that given an instance of input sources $\mathbb{D}$, the sub-DAGs before the window would produce equivalent results according to definition 4.3.2. This result becomes the input source for the window ω and given that the sub-DAGs in ω are equivalent, this means that each sink of $\omega(P)$ is equivalent to the corresponding sink (according to the mapping) of $\omega(Q)$. Hence, the output of the window, which is the input to the pair of sub-DAGs following the window, is identical, and since the operators are isomorphic, the result of the sub-DAGs following the window is equivalent. Thus, $P \equiv Q$. □



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

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

Figure 4.8: Conceptual examples to explain the relation between a "covering window" and version pair equivalence.

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

A key question is how to find such a covering window. Notice that the two sub-DAGs in Figure 4.5 are not equivalent. However, if we include the downstream Filter_k in the covering window to form a new window ω' (shown in Figure 4.6) with a pair of sub-DAGs $\omega'(P)$ and

$\omega'(Q)$, then the two sub-DAGs in ω' are equivalent. This example suggests that we may need to consider multiple windows in order to find one that is equivalent.

4.5.2 EV Restrictions and Valid Windows

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

Definition 4.7 (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'' *as an answer.*

We will relax this definition in Section 4.9, discuss the consequences of relaxing the definition, and propose solutions. There are two types of restrictions.

- Restrictions due to the EV's explicit assumptions: For example, UDP and Equitas support reasoning of certain operators, e.g., Aggregate and SPJ, but not other operators such as Sort.
- Restrictions that are derived due to the modules used by the EV: For example, Equitas [164], Spes [163], and Spark Verifier [56] use an SMT solver [43] to determine if a FOL formula is satisfiable or not. SMT solver is not complete for determining the satisfiability of formulas when their predicates have non-linear conditions [24]. Thus, these EVs require the predicate conditions in their expressions to to be linear to make sure to receive an answer from the solver.

As an example, the following is an example of the explicit and *derived* restrictions of the Equitas [164] to test the equivalence of two queries [4]. Note that restrictions are not unique.

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

R1. The table semantics has to be set semantics.[5]

R2. All operators have to be any of the following types: SPJ, Outer join, and/or Aggregate.

R3. The predicate conditions of SPJ operators have to be linear.

R4. Both queries should have the same number of Outer join operators, if present.

R5. Both queries should have the same number of Aggregate operators, if present.

R6. If they use an Aggregate operator with an aggregation function that depends on the cardinality of the input tuples, e.g., COUNT, then each upstream operator of the Aggregate operator has to be an SPJ operator, and the input tables are not scanned more than once.

Definition 4.8 (Valid window w.r.t an EV)**.** *We say a window is* valid *with respect to an EV if it satisfies the EV's restrictions.*

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

4.5.3 Maximal Covering Window (MCW)

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

Lemma 4.2. *Consider a version pair (P, Q) with a single edit c between them. Suppose*

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

a covering window ω of c is contained in another covering window ω'. If the sub-DAGs in window ω are equivalent, then the sub-DAGs of ω' are also equivalent.

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

Based on Lemma 4.2, we can just focus on covering windows that have as many operators as possible without violating the constraints of the EV. If the EV shows that such a window is not equivalent, then none of its sub-windows can be equivalent. (A subtle case is when the EV does not know if the window ω' is equivalent, but can verify that ω is equivalent. We will discuss this case in Section 4.9.) Based on this observation, we introduce the following concept.

Definition 4.9 (Maximal Covering Window (MCW))**.** *Given a workflow version pair (P, Q) with a single edit operation c, a valid covering window ω is called* maximal *if it is not properly contained by another valid covering window.*

The change c may have more than one MCW, For example, suppose the EV is Equitas. Figure 4.9 shows two MCWs to cover the change of adding the $\mathsf{Filter_h}$ operator. One maximal window ω_1 includes the change $\mathsf{Filter_h}$ and Left Outerjoin on the left of the change. The window cannot include the Classifier operator from the left side because Equitas cannot reason about its semantics. Similarly, the Aggregate operator on the right cannot be included in ω_1 because one of Equitas restrictions is that the input of an Aggregate operator must be an SPJ operator and the window already contains Left OuterJoin. To include the Aggregate operator, a new window ω_2 is formed to exclude Left OuterJoin and include Filter on the right but cannot include Sort because this operator cannot be reasoned by Equitas.

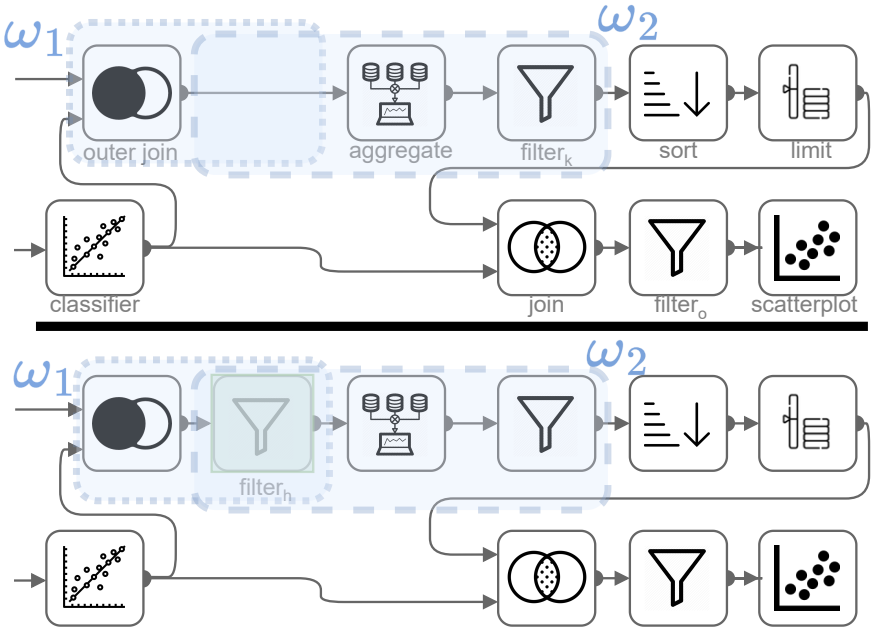


Figure 4.9: Two MCW ω_1 and ω_2 satisfying the restrictions of Equitas to cover the change of adding $\mathsf{Filter_h}$ to v_2.

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

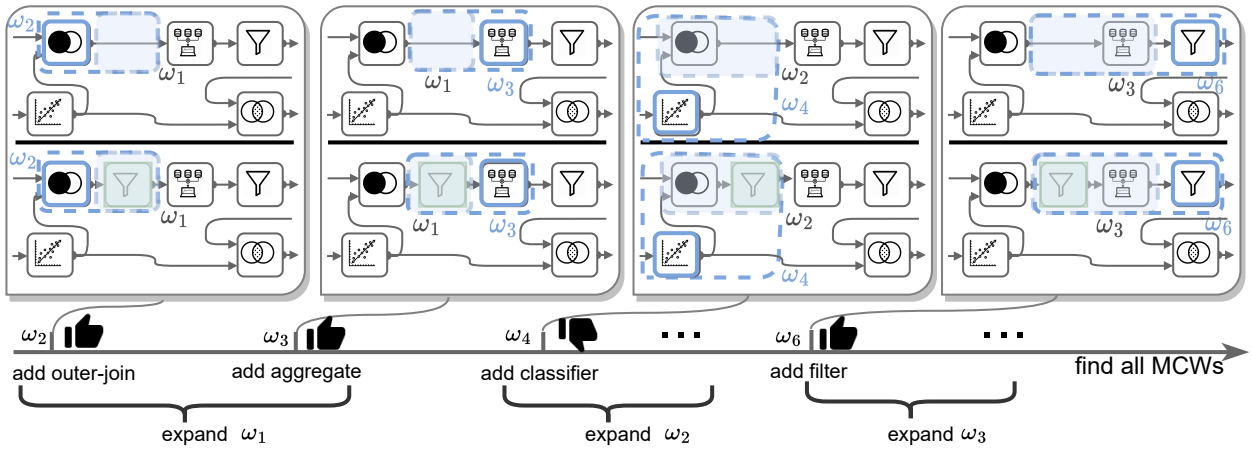


Figure 4.10: Example to illustrate the process of finding MCWs for the change of adding $\mathsf{Filter_h}$ to v_2.

4.5.4 Finding MCWs to Verify Equivalence

Next we study how to efficiently find an MCW to verify the equivalence of two workflow pairs. We present a method shown in Algorithm 4.1. Given a version pair P and Q and a single edit operation c based on the mapping $\mathcal{M}$, the method finds an MCW that is verified by the given EV γ to be equivalent.

Algorithm 4.1: Verifying equivalence of two workflow versions with a single edit

Input: A version pair (P, Q); A single edit c; A mapping $\mathcal{M}$; An EV γ
Output: A flag to indicate if they are equivalent
// a `True` value to indicate the pair is equivalent, a `False` value to indicate the pair is not equivalent, or `Unknown` when the pair cannot be verified
1 $\omega \leftarrow$ create an initial window to include the source and the corresponding target (operator/link) of the edit c
2 $\Omega = \{\omega\}$ // initialize a set for exploring widows
// using memoization, a window is explored only once
3 **while** Ω *is not empty* **do**
4 $\omega_i \leftarrow$ remove one window from Ω
5 **for** *every neighbor of* ω_i **do**
6 **if** *adding neighbor to* ω_i *meets EV's restrictions* **then**
7 add ω_i' (including the neighbor) to Ω
8 **end**
9 **if** *none of the neighbors were added to* ω_i **then**
 // the window is maximal
10 **if** ω_i *is verified equivalent by the EV* **then**
11 **return** `True`
12 **if** ω_i *is verified not equivalent by the EV and the window is the entire version pair* **then**
13 **return** `False`
14 **end**
15 **return** `Unknown`

We use the example in Figure 4.10 to explain the details of Algorithm 4.1. The first step is to initialize the window to cover the source and target operator of the change only (line 1). In this example, for the window ω_1, its sub-DAG $\omega_1(v_2)$ contains only $\mathsf{Filter_h}$ and its corresponding operator using the mapping $\mathcal{M}$ in $\omega_1(v_1)$. Then we expand all the windows created so far, i.e., ω_1 in this case (line 2). To expand the window, we enumerate all possible combi-

nations of including the neighboring operators on both $\omega_1(v_1)$ and $\omega_1(v_2)$ using the mapping. For each neighbor, we form a new window and check if it has not been explored yet. If not, then we check if the newly formed window is valid (lines 5-6).

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

Some EVs [37, 164, 163] return `False` to indicate that the equivalence of the version pair cannot be verified, but it does not necessarily mean that the pair is inequivalent. We take note of these EVs, and in the algorithm mentioned above, we only report `False` if the EV is capable of proving the inequivalence of the pair, such as COSETTE [38].

4.6 Two Versions with Multiple Edits

In the previous section, we assumed there is a single edit operation to transform a workflow version to another version. In this section, we extend the setting to discuss the case where multiple edit operations $\delta = \{c_1 \dots c_n\}$ transform a version P to a version Q. A main challenge is finding covering windows for multiple edits (Section 4.6.1). We address the

challenge by decomposing the version pair into a set of *disjoint* windows. We formally define the concepts of "decomposition" and "maximal decomposition" (Section 4.6.2). We explain how to find maximal decompositions to verify the equivalence of the version pair and prove the correctness of our solution (Section 4.6.4). We analyze the completeness of the proposed algorithm (Section 4.6.5).

4.6.1 Can we use overlapping windows?

When the two versions have more than one edit, they can have multiple covering windows. A natural question is whether we can use covering windows that overlap with each other to test the equivalence of the two versions. We will use an example to show that we cannot do that. The example, shown Figure 4.11, is inspired from the NY Taxi dataset [106] to calculate the trip time based on the duration and starting time. Suppose the Select_x and Select_z operators are deleted from a version v_1 and Select_y operator is added to transform the workflow to version v_2. The example shows two overlapping windows ω and ω', each window is equivalent.

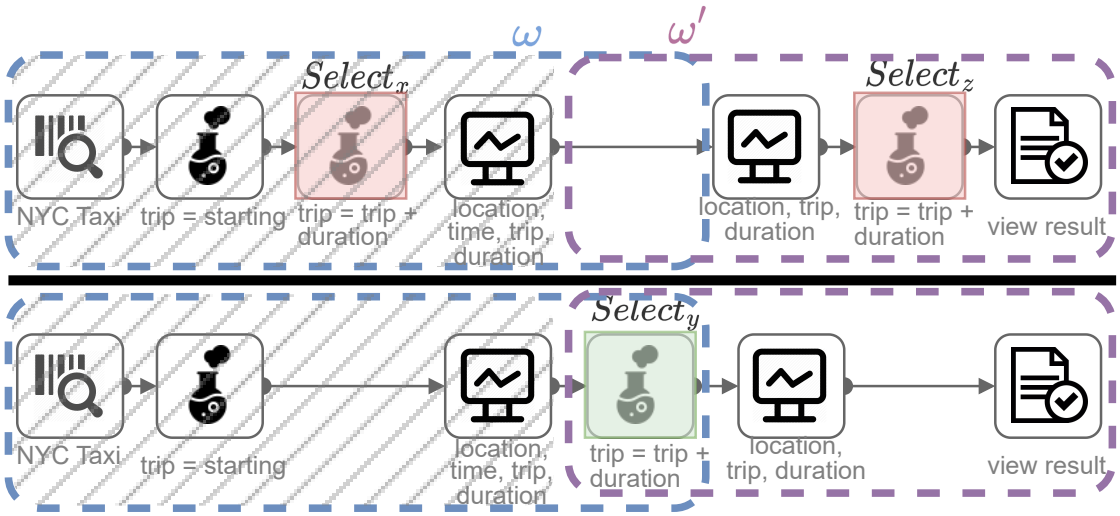


Figure 4.11: In this example, the blue window ω is equivalent and the purple window ω' is also equivalent. But the version pair is not equivalent. The shaded gray area is the input to window ω'.

We cannot say the version pair is equivalent. The reason is that for the pair of sub-DAGs in ω' to be equivalent, the input sources have to be the same (the shaded area in grey in the example). However, we cannot infer the equivalence of the outcome of that portion of the sub-DAG. In fact, the pair of sub-DAGs in the shaded area in this example produce different results. This problem does not exist in the case of a single edit, because the input sources to any *covering* window (in a single edit case) will always be a one-to-one mapping of the two sub-DAGs and there is no other change outside the covering window. The solution in Section 4.5 finds *any* window such that its sub-DAGs are equivalent and cannot be directly used to solve the case of verifying the equivalence of the version pair when there are multiple edits.

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

4.6.2 Version Pair Decomposition

Definition 4.10 (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 4.12 shows a decomposition for the three changes in the running example. The

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

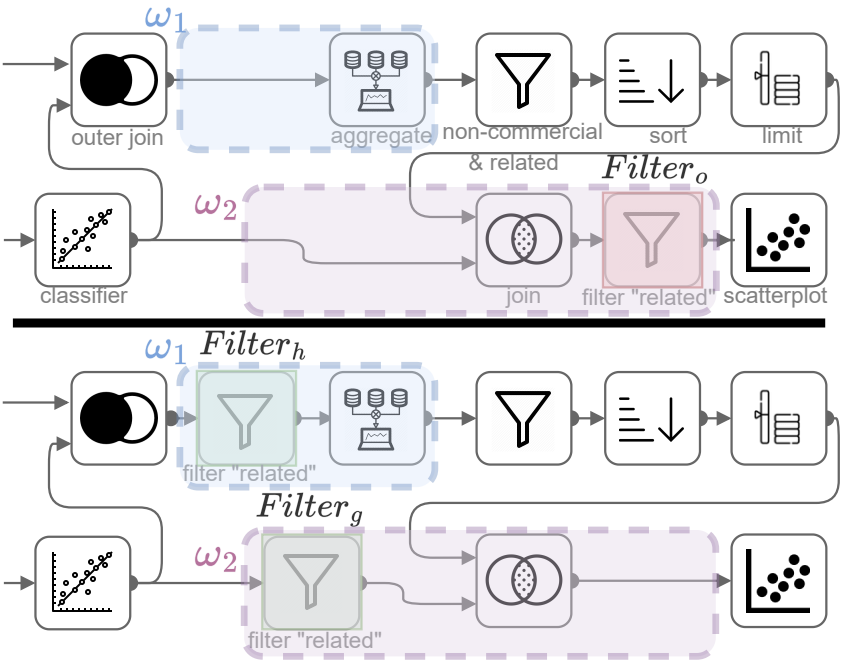


Figure 4.12: A decomposition θ with two covering windows ω_1 and ω_2 that cover the three edits.

Lemma 4.3. *(Corresponding to Lemma 4.1) For a version pair P and Q with a set of edit operations $\delta = \{c_1 \dots c_n\}$ to transform P to Q, if there is a decomposition θ such that every covering window in θ is equivalent, then the version pair is equivalent.*

Proof. Suppose every covering window ω_i in a decomposition θ is equivalent. Every other window that is not covering, its sub-DAGs are structurally identical, according to Definition 4.4.2. Given an instance of input sources $\mathbb{D}$, we can have the following two cases. (`CASE1:`) the input is processed by a pair of structurally identical sub-DAGs that are in a non-covering window. In this case, the pair of sub-DAGs produce an equivalent result since every operator is deterministic according to Assumption 4.3.2. (`CASE2:`) the input is

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

processed by a pair of sub-DAGs in a covering window. In this case, the pair of sub-DAGs produce equivalent result because we assumed each covering window is equivalent. In both cases, the output acts as the input to the following portion of the sub-DAGs (either non-covering or a covering window). This propagation continues along the pair of DAGs until the end, thus the version pair produces equivalent results as shown in Figure 4.13. □

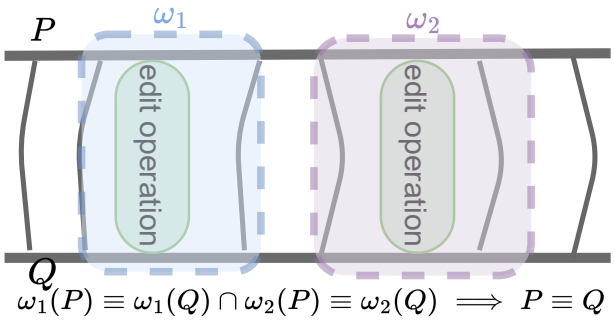


Figure 4.13: Using multiple covering windows on multiple edits to check the equivalence of two versions.

A natural question is how to find a decomposition where each of its windows is equivalent. We could exhaustively consider all the possible decompositions, but the number can grow exponentially as the size of the workflow and the number of changes increase. The following "decomposition containment" concept, defined shortly, helps us reduce the number of decompositions that need to be considered.

Definition 4.11 (Decomposition containment)**.** *We say a decomposition θ is* contained *in another decomposition θ', denoted as $\theta \subseteq_d \theta'$, if every window in θ, there exists a window in θ' that contains it.*

Figure 4.14 shows an example of a decomposition θ' that contains the decomposition θ in Figure 4.12. We can see that in general, if a decomposition θ is contained in another decomposition θ', then each window in θ' is a concatenation of one or multiple windows in θ.

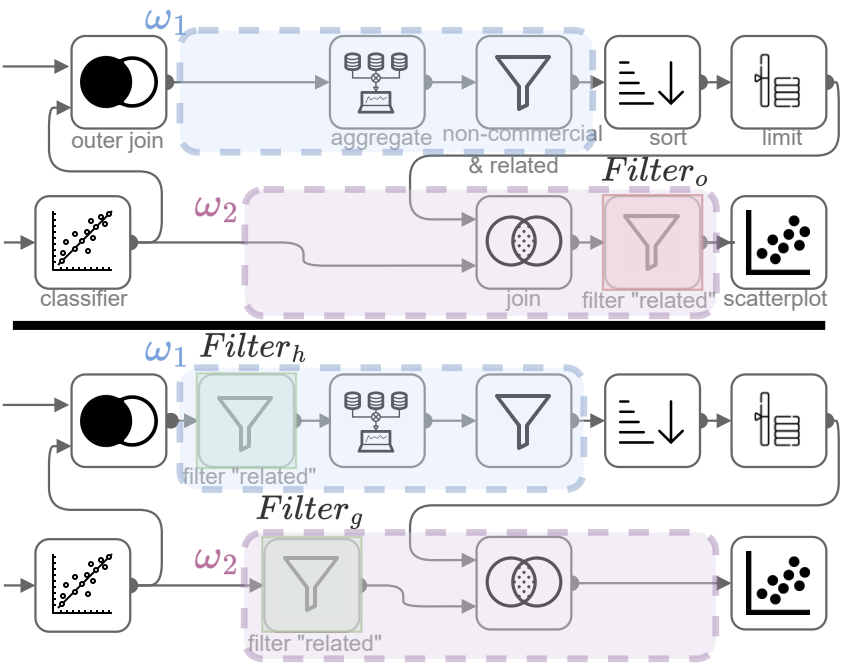


Figure 4.14: Example to show equivalent pair of sub-DAGs of every covering window in a decomposition θ'.

The following lemma, which is a generalization of Lemma 4.2, can help us prune the search space by ignoring decompositions that are properly contained by other decompositions.

Lemma 4.4. *(Corresponding to Lemma 4.2) Consider a version pair P and Q with a set of edit operations $\delta = \{c_1 \dots c_n\}$ from P to Q. Suppose a decomposition θ is contained in another decomposition θ'. If each window in θ is equivalent, then each window in θ' is also equivalent.*

Proof. Suppose each window in a decomposition θ is equivalent and the decomposition is contained in another decomposition θ'. Based on the Definition of decomposition containment 4.6.2, we know that each window in θ is contained in a window in θ'. According to Lemma 4.2, if a window is equivalent then a window that contains it is also equivalent. We can deduce that every window in θ' is equivalent, therefore the version pair is equivalent as per Lemma 4.3. □

4.6.3 Maximal Decompositions w.r.t. an EV

Lemma 4.4 shows that we can safely find decomposition that contain other ones to verify the equivalence of the version pair. At the same time, we cannot increase each window arbitrarily, since the equivalence of each window needs to be verified by the EV, and the window needs to satisfy the restrictions of the EV. Thus we want decompositions that are as containing as possible while each window is still valid. We formally define the following concepts.

Definition 4.12 (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 4.13 (Maximal Decomposition (MD))**.** *We say a valid decomposition θ is* maximal *if no other valid decomposition θ' exists such that θ' properly contains θ.*

The decompositions w.r.t an EV form a unique graph structure, where each decomposition is a node. It has a single root corresponding to the decomposition that includes every operator as a separate window. A downward edge indicates a "contained-in" relationship. A decomposition can be contained in more than one decomposition. Each leaf node at the bottom of the hierarchy is an MD as there are no other decompositions that contain it and the hierarchy may not be balanced. If the entire version pair satisfies the EV's restrictions, then the hierarchy becomes a lattice structure with a single leaf MD being the entire version pair. Each branching factor depends on the number of changes, the number of operators, and the EV's restrictions. Figure 4.15 shows the hierarchical relationships of the valid decompositions of the running example when the EV is Equitas. The example shows two MD θ_{12} and θ_{16}.

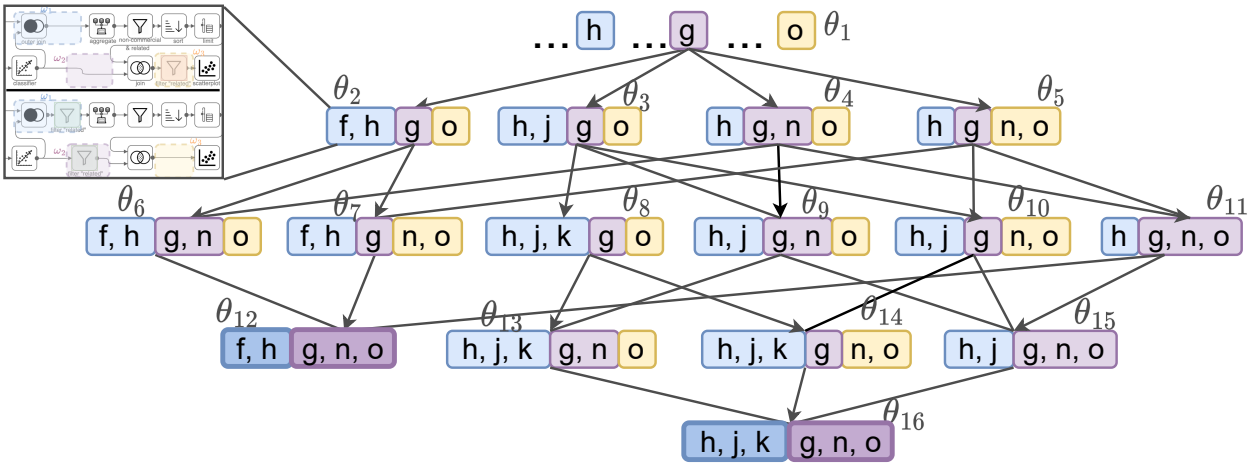


Figure 4.15: Hierarchy of valid decompositions w.r.t an EV. Each letter corresponds to a pair of operators from the running example. We show the containment of covering windows and we omit details of containment of non-covering windows.

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

Now we present an algorithm for finding maximal decompositions shown in Algorithm 4.2. We will explain it using the example in Figure 4.15. We return `True` to indicate the pair is equivalent if there are no changes and the two versions exactly match (Line 1-2). Otherwise, we add the initial decomposition, which includes each operator as a window, to a set of decompositions to be expanded (line 3). In each of the following iterations, we remove a decomposition from the set, and iteratively expand its windows. To expand a window, we follow the procedure as in Algorithm 4.1 to expand its neighbors. The only difference is that the neighbors in this case are windows, and we merge windows if their union is valid (line 10). If a window cannot be further expanded, then we mark the window as maximal to avoid checking it again (line 14). If all of the windows in the decomposition are maximal, we mark the decomposition as maximal, and verify whether each covering window is equivalent by passing it to the given EV (line 17). If all of the windows are verified to be equivalent, we return `True` to indicate that the version pair is equivalent (line 18). If in the decomposition there is only a single window, which includes the entire version pair, and the EV decides

that the window is not equivalent, then the algorithm returns `False` (line 20). Otherwise, we continue exploring other decompositions until there are no more decompositions to explore. In that case, we return `Unknown` to indicate that the equivalence of the version pair cannot be determined (line 22). This algorithm generalizes Algorithm 4.1 to handle cases of two versions with multiple edits.

Algorithm 4.2: Verifying the equivalence of a workflow version pair with one or multiple edits (Baseline)

Input: A version pair (P, Q); A set of edit operations δ and a mapping $\mathcal{M}$ from P to Q; An EV γ
Output: A version pair equivalence flag EQ
// A `True` value indicates the pair is equivalent, a `False` value indicates the pair is not equivalent, and an `Unknown` value indicates the pair cannot be verified

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

Theorem 4.5. *(Correctness). Given a workflow version pair (P, Q), an edit mapping, and a sound EV, 1) if* Veer *returns* `True`*, then $P \equiv Q$, and 2) if* Veer *returns* `False`*, then $P \not\equiv Q$.*

Proof. 1) Suppose $P \not\equiv Q$. According to definition 4.3.2, this means that for a given input

sources $\mathbb{D}$, there is a tuple t that exists in the sink of P but does not exist in the sink of Q. Following Assumption 4.3.2 that multiple runs of a workflow produce the same result, we can infer that there must be a set of edit operations $\delta = \{c_1, \ldots, c_n\}$ to transform P to Q that caused the sink of P to contain the tuple t but does not exist in the sink of Q. Veer must find a valid maximal decomposition θ following Algorithm 4.2. There are four cases the procedure terminates and returns the result:

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

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

`CASE3`: There is a decomposition that is verified to be equivalent by a correct EV, which according to Lemma 4.3, implies that the version pair is equivalent given the assumptions in our setting. However, this is not the case because we assumed $P \not\equiv Q$.

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

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

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

4.6.5 Improving the Completeness of Algorithm 4.2

In general, the equivalence problem for two workflow versions is undecidable [56, 3] (reduced from First-order logic). So there is no verifier that is complete [38]. However, there are

classes of queries that are decidable such as SPJ [163]. In this section, we show factors that affect the completeness of Algorithm 4.2 and propose ways to improve its completeness.

1) Window validity. In line 13 of Algorithm 4.2, if none of the neighbor windows of ω_j can be merged with ω_j to become a valid window, we mark ω_j and stop expanding it, hoping it might be a maximal window. The following example shows that this approach could miss some opportunity to find the equivalence of two versions.

Example 4.1. *Consider the following two workflow versions:*

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

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

Consider the following mapping from P to Q: substituting Project in P with Aggregate in Q and substituting Aggregate in P with Project in Q. Suppose the EV is Equitas *and a covering window ω contains the Project from P and its mapped operator Aggregate from Q. Consider the window expansion procedure in Algorithm 4.2. If we add filter operator of both versions to the window, then the merged window is not valid. The reason is that it violates Equitas's restriction R5 in Section 4.5.2, i.e., both DAGs should have the same number of Aggregate operators. The algorithm thus stops expanding the window. However, if we continue expanding the window till the end, the final window with three operators is still valid.*

Using this final window, we can see that the two versions are equivalent, but the algorithm missed this opportunity. This example shows that even though the algorithm is correct in terms of claiming the equivalence of two versions, it may miss opportunities to verify their equivalence. A main reason is that the Equitas EV does not have the following property.

Definition 4.14 (EV's Restriction Monotonicity)**.** *We say an EV is* restriction monotonic

if for each version pair P and Q, for each invalid window ω, each containing window of ω is also invalid.

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

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

2) Different edit mappings.

Consider two different edit mappings, $\mathcal{M}_1$ and $\mathcal{M}_2$, for the version pair in Example 4.6.5, as shown in Figure 4.16. Let us assume the given EV is Equitas. If we follow the baseline Algorithm 4.2, mapping $\mathcal{M}_1$ results in a decomposition that violates Equitas's **R2** restriction. On the other hand, mapping $\mathcal{M}_2$ satisfies the restrictions and allows the EV to test their equivalence. This example shows that different edit mappings can lead to different decompositions. Notably, the edit distance of the first mapping is 2, while the edit distance of the second one is 4. This result shows

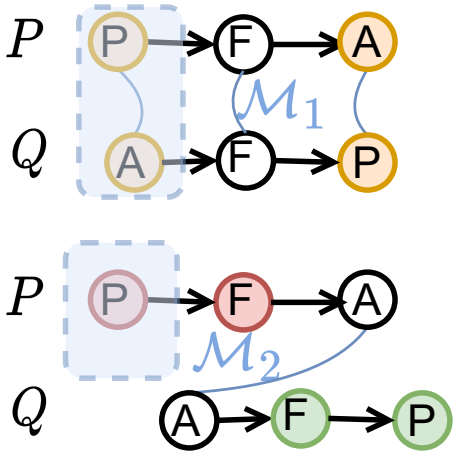


Figure 4.16: An example of two edit mappings, where one leads to a decomposition that satisfies an EV's restrictions, while the other does not.

that a minimum-distance edit mapping does not always produce the best decomposition to show the equivalence.

One way to address this issue is to enumerate all possible edit mappings [120] and perform the decomposition search by calling Algorithm 4.2 for each edit mapping. If the changes between the versions are tracked, then the corresponding mapping of the changes can be treated as the first considered edit mapping before enumerating all other edit mappings.

4.7 Completeness of Veer

In this section, we first discuss the completeness of Veer and the dependence on the completeness of its internal components (§4.7.1). Then we show examples of using different EVs to illustrate the restrictions that a workflow version pair needs to satisfy for Veer to be complete for verifying the pair's equivalence and formally prove its completeness (§4.7.2).

4.7.1 Veer's Completeness Dependency on Internal Components

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

given EV. If the EV is incomplete and returns `Unknown` to all possible valid decompositions generated by Veer, accordingly Veer returns `Unknown`.

The completeness of modern EVs [164, 163, 37, 56, 146] depends on the internal components used. For instance, most EVs [164, 163, 37, 56, 146] model queries as expressions, such as FOL formulas, and utilize a solver, e.g., SMT, to determine the satisfiability of formulas. SMT solvers [43] are complete for testing the satisfiability of linear formulas [24]. Therefore, EVs that use SMT solvers in their internal verification procedure are complete for verifying the equivalence of two queries (workflows or SQL) when the two queries include only linear conditions in their predicates. Likewise, Veer is complete for verifying the equivalence of two workflow versions that satisfy the EV's restrictions. Figure 4.17 illustrates the internal components Veer uses and how these components contribute to Veer's overall completeness.

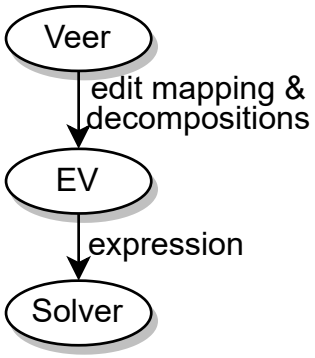


Figure 4.17: Components related to the equivalence verification process.

4.7.2 Restrictions of Some EVs and Veer's Completeness

We use the following examples on three EVs (summarized in Table 4.3) to explain Veer's completeness process. Suppose a given EV is Spes [163]. Spes determines the equivalence of two queries under the "Bag" table semantics. Spes is complete for determining the equivalence of two queries that satisfy the following restrictions [163]: 1) the two queries should contain only SPJ operators; 2) the selection predicates in every query should not include non-linear conditions.

Lemma 4.6. *Given two SPJ workflow versions (P, Q),* Spes *as the EV, and* Spes' *restrictions, if the two versions satisfy* Spes' *restrictions, then* Veer *can determine the equivalence of the pair. That is 1) if the two versions are equivalent, then* Veer *returns* `True`*, 2) and if*

Table 4.3: Example EVs and their restrictions along with how Veer is complete for verifying a version pair that satisfy the EV's restrictions.

EV	EV's restrictions	Explanation of why Veer is complete
Spes [**163**]	1. the pair should not include other than Select-Project-Join (SPJ.) 2. queries should have only linear predicate conditions.	Veer finds all possible windows that satisfy the EV's restrictions, and in this example, the given pair satisfies the restrictions, so Veer can find it.
UDP [**37**]	1. the pair should not include other than Union-SPJ (USPJ.) 2. the two versions must have a bijective mapping.	Veer finds all possible windows that satisfy the EV's restrictions under all possible mappings, and in this example, Veer finds a bijective mapping if there exists one.
Spark Verifier [**56**]	1. the pair should not include other than SPJ-Aggregate (SPJA.) 2. there should not be more than one aggregate operator in each version such that the aggregate is without grouping and outputs a primitive, e.g., MAX and MIN.	Veer finds all possible windows that satisfy the EV's restrictions, and in this example, the given pair satisfies the restrictions, so Veer finds it.

the pair is not equivalent then Veer *returns* `False`.

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

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

a) `False`. According to Algorithm 4.2, Veer returns `False` when there is a decomposition with a single window of the entire version pair and verified inequivalent by the EV. In this case, $P \not\equiv Q$, which contradicts the assumption.

b) `Unknown`. Veer calls Algorithm 4.2 multiple times on all possible mappings. Algorithm 4.2 returns `Unknown` when all valid decompositions w.r.t. the given EV (Spes) are explored and there was no valid decomposition that each of its covering windows was verified equivalent. This contradicts the given constraint that the pair satisfies Spes's restrictions, and Veer finds all possible valid windows and a window that includes the entire version pair is one of the possible valid windows.

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

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

□

Theorem 4.7. *(Completeness.)* Veer *using an EV is complete for determining the equivalence of two workflow versions if the pair satisfies the restrictions of the EV.*

Proof. Suppose the two versions satisfy the restrictions of the EV used in Veer. Since Veer considers all possible mappings and all possible decompositions that satisfy the EV's restrictions, it will for sure find a decomposition with a single window that includes the entire pair because the given pair satisfies the EV's restriction. According to Definition 4.5.2, the EV is able to determine the equivalence of a pair. □

4.8 Veer$^+$: Improving Verification Performance

In this section, we develop four techniques to improve the performance of the baseline algorithm for verifying the equivalence of two workflow versions. We show how to reduce the search space of the decompositions by dividing the version pair into segments (Section 4.8.1). We present a way to detect and prune decompositions that are not equivalent (Section 4.8.2). We also discuss how to rank the decompositions to efficiently explore their search space (Section 4.8.3). Lastly, we propose a way to efficiently identify the inequivalence of two workflow versions (Section 4.8.4).

4.8.1 Reducing Search Space Using Segmentations

The size of the decomposition structure in Figure 4.15 depends on a few factors, such as the number of operators in the workflow, the number of changes between the two versions, and the EV's restrictions. When the number of operators increases, the size of the possible decompositions increases. Thus we want to reduce the search space to improve the performance of the algorithm.

The purpose of enumerating the decompositions is to find all possible cuts of the version pair to verify their equivalence. In some cases a covering window of one edit operation will never overlap with a covering of another edit operation, as shown in Figure 4.18. In this case, we can consider the covering windows of those never overlapping separately. Based on this observation, we introduce the following concepts.

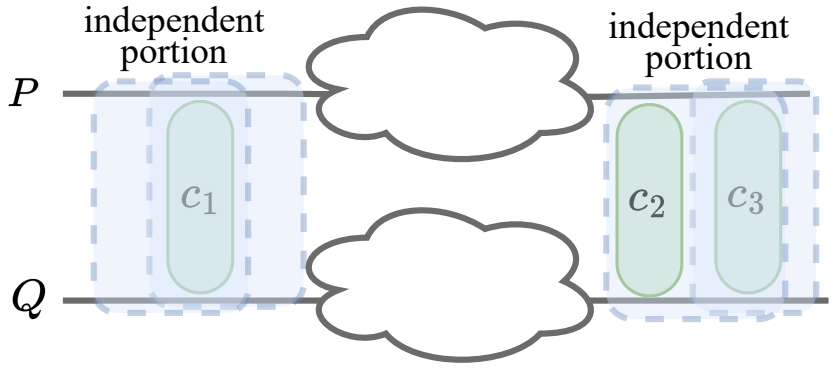


Figure 4.18: An example where any covering window of an edit operation c_1 never overlaps with a covering window of another edit operation c_2 or c_3.

Definition 4.15 (Segment and segmentation)**.** *Consider two workflow versions P and Q with a set of edits $\delta = \{c_1, \ldots, c_n\}$ from P to Q and a corresponding mapping $\mathcal{M}$ from P to Q. A* segment $\mathcal{S}$ *is a window of P and Q under the mapping $\mathcal{M}$. A* segmentation ψ *is a set of disjoint segments, such that they contain all the edits in δ, and there is no valid covering window that includes operators from two different segments.*

A version pair may have more than one segmentation. For example, consider a version pair with a single edit. One segmentation has a single segment, which includes the entire version pair. Another segmentation includes a segment that was constructed by finding the union of MCWs of the edit.

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

Using a segmentation to verify the equivalence of the version pair. As there is no valid covering window spanning over two segments, we can divide the problem of checking the equivalence of P and Q into sub-problems, where each sub-problem is to check the equivalence of the two sub-DAGs in a segment. Then to prove the equivalence of a version pair, each segment in a segmentation needs to be equivalent. A segment is equivalent, if there is any decomposition such that every covering window in the decomposition is equivalent. We can organize the components of the version pair verification problem as an AND/OR tree as shown in the Figure 4.19.

Lemma 4.8. *For a version pair P and Q with a set of edit operations $\delta = \{c_1 \dots c_n\}$ from P to Q, if every segment $\mathcal{S}$ in a segmentation ψ is equivalent, then the version pair is equivalent.*

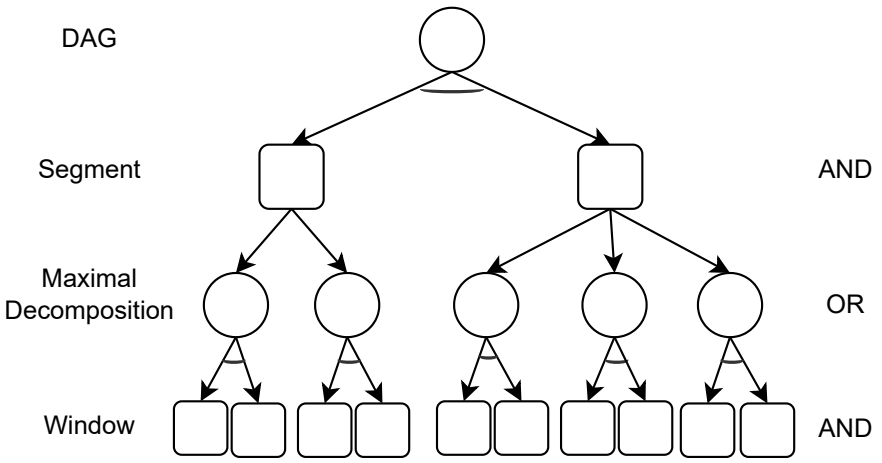


Figure 4.19: A sample abstract AND/OR tree to organize the components of the version pair verification problem.

Proof. Suppose every segment S_i in a segmentation ψ is equivalent. Since according to Definition 4.8.1 a segment is a window and each change is covered in all of the segments in a segmentation, then we can infer that any part of the version pair that is not in a segment is structurally identical. Following the same procedure of the proof for Lemma 4.3, we can say that the result is either from a structurally identical pair of sub-DAGs or from a segment, which is said to be equivalent. We can deduce that the version pair is equivalent. □

Algorithm 4.3 shows how to use a segmentation to check the equivalence of two versions. We first construct a segmentation. For each segment we find if its pair is equivalent by calling Algorithm 4.2. If any segment is not equivalent, we can terminate the procedure early. We repeat this step until all of the segments are verified equivalent and we return `True`. Otherwise we return `Unknown`. For the case where there is a single segment consisting of the entire version pair and Algorithm 4.2 returns `False`, the algorithm re-

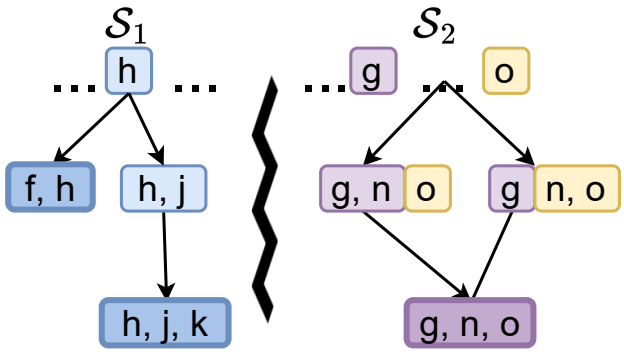


Figure 4.20: Two segments to reduce the decomposition-space of the running example.

turns `False`.

Algorithm 4.3: Using segments to verify the equivalence

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

```
// A True value indicates the pair is equivalent, a False value to indicate the version pair is not
equivalent, and an Unknown value indicates the pair cannot be verified
ψ ← constructSegmentation (P, Q, M)
for every segment S_i ∈ ψ do
    result_i ← Algorithm 4.2 (S_i, δ, M, γ)
    if result_i is not True then
        break
end
if every result_i is True then
    return True
if result_i is False and there is only one segment including the entire version pair then
    return False
return Unknown
```

Figure 4.20 shows the segments of the running example when using Equitas as the EV. Using the second approach for computing a segmentation, we know Equitas does not support the Sort operator, so we divide the version pair into two segments. The first one $\mathcal{S}_1$ includes those operators before Sort, and the second one $\mathcal{S}_2$ includes those operators after the Sort. The example shows the benefit of using segments to reduce the decomposition-space to a total of 8 (the sum of number of decompositions in every segment) compared to 16 (the number of all possible combinations of decompositions across segments) when we do not use segments.

4.8.2 Pruning Stale Decompositions

Another way to improve the performance is to prune *stale* decompositions, i.e., those that would not be verified equivalent even if they are further expanded. For instance, Figure 4.21 shows part of the decomposition hierarchy of the running example. Consider the decompo-

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

Based on this observation, if one of the windows in a decomposition becomes maximal, we can immediately test its equivalence. If it is not equivalent, we can terminate the traversal of the decompositions after this one. To do this optimization, we modify Algorithm 4.2 to test the equivalence of a maximal window after Line 14[7]. If the window is equivalent, we continue the search as before.

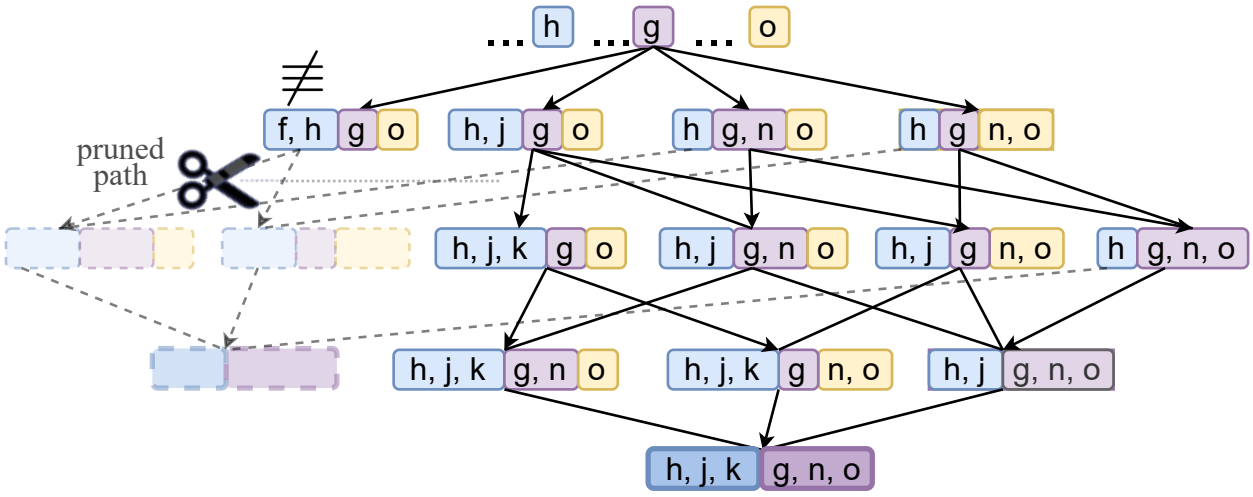


Figure 4.21: Example to show the pruned paths after verifying the maximal window highlighted in blue to be not equivalent.

4.8.3 Ranking-Based Search

Ranking segments within a segmentation. Algorithm 4.3 needs an order to verify those segments in a segmentation one by one. If any segment is not equivalent, then there is no need for verifying the other segments. We want to rank the segments such that we first evaluate

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

the smallest one to get a quick answer for a possibility of early termination. We consider different signals of a segment S to compute its score. Various signals and ranking functions can be used. An example scoring function is $\mathcal{F}(S) = m_S + n_S$, where m_S is its number of operators and n_S is its number of changes. A segment should be ranked higher if it has fewer changes. The reason is that fewer changes lead to a smaller number of decompositions, and consequently, testing the segment's equivalence takes less time. Similarly, if a segment's number of operators is smaller, then the number of decompositions is also smaller and would produce the result faster.

For instance, the numbers of operators in $\mathcal{S}_1$ and $\mathcal{S}_2$ in Figure 4.20 are 4 and 3, respectively. Their numbers of changes are 1 and 2, respectively. The ranking score for both segments is the total of both metrics, which is 5. Then any of the two segments can be explored first, and indeed the example shows that the number of decompositions in both segments is the same.

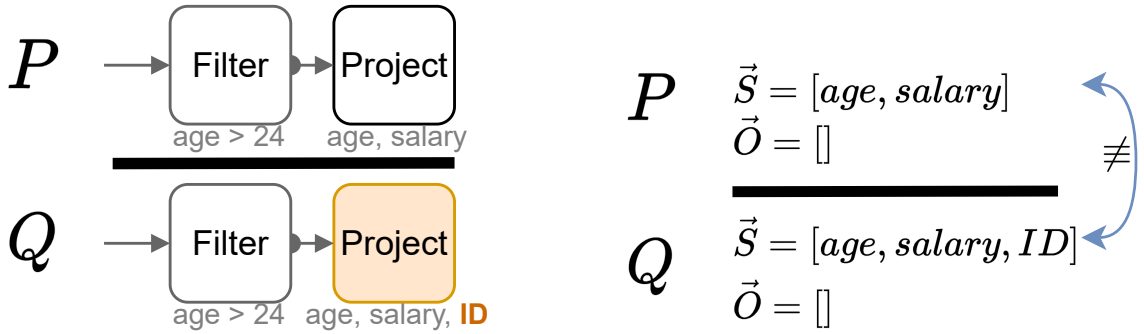
Ranking decompositions within a segment. For each segment, we use Algorithm 4.2 to explore its decompositions. The algorithm needs an order (line 6) to explore the decompositions. The order, if not chosen properly, can lead to exploring many decompositions before finding an equivalent one. We can optimize the performance by ranking the decompositions and performing a best-first search exploration. Again, various signals and ranking functions can be used to rank a decomposition. An example ranking function for a decomposition d is $\mathcal{G}(d) = o_d - w_d$, where o_d is the average number of operators in its covering windows, and w_d is the number of its unmerged windows (those windows that include a single operator and are not merged with a covering window). A decomposition is ranked higher if it is closer to reaching an MD for a chance of finding an equivalent one. Intuitively, if the number of operators in every covering window is large, then it may be closer to reaching an MD. Similarly, if there are only a few remaining unmerged windows, then the decomposition may be close to reaching its maximality.

For instance, decomposition θ_3 in Figure 4.15 has 11 unmerged windows, and the average number of operators in its covering windows is 1. While θ_6 has 10 unmerged windows, and the average number of operators in its covering windows is 2. Using the example ranking function, the score of θ_3 is $1 - 11 = -10$ and the score of θ_6 is $2 - 10 = -8$. Thus, θ_6 is ranked higher, and it is indeed closer to reaching an MD.

4.8.4 Identifying Inequivalent Pairs Efficiently

In this section, we use an example to show how to quickly detect the inequivalence between two workflow versions using a symbolic representation to represent partial information of the result of the sink of each version. Consider the case where two workflow versions P and Q are inequivalent, as shown in Figure 4.22a. The approach discussed so far attempts to find a decomposition in which all its windows are verified equivalent. However, in cases where the version pair is inequivalent, as in this example, such a decomposition does not exist, and the search framework would continue to look for one unsuccessfully. Moreover, detecting the inequivalence of the pair happens if there exists a decomposition that includes the entire version pair and satisfies the given EV's restrictions. Although the cost of maximizing the window and testing if it is valid could be low, testing the equivalence of maximal decompositions by pushing it to the EV incurs an overhead due to the EV's reasoning about the semantics of the window. Thus, we want to avoid sending a window to the EV if we can quickly determine beforehand that the version pair is not equivalent.

To quickly identify the inequivalence between two workflow versions, our approach is to create a lightweight representation that allows us to partially reason about the semantics of the version pair. This approach relies on a symbolic representation similar to other existing methods [89, 164], denoted as $(\vec{S}, \vec{O})$. In this representation, $\vec{S}$ and $\vec{O}$ are lists that represent the projected columns in the table and the columns on which the result table is sorted,



(a) A sample of two inequivalent workflow versions.

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

Figure 4.22: Example of two inequivalent workflow versions and their partial symbolic representation.

respectively. To construct the representation, we follow the same techniques in existing literature [37, 164] by using predefined transformations for each operator. Operators inherit the representation from their upstream/parent operator and update the fields based on their internal logic. In this way, if the list of projected columns (based on $\vec{S}$) of version P is different from those in Q, as in Figure 4.22b, we can know the two versions do not produce the same results. We can apply the same check to the sorted columns.

4.9 Extensions

In this section, we discuss relaxing the definition of EV's restrictions then discuss the consequences of relaxing the definition and propose extending Algorithm 4.2 to handle incomplete EVs and handle multiple given EVs.

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

Definition 4.16 (Relaxed EV's restrictions)**.** Restrictions *of an EV are a set of conditions such that for each query pair if this pair satisfies these conditions, then the EV can attempt to determine the equivalence of the pair.*

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

Handling greedy window verification when an EV is incomplete. In Line 16 of Algorithm 4.2 we push testing the equivalence of a decomposition to the given EV only when the decomposition is marked maximal. The following example shows that this approach could miss some opportunity to find the equivalence of two versions, because the EV is not able to verify the equivalence of the two sub-DAGs in the maximal window.

Example 4.2. *Consider two workflow versions P and Q with a single edit c based on a given mapping $\mathcal{M}$. Let γ be a given EV. Suppose a covering window ω_c satisfies the restrictions of γ, and the EV is able to verify the equivalence of the two sub-DAGs in ω_c. According to Algorithm 4.2, we do not check the equivalence of this window if it is not marked maximal. Let ω' be the only MCW that contains ω. Following Line 16 in Algorithm 4.2, if a window is maximal (ω' in this example) we push testing its equivalence to the EV. However, suppose in this case, the EV returns* `Unknown`*, because EVs are mostly incomplete [38] for verifying two general relational expressions. Since there is no other MCW to test,* Veer *accordingly returns* `Unknown` *for verifying the equivalence of the sub-DAGs in ω'. However, if we pushed testing the equivalence of the smaller window ω_c, then* Veer *would have been able to verify the equivalence of the pair.*

This example highlights the significance of verifying the equivalence of sub-DAGs within smaller windows before expanding to larger windows. The challenge arises when an EV can verify the equivalence of a small window but fails to do so for a larger one. To address this, we modify Line 16 in Algorithm 4.2 to check the equivalence of smaller windows by

backtracking when the maximal window is not verified. This modification ensures that we do have more opportunities to verify the equivalence of the version pair. Note that this approach may introduce a computational overhead due to the repeated checking of each window and not just the maximal ones.

Using multiple EVs. As mentioned earlier, the problem of verifying the equivalence of two relational expressions is undecidable [3]. Thus, any given EV would have limitations and is incomplete for solving the problem of deciding the equivalence of two queries. To harness the capabilities of different EVs, we extend Veer to take in a set of EVs and their associated restrictions. We do not modify Algorithm 4.2. However, we extend the 'isValid' function in Line 9 to encode a window is valid w.r.t which EV so that when we verify the equivalence of the sub-DAGs in the window in Line 17, we call the corresponding EV the window satisfies.

4.10 Experiments

4.10.1 Experimental Setup

Synthetic workload. We constructed four workflows $W1 - W4$ on TPC-DS [140] dataset as shown in Table 4.4. For example, workflow $W1$'s first version was constructed based on TPC-DS Q40, which contains 17 operators including an outer join and an aggregate operator. Workflow $W2$'s first version was constructed based on TPC-DS Q18, which contains 20 operators. We omit details of other operators included in the workflows such as Unnest, UDF, and Sort as these do not affect the performance of the experimental result as we explain in each experiment.

Real-world workload. We analyzed a total of 179 real-world pipelines from Texera [138] as summarized in Appendix A. We show a sample of some workflows used in the experiments

in Appendix B. Among the workflows, 81% had deterministic sources and operators, and we focused our analysis on these workflows. Among the analyzed workflows, 8% consisted primarily of 8 operators, and another 8% had 12 operators. Additionally, 33% of the workflows contained 3 different versions, while 19% had 35 versions. 58% of the versions had a single edit, while 22% had two edits. We also observed that the UDF operator was changed in 17% of the cases, followed by the Projection operator (6% of the time) and the Filter operator (6% of the time). From this set of workflows, we selected four as a representative subset, which is presented as $W5 \ldots W8$ in Table 4.4 and we used IMDB [70] and Twitter [142] datasets.

Table 4.4: 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 operators	17	16	4/5
$W2$	TPC-DS $Q18$	5 joins and 1 aggregate operators	20	20	8/9
$W3$	TPC-DS $Q71$	1 replicate, 1 union, 5 joins, and 1 aggregate operators	23	23	3/4
$W4$	TPC-DS $Q33$	3 replicates, 1 union, 9 joins, and 4 aggregates operators	28	34	2/3
$W5$	IMDB ratio of non-original to original movie titles	1 replicate, 2 joins, 2 aggregate operators	12	12	2/3
$W6$	IMDB all movies of directors with certain criteria	2 replicates, 4 joins, 2 unnest operators	18	20	2/3
$W7$	Tobacco Twitter analysis	1 outer join, 1 aggregate, classifier	14	13	2/3
$W8$	Wildfire Twitter analysis	1 join, 1 UDF	13	12	2/3

Edit operations. Table 4.4 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 [27] for equivalent pairs: These edits are common for rewriting and optimizing workflows, so these edits would produce a version that is *equivalent* to the first version. For example, 'testEmptyProject' is a single edit of adding an empty projection operator. In addition, 'testPushProjectPastFilter' and 'testPushFilterPastAgg' are two example edits that produce more than a single change, in particular, one for deleting an operator and another is for pushing it past another operator. We used a variation of a different number of edits and their placements as we explain in each experiment.

(2) TPC-DS V2.1 [140] iterative edits for inequivalent pairs: These edits are common for exploratory and the early stages of the iterative analytics, so they may produce a version that is *not equivalent* to the first version. Example edits are adding a new filtering condition or changing the aggregate function as in TPC-DS queries. We constructed one version for each workflow using two edit operations from this type of transformations to test our solution when the version pair is not equivalent.

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

Implementation. We implemented the baseline (Veer) and an optimized version (Veer$^+$), by including the optimization techniques presented in Section 4.8, in Java8. We implemented Equitas [164] as the EV in Scala. We evaluated the solution by comparing Veer and Veer$^+$ against a state of the art verifier (Spes), 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.

4.10.2 Comparisons with Other EVs

To our best knowledge, Veer is the first technique to verify the equivalence of complex workflows. To evaluate its performance, we compared Veer and Veer$^+$ against Spes. We chose one equivalent pair and one inequivalent pair of versions with two edits from each workflow. Among the 8 workflows examined, Spes failed to verify the equivalence and inequivalence of any of the pairs, because all of the workflow versions included operators not supported by Spes. In contrast, Veer and Veer$^+$ successfully verified the equivalence of 50% ($W1, W2, W3, W5$) and 75% ($W1 \dots W6$), respectively, of the equivalent pairs as reported in details in Section 4.10.4. 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. Moreover, 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 in our experiments. Veer$^+$ was able to use the heuristic discussed in Section 4.8.4 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 4.5 summarizes the evaluation of the compared techniques.

Table 4.5: Comparison evaluation of Veer and Veer$^+$ against Spes.

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

4.10.3 Evaluating Veer$^+$ Optimizations

We used an equivalent version pair from workflow $W3$ for evaluating the first three optimization techniques discussed in Section 4.8. We used three edit operations: one edit was after the Union operator (which is not supported by Equitas) and two edits (pushFilterPastJoin) were before the Union. We used Veer to verify the equivalence of the pairs, and we tried different combinations of enabling Veer$^+$'s optimization techniques.

Table 4.6 shows the result of the experiments. The worst performance was the baseline itself when all of the optimization techniques were disabled, resulting in a total of $19,656$ decompositions explored in 27 minutes. When only "pruning" was enabled, it was slower than all of the other combinations of enabling the techniques because it tested 108 MCWs for possibility of pruning them. Its performance was better than the baseline thanks to the early termination, where it resulted in $3,614$ explored decompositions in 111 seconds. When "segmentation" was enabled, there were only two segments, and the total number of explored decompositions was lower. In particular, when we combined "segmentation" and "ranking", one of the segments had 8 explored decompositions while the other had 13. If "segmentation" was enabled without "ranking", then the total number of explored decompositions was 430, which was only 2% of the number of explored decompositions when "segmentation" was not enabled. The time it took to construct the segmentation was negligible. When "ranking" was enabled, the number of decompositions explored was around 21. It took an average of 0.17 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 4.6: Result of enabling optimizations ($W3$ with three edits). "S" indicates segmentation, "P" indicates pruning, and "R" indicates ranking. A ✓means the optimization was enabled, a × means the optimization was disabled.

S	P	R	# of decompositions 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	0.82	0.20	1.02
✓	×	×	430	0.51	0.18	0.69
×	✓	✓	20	0.39	0.12	0.52
✓	✓	✓	21	0.20	0.31	0.51
×	×	✓	20	0.07	0.23	0.30
✓	×	✓	21	0.03	0.21	0.24

4.10.4 Comparing Veer and Veer$^+$ on Verifying Two Versions with Multiple Edits

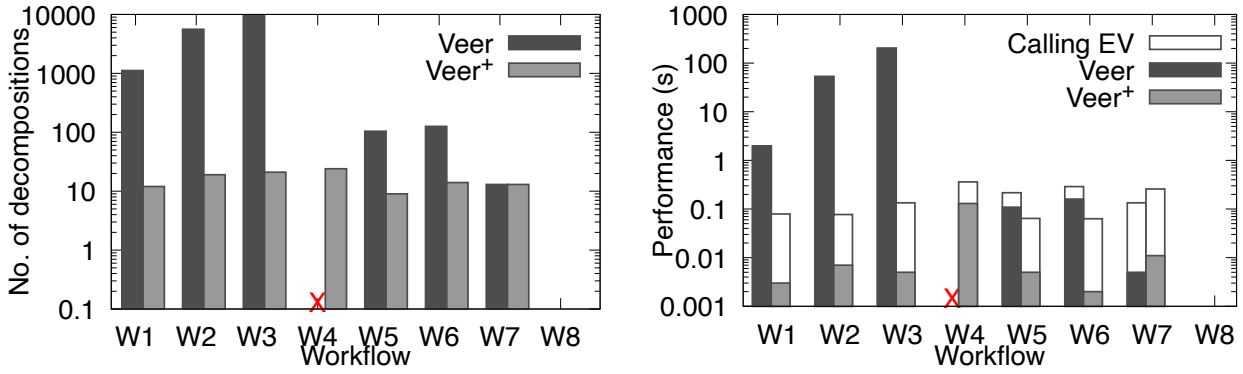
We compared the performance of the baseline and Veer$^+$. We want to know how much time each approach took to test the equivalence of the pair and how many decompositions each approach explored. We used workflows $W1 - W8$ with two edits. We used one equivalent pair and one inequivalent pair from each workflow to evaluate the performance in these two cases. Most workflows in the experiment had one segment, except workflows $W3$, $W5$, and $W6$, each of which has two segments. The overhead of calling 'is maximal' (line 13), 'is valid' (line 9), and 'merge' (line 10) in Algorithm 4.2 was negligible, thus we only report the overhead of calling the EV.

Performance for verifying equivalent pairs. Figure 4.23a shows the number of decompositions explored by each approach. In general, the baseline explored more decompositions, with an average of 3,354 compared to Veer$^+$'s average of 16, which is less than 1% of the baseline. The baseline was not able to finish testing the equivalence of $W4$ in less than an hour. The reason is because of the large number of neighboring windows that were caused by a large number of links in the workflow. Veer$^+$ was able to find a segmentation for $W3$

and $W6$. It was unable to discover a valid segmentation for $W5$ because all of its operators are supported by the EV, while we used the second approach of finding a segmentation as we discussed in Section 4.8.1. We note that the overhead of constructing a segmentation using the second approach was negligible. For workflow $W7$, the size of the windows in a decomposition were small because the windows violated the restrictions of the used EV. Therefore, the "expanding decompositions" step stopped early and thus the search space (accordingly the running time) was small for both approaches. Veer and Veer$^+$ detected that the change on $W8$ was done on a non-supported operator (UDF) by the chosen EV (Equitas), thus the decomposition was not expanded to explore other ones and the algorithm terminated without verifying its equivalence.

Figure 4.23b shows the running time for each approach to verify the equivalence. The baseline took 2 seconds to verify the equivalence of $W1$, and 2 minutes for verifying $W3$. Veer$^+$, on the other hand, had a running time of a sub-second in verifying the equivalence of all of the workflows. Veer$^+$ tested 9 MCWs for a chance of pruning inequivalent decompositions when verifying $W6$. This caused the running time for verifying $W6$ to increase due to the overhead of calling the EV. In general, the overhead of calling the EV was about the same for both approaches. In particular, it took an average of 0.04 and 0.10 seconds for both the baseline and Veer$^+$, respectively, to call the EV.

Performance of verifying inequivalent pairs. Figure 4.24a shows the number of decompositions explored by each approach. Since the pairs are not equivalent, Veer almost exhaustively explored all of the possible decompositions, trying to find an equivalent one. Veer$^+$ explored fewer decompositions compared to the baseline when testing $W3$, thanks to the segmentation optimization. Both approaches were not able to finish testing $W4$ within one hour because of the large number of possible neighboring windows. Veer$^+$ was able to quickly detect the inequivalence of the pairs of workflows $W5 \ldots W8$ thanks to the partial symbolic representation discussed in Scetion 4.8.4, resulting in Veer$^+$ not exploring any



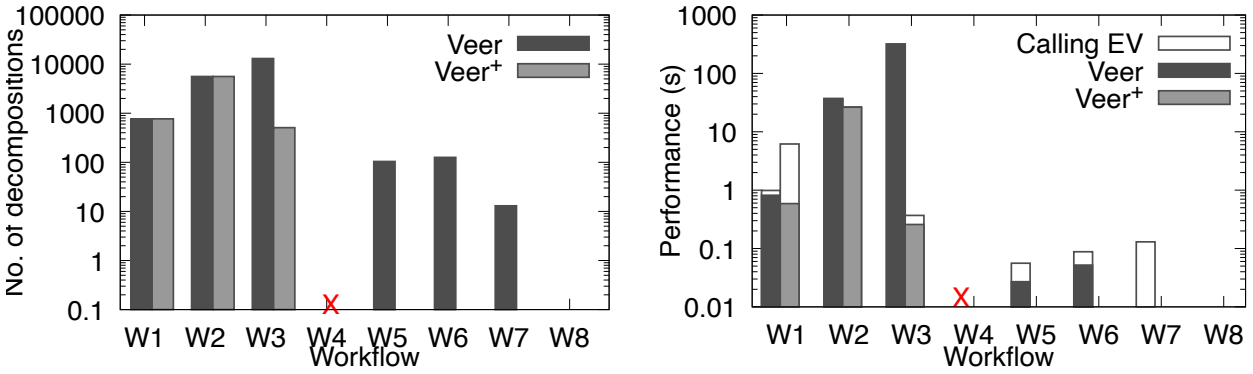
(a) # of explored decompositions. **(b) Running time.**

Figure 4.23: Comparison between Veer and Veer$^+$ for verifying equivalent pairs with two edits. An "×" means the algorithm was not able to finish running within one hour.Overhead of calling EV by Veer is not visible due to the logscale.

decompositions for these workflows.

The result of the running time of each approach is shown in Figure 4.24b. Veer's performance when verifying inequivalent pairs was the same as when verifying equivalent pairs because, in both cases, it explored the same number of decompositions. On the other hand, Veer$^+$'s running time was longer than when the pairs were equivalent for workflows $W1 \ldots W4$. We observe that for $W1$, Veer$^+$'s running time was even longer than the baseline due to the overhead of calling the EV up to 130, compared to only 4 times for the baseline. Veer$^+$ called the EV more as it tried to continuously test MCWs when exploring a decomposition for a chance of pruning inequivalent decompositions. Veer$^+$'s performance on $W3$ was better than the baseline. The reason is that there were two segments, and each segment had a single change. We note that Veer$^+$ tested the equivalence of both segments, even though there could have been a chance of early termination if the inequivalent segment was tested first. The time it took Veer$^+$ to verify the inequivalence of the pairs in workflows $W5 \ldots W8$ was negligible. The heuristic approach was not effective in detecting the inequivalence of the TPC-DS workflows $W1 \ldots W4$. This limitation arises from the technique's reliance on identifying differences in the *final* projected columns, which remained the same across all versions of these workflows (due to the aggregation operator), with most changes occurring

in the filtering conditions.



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

Figure 4.24: 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.

4.10.5 Effect of the Distance Between Edits

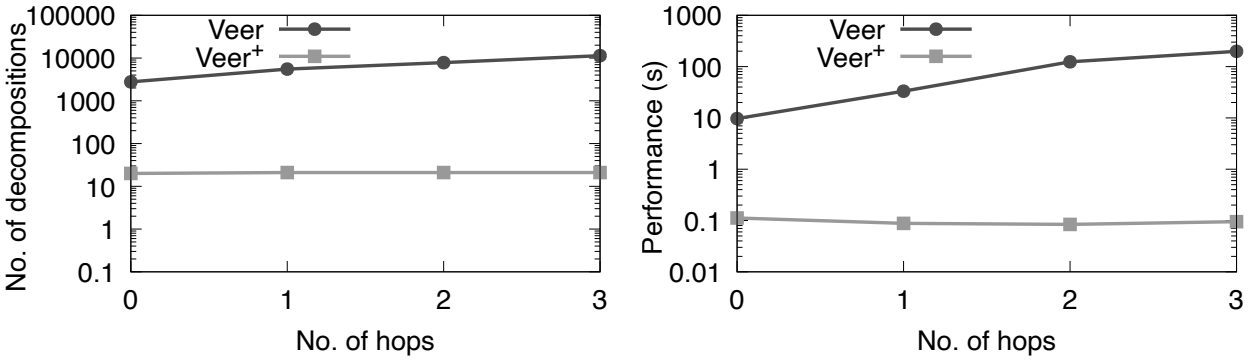
We evaluated the effect of the placement of changes on the performance of both approaches. We are particularly interested in how many decompositions would be explored and how long each approach would take if the changes were far apart or close together in the version DAG. We used an equivalent version 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. For a fair comparison, the operators that were separating the changes were one-to-one operators, i.e., operators with one input and one output links.

Figure 4.25a shows the number of decompositions explored by each approach. The baseline's number of decompositions increased from $2,770$ to $11,375$ as the number of hops increased. This is because it took longer for the two covering windows, one for each edit, to merge into a single one. Before the two covering windows merge, each one produces more decompositions to explore due to merging with its own neighbors. Veer$^+$'s number of explored decompositions remained the same at 21 thanks to the ranking optimization, as once one covering window

includes a neighboring window, its size is larger than the other covering window and would be explored first until both covering windows merge.

Figure 4.25b shows the time each approach took to verify the equivalence of a pair. The performance of each approach was proportional to the number of explored decompositions. The baseline took between 9.7 seconds and 3 minutes, while Veer$^+$'s performance remained in the sub-second range (0.095 seconds).

Effect of the type of changed operators. We note that when any of the changes were on an unsupported operator by the EV, then both Veer and Veer$^+$ were not able to verify their equivalence. We also note that the running time to 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.**

Figure 4.25: Effect of the distance between changes (on $W2$)

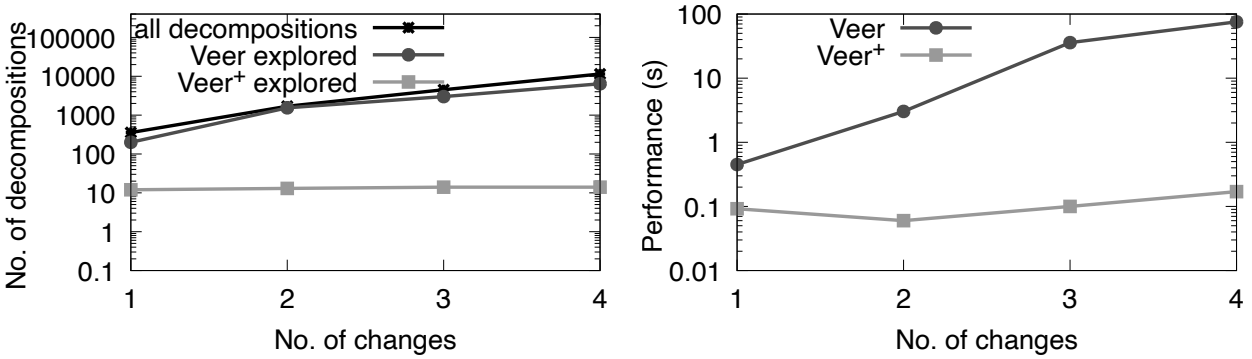
4.10.6 Effect of the Number of Changes

In iterative data analytics, when the task is exploratory, there can be many changes between two consecutive versions. Once the analytical task is formulated, there are typically only minimal changes to refine some parameters [154]. We want to evaluate the effect of the number of changes on the number of decompositions and the time each approach takes to

verify a version pair. The number of changes, intuitively, increases the number of initial covering windows, and consequently, the possible different combinations of merging with neighboring windows increases. We used an equivalent version pair of $W1$ in the experiment.

Figure 4.26a shows the number of decompositions explored by each approach and the total number of "valid" decompositions. The latter increased from 356 to 11,448 as we increased the number of changes from 1 to 4. The baseline explored almost all those decompositions, with an average of 67% of the total decompositions, in order to reach a maximal one 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 4.26b shows the time taken by each approach to verify the equivalence of a pair. Both approaches' time was proportional to the number of explored decompositions. The baseline showed a performance of around 0.42 seconds when there was a single change, up to slightly more than a minute at 75 seconds when there were four changes. Veer$^+$, on the other hand, maintained a sub-second performance with an average of 0.1 seconds.



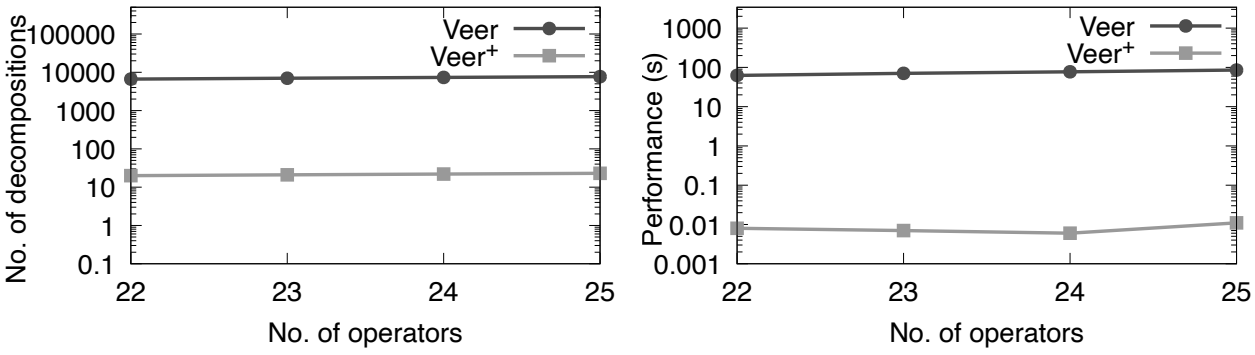
(a) # of explored decompositions. **(b) Running time.**

Figure 4.26: Effect of the number of changes (on $W1$).

4.10.7 Effect of the Number of Operators

We evaluated the effect of the number of operators. We used an equivalent version pair from $W2$ with two edits and varied the number of operators from 22 to 25. We varied the number of operators in two different ways. One was varying the number of operators by including only those supported by the EV. The other type was varying the number of non-supported operators.

Varying the number of supported operators. Figure 4.27a shows the number of explored decompositions. The baseline explored 6,650 decompositions when there were 22 operators, and 7,700 decompositions when there were 25 operators. Veer$^+$ had a linear increase in the number of explored decompositions from 21 to 24 when we increased the number of operators from 22 to 25. 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 4.27b.



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

Figure 4.27: Effect of the number of operators (on $W2$).

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

4.11 Conclusion

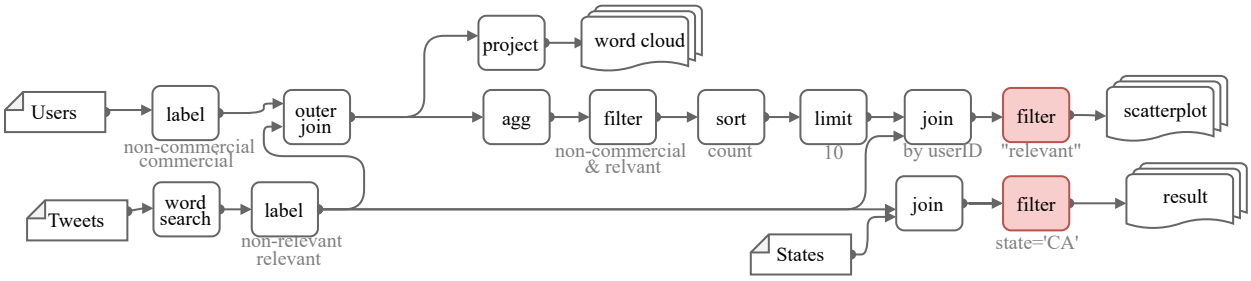
In this chapter, we studied the problem of verifying the equivalence of two workflow versions. We presented a solution called "Veer," which leverages the fact that two workflow versions can be very similar except for a few changes. We analyzed the restrictions of existing EVs and presented a concept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

Chapter 5

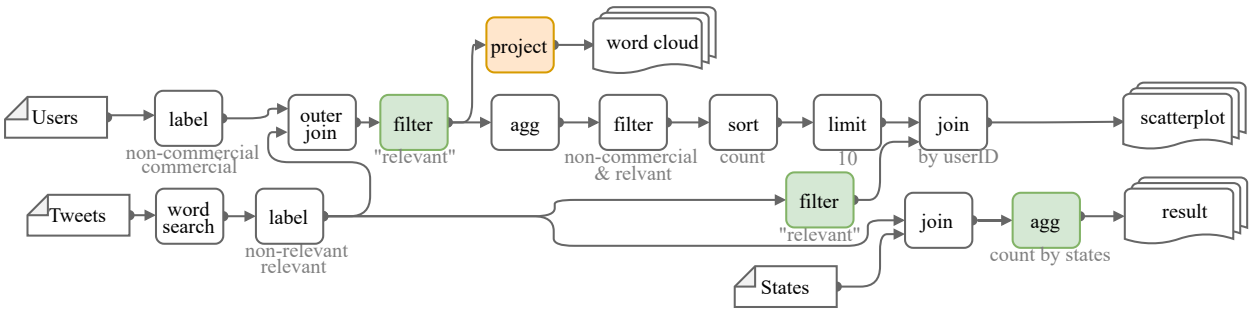
Raven: Accelerating Execution of Iterative Data Analytics by Reusing Results of Previous Equivalent Versions

5.1 Introduction

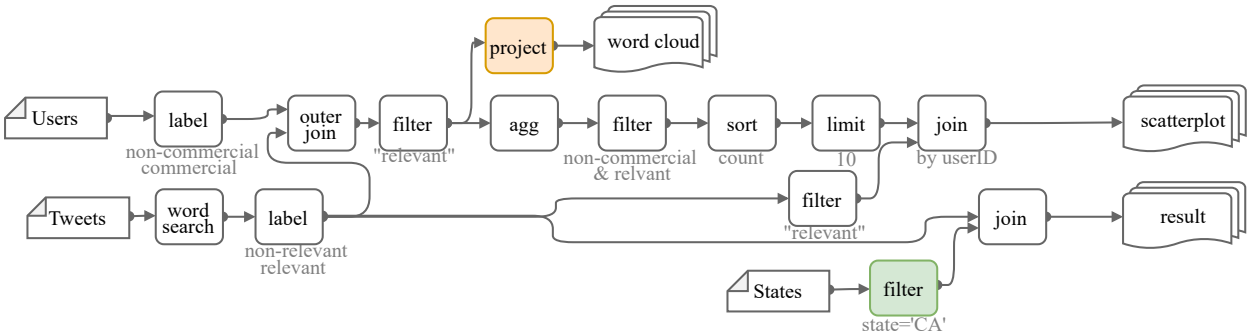
When an analyst employs workflows for data analytics, she starts with a basic workflow and iteratively revises it based on the observed execution results as part of the iterative process of data analytics [45, 154]. She may edit the operators and links in the workflow during each iteration, producing a new *version* of the workflow. Figure 5.1 shows an overview of a workflow for analyzing tweets related to popular wildfires. 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.



(a) Version 1: An initial workflow with three sinks: a *wordcloud*, a *scatterplot*, and a *result*. The highlighted operators in red indicates that they are deleted in a subsequent version.



(b) Version 2: A refined version to optimize the workflow's performance and filter on relevant tweets of all users. Operators in green are newly added in the new version and operators in orange are modified.



(c) Version 3: A refined version to revert some of the earlier changes.

Figure 5.1: Three versions of a workflow for analyzing tweets mentioning a keyword.

Motivation. As an analyst iteratively refines a workflow, many versions can be created. For example, one deployment of Texera [93] recorded a total of 2,424 executions of different versions for one workflow [9]. Tracking the versions of a workflow and the outcomes of their executions has gained significant interest recently [31, 150]. One observation in many applications is that versions of the same workflow frequently produce the same results [163, 76]. 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 [76]. 27% of 9,486 workflows from Ant Financial to detect fraud transactions share common computation, and 6% of them are equivalent [163]. In the running example, the modifications applied to the initial version (in Figure 5.1a) to transform it into the one in Figure 5.1b resulted in the scatterplot sink to produce the same results as the corresponding sink in the initial version.

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 (UDFs) [84]. 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 request to execute a new version, 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, as 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 [49, 76] 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. For example, the two workflow versions in Figure 5.1a and Figure 5.1b 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 [154, 118] to identify redundant and *overlapping* tuples between multiple jobs. However, these works cannot identify the *exact* equivalence of the results of the two workflows.

We want to study the following the problem:

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 that include sinks producing results equivalent to those in the requested execution.*

Challenges and our approach. The problem of testing the equivalence of two queries has been studied for SQL [37], Spark programs [56], and workflow versions, such as Veer discussed in Chapter 4. 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. This approach 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. Moreover, 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 computation.

To address these challenges, we propose a novel framework called Raven, and we make the following contributions in this chapter:

1. We formulate the problem of accelerating the execution of a workflow version by detecting previously-stored semantically equivalent results from previous versions (§5.3).
2. We propose a framework and a novel technique to let a workflow optimizer reuse mate-

rialized results of previous equivalent versions of a workflow execution request through identifying equivalence using Veer (§5.4).

3. We extend Veer to handle the case of verifying the equivalence of versions with multiple sinks (§5.5).
4. We extend Veer to avoid repeated testing on portions of the workflow version pair, by reusing shared computations (§5.6).
5. We propose an approach for ranking the versions using a novel semantic-aware hierarchy to organize the saved results (§5.7).
6. We conduct a thorough experimental evaluation on a real-world dataset and workload to evaluate the solutions and compare them with existing works (§5.8).

5.2 Related Work

There is an extensive work on optimizing the performance of executing complex data-processing pipelines. These works are mainly categorized into rewrite based optimization [146, 16], and materialization reuse optimization [49]. We focus the comparison with the second class, which has been extensively studied as summarized in the following surveys [59, 2, 35].

Exact matching. Compared to general materialization reuse methods, exact matching is a more specific and syntax-based approach, commonly used in systems with high workloads [162, 76, 131]. 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 [72, 85, 113]. Restore [49] caches map-reduce intermediate jobs, while Recycler [102] uses a graph to recycle fine-grained partial query results. Nectar [57] caches

sub-computations that are likely to be reused. Raven aims to identify previous equivalent versions with respect to the final results even when there are structural differences.

Semantic reuse. Prior works such as Eva [154], Acorn [118], and the work in [89] proposed methods to semantically reuse previous results by using techniques, such as converting UDFs to native operators and predicate overlap detection. These methods focus on detecting overlap of tuples between the results but cannot identify that the collection of tuples are entirely equivalent.

Equivalence verification. Some works verify the equivalence of SQL queries under certain assumptions [37, 164]. These solutions cannot reason about UDF semantics, making them unsuitable for detecting the equivalence of two workflow versions, as they contain complex operators such as UDFs. Veer addresses this limitation by verifying the equivalence of two workflow versions with UDFs, and Raven leverages it in its solution.

5.3 Problem Formulation

In this section, we give an overview of the problem setting and we formally formulate the problem.

5.3.1 Iterative Data-Processing Workflows

We consider a workflow DAG as described in Section 4.3. We also consider the edit operations introduced in Section 4.3 to produce different versions, denoted as $\mathcal{V}_w = [v_1, \ldots, v_m]$, of a workflow W based on an edit mapping $\mathcal{M}$.

5.3.2 Workflow's Execution and Result Equivalence

A user submits a query request Q, corresponding to a version v_j, to execute it. The execution produces *artifacts*, which are the results of each sink $s_{ji} \in \mathbb{S}_w$ of the workflow version v_j. These artifacts are saved. In the running example, executing the workflow version v_1 produces two results corresponding to the scatterplot and wordcloud operators.

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 may produce a sink result equivalent to its mapped sink of a previously executed version. For example, in Figure 5.1b, executing the workflow version v_2 produces a result of the scatterplot sink equivalent to the result of the corresponding scatterplot of v_1. In particular, v_2's edit is pushing down the Filter operator and the scatterplot result remains the same. Notice that the result of the word cloud in v_2 is not equivalent to the result of its mapped sink in v_1 because of the addition of a new Filter operator. We consider the Definition of "sink equivalence" as discussed in Section 4.3.

Problem Statement. *Given a workflow version request Q corresponding to a version v_m, a list of previously executed versions of the workflow $\mathcal{V'}_w = [v_1, \ldots, v_{m-1}]$, and a workflow version equivalence verifier (α), we want to find a subset of the previously executed versions $\mathbb{P} \subseteq \mathcal{V'}_w$ such that for every sink s_j of Q, there is a sink s_i of a version $P \in \mathbb{P}$ that is equivalent to s_j, i.e., $s_i \equiv s_j$.*

5.4 Raven: Overview

Figure 5.2 gives an overview of the steps involved in Algorithm 5.1 to detail the optimization lifecycle to accelerate the execution of a workflow version DAG by Raven. Given an execution

[1] We will relax this in a future work as we propose in Chapter 6.2.

request for a workflow version Q (corresponding to the latest version v_m), the optimizer searches for a prior version $P \in [v_1, \ldots, v_{m-1}]$, which has sinks equivalent to some or all of the corresponding sinks in Q. It takes the following steps.

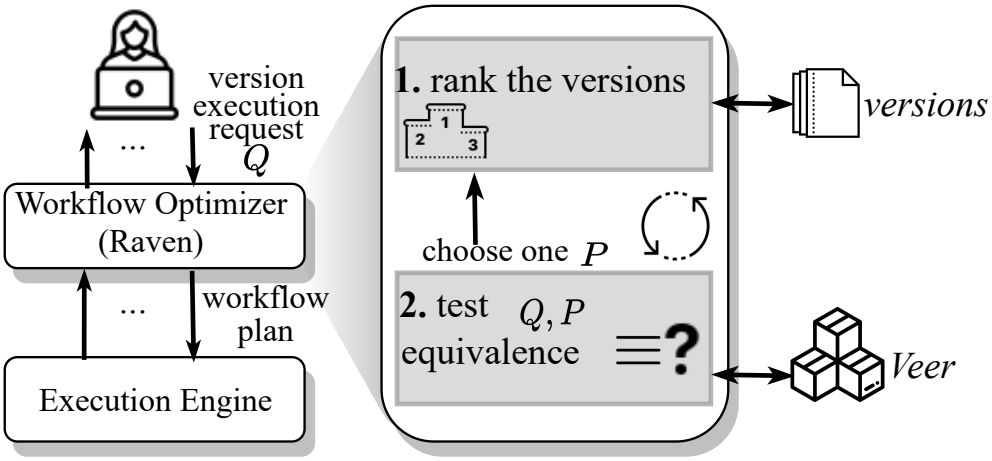


Figure 5.2: Overview of Raven's framework.

Step 1. Ranking the prior versions. Raven ranks the previous versions in the order of their likelihood of being equivalent to the current one (Line 1). To do this, we propose a two-stage approach. First, Raven organizes the sinks belonging to previously-executed versions in a hierarchy by modeling the sinks 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. The second stage is to rank the versions that have the same sink representation by using an edit mapping between the pairwise of Q and every other prior version P. Raven chooses a prior version with the highest rank to test its equivalence with the current one (Section 5.7).

Step 2: Testing the equivalence of the version pair. Raven uses an equivalence verifier to test the equivalence of the pair of versions (Line 4). The verifier returns a flag to indicate if the sinks in the two versions are equivalent. When we invoke the verifier 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. In this work, we show how Raven

Algorithm 5.1: Rewrite a workflow version to reuse previous equivalent results

Input:
A workflow version Q,
A list of previous executed versions of the workflow ${\mathcal{V}'}_w = [v_1, \ldots, v_{m-1}]$,
An equivalence verifier α;
Output: A rewritten workflow Q' to reuse equivalent results from previous versions.

1 $\mathbb{P} \leftarrow$ pruneAndRank(${\mathcal{V}'}_w$) // prune and rank previous versions
2 **while** $\mathbb{P}$ *is not empty AND there are sinks in* Q *that are unmarked* **do**
3 remove a version P from $\mathbb{P}$
4 $\mathbb{S} \leftarrow$ findEQsinks(α, P, Q) // find a set of equivalent sinks
5 mark every s_i in $\mathbb{S}$
6 **end**
7 $Q' \leftarrow$ rewrite Q
8 **return** Q'

uses Veer as the verifier and show how Raven extends Veer to return a set of equivalent sinks (Section 5.5) and avoid repeated computation by performing memoization (Section 5.6).

We repeat the above steps till all the sinks in the version Q can be answered using prior versions, or there are no more *selected* previous versions left to check (Line 2). An improvement is to re-rank the remaining workflow versions after finding the equivalent sinks in Line 5 by only considering the unmarked sinks. We leave this improvement to future work. Lastly, Raven rewrites the workflow version to reuse previous equivalent results by replacing the path of operators leading to the sinks with a Load operator to load the saved results (Line 7). The operators in the replaced path must not be ancestors of other sinks that are not equivalent to a previous version because these operators need to be executed to produce the results.

5.5 Equivalence Verification of Multiple Sink Pairs in Two Workflow Versions

We want to know which sink operators in a workflow version Q are equivalent to the corresponding sinks under a mapping in a prior version P. We can ask an equivalence verifier to verify which sinks are equivalent. However, verifiers expect a version pair with a single sink as an input and return a Boolean flag to indicate whether the two sinks are equivalent. In this section, we propose a baseline approach that divides the version pair into sub-DAGs such that each sub-DAG includes a single sink, and we call the verifier iteratively for each upstream sub-DAG of a sink.

5.5.1 Dividing the Version Pair into sub-DAGs with a Single Sink

A way to verify which sinks in a version are equivalent to the corresponding sinks in another version is to divide each version's DAG into sub-DAGs, where each sub-DAG includes a sink and its ancestor operators. We use the running example to explain this approach in Algorithm 5.2. Line 4 shows that for every sink $s \in \mathbb{S}_Q$ of version Q, we do the following. First, we construct the sub-DAG of Q consisting of the ancestor operators of s and their edges, with s as the only sink, denoted as $\pi(Q, s)$ (Line 5). Similarly, we construct the sub-DAG of P consisting of the ancestor operators of $\mathcal{M}(s)$ and their edges, with $\mathcal{M}(s)$ as the only sink, denoted as $\pi(P, \mathcal{M}(s))$ (Line 6). In the running example, there are three sinks. In particular, "Wordcloud," "Scatterplot," and "Result" sink operators. We first construct a pair of sub-DAGs, where each sub-DAG includes the upstream operators that reach the Scatterplot sink, as shown in Figure 5.3. Likewise, we construct a pair of sub-DAGs, where each sub-DAG includes upstream operators that can reach the Wordcloud operator, as depicted in Figure 5.4. We perform the same for the "Result" sink. Once we

construct the sub-DAGs for every sink in every version, we call the verifier to verify the equivalence of a pair of sub-DAGs corresponding to a sink s. If the verifier decides that the version pair is equivalent (Line 7), we add s to a set of equivalent sinks $\mathbb{S}$ (Line 8). We repeatedly call the verifier for every pair of sub-DAGs that correspond to the ancestors of a sink. Finally, we return the set of verified equivalent sinks (Line 10).

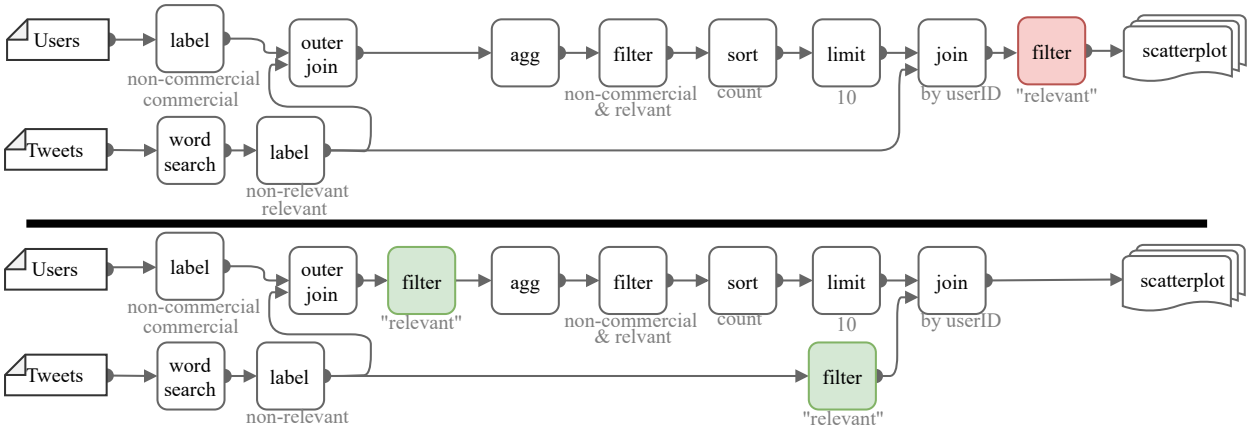


Figure 5.3: A pair of sub-DAGs on the first two versions that include the upstream operators of the Scatterplot sink from the running example.

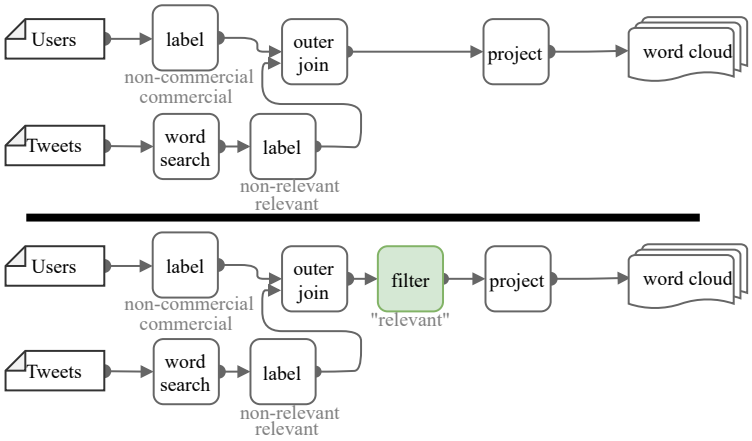


Figure 5.4: A pair of sub-DAGs that include the upstream operators of the Wordcloud sink.

Overlapping computation. A main limitation of the baseline approach is the high overlap and repeated computation that the verifier performs every time it verifies a pair of sub-DAGs. For example, suppose the verifier is Veer. When Veer tests the equivalence of the sub-DAGs that correspond to the Scatterplot sink, as shown in Figure 5.3, it expands window ω_1 and

Algorithm 5.2: Verifying the equivalence of a workflow version pair with multiple sinks (a baseline approach)

Input:
A version pair (P, Q);
An equivalence verifier α;
A set of edit operations δ and a mapping $\mathcal{M}$ from P to Q;
Output: A set of equivalent sinks

```
1  if δ is empty then
2  |  return sinks of 𝕊_P ∩ 𝕊_Q // mapped sinks
3  𝕊 = {} // an initial set of equivalent sinks
4  for s ∈ 𝕊_Q do
5  |  q_s ← π(Q, s) // construct the sub-DAG of Q consisting of ancestor operators of sink s
6  |  p_s ← π(P, M(s)) // construct the sub-DAG of P consisting of ancestor operators of M(s)
7  |  if α(p_s, q_s, δ_s, M) returns True then
8  |  |  add s to 𝕊
9  end
10 return 𝕊
```

tests if the window 'isValid' (Line 9 in Algorithm 4.2). If the window is valid, Veer expands it until reaching maximality by testing if the window satisfies "isMaximal" (Line 16). Lastly, if the window is maximal, the equivalence of its sub-DAGs is tested by passing the pair in the window to the EV (Line 17). Notice that Veer repeats the above steps when verifying the equivalence of the sub-DAGs that correspond to the Wordcloud sink, as shown in Figure 5.4. Moreover, Section 4.10 reported that the time taken to verify a window by the EV takes, on average, 87% of the total time. Since some of these windows may have been checked in previous iterations when testing other pairs, memoizing the results of previous windows' equivalence checks can help improve the performance.

Next we propose a way to avoid the aforementioned repeated computation by extending Veer to take in a pair of workflow versions with multiple sinks and returning a set of equivalent sinks (Section 5.6.1). We also extend Veer to reuse equivalence tests (Section 5.6.2).

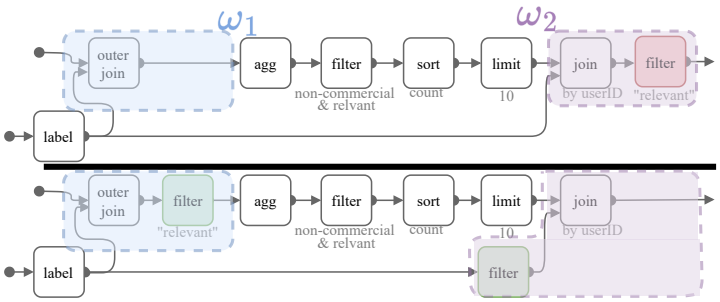


Figure 5.5: A maximal decomposition of the pair of sub-DAGs that include the upstream operators of the Scatterplot sink from the first two versions in the running example. For simplicity, we only show the covering windows throughout the chapter.

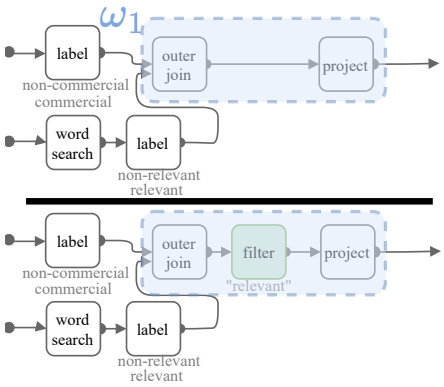


Figure 5.6: A maximal decomposition of the pair of sub-DAGs that include the upstream operators of the Wordcloud sink.

5.6 Avoiding Repeated Computation in Veer

In this section, we extend Veer to return a set of equivalent sinks by modifying the definition of decompositions and equivalence to be w.r.t. a sink (Section 5.6.1). We also propose a way to group sub-DAGs of windows in equivalence classes to reuse information about the equivalence of previously tested windows (Section 5.6.2).

5.6.1 Using a Decomposition Verification for Multiple Sinks

To extend Veer to return a set of equivalent sinks when examining the equivalence of each covering window in a decomposition, we need to know which sink of each covering window affects the result of the final sink of the version DAG. We achieve that by annotating and encoding each operator's reachability to every sink.

Definition 5.1 (Reachability of a window to a sink)**.** *We say a window ω of two versions P and Q is reachable to a sink s if any of the sub-DAGs $\omega(P)$ or $\omega(Q)$ includes an operator that is an ancestor to the sink s.*

Figure 5.7 shows a window ω and its reachability to both the Scatterplot and the Wordcloud sinks in the running example. Now we extend Definition 4.5 of the equivalence of sub-DAGs in a window to be with respect to a sink.

Definition 5.2 (Equivalence of the two sub-DAGs w.r.t. a sink)**.** *For a version pair P and Q, we say the two sub-DAGs $\omega(P)$ and $\omega(Q)$ of a window ω are* equivalent *with respect to a sink s, denoted as "$\omega(P) \equiv_s \omega(Q)$," if every sink (an operator without its downstream operator in the window) o_i in the window reachable to s is equivalent, as a stand-alone query, to its mapped sink $\mathcal{M}(\omega(o_i))$.*

For example, the two sub-DAGs in Figure 5.7 are equivalent w.r.t. the "Scatterplot" sink in the running example because the two "Filter" operators, which are annotated with a red circle in the window and can reach the scatterplot, are equivalent. The two sub-DAGs are not equivalent w.r.t. the "Wordcloud" sink in the running example because the two "Project" operators, which are annotated with a blue in the window and can reach the Wordcloud, are not equivalent.

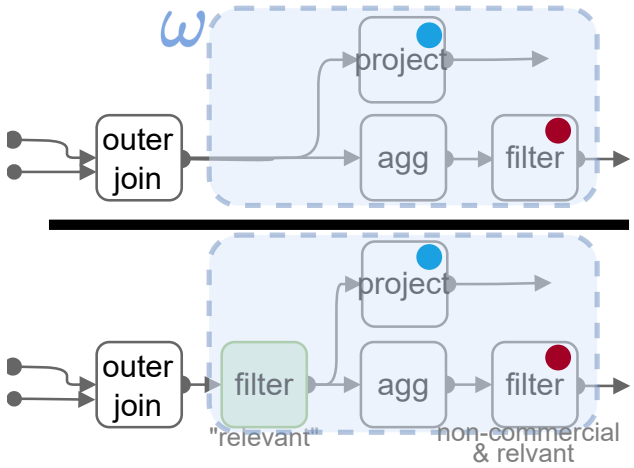


Figure 5.7: A window including operators (shown as a Blue circle) that can reach the Wordcloud sink, and operators (shown as a Red circle) that can reach the Scatterplot sink. Those operators that are not annotated can reach both sinks.

Lemma 5.1. *For a version Q with a set of sinks $\mathbb{S}_Q = \{s_1 \dots, s_n\}$, a version P, and edit operations $\delta = \{c_1 \dots c_o\}$ based on an edit mapping $\mathcal{M}$ to transform P to Q, if there is a*

decomposition θ such that every covering window in θ that reaches a sink $s \in \mathbb{S}_Q$ is equivalent w.r.t. sink s, then sink s is equivalent to its mapped sink $\mathcal{M}(s)$.

Proof. Assume a decomposition θ and every covering window $\omega_i \in \theta$ reachable to a sink s, is equivalent w.r.t. s. Every other window in θ is either a covering window and not reachable to s, i.e., does not include an operator that is an upstream of s, or a not covering window, i.e., its sub-DAGs are structurally identical, according to Definition 4.4.2. Given an instance of input sources $\mathbb{D}$, we can have the following three cases. (`Case1:`) the input sources are processed by a pair of sub-DAGs in a covering window reachable to s. In this case, the sink operators in the window that act as the input to the following windows and are reachable to the sink s are equivalent based on our assumption. (`Case2:`) the input sources are processed by a pair of sub-DAGs in a covering window not reachable to s. In this case, the result of the pair of sub-DAGs does not act as the input to any other window that is reachable to s. (`Case3:`) the input sources are processed by a pair of structurally identical sub-DAGs that are in a non-covering window. In this case, the pair of sub-DAGs produce an equivalent result since every operator is deterministic according to Assumption 5.3.2. In all cases, the output acts as the input to the following portion of the sub-DAGs (either non-covering, a covering window reachable to s, or a covering window non-reachable to s). This propagation continues along the pair of DAGs until the end, thus the two sinks s produce equivalent results. □

We modify Algorithm 4.2 in Veer to return a set of equivalent sinks from a version pair as we detail in Algorithm 5.3 using an example depicted in Figure 5.8. We follow the same procedure in the original Algorithm 4.2 to initialize a decomposition and expand it to reach maximality. When a decomposition θ is marked maximal (Line 7), we do the following. For every covering window in the decomposition, we encode the window with the information about the sinks it reaches. Then we test every covering window marked reachable to a sink s if it is equivalent w.r.t. the sink (Line 9). If this is true, then we add the sink s

into a set of equivalent sinks $\mathbb{S}$ (Line 10). Figure 5.8 shows a maximal decomposition that includes two covering windows. Covering window ω_1 reaches both the Scatterplot and the Wordcloud sink operators. Covering window ω_2 includes operators that are ancestors to only the Scatterplot operators. Both ω_1 and ω_2, which are covering windows that are reachable to the Scatterplot, are verified to be equivalent w.r.t. the Scatterplot sink. Thus, we add the Scatterplot operator to the set of equivalent sinks. Only window ω_1 is reachable to the Wordcloud sink. We test its equivalence w.r.t. the Wordcloud operator, and the window is verified as not equivalent by the EV. We repeat the above steps until either all sinks in Q are verified to be equivalent to their corresponding sinks in P, or there are no more decompositions to maximize and test.

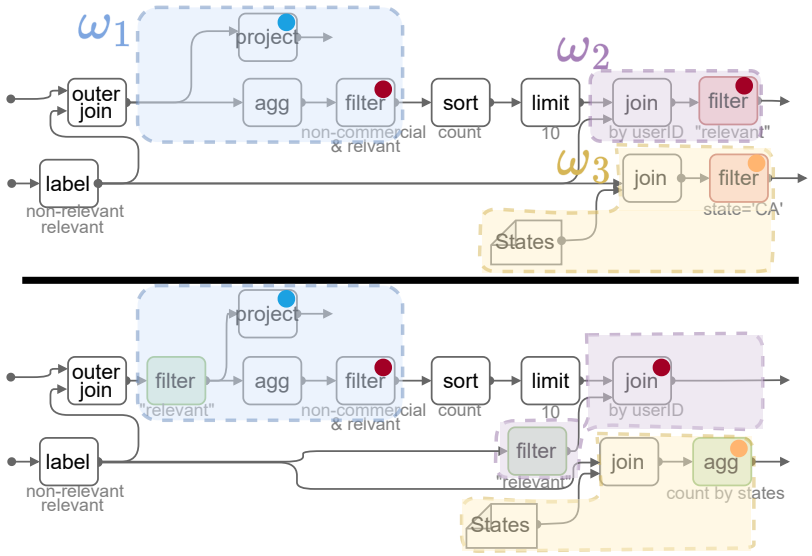


Figure 5.8: An MD with three windows, including ω_1 that can reach both Scatterplot (shown as a Red circle) and Wordcloud (shown as a Blue circle) sink operators, ω_2 that can reach the Scatterplot sink, and ω_3 that can reach the Result sink (shown as an Orange circle).

Now we show in the following Lemma 5.2 that the extended Algorithm 5.3 does not miss the chance of verifying the equivalence of a sink s that can be verified by the baseline Algorithm 5.2.

Lemma 5.2. *Consider a version Q with a set of sinks $\mathbb{S}_Q = \{s_1 \ldots, s_n\}$, a version P, an*

Algorithm 5.3: Verifying the equivalence of a workflow version pair with multiple sinks (an improved approach)

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 set of equivalent sinks

1 **if** δ *is empty* **then**
2 **return** `sinks of` $\mathbb{S}_P \cap \mathbb{S}_Q$ // mapped sinks
3 $\theta \leftarrow$ decomposition with each operator as a window
4 $\Theta = \{\theta\}$ // initial set of decompositions
5 $\mathbb{S} = \{\}$ // an initial set of equivalent sinks
6 **while** Θ *is not empty AND* $\mathbb{S} \subset \mathbb{S}_Q$ **do**
  // Lines 6 - 15 from Algorithm 4.2 to find a possible maximal decomposition θ_i
7 **if** *every covering window in* θ_i *is marked* **then**
    // i.e., θ_i is indeed a maximal decomposition
8 **if** *there exists a covering window* ω *reachable to a sink* $s \notin \mathbb{S}$ **then**
      // i.e., s is not verified equivalent yet
9 **if** γ *verifies each covering window in* θ_i *that is reachable to a sink* s *to be equivalent w.r.t.* s **then**
10 add s to $\mathbb{S}$
11 **end**
12 **return** $\mathbb{S}$

EV γ, and edit operations $\delta = \{c_1 \dots c_o\}$ based on an edit mapping $\mathcal{M}$ to transform P to Q. If Algorithm 5.2 finds a maximal decomposition θ for a sink s_i such that each covering window in θ is verified equivalent, then Algorithm 5.3 will find a maximal decomposition θ' such that $\theta \subseteq \theta'$ and each covering window (in θ') that reaches s_i is verified equivalent.

Proof. Suppose Algorithm 5.2 finds a maximal decomposition θ w.r.t. γ such that a sink s_i in workflow version Q is verified equivalent to its mapped sink $\mathcal{M}(s_i)$ in version P. Notice Algorithm 5.3 considers all maximal decompositions. We construct a maximal decomposition θ' from θ using the following steps. We construct a decomposition on the version pair by including every window in θ. Each remaining operator in the version pair not in θ forms its own window. Then we merge every pair of windows if they satisfy the restrictions of γ. We do this merging step iteratively until we cannot merge any two windows. We can infer

that $\theta \subseteq_d \theta'$, and that θ' is maximal. The operators in θ' not in θ do not reach the sink s_i; otherwise, it contradicts the assumption that θ is a maximal decomposition found by Algorithm 5.2. From Lemma 5.1, a sink is equivalent to its mapped sink if every covering window that reaches the sink is equivalent w.r.t. the sink, which is the case in θ. We can deduce that using the decomposition θ', Algorithm 5.3 can verify that the sink s_i is equivalent to its mapped sink $\mathcal{M}(s_i)$. □

Figure 5.9 shows an abstract representation of the proof.

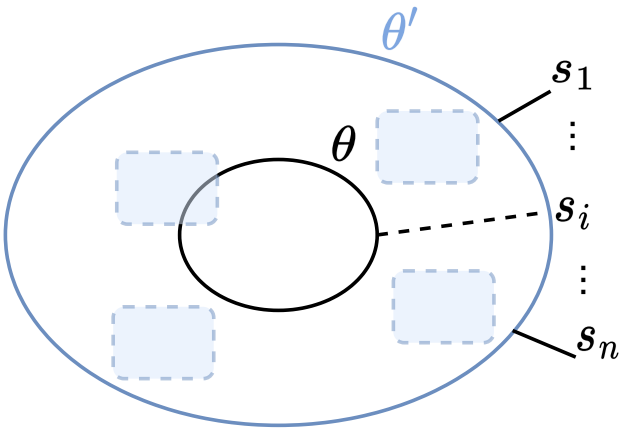


Figure 5.9: An abstract example to show the proof of Lemma 5.2.

Extending Veer$^+$ to rank decompositions with multiple sinks.

In Section 4.8 we proposed a few optimization techniques to improve the performance of finding maximal decompositions, namely the "segmentation" and "ranking" techniques. Segmentation divides the version pair DAG into independent portions called "segments" (denoted as $\mathcal{S}$), where each segment includes decompositions such that there is no valid window in a decomposition that includes operators from two different segments. Veer$^+$ proposes two ranking techniques. One is to rank which segment to evaluate first, and the other is to rank which decomposition within a segment to expand first. Next we discuss how we extend Veer$^+$'s ranking to include a metric to score segments and decompositions, including the information on the different sinks.

We consider $\mathcal{F}(S) = m_S + n_S + r_S$ as an example scoring function to rank a segment $\mathcal{S}$. The details of m_S and n_S are in Chapter 4. r_S indicates the number of sinks the segment reaches. The higher the number is, the higher the segment is ranked, as it indicates the segment can give us more answers about the equivalence of multiple sinks and the possibility of early termination. Similarly, an example ranking function for a decomposition d is $\mathcal{G}(d) = o_d - w_d + r_d$, where o_d and w_d are detailed in Chapter 4. r_d indicates the number of sinks the decomposition reaches. The higher the number is, the higher the decomposition is ranked, as it indicates the decomposition can give us more answers about the equivalence of multiple sinks and the possibility of early termination.

5.6.2 Grouping Sub-DAGs of Windows in Equivalence Classes

Consider the following example of optimizing the execution of four different versions in a workflow, as depicted in Figure 5.10. Suppose a user submits an execution request for the v_2 version of the workflow. Based on Raven's procedure, it tests the equivalence of v_2 with a prior version v_1 using Veer. After decomposing the pair and finding an MD, Veer verifies the sub-DAGs' equivalence in each covering window within the MD. In this example, a covering window $\omega_1 = \langle \omega_1(v_1), \omega_1(v_2) \rangle$ (depicted in blue in the figure) of an MD needs to be tested for equivalence. Suppose the EV can verify the equivalence of the two sub-DAGs in the window in this example.

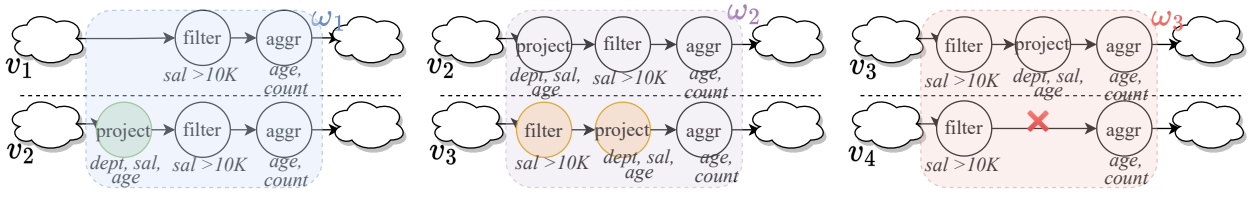


Figure 5.10: Example of testing three pairs of four versions to show three different windows and the windows' sub-DAGs belong to the same equivalence class.

Suppose in a separate iteration for optimizing the performance of version v_3, Raven tests the

equivalence of the pair (v_2, v_3) by pushing the pair to Veer. Again, Veer internally decomposes the pair into windows. One of the covering windows is $\omega_2 = \langle \omega_2(v_2), \omega_2(v_3) \rangle$ (depicted in purple in the figure). The two sub-DAGs in ω_2 are proved to be equivalent by the EV. Note that the sub-DAG $\omega_2(v_2)$ is equally or structurally the same as $\omega_1(v_2)$ as defined in Recycler [102] and Restore [49] [2]. Recall that $\omega_1(v_2)$ was tested for equivalence with the sub-DAG $\omega_1(v_1)$.Then we deduce that the sub-DAGs $\omega_1(v_1)$ and $\omega_2(v_3)$ are equivalent by transitivity [28].

Now, when Raven optimizes the execution of v_4, it pushes a pair (v_3, v_4) to Veer to test the pair's equivalence. Suppose one of the covering windows that Veer needs to test its equivalence is $\omega_3 = \langle \omega_3(v_3), \omega_3(v_4) \rangle$ (depicted in red in the figure). Notice that the sub-DAG $\omega_3(v_3)$ is equal to $\omega_2(v_3)$ and the sub-DAG $\omega_3(v_4)$ is equal to $\omega_1(v_1)$. This means that testing the equivalence of the sub-DAGs $\omega_3(v_3)$ and $\omega_3(v_4)$ is the same as testing the equivalence of the sub-DAGs $\omega_2(v_3)$ and $\omega_1(v_1)$. The latter pair was proven to be equivalent by transitivity in an earlier iteration. Had we stored the results of the previous covering window equivalence tests, Veer would have avoided testing the equivalence of pairs that were tested before.

To store the result of equivalence tests, we group sub-DAGs of tested windows in *equivalence classes*. An equivalence class is a set of equivalent elements. Each element in a class is a sub-DAG from a window. The idea is to maintain a mapping of each tested sub-DAG to its equivalence class number. For simplicity of the discussion, we explain the solution where a window has a single sink, and we generalize to the case where a window has multiple sinks in Section 5.6.2.1. We revisit the example in Figure 5.10 to explain how we modify Algorithm 5.3 to memoize and use previous tests on the equivalence of windows. We extend Veer, as shown in Algorithm 5.4, to let it check if the sub-DAGs in a covering window were verified before by performing a lookup of a sub-DAG. Each stored sub-DAG includes a pointer to the equivalence class the sub-DAG belongs to.

[2]For simplicity of the discussion, we say two sub-DAGs are *equal* to indicate their exact structure match.

Algorithm 5.4: Verifying the equivalence of a workflow version pair with multiple sinks (Reusing equivalence result of previously-tested sub-DAGs).

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 set of equivalent sinks

// Lines 1-16 in Algorithm 4.2

```
1  switch lookup each sub-DAG in ω_j do
2      case both sub-DAGs were not seen before do
3          if γ proves the window is equivalent then
4              assign both sub-DAGs the same class
5          else
6              assign each sub-DAG a new class
7          end
8      case one sub-DAG only was seen before do
9          if γ proves the window is equivalent then
10             assign unseen sub-DAG the same class as the seen sub-DAG
11         else
12             assign the unseen sub-DAG a new class
13         end
14     case both sub-DAG were seen before do
15         if both sub-DAGs are not in the same class and they were never tested
            together before then
16             test their equivalence using γ and assign the appropriate class
17 end
```

// Line 20 from Algorithm 4.2

The lookup 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 a new identifying label (Line 4). 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 (Line 6). Then Veer store the sub-DAG and assigns a pointer to the sub-DAG's equivalence class. Following the example in Figure 5.10, testing the sub-DAGs in ω_1 falls under this case, as both sub-DAGs were not tested before. After the EV proves that the two sub-DAGs are equivalent, Veer stores the new sub-DAGs and assigns

their equivalence pointers the same equivalence class, say in this example, a value of 1.

2. One sub-DAG only was tested before: Veer pushes the pair to the EV, and if the EV proves the pair is equivalent, then we add the unseen sub-DAG to the same equivalence class as the other one (Line 10). Otherwise, we create a new equivalence class for the unseen sub-DAG (Line 12). In the above example, testing the equivalence of the sub-DAGs in the window ω_2 falls under this case. The reason is because the sub-DAG $\omega_2(v_1)$ was previously tested but $\omega_2(v_2)$ was not. After the EV proves the equivalence of the pair in the window, Veer stores the new sub-DAG $\omega_2(v_2)$ and assigns it with the same equivalence class as $\omega_2(v_1)$, i.e., 1. A subtle case is when the EV is incomplete and returns "Unknown" as an answer for the pair's equivalence test. One way to solve this is to replace the previously-tested sub-DAG with another sub-DAG from the same class, and repeat this replacement until we get an answer from the EV other than "Unknown" or there are no other sub-DAGs in the 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. If the two equivalence classes are the same, Veer marks the pair's equivalence, and there is no need to push testing the pair's equivalence to the EV. Otherwise, every sub-DAG is in a different equivalence class. Thus, we ask the EV to verify if the two sub-DAGs are equivalent in order to merge the two equivalence classes of the sub-DAGs into one. If the EV determines that the two sub-DAGs are equivalent, we merge the two classes and update the sub-DAGs' pointers to point to the newly merged class. In the above example, the two sub-DAGs in window ω_3 were both seen before, and their equivalence class is 1. Following this procedure, we can see the benefit of not pushing the last pair of the covering window ω_3 to the EV, as we know their equivalence from previous tests.

Memoizing the check of two different equivalence classes: In order to prevent redundant tests of two equivalence classes to determine whether they need to be merged or not, we

employ a memoization technique. We maintain a 2-D matrix of flags associated with the equivalence classes, as depicted in Figure 5.11. This matrix allows us to track whether a pair of equivalence classes have already been tested. We update the flag in the matrix each time we verify the equivalence of two distinct equivalence classes.

EC	1	2	3	⋯
1	O	X	X	
2	X	O	X	
3	X	X	O	...

Figure 5.11: A sample 2-D matrix for storing the equivalence tests between a pair of equivalence classes. A cell initially is "X" and is changed to "O" when the two classes are tested.

5.6.2.1 Looking up a Sub-DAG by Performing a Structure Match

A natural question is how to lookup whether a sub-DAG has been tested before. Veer maintains a "recycler DAG" [102], which is a structural representation of the previously-tested sub-DAGs. Each node in the recycler DAG represents an operator of a sub-DAG. The recycler DAG combines and groups the sub-DAGs from previous tests. When we want to check if a sub-DAG has been tested before, we traverse the sub-DAG simultaneously with the recycler DAG, matching each node from the sub-DAG with the corresponding node in the recycler DAG. If a node matches, we continue the traversal. If a node is not matched, we update the recycler DAG to insert the new node. Two nodes v and u exactly match if: (i) v and u represent the same operator type; (ii) they have the same properties; (iii) their upstream operators also match [102]. Lastly, in the recycler DAG, nodes without outgoing edges, i.e., leaves, include pointers indicating their respective equivalence classes. These pointers signify the equivalence class of the sub-DAG, which includes the upstream nodes that lead to each particular leaf. Figure 5.12 shows an example of a recycler DAG for storing

the sub-DAGs in the three windows of the example in Figure 5.10.

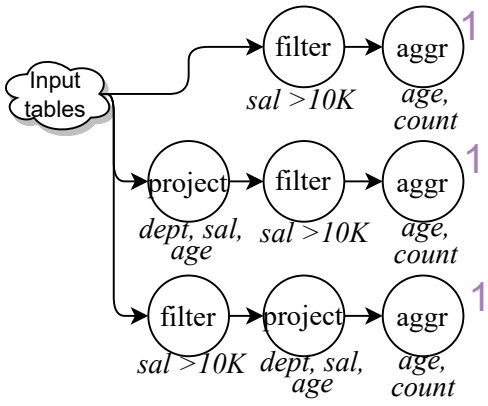


Figure 5.12: Example Recycler DAG to group the sub-DAGs of three windows and their equivalence class.

5.7 Ranking Versions for Equivalence Check

When given a new workflow version execution request Q, we want to rank the prior versions before verifying the equivalence of version Q and each prior version $P \in \mathbb{P}$. In this section, we discuss how Raven follows a two-stage approach to ranking the prior executed versions. In the first stage, Raven *prunes* the inequivalent versions based on a partial representation of the semantics of the sinks in a version following a novel structure to organize the versions (Section 5.7.1). In the second stage, Raven ranks the chosen prior versions from the first stage based on their edit distance compared to version Q (Section 5.7.2).

5.7.1 Ranking Versions by their Semantic Results of the Sinks

To efficiently identify reusable artifacts across different versions, we need a lightweight fingerprint representation that models the semantics of the sinks' results. In this section, we use "views", "artifacts", and "sinks" interchangeably. 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 on, 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 [37, 89] represented by T in this work.

Representation construction. To construct the view representation, we follow the same techniques in existing literature [37, 164] 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 5.3 (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 5.13 shows a sample V2-structure to organize the sinks in the running example. Each node has a physical pointer to its saved result of the sink.

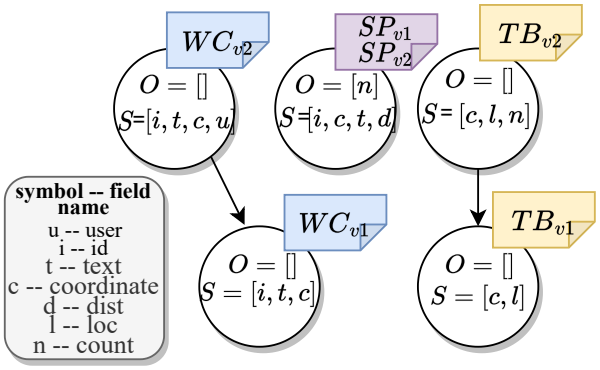


Figure 5.13: A sample V2-structure to organize the saved results of sinks from the first two versions in the running example.

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. We first construct the view representation of the word cloud sink, $\vec{S} = [i, c]$ 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 2.12b 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 the given verifier 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 one pair to the verifier, instead of iterating over every past version. Finally, because a node can have multiple parent nodes, we memoize if the node was visited for the particular current version to avoid exploring the same node multiple times and performing the test.

We repeat this process for each sink of the version Q. Once we identify a node for each sink, we may end up with a list of versions. Raven starts the second stage to rank those versions. We rank a version P_a higher than another version P_b if the version P_a is more similar to the version Q compared to the similarity of the version pair (P_b, Q). As an example, we use the edit distance as one factor as we explain next.

5.7.2 Ranking Versions Based on Edit Mapping

We rank a version with less edit to Q higher than another. 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 [23]. 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 find an equivalent version,

Example 5.1. *Consider the following three versions:*

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

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

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

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 [23]. We exploit the analysts' interactions recorded when refining the workflow versions iteratively in the form of deltas to maintain an incremental sequence of the deltas for a lighter-weight mapping.

5.8 Experiments

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

5.8.1 Experimental Setup

Real workload. We created similar workflows, which are presented as $W1 \ldots W4$ in Table 5.1, from the collected real workflows from one deployment of Texera [138] as summarized in Appendix A. We show a sample of some workflows used in the experiments in Appendix B. We used IMDB [70] ($\approx 3GB$) and Twitter [142] ($\approx 0.5GB$) datasets. All versions included UDF operators. The average time it took to execute a version without reuse is 1.9 minutes.

Table 5.1: Workloads used in the experiments.

Workflow#	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 [102] baseline, which compares a workflow query DAG with previously executed workflow DAGs by examining their structures for equality. We implemented a basic Raven, denoted as Raven_b, which is a basic approach that iterates over past versions without ranking them, divides the version pair into sub-DAGs of a single sink then calls Veer, and uses Veer without enabling reusing previous tests on windows. We implemented Raven_a, which is advanced Raven and included ranking past versions and extending Veer to return a set of equivalent sinks and reuse previous equivalence tests. We implemented the Veer 4 verifier and used Equitas [164] as its EV. We implemented the baseline and Raven using Java8 in Texera [93]. 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.

5.8.2 Identifying Reuse

We evaluated Raven's effectiveness in identifying semantic equivalence of workflows with UDFs compared to the baseline. Table 5.2 shows the results of the experiment. Recycler successfully identified 25% of the equivalent cases, while both Raven_b and Raven_a successfully identified 60% of the equivalent cases. Recycler failed to rewrite any of the workflow versions to reuse the identified equivalent results. On the other hand, Raven_b and Raven_a were able to rewrite the workflows to reuse the results for 40% of the equivalent sinks. 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.

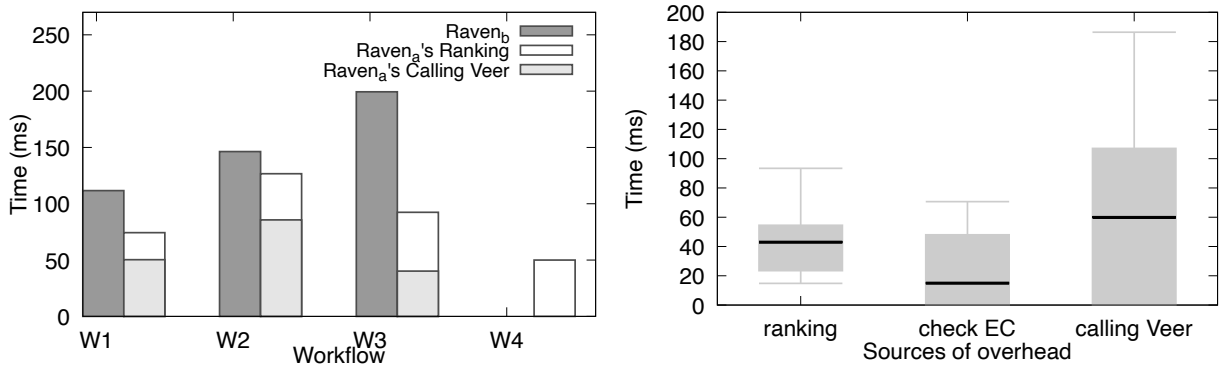
Table 5.2: Comparison evaluation of Raven_b and Raven_a against Recycler.

Approach	% of identified equivalent sinks	% of used equivalent sinks
Recycler	25.0	0.0
Raven_b	60.0	40.0
Raven_a	60.0	40.0

5.8.3 Overhead Analysis

We recorded the overhead of each approach and the time each approach spent to identify reuse. Figure 5.14a 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 5.14a. Raven_b and Raven_a had more overhead than Recycler because they needed to invoke Veer multiple times. The overhead of Raven_a (up to 126.6 ms on $W2$) is less than Raven_b (up to 199.4 ms on $W3$) because it used the equivalence class concept and the ranking approach to optimize and reduce the time spent on Veer.

Breakdown of Raven$_a$'s overhead. Figure 5.14b shows the time Raven$_a$ spent on "ranking", "checking the equivalence class of windows", and "calling Veer" for each version of the workflow. Each boxplot represents the following: 1^{st} percentile for the bottom line, 25^{th} for the beginning of the box and 75^{th} percentile for the end of the box, median is represented by a line crossing the body of the box, and the 99^{th} percentile for the top line. In general the performance of the "ranking" ranged between 14 and 97 milliseconds with an average of 42 milliseconds. The time it took to find equivalence classes of sub-DAGs in the window (check EC) ranged from a few milliseconds to 72 milliseconds with an average of 24.9 milliseconds. There was a high variation of the time taken to "call Veer" (a few milliseconds to 190 ms) caused by the following reasons: either because the two versions did not have any changes (due to reverting the changes to a previous version), thus Veer's call terminated early; or because Veer identified equivalence reuse using the equivalence classes and it internally did not call the EV.



(a) Time taken to identify reuse. **(b) Breakdown of Raven$_a$'s overhead.**

Figure 5.14: Overhead analysis of the solutions.

5.8.4 Execution Speedup

Figure 5.15 shows the speedup of the approaches. Because Recycler failed to rewrite any of the workflow versions to reuse the identified equivalent results, it had a speedup of 1. On the other hand, Raven$_b$ and Raven$_a$ were able to rewrite the workflows to reuse the results for 40%

of the equivalent sinks, yielding speedups of up to 322 using Raven_b and 747 using Raven_a for $W3$. Overall, Raven_a outperformed Raven_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.

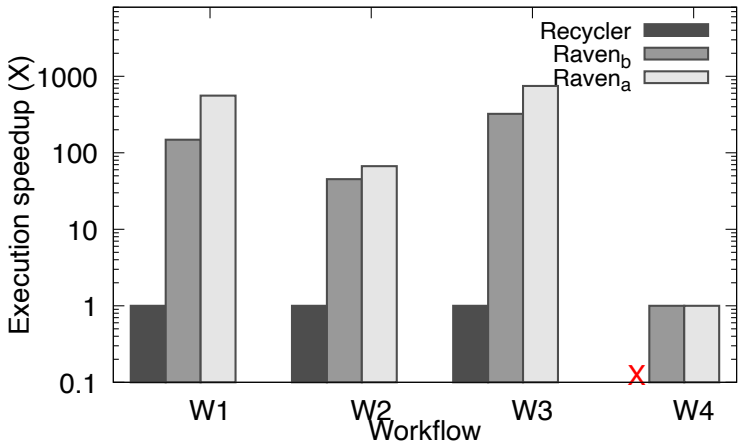


Figure 5.15: Effectiveness of Raven on execution speedup. An "X" indicates the workflow was not supported by the solution.

5.8.5 Effect of the Number of Sinks

We evaluated the performance of verifying a pair with multiple sinks using the baseline approach discussed in Algorithm 5.2 against extending Veer to handle multiple sinks, as in Algorithm 5.3. We can see that as we increase the number of sinks, the time it takes to verify a pair also increases up to 243 ms for the baseline as shown in Figure 5.16, while the time taken to do the verification by Veer remains the same around 70 ms.

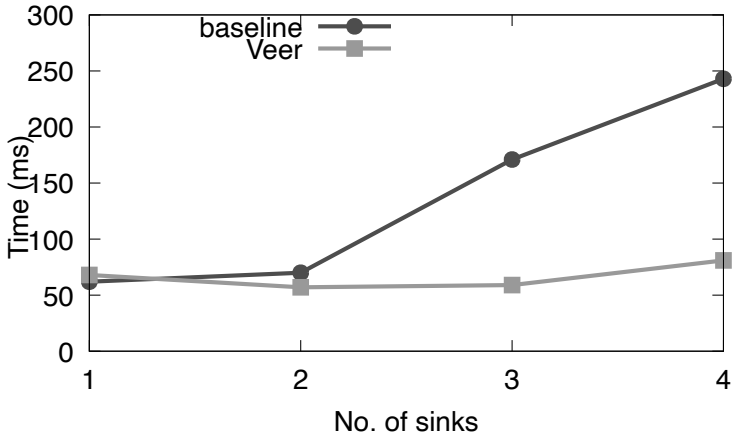


Figure 5.16: Effect of the number of sinks on the performance of verifying a version pair.

5.9 Conclusion

In this chapter, 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 using a verifier. We discussed how Raven extends Veer to handle verifying a pair with multiple sinks, and to reuse previous computations. We also showed how Raven follows a two-stage approach to ranking the versions. We evaluated the effectiveness of the solution on real workflows from Texera, observing actual speedup benefits of up to several orders of magnitude in some cases.

Chapter 6

Conclusions and Future Work

Overall, we studied the challenges and opportunities of iterative data analytics using complex data-processing workflows. This dissertation contributes to the advancement of GUI-based data-processing systems by integrating visualization, version control, and equivalence verification for optimizing the execution of data-processing tasks. By bridging these areas, analysts are empowered with enhanced tools and techniques for effective data analytics, leading to improved productivity and reproducibility. In this chapter, we present the conclusions of the four works presented in this dissertation and motivate the future work.

6.1 Conclusions

In Chapter 2, we addressed the problem of visualizing large spatial networks in a progressive setting and introduced GSViz, a general-purpose middleware solution. GSViz effectively resolves this problem along with its associated challenges, particularly visual clutter. We proposed three techniques to mitigate visual clutter, including an edge-aware vertex clustering algorithm. Additionally, we presented a tree-like data structure that enhances the efficiency

of searching for compatible edges, enabling incremental and efficient bundling. Furthermore, we integrated these two techniques, addressing the challenges that arise from their combined use, such as the frequent updating of visual results. We also discussed how our solution supports various interaction techniques like zooming and panning. We also discussed how we leverage the hierarchical structure, in which supports zooming, to further reduce visual clutter.

In Chapter 3, we presented Drove, a holistic end-to-end approach to provide an infrastructure to allow users to orchestrate, refine, and execute workflows while ensuring the reproducibility of their experiments. We addressed the challenge of enabling effective tracking in workflow-based systems and presented our innovative solution, which leverages the concept of maintaining deltas and tracking the edit operations performed. This lightweight approach provides users with the necessary tools to track and analyze changes in their workflows, enabling a more streamlined and transparent workflow management process.

In Chapter 4, we studied the problem of verifying the equivalence of two workflow versions. We presented a solution called "Veer," which leverages minor edits between two workflow versions. We analyzed the restrictions of existing EVs and proposed a concept called a "window" to leverage the existing solutions for verifying equivalence. We proposed a verification algorithm using "windows" to verify the equivalence of a version pair with a single edit. We discussed the challenges of testing the equivalence of a version pair with multiple edits and proposed a baseline algorithm. We proposed optimization techniques to speed up the baseline's performance. We discussed the correctness and completeness of the equivalence verification algorithm. We conducted a thorough experimental study and showed the high efficiency and effectiveness of the solution.

In Chapter 5, 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 discussed how Raven uses Veer in its modules to test the pair's equivalence.

We presented a ranking based on the semantics of the saved results corresponding to the sinks in the version. We also discussed how Raven extends Veer to handle multiple sinks. Moreover, Raven extends Veer to avoid pushing newly constructed windows into an EV if the window was tested before in previous iterations. We group the windows' sub-DAGs into equivalence classes when storing them. We saw speedup of benefits of up to several orders of magnitude in some cases.

6.2 Future Work

GSViz stores the clustering hierarchy and the PEB-tree in memory and follows a heuristic approach to reducing the tree size by merging nodes when a tree size meets a certain threshold. We plan to devise a cost-based technique to reduce the tree size to satisfy a given memory budget. Moreover, GSViz currently follows a heuristic and greedy approach to clustering the vertices and bundling the edges. Although the algorithm is bounded by the range radius ρ for clustering and the compatibility score in edge bundling, the quality and accuracy are not compromised. However, we are interested in proposing an objective function to trade off the visualization accuracy and performance efficiency in the future.

Drove assumes that the workflow execution is completed when retrieving and displaying its details. To handle ongoing executions, we need to be able to serialize the states of the execution. We plan to tackle this problem in future work. Moreover, we plan to capture more environmental information, such as details of other jobs or processes running, as these may affect the performance or result of the experiment. We also plan to allow the user to compare a pair of executions and their corresponding results in a fine-grained fashion, e.g., by highlighting the different tuples. One approach to achieve highlighting the difference between two results is to compare every pair-wise tuple from the two results, which can be expensive, but this is sufficient when the user views a small number of tuples at a time on

the screen.

Veer tests the equivalence or inequivalence of two workflow versions. We want to extend the solution to detect the containment of the two versions. The current window-based solution works for identifying equivalence but not containment. The reason is from the two outputs of a non-covering window that act as the input to a covering window (whether they are equivalent or one contains the other), we cannot infer how the data is transformed in each covering window. One way to track that is to encode the order and structure of the windows and the data flow. Another future improvement is that the solution in Veer assumes the workflow version is a DAG, i.e., there is no loop ore recursion, but we want to propose ways to handle versions that contain loop blocks and recursion. We also want to propose a more powerful EV that handles reasoning the semantics of operators beyond relational ones, such as UDFs. The objective is to incorporate this powerful EV into the Veer search framework. One way to do so is to propose an API that the UDF developer needs to adhere to in order to be able to capture the semantics of the UDF operator. One example API is $\langle T, S, C, O \rangle$ [9]. 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 result is ordered in.

Raven assumes the operators' logic is deterministic, and the data sources are static and not changing. We want to extend the solution to detect and handle non-determinism. There can be many ways to detect non-deterministic workflow versions as follows: a) the user specifies that the workflow includes sources of non-determinism, b) the developer of the operator indicates the logic of the developed operator includes non-determinism, e.g., in PostgreSQL the CREATE FUNCTION expects a user to specify that a UDF is VOLATILE [114], c) Raven applies logic such as to look for certain keywords, e.g., "random," to detect non-determinism. To handle non-deterministic versions, we can apply some of the techniques proposed to support

reproducibility [55]. These include for example, using pseudo-random number generators that take a seed and generate seemingly random numbers in a deterministic fashion [55]. Moreover, we want the Raven optimizer to follow a best-effort optimization given a time budget. Once there is a time budget, we need to maximize the chance that Raven chooses a set of the most promising workflows (those that are equivalent). To this end, we plan to study the semantic similarity between the workflows to return a set of the most semantically similar workflow versions to a given version.

Bibliography

[1] J. Abello, F. van Ham, and N. Krishnan. Ask-graphview: A large scale graph visualization system. *IEEE Trans. Vis. Comput. Graph.*, 12(5):669–676, 2006.

[2] S. Abiteboul and O. M. Duschka. Complexity of answering queries using materialized views. In A. O. Mendelzon and J. Paredaens, editors, *Proceedings of the Seventeenth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems, June 1-3, 1998, Seattle, Washington, USA*, pages 254–263. ACM Press, 1998.

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

[4] F. N. Afrati, C. Li, and P. Mitra. On containment of conjunctive queries with arithmetic comparisons. In *EDBT*, pages 459–476, 2004.

[5] P. K. Agarwal, K. Fox, K. Munagala, A. Nath, J. Pan, and E. Taylor. Subtrajectory clustering: Models and algorithms. In J. V. den Bussche and M. Arenas, editors, *Proceedings of the 37th ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems, Houston, TX, USA, June 10-15, 2018*, pages 75–87. ACM, 2018.

[6] Y. Ahmad, O. Kennedy, C. Koch, and M. Nikolic. Dbtoaster: Higher-order delta processing for dynamic, frequently fresh views. *Proc. VLDB Endow.*, 5(10):968–979, 2012.

[7] R. Alhajj and J. G. Rokne, editors. *Encyclopedia of Social Network Analysis and Mining, 2nd Edition*. Springer, 2018.

[8] S. M. Ali, N. Gupta, G. K. Nayak, and R. K. Lenka. Big data visualization: Tools and challenges. In *2016 2nd International Conference on Contemporary Computing and Informatics (IC3I)*, pages 656–660, 2016.

[9] S. Alsudais. Drove: Tracking execution results of workflows on large data. In Z. Bao and T. K. Sellis, editors, *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*, volume 3186 of *CEUR Workshop Proceedings*. CEUR-WS.org, 2022.

[10] S. Alsudais, Q. Bai, S. Zhao, and C. Li. Gsviz: progressive visualization of geospatial influences in social networks. In *SIGSPATIAL*, 2022.

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

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

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

[14] Apache Spark http://spark.apache.org.

[15] D. Auber. Tulip - A huge graph visualization framework. In M. Jünger and P. Mutzel, editors, *Graph Drawing Software*, pages 105–126. Springer, 2004.

[16] Q. Bai, S. Alsudais, C. Li, and S. Zhao. Maliva: Using machine learning to rewrite visualization queries under time constraints. In J. Stoyanovich, J. Teubner, N. Mamoulis, E. Pitoura, and J. Mühlig, editors, *Proceedings 26th International Conference on Extending Database Technology, EDBT 2023, Ioannina, Greece, March 28-31, 2023*, pages 157–170. OpenProceedings.org, 2023.

[17] J. Bao, Y. Zheng, D. Wilkie, and M. F. Mokbel. Recommendations in location-based social networks: a survey. *GeoInformatica*, 19(3):525–565, 2015.

[18] M. Bastian, S. Heymann, and M. Jacomy. Gephi: An open source software for exploring and manipulating networks. In E. Adar, M. Hurst, T. Finin, N. S. Glance, N. Nicolov, and B. L. Tseng, editors, *Proceedings of the Third International Conference on Weblogs and Social Media, ICWSM 2009, San Jose, California, USA, May 17-20, 2009*. The AAAI Press, 2009.

[19] L. Battle, R. Chang, and M. Stonebraker. Dynamic prefetching of data tiles for interactive visualization. In F. Özcan, G. Koutrika, and S. Madden, editors, *Proceedings of the 2016 International Conference on Management of Data, SIGMOD Conference 2016, San Francisco, CA, USA, June 26 - July 01, 2016*, pages 1363–1375. ACM, 2016.

[20] T. Beleche, J. Ruhter, A. Kolbe, J. Marus, L. Bush, and B. Sommers. Covid-19 vaccine hesitancy: Demographic factors, geographic patterns, and changes over time. *Published online*, 27, 2021.

[21] N. Bikakis, J. Liagouris, M. Krommyda, G. Papastefanatos, and T. K. Sellis. graphvizdb: A scalable platform for interactive large graph visualization. In *32nd IEEE International Conference on Data Engineering, ICDE 2016, Helsinki, Finland, May 16-20, 2016*, pages 1342–1345. IEEE Computer Society, 2016.

[22] V. D. Blondel, J.-L. Guillaume, R. Lambiotte, and E. Lefebvre. Fast unfolding of communities in large networks. *Journal of statistical mechanics: theory and experiment*, 2008(10):P10008, 2008.

[23] D. B. Blumenthal, N. Boria, J. Gamper, S. Bougleux, and L. Brun. Comparing heuristics for graph edit distance computation. *VLDB J.*, 29(1):419–458, 2020.

[24] 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.*, 20(4):25:1–25:36, 2019.

[25] I. Boyandin, E. Bertini, and D. Lalanne. Using flow maps to explore migrations over time. In *Geospatial Visual Analytics Workshop in conjunction with The 13th AGILE International Conference on Geographic Information Science*, volume 2, 2010.

[26] F. Brodkorb, A. Kuijper, G. L. Andrienko, N. V. Andrienko, and T. von Landesberger. Overview with details for exploring geo-located graphs on maps. *Inf. Vis.*, 15(3):214–237, 2016.

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

[28] A. K. Chandra and P. M. Merlin. Optimal implementation of conjunctive queries in relational data bases. In J. E. Hopcroft, E. P. Friedman, and M. A. Harrison, editors, *Proceedings of the 9th Annual ACM Symposium on Theory of Computing, May 4-6, 1977, Boulder, Colorado, USA*, pages 77–90. ACM, 1977.

[29] B. Chandra and S. Sudarshan. Automated grading of SQL queries. *IEEE Data Eng. Bull.*, 45(3):17–28, 2022.

[30] L. Chang, W. Li, L. Qin, W. Zhang, and S. Yang. pscan: Fast and exact structural graph clustering. *IEEE Trans. Knowl. Data Eng.*, 29(2):387–401, 2017.

[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] C. Chen, S. Hwang, and Y. Oyang. An incremental hierarchical data clustering algorithm based on gravity theory. In *Advances in Knowledge Discovery and Data Mining, 6th Pacific-Asia Conference, PAKDD 2002, Taipei, Taiwan, May 6-8, 2002, Proceedings*, volume 2336 of *Lecture Notes in Computer Science*, pages 237–250. Springer, 2002.

[33] L. Chen, Y. Gao, Y. Zhang, C. S. Jensen, and B. Zheng. Efficient and incremental clustering algorithms on star-schema heterogeneous graphs. In *35th IEEE International Conference on Data Engineering, ICDE 2019, Macao, China, April 8-11, 2019*, pages 256–267. IEEE, 2019.

[34] F. Chirigati, R. Rampin, D. E. Shasha, and J. Freire. Reprozip: Computational reproducibility with ease. In *SIGMOD*, 2016.

[35] R. Chirkova, C. Li, and J. Li. Answering queries using materialized views with minimum size. *VLDB J.*, 15(3):191–210, 2006.

[36] E. Cho, S. A. Myers, and J. Leskovec. Friendship and mobility: user movement in location-based social networks. In C. Apté, J. Ghosh, and P. Smyth, editors, *Proceedings of the 17th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, San Diego, CA, USA, August 21-24, 2011*, pages 1082–1090. ACM, 2011.

[37] 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.

[38] 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.

[39] Clustering millions of points on a map with supercluster. `https://blog.mapbox.com/clustering-millions-of-points-on-a-map-with-supercluster-272046ec5c97`. Accessed: 2019-10-23.

[40] A. Crotty, A. Galakatos, E. Zgraggen, C. Binnig, and T. Kraska. The case for interactive data exploration accelerators (ideas). In C. Binnig, A. Fekete, and A. Nandi, editors, *Proceedings of the Workshop on Human-In-the-Loop Data Analytics, HILDA@SIGMOD 2016, San Francisco, CA, USA, June 26 - July 01, 2016*, page 11. ACM, 2016.

[41] W. Cui, H. Zhou, H. Qu, P. C. Wong, and X. Li. Geometry-based edge clustering for graph visualization. *IEEE Trans. Vis. Comput. Graph.*, 14(6):1277–1284, 2008.

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

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

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

[45] 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*, pages 1962–1976. ACM, 2022.

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

[47] Eclipse JGit `https://www.eclipse.org/jgit/`, 2023. last accessed: 2023-01-08.

[48] A. Eldawy, M. F. Mokbel, and C. Jonathan. Hadoopviz: A mapreduce framework for extensible visualization of big spatial data. In *32nd IEEE International Conference on Data Engineering, ICDE 2016, Helsinki, Finland, May 16-20, 2016*, pages 601–612. IEEE Computer Society, 2016.

[49] I. Elghandour and A. Aboulnaga. Restore: Reusing results of mapreduce jobs. *VLDB'12*, 2012.

[50] M. Ester, H. Kriegel, J. Sander, M. Wimmer, and X. Xu. Incremental clustering for mining in a data warehousing environment. In *VLDB'98, Proceedings of 24rd International Conference on Very Large Data Bases, August 24-27, 1998, New York City, New York, USA*, pages 323–333. Morgan Kaufmann, 1998.

[51] R. Fagin, P. G. Kolaitis, R. J. Miller, and L. Popa. Data exchange: semantics and query answering. *Theor. Comput. Sci.*, 336(1):89–124, 2005.

[52] M. Fiedler et al. State-of-the-art with regards to user-perceived quality of service and quality feedback. In *Euro-NGI Deliverable D. WP. JRA. 6. 1. 1.* 2004.

[53] S. Frey, F. Sadlo, K. Ma, and T. Ertl. Interactive progressive visualization with space-time error control. *IEEE Trans. Vis. Comput. Graph.*, 20(12):2397–2406, 2014.

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

[55] K. Greff, A. Klein, M. Chovanec, F. Hutter, and J. Schmidhuber. The sacred infrastructure for computational research. In *SCIPY*, 2017.

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

[57] P. K. Gunda, L. Ravindranath, C. A. Thekkath, Y. Yu, and L. Zhuang. Nectar: Automatic management of data and computation in datacenters. In R. H. Arpaci-Dusseau and B. Chen, editors, *9th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2010, October 4-6, 2010, Vancouver, BC, Canada, Proceedings*, pages 75–88. USENIX Association, 2010.

[58] T. Guo, K. Feng, G. Cong, and Z. Bao. Efficient selection of geospatial data on maps for interactive and visualized exploration. In *Proceedings of the 2018 International Conference on Management of Data, SIGMOD Conference 2018, Houston, TX, USA, June 10-15, 2018*, pages 567–582. ACM, 2018.

[59] A. Y. Halevy. Answering queries using views: A survey. *The VLDB Journal*, 10(4):270–294, Dec. 2001.

[60] S. Y. Han, K. C. Clarke, and M. Tsou. Animated flow maps for visualizing human movement: Two demonstrations with air traffic and twitter data. In A. Magdy, X. Zhou, and Y. Huang, editors, *Proceedings of the 1st ACM SIGSPATIAL Workshop on Analytics for Local Events and News, Redondo Beach, CA, USA, November 7-10, 2017*, pages 5:1–5:10. ACM, 2017.

[61] S. Hasan, S. V. Ukkusuri, and X. Zhan. Understanding social influence in activity location choice and lifestyle patterns using geolocation data from social media. *Frontiers ICT*, 3:10, 2016.

[62] Hierarchical clustering. `https://github.com/mapbox/supercluster`.

[63] D. Holten. Hierarchical edge bundles: Visualization of adjacency relations in hierarchical data. *IEEE Trans. Vis. Comput. Graph.*, 12(5):741–748, 2006.

[64] D. Holten and J. J. van Wijk. Force-directed edge bundling for graph visualization. *Comput. Graph. Forum*, 28(3):983–990, 2009.

[65] S. Huang, L. Xu, J. Liu, A. J. Elmore, and A. G. Parameswaran. Orpheusdb: Bolt-on versioning for relational databases. *VLDB*, 2017.

[66] C. Hung, W. Peng, and W. Lee. Clustering and aggregating clues of trajectories for mining trajectory patterns and routes. *VLDB J.*, 24(2):169–192, 2015.

[67] C. Hurter, O. Ersoy, S. I. Fabrikant, T. R. Klein, and A. C. Telea. Bundled visualization of dynamicgraph and trail data. *IEEE Trans. Vis. Comput. Graph.*, 20(8):1141–1157, 2014.

[68] C. Hurter, O. Ersoy, and A. Telea. Graph bundling by kernel density estimation. *Comput. Graph. Forum*, 31(3):865–874, 2012.

[69] C. Hurter, O. Ersoy, and A. Telea. Smooth bundling of large streaming and sequence graphs. In *IEEE Pacific Visualization Symposium, PacificVis 2013, February 27 2013-March 1, 2013, Sydney, NSW, Australia*, pages 41–48. IEEE Computer Society, 2013.

[70] Imdb datasets website. `https://www.imdb.com/interfaces/`.

[71] Imdb workload website. `https://github.com/juanmanubens/SQL-Advanced-Queries/blob/master/imdb.sql`.

[72] M. Ivanova, M. L. Kersten, N. J. Nes, and R. Goncalves. An architecture for recycling intermediates in a column-store. In U. Çetintemel, S. B. Zdonik, D. Kossmann, and N. Tatbul, editors, *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2009, Providence, Rhode Island, USA, June 29 - July 2, 2009*, pages 309–320. ACM, 2009.

[73] `https://www.rfc-editor.org/rfc/rfc6902`, 2023. last accessed: 2023-01-07.

[74] T. S. Jayram, P. G. Kolaitis, and E. Vee. The containment problem for REAL conjunctive queries with inequalities. In S. Vansummeren, editor, *Proceedings of the Twenty-Fifth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems, June 26-28, 2006, Chicago, Illinois, USA*, pages 80–89. ACM, 2006.

[75] J. Jia, C. Li, and M. J. Carey. Drum: A rhythmic approach to interactive analytics on large data. In *BigData*, pages 636–645, 2017.

[76] A. Jindal, K. Karanasos, S. Rao, and H. Patel. Selecting subexpressions to materialize at datacenter scale. *Proc. VLDB Endow.*, 11(7):800–812, 2018.

[77] Jupyter Notebook Website, `https://jupyter.org/`.

[78] M. A. Khan, L. Xu, A. Nandi, and J. M. Hellerstein. Data tweening: Incremental visualization of data transforms. *Proc. VLDB Endow.*, 10(6):661–672, 2017.

[79] J. Kim and J. Lee. Community detection in multi-layer graphs: A survey. *SIGMOD Rec.*, 44(3):37–48, 2015.

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

[81] Knime Website, `https://www.knime.com/`.

[82] Knime workflows website. `https://hub.knime.com/search?type=Workflow&sort=maxKudos`.

[83] J. Kossmann, T. Papenbrock, and F. Naumann. Data dependencies for query optimization: a survey. *VLDB J.*, 31(1):1–22, 2022.

[84] A. Kumar, Z. Wang, S. Ni, and C. Li. Amber: A debuggable dataflow system based on the actor model. *Proc. VLDB Endow.*, 13(5):740–753, 2020.

[85] M. Kunjir, B. Fain, K. Munagala, and S. Babu. ROBUS: fair cache allocation for data-parallel workloads. In S. Salihoglu, W. Zhou, R. Chirkova, J. Yang, and D. Suciu, editors, *Proceedings of the 2017 ACM International Conference on Management of Data, SIGMOD Conference 2017, Chicago, IL, USA, May 14-19, 2017*, pages 219–234. ACM, 2017.

[86] A. Lambert, R. Bourqui, and D. Auber. Winding roads: Routing edges into bundles. *Comput. Graph. Forum*, 29(3):853–862, 2010.

[87] J. Lee, J. Han, X. Li, and H. Gonzalez. *TraClass*: trajectory classification using hierarchical region-based and trajectory-based clustering. *Proc. VLDB Endow.*, 1(1):1081–1094, 2008.

[88] J. Lee, J. Han, and K. Whang. Trajectory clustering: a partition-and-group framework. In C. Y. Chan, B. C. Ooi, and A. Zhou, editors, *Proceedings of the ACM SIGMOD International Conference on Management of Data, Beijing, China, June 12-14, 2007*, pages 593–604. ACM, 2007.

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

[90] G. Li, S. Chen, J. Feng, K. Tan, and W. Li. Efficient location-aware influence maximization. In C. E. Dyreson, F. Li, and M. T. Özsu, editors, *International Conference on Management of Data, SIGMOD 2014, Snowbird, UT, USA, June 22-27, 2014*, pages 87–98. ACM, 2014.

[91] J. Li, T. Sellis, J. S. Culpepper, Z. He, C. Liu, and J. Wang. Geo-social influence spanning maximization. In *34th IEEE International Conference on Data Engineering, ICDE 2018, Paris, France, April 16-19, 2018*, pages 1775–1776. IEEE Computer Society, 2018.

[92] L. D. Lins, J. T. Klosowski, and C. E. Scheidegger. Nanocubes for real-time exploration of spatiotemporal datasets. *IEEE Trans. Vis. Comput. Graph.*, 19(12):2456–2465, 2013.

[93] X. Liu, Z. Wang, S. Ni, S. Alsudais, Y. Huang, A. Kumar, and C. Li. Demonstration of collaborative and interactive workflow-based data analytics in texera. *Proc. VLDB Endow.*, 15(12):3738–3741, 2022.

[94] Z. Liu, B. Jiang, and J. Heer. *imMens*: Real-time visual querying of big data. *Comput. Graph. Forum*, 32(3):421–430, 2013.

[95] Z. Luo, S. H. Yeung, M. Zhang, K. Zheng, L. Zhu, G. Chen, F. Fan, Q. Lin, K. Y. Ngiam, and B. C. Ooi. Mlcask: Efficient management of component evolution in collaborative data analytics pipelines. In *ICDE*, 2021.

[96] S. T. Mai, M. S. Dieu, I. Assent, J. Jacobsen, J. Kristensen, and M. S. Birk. Scalable and interactive graph clustering algorithm on multicore cpus. In *33rd IEEE International Conference on Data Engineering, ICDE 2017, San Diego, CA, USA, April 19-22, 2017*, pages 349–360. IEEE Computer Society, 2017.

[97] E. Masciari. A framework for trajectory clustering. In N. Trigoni, A. Markham, and S. Nawaz, editors, *GeoSensor Networks, Third International Conference, GSN 2009, Oxford, UK, July 13-14, 2009. Proceedings*, volume 5659 of *Lecture Notes in Computer Science*, pages 102–111. Springer, 2009.

[98] H. Miao, A. Chavan, and A. Deshpande. Provdb: Lifecycle management of collaborative analysis workflows. In *HILDA@SIGMOD*, 2017.

[99] H. Miao and A. Deshpande. Provdb: Provenance-enabled lifecycle management of collaborative data analysis workflows. *IEEE Data Eng. Bull.*, 2018.

[100] H. Miao, A. Li, L. S. Davis, and A. Deshpande. Towards unified data and lifecycle management for deep learning. In *ICDE'17*, 2017.

[101] D. Moritz, D. Fisher, B. Ding, and C. Wang. Trust, but verify: Optimistic visualizations of approximate queries for exploring big data. In *Proceedings of the 2017 CHI Conference on Human Factors in Computing Systems, Denver, CO, USA, May 06-11, 2017*, pages 2904–2915. ACM, 2017.

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

[103] J. Nesetril and P. O. de Mendez. *Sparsity - Graphs, Structures, and Algorithms*, volume 28 of *Algorithms and combinatorics*. Springer, 2012.

[104] Q. H. Nguyen, P. Eades, and S. Hong. Streameb: Stream edge bundling. In *Graph Drawing - 20th International Symposium, GD 2012, Redmond, WA, USA, September 19-21, 2012, Revised Selected Papers*, volume 7704 of *Lecture Notes in Computer Science*, pages 400–413. Springer, 2012.

[105] Q. H. Nguyen, P. Eades, and S. Hong. Towards faithful graph visualizations. *CoRR*, abs/1701.00921, 2017.

[106] NYC Taxi Data, `https://www1.nyc.gov/site/tlc/about/tlc-trip-record-data.page`.

[107] DVC Website `https://dvc.org/`, 2023. last accessed: 2023-01-07.

[108] Optimizing apache spark udfs website. `https://www.databricks.com/session_eu20/optimizing-apache-spark-udfs`.

[109] Orange data mining workflows. `https://orangedatamining.com/workflows/`.

[110] Y. Park, M. J. Cafarella, and B. Mozafari. Visualization-aware sampling for very large databases. In *32nd IEEE International Conference on Data Engineering, ICDE 2016, Helsinki, Finland, May 16-20, 2016*, pages 755–766. IEEE Computer Society, 2016.

[111] B. K. Patra, V. Ollikainen, R. Launonen, S. Nandi, and K. S. Babu. Distance based incremental clustering for mining clusters of arbitrary shapes. In *Pattern Recognition and Machine Intelligence - 5th International Conference, PReMI 2013, Kolkata, India, December 10-14, 2013. Proceedings*, volume 8251 of *Lecture Notes in Computer Science*, pages 229–236. Springer, 2013.

[112] N. Pelekis, P. Tampakis, M. Vodas, C. Panagiotakis, and Y. Theodoridis. In-dbms sampling-based sub-trajectory clustering. In V. Markl, S. Orlando, B. Mitschang, P. Andritsos, K. Sattler, and S. Breß, editors, *Proceedings of the 20th International Conference on Extending Database Technology, EDBT 2017, Venice, Italy, March 21-24, 2017*, pages 632–643. OpenProceedings.org, 2017.

[113] L. L. Perez and C. M. Jermaine. History-aware query optimization with materialized intermediate views. In I. F. Cruz, E. Ferrari, Y. Tao, E. Bertino, and G. Trajcevski, editors, *IEEE 30th International Conference on Data Engineering, Chicago, ICDE 2014, IL, USA, March 31 - April 4, 2014*, pages 520–531. IEEE Computer Society, 2014.

[114] PostgreSQL UDF Create Function Website, `https://www.postgresql.org/docs/current/sql-createfunction.html`.

[115] M. Procopio, C. Scheidegger, E. Wu, and R. Chang. Selective wander join: Fast progressive visualizations for data joins. *Informatics*, 6(1):14, 2019.

[116] V. Raghupathi, J. Ren, and W. Raghupathi. Studying public perception about vaccination: A sentiment analysis of tweets. *International journal of environmental research and public health*, 17(10):3464, 2020.

[117] S. Rahman, M. Aliakbarpour, H. Kong, E. Blais, K. Karahalios, A. G. Parameswaran, and R. Rubinfeld. I've seen "enough": Incrementally improving visualizations to support rapid decision making. *Proc. VLDB Endow.*, 10(11):1262–1273, 2017.

[118] 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*, pages 206–219. ACM, 2019.

[119] RapidMiner Website, `https://rapidminer.com/`.

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

[121] R. Rosenholtz, Y. Li, Z. Jin, and J. Mansfield. Feature congestion: A measure of visual clutter. *Journal of Vision - J VISION*, 6:827–827, 06 2010.

[122] A. Roy, A. Jindal, P. Gomatam, X. Ouyang, A. Gosalia, N. Ravi, S. Mann, and P. Jain. Sparkcruise: Workload optimization in managed spark clusters at microsoft. *Proc. VLDB Endow.*, 14(12):3122–3134, 2021.

[123] L. Rupprecht, J. C. Davis, C. Arnold, Y. Gur, and D. Bhagwat. Improving reproducibility of data science pipelines through transparent provenance capture. *VLDB*, 2020.

[124] Y. Sagiv and M. Yannakakis. Equivalences among relational expressions with the union and difference operators. *J. ACM*, 27(4):633–655, 1980.

[125] M. Sarwat, J. J. Levandoski, A. Eldawy, and M. F. Mokbe. Umn sarwat foursquare dataset (september 2013), 2013.

[126] V. Satuluri, S. Parthasarathy, and Y. Ruan. Local graph sparsification for scalable clustering. In T. K. Sellis, R. J. Miller, A. Kementsietsidis, and Y. Velegrakis, editors, *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2011, Athens, Greece, June 12-16, 2011*, pages 721–732. ACM, 2011.

[127] M. Schmidt, M. Meier, and G. Lausen. Foundations of SPARQL query optimization. In L. Segoufin, editor, *Database Theory - ICDT 2010, 13th International Conference, Lausanne, Switzerland, March 23-25, 2010, Proceedings*, ACM International Conference Proceeding Series, pages 4–33. ACM, 2010.

[128] S. Schöttler, Y. Yang, H. Pfister, and B. Bach. Visualizing and interacting with geospatial networks: A survey and design space. *CoRR*, abs/2101.06322, 2021.

[129] D. Selassie, B. Heller, and J. Heer. Divided edge bundling for directional network data. *IEEE Trans. Vis. Comput. Graph.*, 17(12):2354–2363, 2011.

[130] Z. Shang, E. Zgraggen, B. Buratti, P. Eichmann, N. Karimeddiny, C. Meyer, W. Runnels, and T. Kraska. Davos: A system for interactive data-driven decision making. *Proc. VLDB Endow.*, 14(12):2893–2905, 2021.

[131] Y. N. Silva, P. Larson, and J. Zhou. Exploiting common subexpressions for cloud query processing. In A. Kementsietsidis and M. A. V. Salles, editors, *IEEE 28th International Conference on Data Engineering (ICDE 2012), Washington, DC, USA (Arlington, Virginia), 1-5 April, 2012*, pages 1337–1348. IEEE Computer Society, 2012.

[132] Y. Skadberg and J. R. Kimmel. Visitors' flow experience while browsing a Web site: its measurement, contributing factors and consequences. *Computers in Human Behavior*, 20(3):403–422, 2004.

[133] Tableau website. `https://www.tableau.com`.

[134] T. Takahashi, H. Shiokawa, and H. Kitagawa. SCAN-XP: parallel structural graph clustering algorithm on intel xeon phi coprocessors. In A. Arora, S. Roy, and A. Bhattacharya, editors, *Proceedings of the 2nd International Workshop on Network Data Analytics, NDA@SIGMOD 2017, Chicago, IL, USA, May 19, 2017*, pages 6:1–6:7. ACM, 2017.

[135] P. Tampakis, N. Pelekis, N. V. Andrienko, G. L. Andrienko, G. Fuchs, and Y. Theodoridis. Time-aware sub-trajectory clustering in hermes@postgresql. In *34th IEEE International Conference on Data Engineering, ICDE 2018, Paris, France, April 16-19, 2018*, pages 1581–1584. IEEE Computer Society, 2018.

[136] W. Tao, X. Liu, Ç. Demiralp, R. Chang, and M. Stonebraker. Kyrix: Interactive visual data exploration at scale. In *9th Biennial Conference on Innovative Data Systems Research, CIDR 2019, Asilomar, CA, USA, January 13-16, 2019, Online Proceedings*. www.cidrdb.org, 2019.

[137] A. Telea and O. Ersoy. Image-based edge bundles: Simplified visualization of large graphs. *Comput. Graph. Forum*, 29(3):843–852, 2010.

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

[139] C. Tominski, J. Abello, and H. Schumann. CGV - an interactive graph visualization system. *Comput. Graph.*, 33(6):660–678, 2009.

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

[141] C. Turkay, E. Kaya, S. Balcisoy, and H. Hauser. Designing progressive and interactive analytics processes for high-dimensional data analysis. *IEEE Trans. Vis. Comput. Graph.*, 23(1):131–140, 2017.

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

[143] M. Vartak, H. Subramanyam, W. Lee, S. Viswanathan, S. Husnoo, S. Madden, and M. Zaharia. Modeldb: a system for machine learning model management. In *HILDA@SIGMOD'16*, 2016.

[144] S. Wang, Z. Bao, J. S. Culpepper, T. Sellis, and X. Qin. Fast large-scale trajectory clustering. *Proc. VLDB Endow.*, 13(1):29–42, 2019.

[145] X. Wang, X. Niu, J. Zhu, and Z. Liu. An approach to spatiotemporal trajectory clustering based on community detection. *Wirel. Commun. Mob. Comput.*, 2021:5582341:1–5582341:10, 2021.

[146] 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*, pages 94–107. ACM, 2022.

[147] H. Wei, J. Sankaranarayanan, and H. Samet. Measuring spatial influence of twitter users by interactions. In A. Magdy, X. Zhou, and Y. Huang, editors, *Proceedings of the 1st ACM SIGSPATIAL Workshop on Analytics for Local Events and News, Redondo Beach, CA, USA, November 7-10, 2017*, pages 2:1–2:10. ACM, 2017.

[148] D. Wen, L. Qin, Y. Zhang, L. Chang, and X. Lin. Efficient structural graph clustering: an index-based approach. *VLDB J.*, 28(3):377–399, 2019.

[149] A. T. Wilson, M. D. Rintoul, and C. G. Valicka. Exploratory trajectory clustering with distance geometry. In D. D. Schmorrow and C. M. Fidopiastis, editors, *Foundations of Augmented Cognition: Neuroergonomics and Operational Neuroscience - 10th International Conference, AC 2016, Held as Part of HCI International 2016, Toronto, ON, Canada, July 17-22, 2016, Proceedings, Part II*, volume 9744 of *Lecture Notes in Computer Science*, pages 263–274. Springer, 2016.

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

[151] D. Xin, S. Macke, L. Ma, J. Liu, S. Song, and A. G. Parameswaran. Helix: Holistic optimization for accelerating iterative machine learning. *Proc. VLDB Endow.*, 12(4):446–460, 2018.

[152] H. Xu, Y. Zhou, W. Lin, and H. Zha. Unsupervised trajectory clustering via adaptive multi-kernel-based shrinkage. In *2015 IEEE International Conference on Computer Vision, ICCV 2015, Santiago, Chile, December 7-13, 2015*, pages 4328–4336. IEEE Computer Society, 2015.

[153] X. Xu, N. Yuruk, Z. Feng, and T. A. J. Schweiger. SCAN: a structural clustering algorithm for networks. In P. Berkhin, R. Caruana, and X. Wu, editors, *Proceedings of*

the 13th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, San Jose, California, USA, August 12-15, 2007, pages 824–833. ACM, 2007.

[154] Z. Xu, G. T. Kakkar, J. Arulraj, and U. Ramachandran. EVA: A symbolic approach to accelerating exploratory video analytics with materialized views. In Z. Ives, A. Bonifati, and A. E. Abbadi, editors, *SIGMOD '22: International Conference on Management of Data, Philadelphia, PA, USA, June 12 - 17, 2022*, pages 602–616. ACM, 2022.

[155] Z. Xu, Y. Ke, Y. Wang, H. Cheng, and J. Cheng. A model-based approach to attributed graph clustering. In K. S. Candan, Y. Chen, R. T. Snodgrass, L. Gravano, and A. Fuxman, editors, *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2012, Scottsdale, AZ, USA, May 20-24, 2012*, pages 505–516. ACM, 2012.

[156] J. Yu and M. Sarwat. Turbocharging geospatial visualization dashboards via a materialized sampling cube approach. In *36th IEEE International Conference on Data Engineering, ICDE 2020, Dallas, TX, USA, April 20-24, 2020*, pages 1165–1176. IEEE, 2020.

[157] C. Zhang, L. Shou, K. Chen, G. Chen, and Y. Bei. Evaluating geo-social influence in location-based social networks. In X. Chen, G. Lebanon, H. Wang, and M. J. Zaki, editors, *21st ACM International Conference on Information and Knowledge Management, CIKM'12, Maui, HI, USA, October 29 - November 02, 2012*, pages 1442–1451. ACM, 2012.

[158] T. Zhang, R. Ramakrishnan, and M. Livny. BIRCH: an efficient data clustering method for very large databases. In *Proceedings of the 1996 ACM SIGMOD International Conference on Management of Data, Montreal, Quebec, Canada, June 4-6, 1996*, pages 103–114. ACM Press, 1996.

[159] Y. Zhang, F. Xu, E. Frise, S. Wu, B. Yu, and W. Xu. Datalab: a version data management and analytics system. In *BIGDSE@ICSE'16*, 2016.

[160] H. Zhou, P. Xu, X. Yuan, and H. Qu. Edge bundling in information visualization. *Tsinghua Science and Technology*, 18(2):145–156, 2013.

[161] H. Zhou, X. Yuan, W. Cui, H. Qu, and B. Chen. Energy-based hierarchical edge clustering of graphs. In *IEEE VGTC Pacific Visualization Symposium 2008, PacificVis 2008, Kyoto, Japan, March 5-7, 2008*, pages 55–61. IEEE Computer Society, 2008.

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

[163] Q. Zhou, J. Arulraj, S. B. Navathe, W. Harris, and J. Wu. SPES: A symbolic approach to proving query equivalence under bag semantics. In *38th IEEE International Conference on Data Engineering, ICDE 2022, Kuala Lumpur, Malaysia, May 9-12, 2022*, pages 2735–2748. IEEE, 2022.

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

[165] Y. Zhou, H. Cheng, and J. X. Yu. Graph clustering based on structural/attribute similarities. *Proc. VLDB Endow.*, 2(1):718–729, 2009.

[166] W. Zhu, W. Peng, L. Chen, K. Zheng, and X. Zhou. Exploiting viral marketing for location promotion in location-based social networks. *ACM Trans. Knowl. Discov. Data*, 11(2):25:1–25:28, 2016.

Appendix A

Real-workflow Workload Statistics

The following Figure A.1 shows the details of a real workload collected from one deployment of Texera [138].

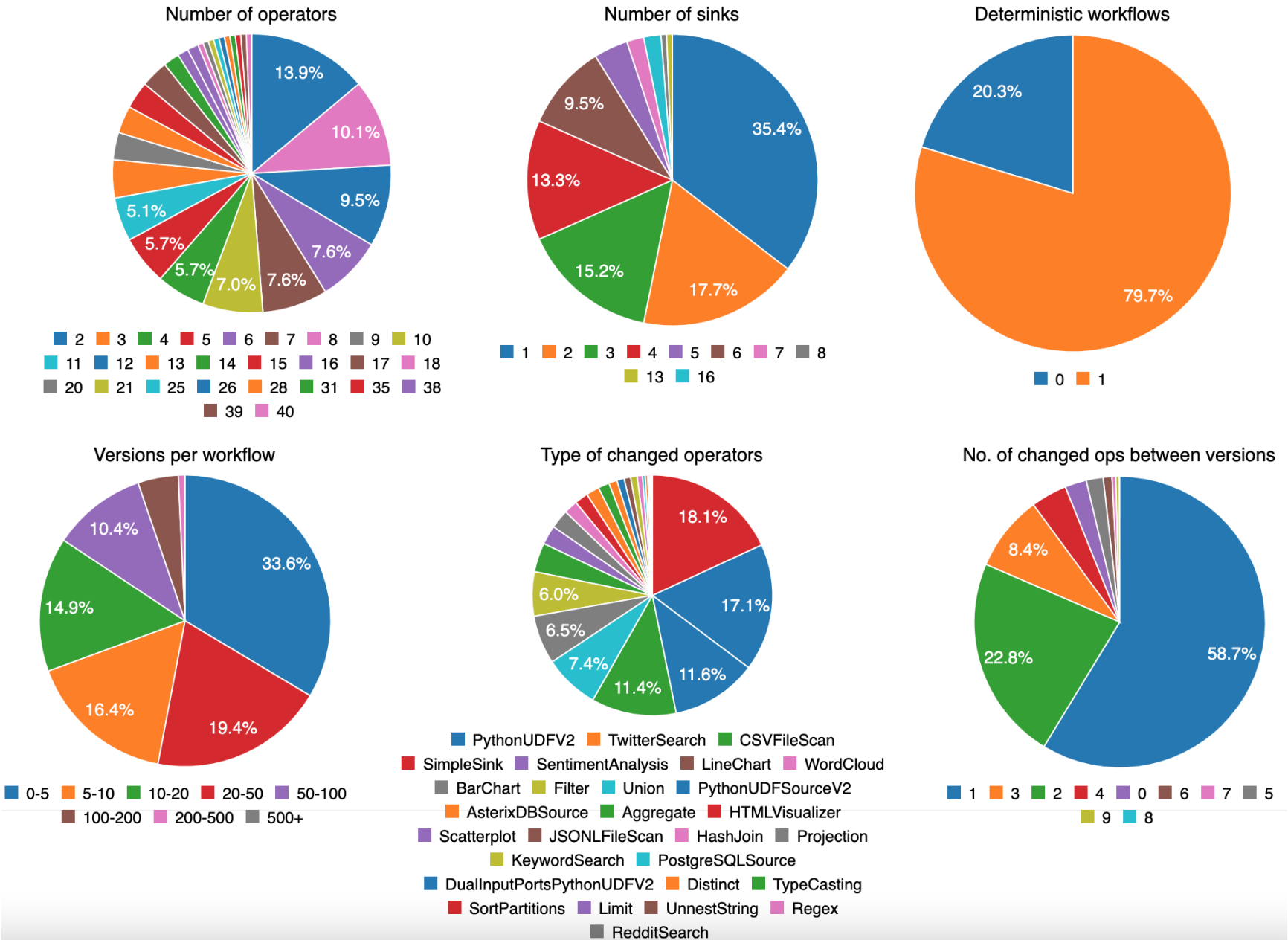


Figure A.1: Details of workflow from one real workload

Appendix B

Sample Real Workflows

The following Figures B.1- B.7 show samples of the initial version of the workflows used in some of the experiments in Chapter 4 and Chapter 5. The UDF operator included in the transformed TPC-DS queries to Texera workflows, is to include the logic of Order By, because Sort operator was not part of the Texera system at the time the workflow was constructed. The workflows include in some cases a sequence of filter operators, because at the time the workflows were constructed, Texera did not support the AND operation to join multiple predicate conditions. The sample workflows include a single sink only to show a portion of the task.

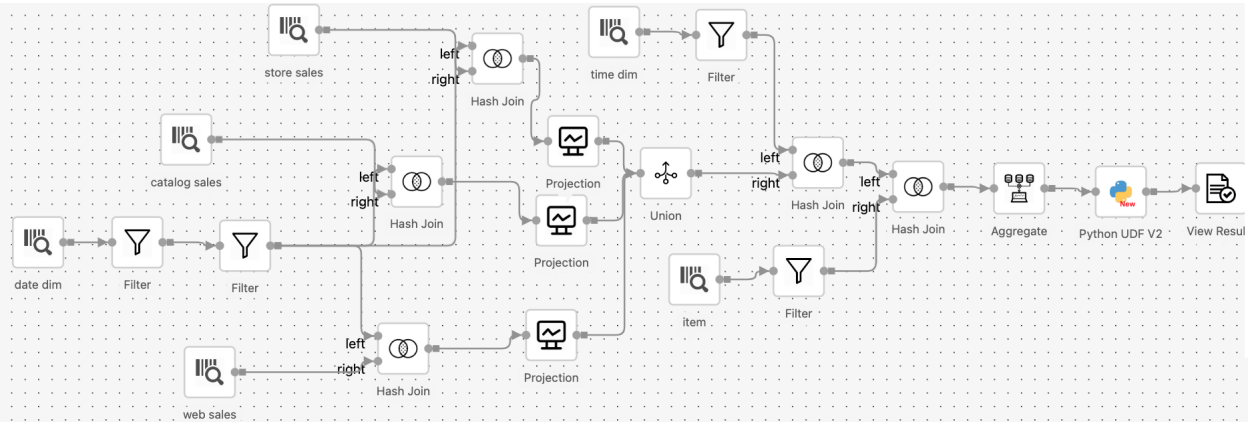


Figure B.1: Sample workflow of TPC-DS Q71 in Texera.

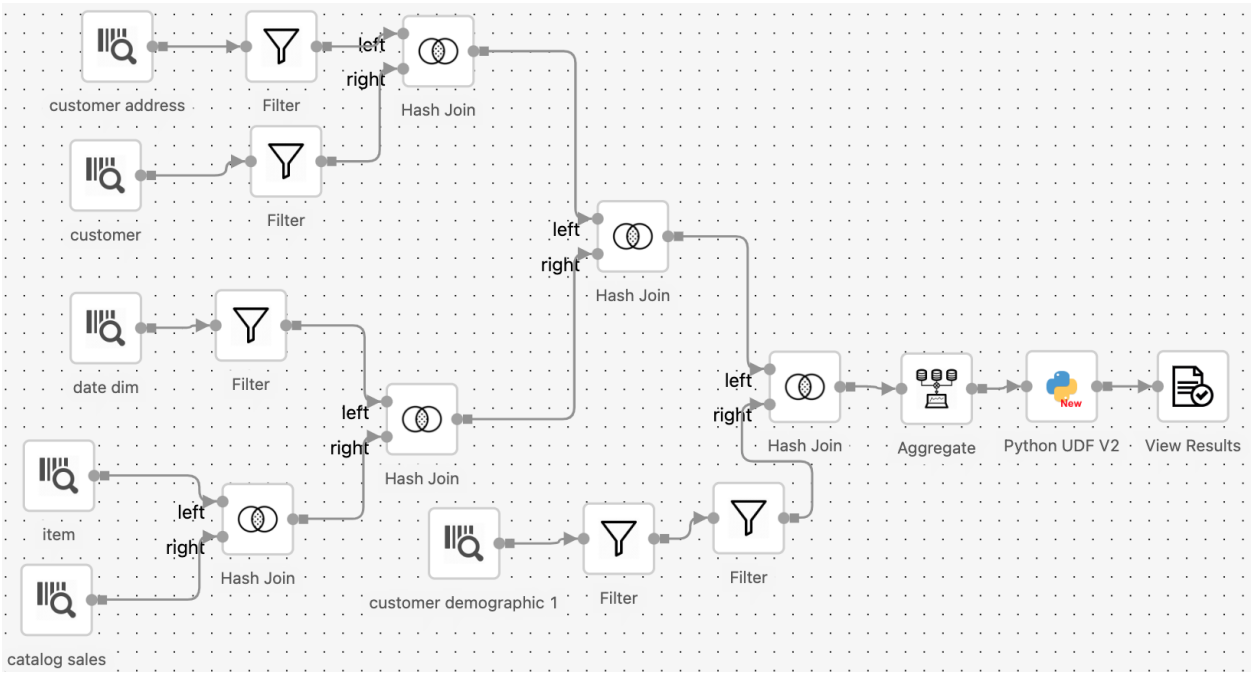


Figure B.2: Sample workflow of TPC-DS Q18 in Texera.

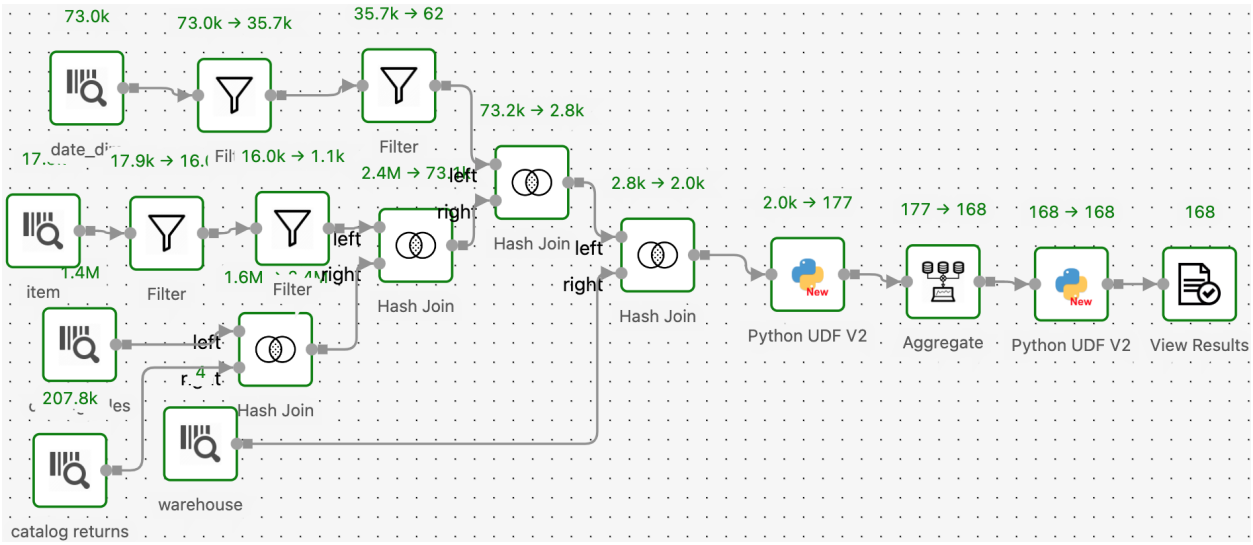


Figure B.3: Sample workflow of TPC-DS Q40 in Texera.

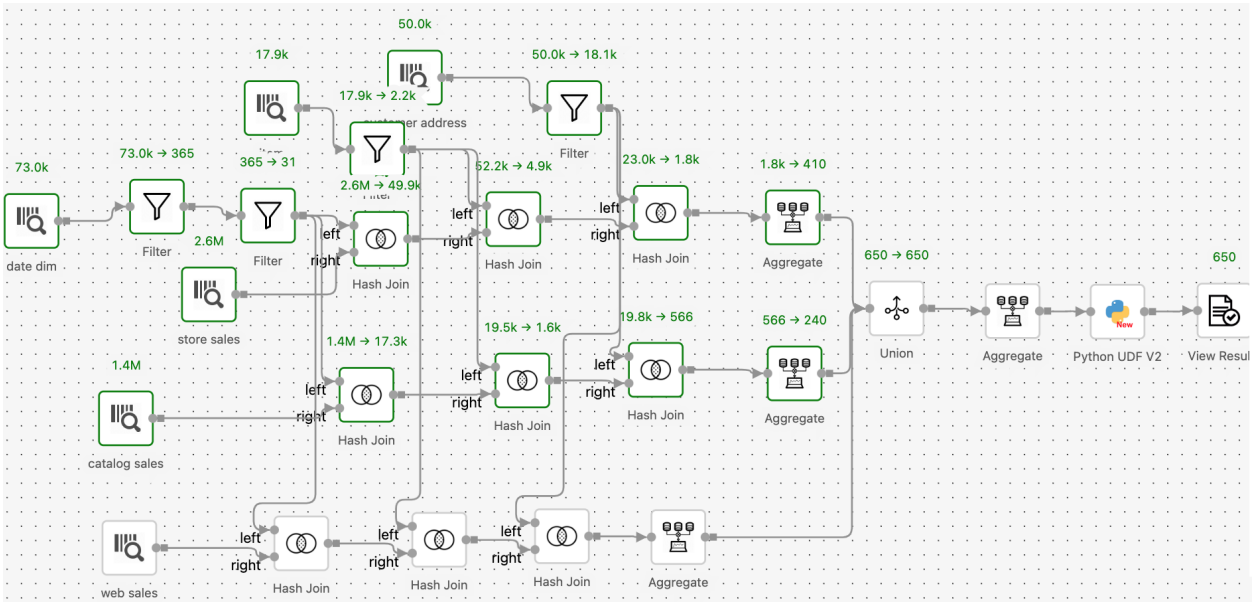


Figure B.4: Sample workflow of TPC-DS Q33 in Texera.

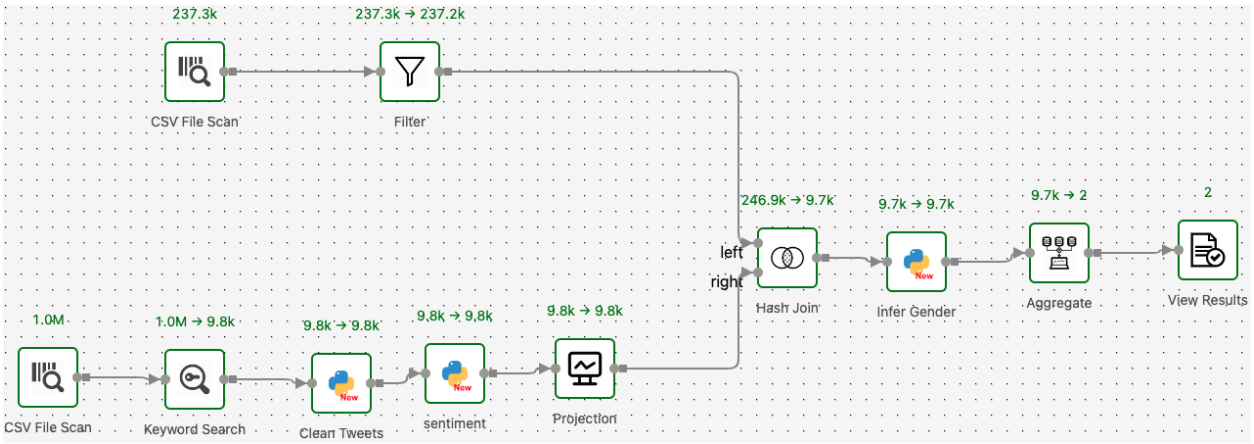


Figure B.5: Sample workflow of Tweets to infer the gender of the Tweeter.

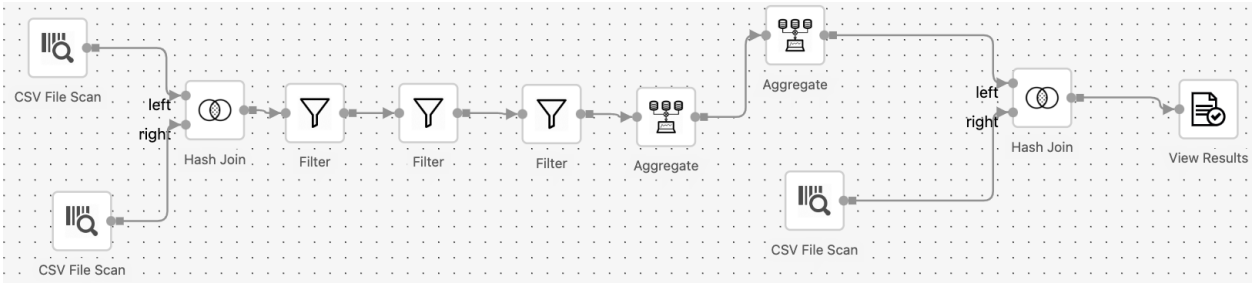


Figure B.6: Sample workflow of IMDB movies to calculate the ratio of original to non-original movies.

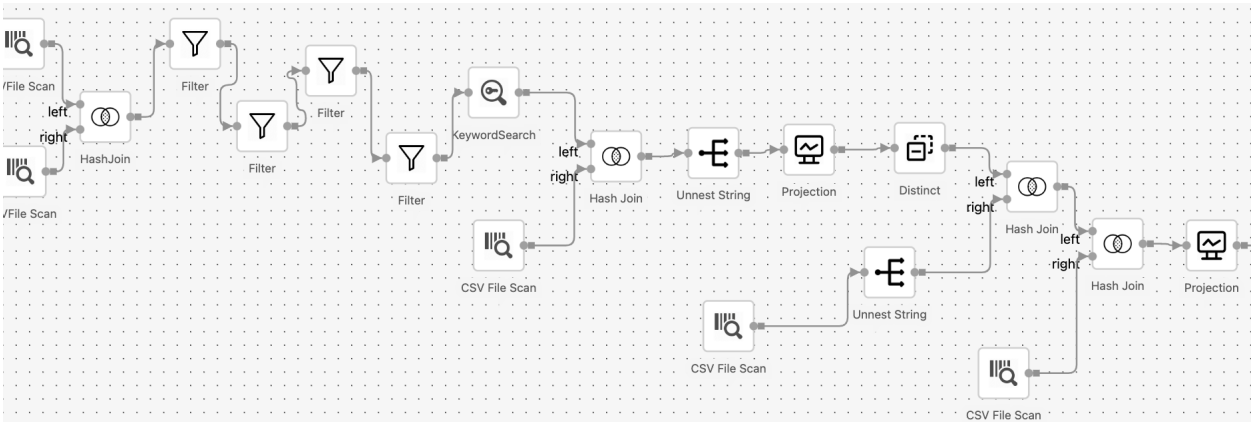


Figure B.7: A portion of a workflow to get all movies of directors that have certain criteria.