THE UNIVERSITY OF CHICAGO

RECONSTRUCTING THE LINEAGE OF ARTIFACTS IN DATA LAKES

A DISSERTATION SUBMITTED TO
THE FACULTY OF THE DIVISION OF THE PHYSICAL SCIENCES
IN CANDIDACY FOR THE DEGREE OF
DOCTOR OF PHILOSOPHY

DEPARTMENT OF COMPUTER SCIENCE

BY
MOHAMMED SUHAIL REHMAN

CHICAGO, ILLINOIS
MARCH 2023

TABLE OF CONTENTS

LIST OF FIGURES vi

LIST OF TABLES vii

ACKNOWLEDGMENTS viii

ABSTRACT ix

1 INTRODUCTION 1
1.1 Thesis Statement 2
1.2 Thesis Outline 2

2 BACKGROUND 4
2.1 Motivation 4
2.2 Related Work 6
2.2.1 Research in Data Lineage 6
2.2.2 Online Lineage Capture / Curation / Version Control 7
2.2.3 Data Discovery 7
2.2.4 Query Synthesis 8
2.2.5 Document Provenance Systems 8
2.2.6 Dataframe Benchmarking 9

3 THE RELIC SYSTEM 11
3.1 Introduction 11
3.2 Problem Definition 16
3.2.1 Notation 16
3.2.2 The Lineage Inference Task 17
3.2.3 Types of Transformations 18
3.3 Lineage Inference Technique 20
3.3.1 Initialization 21
3.3.2 Building the Lineage Graph 23
3.3.3 Inferring Point Preserving Transformations 25
3.3.4 Inferring Non-Point-Preserving Transformations 27
3.3.5 Breaking Ties when Selecting Edges 33
3.4 Experimental Evaluation 34
3.4.1 Datasets Used 35
3.4.2 Configurations to be Evaluated 37
3.4.3 Overall Accuracy 38
3.4.4 Individual Detector Performance 39
3.4.5 Time to Infer Workflows 40
3.5 Conclusions and Future Work 41
3.6 Relaxing Relic's Assumptions 42

3.6.1 Effect of Materialization Rate on Accuracy . . . 42
3.6.2 Unmatched Schema . . . 43
3.6.3 Multiple Workflows . . . 45

4 SCALING RELIC . . . 50
4.1 Motivation . . . 50
4.2 Preliminaries . . . 54
4.2.1 MinHashing . . . 54
4.2.2 One Permutation Hashing . . . 55
4.2.3 K-Minimum Value Sketches . . . 55
4.2.4 Locality Sensitive Hashing . . . 56
4.3 Coupled Containment Sketches . . . 57
4.3.1 Sketch Construction . . . 57
4.3.2 Index Construction . . . 58
4.3.3 Querying the Index for Lineage . . . 59
4.3.4 Ranking the Results . . . 60
4.4 Implementation . . . 61
4.4.1 Sketching Techniques Used . . . 61
4.4.2 Software Implementation . . . 62
4.5 Evaluation . . . 63
4.5.1 Datasets Used . . . 64
4.5.2 Competing Systems . . . 65
4.5.3 Evaluation Metrics . . . 66
4.6 Results . . . 67
4.6.1 Index Query Accuracy . . . 67
4.6.2 Result Ranking . . . 68
4.6.3 Baseline Performance . . . 78
4.6.4 Index Generation and Query Runtimes and Space Requirements . . . 79
4.7 Conclusions . . . 82

5 THE FUZZYDATA WORKFLOW SPECIFICATION SYSTEM . . . 83
5.1 FuzzyData Design . . . 85
5.1.1 Data Model . . . 86
5.1.2 Implementation . . . 88
5.1.3 Generating Random Artifacts . . . 89
5.1.4 Generating and Replaying Random Workflows . . . 89
5.1.5 Instrumentation . . . 92
5.1.6 Clients Implemented . . . 93
5.2 Use Cases . . . 103
5.2.1 Fuzzy Testing Suite for Dataframe Systems . . . 104
5.2.2 Encoding and Replaying an Existing Workflow . . . 104
5.2.3 Scaling and Replaying a Generated Workflow . . . 106
5.3 Deployment on Modin . . . 108
5.4 Conclusions and Future Work . . . 108

6 CONCLUSIONS 109

REFERENCES 110

LIST OF FIGURES

2.1 The Compliance/Accessibility Trade-off in Data Lakes 5

3.1 Example of a Typical Data Analysis Workflow 11
3.2 Motivating Example for the Lineage Inference Problem. 12
3.3 Point-Preserving and Non-Point-Preserving Operations 18
3.4 RELIC Running Example 21
3.5 Cell-Level Comparison of Artifacts 25
3.6 Lineage Recovery Accuracy 39
3.7 RELIC Runtime 47
3.8 Materializaiton Rate vs Accuracy 48
3.9 Accuracy of Schema Matching 48
3.10 RELIC's Accuracy with Unmatched Schema 49

4.1 Client-Server Interaction Model using Apache Arrow. 63
4.2 Index Recall Results for Groupby Queries on the GitTables Dataset 69
4.3 Index Recall Results for Join (Left) Queries on the GitTables Dataset 70
4.4 Index Recall Results for Join (Right) Queries on the GitTables Dataset 71
4.5 Index Recall Results for Pivot Queries on the GitTables Dataset 72
4.6 Index Recall Results for Groupby Queries on the Synthetic Dataset 73
4.7 Index Recall Results for Join (Left) Queries on the Synthetic Dataset 74
4.8 Index Recall Results for Join (Right) Queries on the Synthetic Dataset 75
4.9 Index Recall Results for Pivot Queries on the Synthetic Dataset 76

5.1 The design of FuzzyData. 84
5.2 DAG of a randomly generated workflow by FuzzyData 92
5.3 Real-World Workflow Replay 107
5.4 Results of the Scaling Experiments 107

LIST OF TABLES

3.1 Table Notation Used in this thesis 17
3.2 Set of Operations $\mathcal{O}$ Under Consideration 17
3.3 List of Real Workflows 36
3.4 Individual Stage Performance 40
3.5 Schema Matching Runtime 44
3.6 F1 score and cross-workflow false positive edges for RELIC on multiple workflows. 46

4.1 Example Table R 51
4.2 Example Table S 51
4.3 Example Table T 52
4.4 Example Table U 52
4.5 Example Table V 52
4.6 GitTables Dataset Filtering and Query Class Generation 64
4.7 Ranking Recall Results for the GitTables Dataset 77
4.8 Ranking Recall Results for the GitTables Dataset using 3-Column Sketches . . . 78
4.9 Baseline Ranking Recall Results for the GitTables Dataset 79
4.10 Baseline Ranking Recall Results for the Synthetic Dataset 80
4.11 Runtime Performance for Index Generation 80
4.12 Runtime Performance for Querying the Index 80
4.13 Runtime Performance for Ranking Candidates 81
4.14 Baseline Runtime Comparison 81
4.15 Index Size for the GitTables Dataset 81
4.16 Index Sizes as a Function of the Number of Rows 81

5.1 Transformation Implementation Matrix 86
5.2 An example artifact generated using FUZZYDATA 89
5.3 Parameters for generating synthetic workflows. 90

ACKNOWLEDGMENTS

I sincerely thank my thesis advisor, Aaron Elmore, for their invaluable guidance, support, and encouragement throughout this research project. Aaron has been extraordinarily patient and understanding as he ensured that I saw this project to completion. I am also grateful to my committee members, Raul Castro Fernandez and Michael J. Franklin, for their valuable insights and feedback as this project progressed. I would also like to acknowledge the support and assistance of the Department of Computer Science, University of Chicago, in providing the necessary resources and equipment for this research.

Additionally, I would like to thank my colleagues, friends, and family for their support and encouragement throughout this journey. My father nurtured my interest in computer science early on by ensuring an endless supply of equipment and software to play with during my formative years. My teachers at MES Indian School, Doha, Qatar, also significantly shaped my interest in computer science. I was nudged into research by Prof. P.J. Narayanan at the International Institute of Information Technology, Hyderabad. He saw a spark of interest in me and encouraged me to pursue research. Prof. Majd F. Sakr at Carnegie Mellon University, Qatar, was instrumental in providing valuable guidance and support as I navigated the then-novel areas of Cloud Computing which piqued my interest in large-scale Data Management. I would also like to thank my friends, especially my roommates, for their support and encouragement during this research project.

My wife, Mizaj, has been my rock throughout this journey. She has been my constant source of support and encouragement and has been instrumental in ensuring that I could focus on my research. I would also like to thank my parents, who have been my constant source of inspiration and motivation. They have always been my biggest cheerleaders and have encouraged me to follow my dreams.

ABSTRACT

The explosive growth of data-driven fields such as machine learning and data science has led to a proliferation of large amounts of data and systems, tools, and techniques to acquire, clean, process, prepare, curate, wrangle, and analyze data. This has led to the creation of data lakes – large repositories of data often used as a central source of truth for data-driven applications. However, the lack of lineage information in data lakes can affect the quality of data processed and the insights derived from data lakes. Existing solutions for lineage involve manual annotation of lineage information or capturing lineage as data is manipulated and transformed. This does not solve the problem of lineage and quality for data generated in the past and is often cited as an impediment to the overall vision of reducing the time to insight from vast amounts of data organized in a central data lake. This thesis proposes using similarity metrics to infer the lineage of data artifacts in data lakes. We show the feasibility of recovering the lineage of data artifacts under varying assumptions of the availability of metadata using Relic. We then scale Relic using sketching and indexing techniques and show that we can answer complex lineage queries accurately and efficiently with a suitable index structure. Finally, we also introduce FuzzyData, a dataframe benchmarking system that can generate dataframe workflows of varying complexity using different dataframe clients to benchmark and test dataframe-based systems.

CHAPTER 1

INTRODUCTION

Data lineage, also called provenance or pedigree, is information about data's origin and creation process. Lineage information is used to understand the semantics of processed data, tracing errors from the result of a transformation back to its input data, and estimating the quality of derived data [50]. Data lineage is a crucial component that enables error tracing, troubleshooting, quality estimation, and knowledge building in a variety of data systems ranging from relational databases [48], OLAP warehousing systems [20], NoSQL systems [95] and systems for the Machine Learning Lifecycle [94]. Quality lineage is crucial for building productive and sustainable data lakes; the lack of lineage information can affect the quality of data processed and the insights derived from data lakes [62].

Prior work in database lineage deals with lineage has always been discussed from a *capture*, *record* or *annotate* perspective [11, 34, 44, 48, 63, 94]. The assumption is that the lineage management system is running in conjunction with the core database software, monitoring accesses and logs or allowing for API calls to record the lineage information for every artifact accessed during each data life-cycle activity. These systems effectively solve the lineage problem going forward; none of the proposed solutions allow for retrospective tagging/analysis of data artifacts generated in the past.

On the other hand, existing data discovery solutions such as [15, 96, 98] do not consider lineage at all; they use techniques from information retrieval to process data from a query-focused perspective. These systems are designed to handle queries such as (*Retrieve all artifacts that match a set of keywords or is a likely join or union candidate for a specific input table*).

To our knowledge, there has not been any work on inferring the lineage of data artifacts in a *retrospective* manner, i.e., long after they have been created, without any supporting metadata about the artifacts that could help with the inference procedure. Such a scenario

is common within poorly maintained and governed data lakes and is often cited as an impediment to the overall vision of reducing the time to insight from vast amounts of data organized in a central data lake [62].

This disseratation aims to be the seminal retrospective data lineage thesis - we would like to outline the problem statement, its feasibility, applications, and solutions for the problem.

1.1 Thesis Statement

My thesis statement is as follows: *Given an expected grammar of data transformations, can we retrospectively use content similarity to:*

- Organize data lake artifacts into the individual workflows that generated them, and,
- Infer a lineage tree for the individual workflows that most closely resemble their ground truth derivations,
- In a manner that most closely represents the ground truth lineage, as measured by the edge F1 score?

1.2 Thesis Outline

This disseratation is organized as follows:

- Chapter 2 provides a background on the problem of data lineage, and the related work in the area.
- Chapter 3 introduces the problem of *retrospective lineage inference* (RLI) and outlines a first solution, Relic, which is used to solve the RLI problem under a set of strong assumptions. We then relax these assumptions and demonstrate the feasibility of solving the RLI problem by extending Relic to support multiple workflows, workflows with different materialization rates, and workflows with unmatched schemas.

- Chapter 4 evaluates different sketching techniques to improve the scalability of RELIC. We build a specialized contianment index that can be used to answer lineage queries, using existing Locality-Sensitive Hashing (LSH) and K-Minimum Value sketches. We show that the most time-consuming detectors in RELIC, namely the *join, groupby* and *pivot* detectors can be sped up by an order of magnitude by using these sketching and indexing techniques.

- Chapter 5 introduces a dataframe benchmarking system, FUZZYDATA, which can be used to generate dataframe workflows of varying complexity using different dataframe clients. We usde fuzzydata to generate a set of synthetic workflows for RELIC, and it has been further deployed to perform fuzzy testing of dataframe systems like modin.

- Chapter 6 presents the conclusions and future work.

CHAPTER 2

BACKGROUND

2.1 Motivation

Relational databases have been the mainstay of data processing systems since the advent of System R [8]. Within the relational model, operational data collection and storage were typically handled by transaction-focused OLTP systems, and analytical queries, reporting, and data exploration tasks were handled by read-query-optimized OLAP (data warehousing) systems. Decades of database research have helped in the fine-tuned performance optimizations of both these types of systems, as well as the development of hybrid engines that can be used to handle both types of workloads for organized, schema-defined data [47].

However, the explosion in data as part of the expansion of the Internet, smartphones, and sensor networks along with the ever-increasing requirements for complex analyses, machine learning, and analytics have resulted in a shift away from these traditional, rigid, schema-first relational databases [9].

Data analysts and machine learning practitioners often have to process large amounts of data from disparate sources, using multiple tools and processes and collaboratively, often using *lowest common denominator* data interchange formats such as CSV, Apache Parquet, and so on. The resulting decentralization of data away from relational database systems has prompted the industry to converge around the idea of *data lakes* – centralized, shared data repositories that allow for the storage of both raw data from multiple sources, as well as cleaned, transformed data and machine learning models [62]. The availability of cheap, easy to integrate distributed file systems such as HDFS and cloud storage such as S3 hastened the move towards this paradigm, making it easy to set up an internet-connected data collection and retention system that is instantly scalable on-demand. According to [7], most Fortune 500 companies operate a two-tiered architecture where a data lake is used to ingest and

feed non-relational analytics and machine learning pipelines, while some data is ETL-ed into warehouse systems for more traditional OLAP workloads.

While the data lake vision promises to enable data discovery, reduce time-to-insight for real-time analytics, and enable quick machine learning training and inference turnarounds; more often than not, the data lake reality is that of a *data swamp*, a so-called data dumping ground with data curation tasks taking a back seat. In data swamps, the data quality ends up being suspect and only gets worse over time.

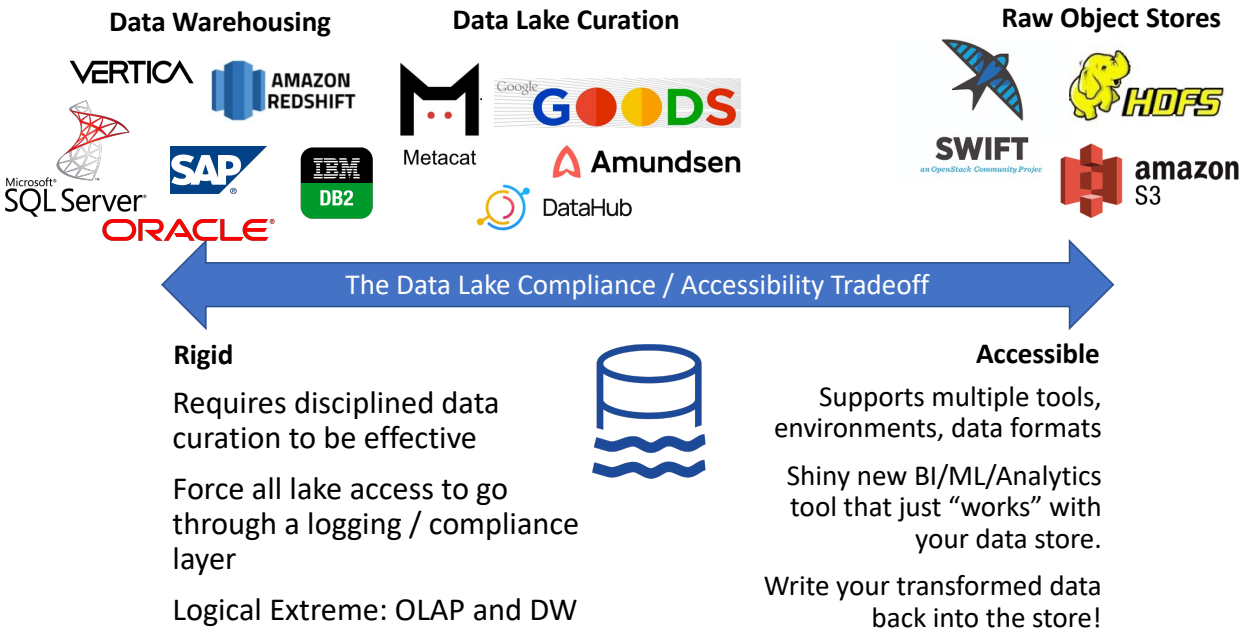


Figure 2.1: The Compliance/Accessibility Trade-off in Data Lakes

This situation is in part due to a rigidity/accessibility trade-off (Figure 2.1) that encompasses database systems, data lakes, and other data operations. Traditional relational databases and data warehouses figure on the left, with rigid schema acting as a compliance layer which forces data to be well-structured and enough metadata (in the form of logging/access control/schema and constraint management) to discern lineage information [62].

On the other hand, Big data architectures and data lakes are an evolving market with

many emerging technologies that are vying for market share. In order to build a customer base and ease customer adoption, there will be a natural tendency to skew right in Figure2.1 (towards less rigid and accessible solutions that fit into the existing data systems). The resulting provenance crisis in data lakes is a significant problem that impedes the original data lake vision. As we will see in Section 2.2, existing solutions are insufficient to discern lineage in such data swamps in a retrospective manner and require new tools and techniques to aid in the organization of said swamps. This thesis will focus on inferring the lineage of these data artifacts in a retrospective manner; using tools and techniques adapted and modified from various related fields as follows:

2.2 Related Work

Work related to this dissertation can be broadly classified into the following themes:

2.2.1 Research in Data Lineage

Ikeda*et al.*have summarized traditional lineage research in a survey [50], while providing a taxonomy of lineage systems for relational and probabilistic databases. Cui*et al.* [20] proposed a taxonomy of transformations for the purposes of lineage tracing. The categorization is based on the number of input and output tuples involved in transformation; dispatchers (1:N), aggregators(N:1), and black-box transformations; the paper provides the formalism and tracing procedures for each of these transformation types assuming that the lineage tracing procedure is capturing both the input and output tuples as they are being processed during query execution. A more recent survey by Herschel*et al.* [46] provides an application-centric overview of all of the workflow lineage solutions proposed in the domains of science, business, data analytics, and general programming.

2.2.2 Online Lineage Capture / Curation / Version Control

: Systems such as OrpheusDB [48] and ProvDB [60] provide (git-style) version control for relational databases, they have substantial barriers to adoption—and are unlikely to be used in unstructured, ad-hoc data exploration settings like in data lakes. For data lakes, lineage capture systems both from the industry [1, 7, 11, 34, 36, 63] and academia [4, 11, 15, 48, 97] either require access to code or explicit API calls to register and link artifacts in a curated fashion, which requires compliance and human effort to document lineage while artifacts are being created and offer no help in a retrospective analysis of said artifacts. At the time of writing, the Lakehouse architecture described in [7] is gaining traction among the plethora of solutions being proposed by companies in managing data lake complexity going forward.

2.2.3 Data Discovery

: ReConnect [2] and Rediscover [3] attempt to discover the relationship for a given dataset pair. These papers define a space of relevant relationships, generate the conditions for each relationship based on row and column statistics, and then suggest a relationship for a given dataset pair by examining the conditions. ReConnect relies on user feedback to validate candidate relationships, while ReDiscover uses a machine learning model to predict the relationship. However, both ReConnect and ReDiscover only consider a limited relationship set, i.e., containment, augmentation, complementation, template, and incompatible, and cannot handle any mix of them. Systems such as Aurum [15] use sampling and estimation techniques to build, maintain, and query datasets in an Enterprise Knowledge Graph (EKG). The system uses similarity metrics like Jaccard distance and containment to find tables with columns that are most similar to a source table; JOISE [98] speeds up searches for top-k joinable tables in data lakes using a novel set overlap similarity search system. Aurum and JOISE assist in query-centric similarity search; neither system makes any inferences about lineage.

2.2.4 Query Synthesis

A long line of work in Query Reverse Engineering (QRE) and Query By Example (QBE) [21, 29, 52, 80, 84, 93] focuses on reverse-engineering SQL queries that are used to transform one artifact to another. This approach has several issues that make it challenging to apply in our problem context. Firstly, most of this work constrains the inferred SQL queries to some subset of Select-Project-Join-Aggregate (SPJA) and assumes a perfect solution exists in that search space. Second, the input and output artifacts are explicitly labeled, usually within the context of a normalized database schema with established join paths via PK/FK constraints. We argue that artifacts generated by modern data analytics systems do not satisfy either of these conditions. Besides SQL queries, manual edits, scripts, and programs can also be involved in data curation, transformation, and feature engineering. We also note that inferring a concise SQL query itself is a computationally hard problem [93]. We may complicate the problem further if we try to formulate the delta between two artifacts as SQL queries. Finally, current work focuses on a single pair of datasets. Instead, we aim to summarize the relationship among a collection of datasets, which poses additional challenges since it involves all possible dataset pairs.

2.2.5 Document Provenance Systems

A separate line of work explores retrospective lineage using similarity scores in information retrieval for plain text documents. Deolalikar *et al.* [24] demonstrate a system that combines content analysis (cosine similarity over TF-IDF vectors) with filesystem timestamps to generate a list of documents, ordered by the time that is most relevant to a given document. Similarity [22, 23] uses TF-IDF over-extracted named entities to discover provenance using semantic similarity over a set of documents. These approaches are fine-tuned for documents and are unlikely to yield good results over large data tables, as shown by [86], which uses a complex scheme to recover the semantic meaning of tables present in the web corpus and

utilize the semantic keywords and relevant information to power a table search system. In our context, we deal with multiple tables with varying degrees of semantic information or named entities (e.g., a time-series table of sensor values may have little semantic information or named entities that can be extracted from the table). Since the tables are assumed to have some derivation or transformation relationship to one another, they are likely to saturate these semantic information signals to the extent that makes it difficult to assess the fine-grained differences between versions of these individual tables. Table similarity is discussed in [64], which uses a combined Schema and Data similarity metric to measure the distance between two web tables and is an approach that is most closely related to our proposal.

2.2.6 Dataframe Benchmarking

Benchmarking and database workload generation systems have a long and storied history in academia and industry [38].

The TPC benchmark suite [19] is the most popular benchmark suite for database systems and encompasses transactional processing, data warehousing, data integration, big-data systems, and others. Vogelsgesang et al. [88] have empirically shown that typical data visualization workloads originating from BI application dashboards such as Tableau consist of complex nested sub-queries, which are very different from typical TPC workloads.

The rise in popularity of the Dataframe-based data analysis workflows has resulted in a large number of systems that implement the API while making improvements to the memory model [32, 87], leverage parallel and distributed computing [54, 61, 75], heterogeneous hardware [81] and usability improvements [55]. However, the lack of a standardized workload or performance benchmark for dataframe systems results in primarily anecdotal reports of performance improvements. One prominent example is the H20.ai benchmark website [41], which compares join and groupby query performance against dataframes of various sizes

(0.5, 5, and 50GB at the time of writing); the website is designed to run periodically against the latest versions of popular "database-like" systems in an automated fashion. Among published work, the AFrame system [78] used a version of the scalable Wisconsin benchmark [38], consisting of mostly integer columns and a few string patterns with varying cardinality to show performance improvements. Sanzu [89] is a big-data benchmark suite designed to test data science environments with a wide variety of operations such as wrangling and machine learning but consists of static queries and datasets. Similarly, Böhm et al. [14] conducts a deep dive into dask's runtime and scheduler system to find performance bottlenecks; the benchmark suite is again a set of static tasks/operations performed against static datasets.

YCSB [18], is a popular NoSQL benchmark system that is aimed at cloud-serving systems that support a limited set of queries (insert/update/read/scan), with the option of scaling and mixing query types in a workload. YCSB's popularity can be attributed to its extensible client-server model – a new key-value store or database client can be written using the YCSB API and run existing workload sets to compare against other YCSB clients. We have sought to emulate this design pattern with FuzzyData.

CHAPTER 3

THE RELIC SYSTEM

3.1 Introduction

The emergence of collaborative data science and machine learning (ML) platforms has made it possible for data scientists and analysts to manipulate, process, and analyze modest to large amounts of tabular data in an exploratory or ad-hoc manner. This data analysis *workflow* typically involves dealing with raw data from one or more sources, followed by multiple stages of data preparation, spanning ingestion, cleaning, transformation/analysis, and export. These stages are often performed by a range of tools, spanning data analysis scripts, and/or computational notebooks, and/or spreadsheet systems. Consider a typical ML preparation workflow in Figure 3.1: here, a dataset is sourced from the Internet, wrangled, and cleaned for use as input to an ML training algorithm. Depending on the trained model's accuracy, the data scientist may go back to the original dataset and perform additional wrangling or feature engineering, producing new artifacts for training.

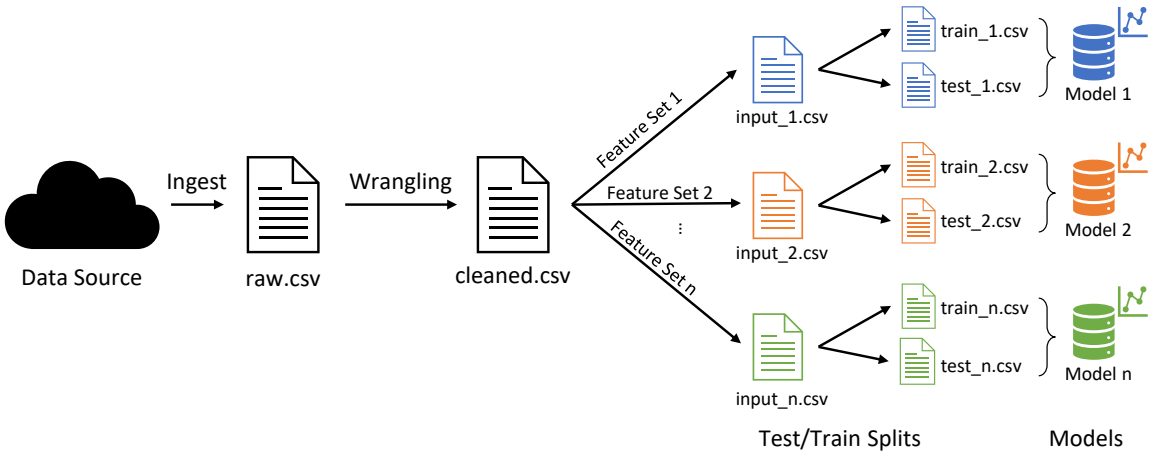
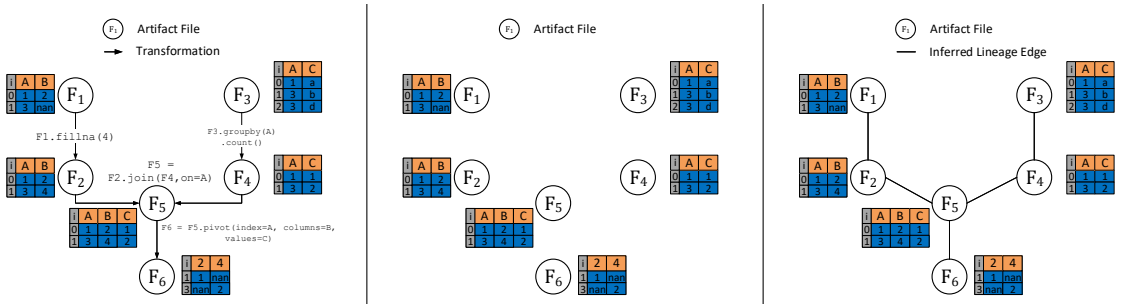


Figure 3.1: Typical data Analysis/ML prep workflow, with each stage transforming the input and producing artifact file(s).



(a) A data analysis workflow represented as a graph (Ground Truth).

(b) Set of artifacts without any derivation information (Input).

(c) Lineage graph reconstructed from artifact contents (Output).

Figure 3.2: Motivating Example for the Lineage Inference Problem.

Intermediate results between the stages of data analysis workflows are often stored as *artifacts* in a file system that is shared across the organization, a so-called *data lake* [43, 79, 83]. However, the processes used to generate them, and the lineage information (i.e., the source artifacts that a given artifact is derived from), are rarely recorded. As such, the lack of this lineage makes it difficult to support *(i)* reproducibility and debugging, e.g., *how did I derive this artifact that led to high accuracy on my ML task* or *which downstream artifacts were derived from this artifact*, *(ii)* maintenance, e.g., *which artifacts represent "abandoned" trials during experimentation—and therefore can be discarded*, *(iii)* explainability, e.g., *can I make sense of the experimentation workflow that led to these artifacts at a high level*, and *(iv)* discovery, e.g., *can I find a pre-cleaned version of this dataset.* Thus, the original driving vision of data lakes, with the goal of making it seamless to discover, make sense of, and integrate datasets across an organization, is hard to accomplish without lineage [73, 79].

Recognizing the fact that lineage is essential to make data lakes useful, a number of industry projects have emerged recently, targeted at enabling users to explicitly record lineage metadata during data analysis workflows, including Amundsen [34], Marquez [36], Apache Atlas [33], Egeria [35], Uber's DataBook [85], LinkedIn's DataHub [26], Netflix's Metacat [63], and AirBnB's DataPortal [1]. There are also a number of academic projects in this space, enabling users to provide and make use of lineage metadata [4, 12, 30, 48, 97].

Despite all of these projects and initiatives, data scientists in most organizations are reluctant to explicitly record lineage information along with their artifacts during their workflows due to the additional overhead it introduces into the rapid experimentation typical in data science. The culture of disciplined provenance management and version control is not as widespread among data scientists as it is with software development teams [12, 83]. Data scientists may end up recording the lineage metadata only for their "final" artifacts, as opposed to recording it every step along the way. Moreover, even if organizations do eventually adopt one of the lineage-recording tools described previously, all of the organizations' artifacts until that point would still lack lineage information, requiring solutions that can help "ingest" and "backfill" these past artifacts into these lineage-recording tools. Another potential solution is instrument existing tools such that they automatically capture lineage [16, 40, 44, 46, 68, 90]; however, these approaches limit the flexibility in tooling available to data scientists, who often prefer using tools that are hard to instrument—and the problem of "backfilling" past artifacts still exists with this approach.

Consider the worst-case scenario—one where code and documentation for a data analysis workflow are absent, wherein a user has the daunting task of understanding the purpose and context for several artifact files that contain tabular data, say from a data dump or backup. The file system metadata of these artifact files (such as file creation/modification time) may be absent or unreliable[1], in which case we do not have any reliable temporal ordering of these artifact files; indeed, even with this information, it is hard to determine which artifact files were the sources for a certain one, given the inherent non-linearity in data science experimentation.

In this work, we introduce the problem of *retrospective lineage inference*, with the goal of recovering the lineage of artifact files (or simply artifacts) generated during a data wrangling or analysis workflow. We rely solely on the content of said artifact files and determine

1. This is typical in case of an ad-hoc copy or improperly configured file backup operation.

their relationships with one another via an *inferred lineage graph*. As an example, consider Figure 3.2, which represents a cleaning and integration workflow consisting of 6 artifacts. Figure 3.2a illustrates the ground-truth lineage graph, which represents the original workflow, but is unavailable as a retrospective reference. Each edge of this graph has been labeled with the exact operation used to derive a target artifact from its source. The operations shown are from Python/Pandas but could be in other languages, such as SQL. Figure 3.2b depicts the input to the retrospective lineage inference problem: the artifacts themselves, without any lineage information or order. To keep the problem a bit more tractable, our focus is on just identifying the edges between artifacts, but not the exact operation (such as `f1.fillna(4)`). This level of source-target relationships is similar to the types of relationships recorded explicitly by users in industry lineage metadata capture solutions and is therefore compatible with them. Due to this simplification, our goal is both *more ambitious* and *shallower* than prior work on query reverse engineering (QRE) [10, 29, 39, 52]: we are inferring multiple relationships between artifacts at the same time (unlike QRE, which only does so for a pair of artifacts), while not inferring the exact operation, just source-target relationships (unlike QRE, which attempts to determine the exact operation). If needed, we could perform QRE on each edge after retrospective lineage inference is complete if more fine-grained derivation relationships are needed. We compare against QRE and other related work in more detail in Section 2.2. We also do not attempt to infer the directionality of the operations (note that the edges in Figure 3.2c are un-directed). The inferred lineage graph is un-directed because it may be impossible to infer the directionality of certain operations (columns/rows added vs. removed). If file creation times are present, the directionality is easy to determine; therefore, in our formulation, we focus on the undirected version.

Despite these simplifications, retrospective lineage inference is a rather challenging problem: we have n artifact files, no input or output labels, no fixed schema, and therefore a space of $\binom{n}{2}$ relationships (and more, if we consider joins) that need to be assessed to con-

struct the lineage graph, essentially making our problem one of query reconstruction at scale. The second challenge emerges from the wide variety of operations that can be performed on tabular data in a typical workflow. The set of operations not only includes traditional relational operations but may include spreadsheet-style editing of individual cells, and arbitrary UDFs, in addition to re-ordering of tuples and columns. Furthermore, there may be no strict schema, the files may be indexed (in the dataframe sense) arbitrarily or not have indexes at all, and there may be no known relationships between the data present in these artifact files.

In this chapter, we present RELIC, our first solution for retrospective lineage inference. RELIC infers the lineage of artifacts generated in data analysis workflows retrospectively, without access to the underlying scripts or instructions that generated said artifacts. The intuition underlying RELIC is straightforward: artifacts that exhibit a higher similarity to one another are more likely to have been derived from one another, in line with Occam's razor. This intuition follows from prior work exploring data versioning in such workflows [48, 93], as well as work exploring retrospective lineage in other settings, such as software lineage inference [51]. However, the implementation of this similarity differs on the type of operations employed. We consider scenarios that involve the manipulation of tabular data using operations that encompass typical data integration, cleaning, analysis, and feature engineering workloads—including operations that preserve one-to-one relationships between rows and columns across artifacts (such as filters, projections, or pointwise edits), and those that don't (such as group-bys or pivots). RELIC will help with dataset curation, finding authoritative versions of a dataset, or forensic analysis of data recovered from a backup or data dump. Our contributions are as follows:

- We formalize the problem of *retrospective lineage inference*, wherein we infer the lineage graph of artifact generated as a result of data analysis operations. (Section 3.2).
- We introduce a novel characterization of artifact transformations based on the preservation of column and row mappings before and after transformations. This is then used

to design similarity metrics and operation-specific detectors to infer workflow lineage in a systematic manner as part of our solution, RELIC (Section 3.3).

- Being a novel problem with no standardized benchmarks or workloads, we design and evaluate our technique on both real and synthetic workloads (Section 3.4), discuss related work (Section 2.2) before wrapping up with the key takeaways and directions for future work (Section 3.5).

3.2 Problem Definition

We are given a set of tabular artifact files with no additional metadata, including versioning or temporal ordering information available to us. Our task is to infer the underlying lineage graph that describes the evolution of the files in the workflow.

3.2.1 Notation

Consider a set of n tabular artifact files $\mathcal{F} = \{F_1, \ldots, F_n\}$. These files are assumed to belong to a data science workflow, represented as a ground truth lineage graph $\mathcal{W} = (\mathcal{F}, E)$, where each edge $(F_u, F_v) \in E$ is labeled with some operation $o \in \mathcal{O}$, such that $o(F_u) = F_v$. . $\mathcal{O}$ represents the set of all possible transformation operations in the setting under consideration (such as pandas, R or SQL transforms). A special case is *join* ($\bowtie \in \mathcal{O}$), which associates three artifacts together as $F_r \bowtie F_s = F_t$. This allows for two edges $(F_r, F_t), (F_s, F_t) \in E$ to be part of the same operation.

Within each file F_i, the tabular notation used in this thesis is illustrated in Table 3.1, consisting of n row ids $R_{F_i} = \{r_1 \ldots r_n\}$, m column labels $C_{F_i} = \{c_1, \ldots, c_m\}$ and $m \times n$ individual values in tabular form $V_{F_i} = \left\{v_{(c_1,r_1)}, \ldots, v_{(c_m,r_n)}\right\}$.

Finally, $\mathcal{O}$, the set of operations used to transform these artifacts are listed in Table 3.2. From our analysis of typical data analysis workflows in the wild (Section 3.4.1), we believe

Row-Ids (R_{F_i})	Column Labels (C_{F_i}) c_1	$\dots$	c_m
r_1	$v_{(c_1,r_1)}$	$\dots$	$v_{(c_m,r_1)}$
$\vdots$	$\vdots$	$\ddots$	$\vdots$
r_n	$v_{(c_1,r_n)}$	$\dots$	$v_{(c_m,r_n)}$

Table 3.1: Notation used to describe an artifact file F_i. The set of cell-values in this table is denoted as $V_{F_i} = \{v_{(c_1,r_1)} \dots v_{(c_m,r_n)}\}$.

Operation	**Pandas Examples**
Point Edits	`df.fillna('0')`
Row Selection	`df.iloc , df.loc, df.sample`
Row Concatenation	`df3 = pd.concat([df1,df2])`
Column Modifications	`df[['column1', 'column2']], df.drop(columns=['column1'])`
Derived Column	`df['column2'] = df['column2'].apply(func)`
Joins	`df3 = df1.merge(df2, on='keycol')`
GroupBy/Aggregation	`df.groupby(['column1' ...]).sum()`
Pivot	`df.pivot(index='col1', columns='col2', values='col3')`

Table 3.2: Set of operations $\mathcal{O}$ under consideration, and example Pandas functions on a dataframe (`df`).

this set covers a wide range of data analysis/transformation operations. While our technique is optimized for these operations, Section 3.5 describes ways to extend this set to cover additional operation types.

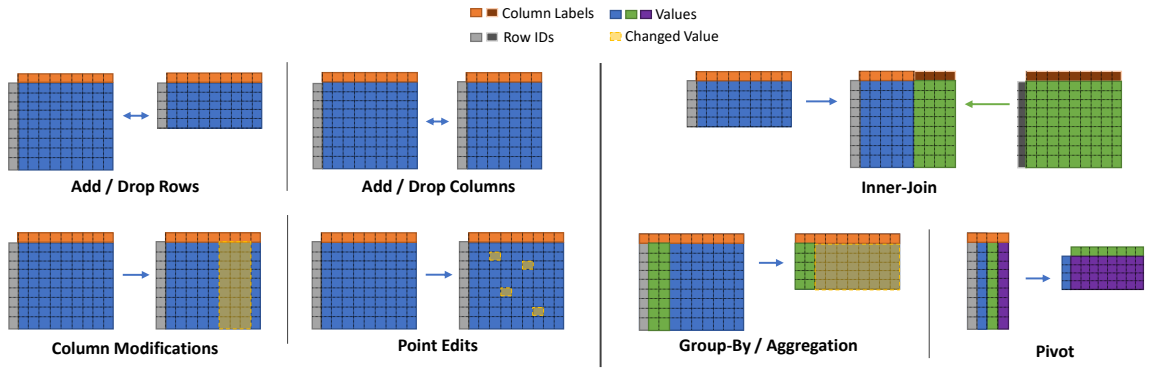
3.2.2 The Lineage Inference Task

Our task is formalized as follows:

Problem 1 (Lineage Inference)**.** *Given a set of artifact files $\mathcal{F}$ and the set of all possible operations $\mathcal{O}$, infer the un-directed workflow lineage graph edges E' that most closely resembles the ground-truth graph represented using lineage edges E, as measured by the precision, recall and F1 score.*

The ground truth graph is assumed to be a connected, *directed acyclic graph* (DAG), commonly used to represent cycle-free[2] data workflows. As a result, the ground truth graph must have a minimum of $(n-1)$ edges to be weakly connected and can have up to $\frac{n(n-1)}{2}$ edges to remain acyclic. In our work, we choose to terminate our algorithm as soon as the graph is connected with $(n-1)$ edges. Although the ground-truth lineage graph of the workflow is directed, the directionality of certain operations may be impossible to disambiguate (e.g., a dropped column can be viewed as adding a new derived column in the other direction). Note that the ground truth may also contain cycles, but for simplicity, we target a DAG at the cost of lower precision. Section 3.5 outlines how we can extend our framework to infer directionality, operation type, and operation parameters as part of a larger lineage inference task.

3.2.3 Types of Transformations



(a) Point Preserving Transformations (PPT)

(b) Non- Point Preserving Transformations (NPPT)

Figure 3.3: Types of transformations and how row-ids, columns labels, and values change between these transformations.

We are primarily interested in recovering the lineage of artifacts produced during data science, including data analysis, machine learning preparation, and data integration work-

2. In a retrospective setting, we are viewing a point-in-time snapshot of artifact files, and we cannot observe any new mutations to an existing artifact, so we assume a cycle-free graph.

loads. Such workflows commonly contain spreadsheet-style editing operations performed at the cell level or more complex operations such as *pivots*. The focus in this thesis is Pandas, but our techniques are generalizable to other frameworks. Our key insight in this work is the categorization of transformations into two types: *point-preserving*, and *non-point preserving*. The categorization helps us in inferring transformation edges while gradually building the lineage graph.

An $o(F_u) = F_v$ transformation results in a mapping of each of the source rows in R_{F_u} to either 0, 1, or N rows in R_{F_v}. As an example, the column add/drop transformation has a 1:1 mapping of rows from source to destination. In contrast, the group-by operation, which aggregates multiple rows in the source to a single row in the target, results in an N:1 mapping of rows from source to destination. This can be used as a basis for classifying different types of transformations.

Point-Preserving Transformations: We define a *point-preserving transformation* (PPT) as one which has a 1:1, 1:0 or 0:1 row mapping between source and destination artifacts. This means that a destination row either exists as a modified version of an existing row in the source (1:1), is filtered out (1:0), or is a new row(0:1). For example, in Figure 3.3a, all the operations shown have a row-mapping between the source artifact and destination artifact of either 1:0 or 1:1. PPTs provide a positional "anchor" to map individual cell values across the transformation, allowing us to reason about the similarities between two artifacts in a fine-grained, cell-level manner. Note that column additions and deletions are designated as PPTs in our classification, and thus artifacts transformed via a PPT do not need to have exactly matching schema.

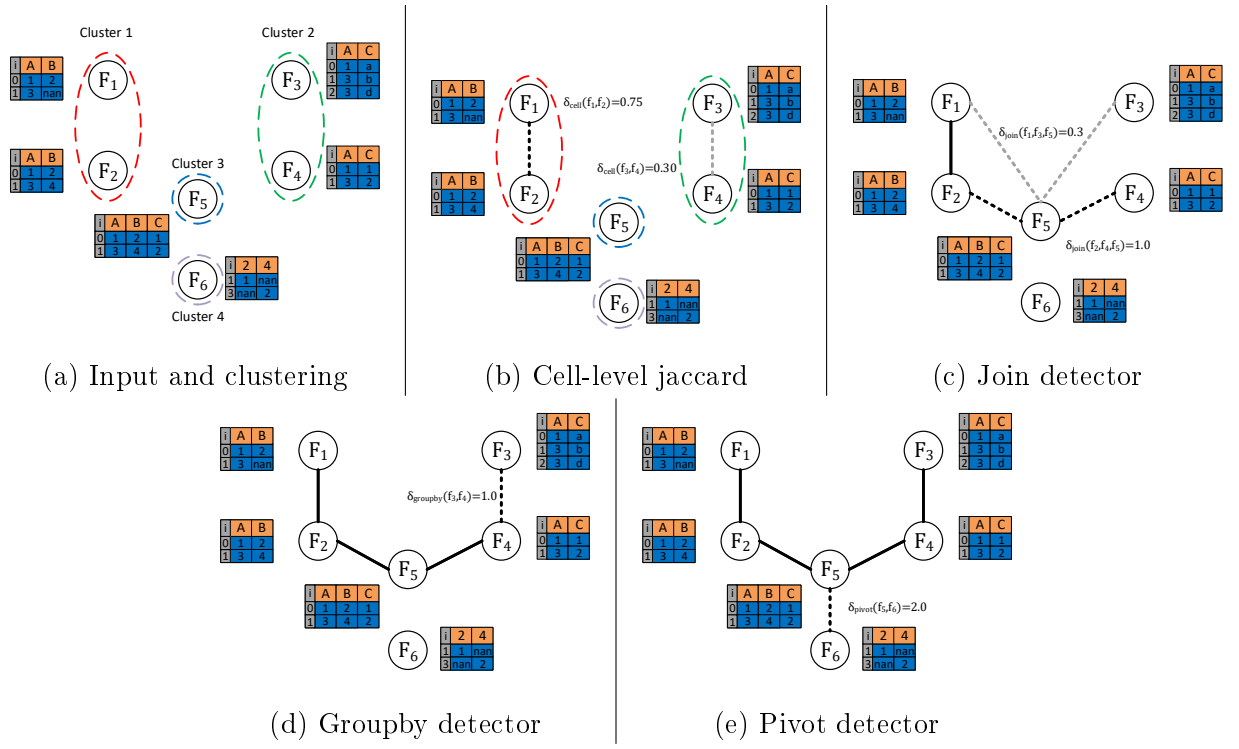
Non-Point-Preserving Transformations: Transformations that have an N:1, 1:N, or N:N row mapping between source and destination artifacts are *non-point preserving transformations* (NPPTs). This means that a destination row represents either a collection of rows from the source (N:1, in case of a groupby/aggregation), or a single row can be replicated (1:N, in

case of a join), or there may be no direct mapping between records (N:N such as pivots or melts). NPPTs are harder to characterize as there is no obvious way to map values before and after a transformation, which makes inferring the lineage of such operations difficult. Figure 3.3b illustrates how the mapping of values across the transformation can vary. Note that joins are a special case; in the case of a PK-FK join, one of the source artifacts (the one with the primary key) may have a 1:1 row mapping, while the other artifact (with the foreign key) may have a 1:N mapping.

3.3 Lineage Inference Technique

At the heart of the lineage inference problem is an edge selection problem from the space of $\binom{\mathcal{F}}{2}$. Our methodology seeks to find the artifacts that are most similar to each other first. We start by inferring minor point-preserving edits among artifacts that share the same schema and then move on to identify more elaborate non-point-preserving transformations. We break down the lineage inference technique into the following steps:

- Initialize and pre-process the input artifacts; identify row and column mappings for all artifacts, and cluster the artifacts by schema. (Section 3.3.1)
- Within each cluster, compute the pair-wise similarity of artifacts and place edges on the most similar artifacts. (Section 3.3.3)
- Between clusters, find pairs of artifacts that may be lineage-linked using operation-specific detectors and use them to bridge and merge disconnected artifacts. (Section 3.3.4)
- Repeat the previous step until the generated lineage graph is a single, connected cluster consisting of all artifacts, or the remaining edges are below some similarity/detector threshold for consideration.



(a) Input and clustering (b) Cell-level jaccard (c) Join detector

(d) Groupby detector (e) Pivot detector

Figure 3.4: Running example for our lineage inference technique, showing edges added to the inferred lineage graph

.

3.3.1 Initialization

To compare the cell values of two artifacts, we must first map the individual cell values across artifacts using their row ids and column labels. Artifacts that have consistent row ids and column labels can have their cell values compared easily. On the other hand, if rows are re-ordered, or columns have been renamed or replaced, we will have to rely on more complex mapping techniques as follows:

Recovering Column Mappings

Identifying column mappings across tables is an instance of the so-called schema matching problem, for which many techniques have been summarized by Rahm *et al.* [70]. These techniques typically exploit information from the schema (e.g., column name and column type) or the content of these columns. In this work, we implement schema matching solely

based on the column name and column type. That is, when both the column name and column type are the same, we claim that these two columns are matched, and we have found this assumption to be sufficient for our setting.

Recovering Row Mappings

Pandas dataframes contain an explicit row-index by default and is typically the first column when materialized to CSV. Many NPPTs, however, replace the existing row-index with other values, which then requires additional effort to identify. One natural method is to examine the uniqueness of column combinations (UCCs)—if the ratio between the value set cardinality and the total number of values in the column is above some threshold, e.g., 0.9, the column combination can be used as a primary key [45]. In the absence of an appropriate primary key, we may have to rely on more complex record linkage techniques [17] which may be computationally expensive. In our work, we assume the first column of an artifact to be the index of the artifact (as is typically the case with pandas dataframes, emperically shown in [91]), and if the row-indices of two artifacts do not share at least 50% of their values, we re-index the artifacts to align them. In case there are no pairs of columns that serve as an appropriate index, the artifact cells are compared in their physical, materialized order, which we have also found to be useful in our setting.

Clustering

Once the row and column mappings are computed and verified among the set of input artifacts, RELIC clusters the artifacts by the exact column set, thereby restricting the initial set of edges inferred to be among those artifacts that have the same set of column labels. The clustering procedure is as simple as constructing a map from sets of column labels to artifacts. This can be done by reading the column labels for each artifact and does not require scanning any of the data values. As part of our running example, Figure 3.4a shows the

artifacts grouped into four clusters with column sets of $\{\{A, B\}, \{A, C\}, \{A, B, C\}, \{2, 4\}\}$.

3.3.2 Building the Lineage Graph

Once we have all the artifacts ready for pairwise comparison, we can start the inference procedure (Algorithm 1). Starting from an edge-less graph consisting of all the vertices, we begin by computing the pairwise similarity of all artifacts within a cluster. Since all the artifacts within a cluster share the same schema, they may be related through point-preserving transformations, and we use the techniques laid out in Section 3.3.3 to infer their lineage. Once we have exhausted the search for edges within clusters, we can look for the remaining edges between disconnected components. If there are no point-preserving edges between artifacts above some threshold (say, ϵ_{intra_cell} within clusters and ϵ_{inter_cell} between clusters), we then proceed to check for non-point preserving operations as outlined in Section 3.3.4. This procedure is repeated until the entire graph is connected or there are no remaining edges with similarity scores above some threshold.

Algorithm 1: InferLineage($\mathcal{F}$)

Input: Set of artifact files $\mathcal{F}$
Output: $\mathcal{W} = (\mathcal{F}, E)$ with lineage edges added

Initialize $\mathcal{W} \leftarrow (\mathcal{F}, \phi)$;
$C \leftarrow cluster(\mathcal{F})$;
foreach $C_i \in C$ **do**
  $\mathcal{W} \leftarrow \mathcal{W} \cup$ AddLineageEdges($\mathcal{W}[C_i], \delta_{cell}, \epsilon_{intra\_cell}$)
$\mathcal{W} \leftarrow \mathcal{W} \cup$ AddLineageEdges($\mathcal{W}, \delta_{join}, \epsilon_{join}$);
$\mathcal{W} \leftarrow \mathcal{W} \cup$ AddLineageEdges($\mathcal{W}, \delta_{cell}, \epsilon_{inter\_cell}$);
$\mathcal{W} \leftarrow \mathcal{W} \cup$ AddLineageEdges($\mathcal{W}, \delta_{contain}, \epsilon_{contain}$);
$\mathcal{W} \leftarrow \mathcal{W} \cup$ AddLineageEdges($\mathcal{W}, \delta_{groupby}, \epsilon_{groupby}$);
$\mathcal{W} \leftarrow \mathcal{W} \cup$ AddLineageEdges($\mathcal{W}, \delta_{pivot}, \epsilon_{pivot}$);
return $\mathcal{W}$

Algorithm 2:
AddLineageEdges($\mathcal{W}, \delta_{detector}, \epsilon_{detector}$)

Input: Partially connected graph $\mathcal{W} = (\mathcal{F}, E)$,
Scoring / Detector Function $\delta_{detector}$,
Scoring / Detector Threshold $\epsilon_{detector}$
Output: $\mathcal{W}$ with detector lineage edges added

while $\mathcal{W}$ *is not connected* **do**
  $K \leftarrow components(\mathcal{W})$;
  if $\delta_{detector} = \delta_{join}$ **then**
    $f_x, f_y, f_z \leftarrow \underset{\substack{f_x \in K_i, f_y \in K_j, f_z \in K_k \\ x \neq y \neq z}}{\arg\max} \left\{ \delta_{join}(f_x, f_y, f_z) \right\}$;
    if $\delta_{join}(f_x, f_y, f_z) \geq \epsilon_{join}$ **then**
      $\bar{E} \leftarrow (f_x, f_z), (f_y, f_z)$;
  else
    $f_x, f_y \leftarrow \underset{\substack{f_x \in K_i, f_y \in K_j \\ x \neq y}}{\arg\max} \left\{ \delta_{detector}(f_x, f_y) \right\}$;
    if $\delta_{detector}(f_x, f_y) \geq \epsilon_{detector}$ **then**
      $\bar{E} \leftarrow (f_x, f_y)$;
  if $\bar{E} = \emptyset$ **then**
    break;
  else
    $\mathcal{W} \leftarrow (\mathcal{F}, E \cup \bar{E})$;
return $\mathcal{W}$;

3.3.3 *Inferring Point Preserving Transformations*

We can represent the similarity between two artifact files F_u and F_v by looking at the cells that are modified between them. Each cell $v_{(c_i,r_j)}$ can be represented as the column label, row id and value triple $(c_i, r_j, v_{i,j})$. Two cells from two different artifacts are the same if and only if they share the same column label, row id, and value.

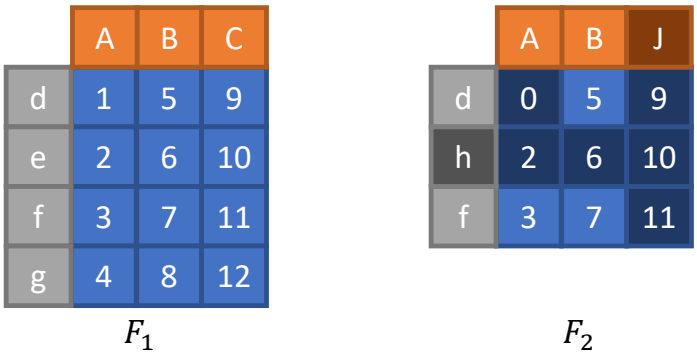


Figure 3.5: Artifacts F_1 and F_2 compared at the cell-level. Based on our definitions, only the cell values 3, 5 and 7 are in common between these two artifacts.

Cell-Level Jaccard Similarity

The cell-level Jaccard similarity is defined as the Jaccard similarity between set of cell values V_{F_i} and V_{F_j}. Formally, we can compute $\delta_{cell}(F_i, F_j)$ as in Equation 3.3.1, where $|V_{F_i} \cap V_{F_j}|$ is the number of common cells between F_i and F_j and $|V_{F_i} \cup V_{F_j}|$ is the number of the union of cells in F_i and F_j.

$$\delta_{cell}(F_i, F_j) = \frac{|V_{F_i} \cap V_{F_j}|}{|V_{F_i} \cup V_{F_j}|} \tag{3.3.1}$$

For the artifacts in Figure 3.5, the number of cells in common is 3, and the union is 18, since we consider the cells in row h and Column J in F_2 to be distinct from the values in row e and column C in F_2. Thus, $\delta_{cell}(F_1, F_2) = \frac{3}{18} \simeq 0.167$.

Cell-Level Jaccard Containment Similarity

Certain operations (such as row sampling, concatenation, drop columns etc.) exhibit low cell-level jaccard similarity, especially when one artifact is similar to a very small fraction of the other. Consider the case of a sample operation that takes a 1 percent sample from a larger artifact. Such an operation will only register a 1 percent similarity score using the cell-level Jaccard similarity metric (Equation 3.3.1). However, in these cases each artifact is fully contained in the other. We have found it to be useful to incorporate this reasoning in our technique, by employing a containment score, described below (Equation 3.3.2):

$$\delta_{contain}(F_i, F_j) = \frac{|V_{F_i} \cap V_{F_j}|}{\min(|V_{F_i}|, |V_{F_j}|)} \tag{3.3.2}$$

$\delta_{contain}$ thus has a value between 0 (when there are no cells in common) and 1 (one of the artifacts is fully contained in the other). When invoked, we find a pair of artifacts that exhibits the highest containment score and add an edge connecting them to the inferred lineage graph. For the artifacts in Figure 3.5, the number of cells in common is 3, and the size of the smallest artifact is 9. Thus, $\delta_{contain}(F_1, F_2) = \frac{3}{9} \simeq 0.333$.

$\delta_{contain} \geq \delta_{cell}$ by definition, and hence $\delta_{contain}$ can produce a large number of false positive edges, especially if there is an artifact that has its values contained in many other artifacts. We hence always invoke the containment score only after we have exhausted all δ_{cell} edges in the workflow. Computing δ_{cell} and $\delta_{contain}$ are straight-forward. Given two artifacts F_i and F_j, we first build a hash table with all cells in F_i, and then probe cells in F_j using the built hash table. Thus, the time complexity for computing the cell-level delta is $O(|F_i| + |F_j|)$, where $|F_i|$ is the number of cells in F_i, i.e., $|V_{F_i}|$.

3.3.4 Inferring Non-Point-Preserving Transformations

Recall that NPPTs are transformations that contain 1:N, N:1 or N:N row mappings. Such transformations are difficult to infer via PPT techniques. Our initial approach to infer these transformation edges was to use a generalized similarity score that relied on value containment of all columns between the source and destination artifacts, akin to a dataset similarity search implemented in [30]. This approach was promising for some operations like groupbys, but it did not perform adequately in case of joins and pivots. We, therefore, designed operation-specific detectors. These detectors look for value containment across cell boundaries in specific patterns indicative of certain operations. We shall now describe the design of three such detectors, namely the *join*, *groupby* and *pivot* detectors.

The Join Detector

The join detector allows us to locate and infer join operations in the workflow, wherein two source artifacts F_r and F_s are joined on a common key to produce a destination artifact F_t[3].

A natural join can be viewed as a column concatenation between different artifacts that share a common key. We design our detector around finding value containment from each side of the join, in a manner that exhibits coherent group containment with the join key, similar to a technique used in [52]. Unlike the other detectors in RELIC, the join detector works on triples of artifacts rather than pairs (Line 4 in Algorithm 2); Given some triple (F_i, F_j, F_k), the join detector assigns a score between 0 and 1 that indicates the likelihood that two of the three artifacts were naturally joined to create the third, as follows:

1. We first determine the two source artifacts and the destination artifact of a possible join by checking the column labels. We assign the source labels F_r, F_s and destination

3. We concentrate on natural joins between two source artifacts F_r and F_s to produce a destination artifact F_t. We additionally assume that the join be performed on a single key column that is common in the source artifacts and present in the destination artifact. These assumptions were made to reduce the search space for our detector. They can be relaxed, but the runtime performance and accuracy will be affected.

label F_t to the artifacts F_i, F_j, F_k such that the following conditions are satisfied:

$$\left(C_{F_r} \cup C_{F_s}\right) \supseteq C_{F_t} \tag{3.3.3a}$$

$$C_{rt} = \left(C_{F_r} \cap C_{F_t}\right) - C_{F_s} \neq \emptyset \tag{3.3.3b}$$

$$C_{st} = \left(C_{F_s} \cap C_{F_t}\right) - C_{F_r} \neq \emptyset \tag{3.3.3c}$$

These conditions can be used to figure out the source and target assignments for a possible natural join. The target artifact F_t must not get columns from other sources (Equation 3.3.3a) and we verify that there are non-null column contributions (C_{rt} and C_{st}) from either source artifact (Equations 3.3.3b, 3.3.3c). The order of the source artifacts does not matter for the detector, so F_s and F_r are interchangeable. No assignment of source and target artifacts satisfies these conditions, then we stop and assign $\delta_{join}(F_i, F_j, F_k) = 0$, and Algorithm 2 can move on to the next triple to be evaluated.

2. Compute the join score for this triple $\delta_{join}(F_r, F_s, F_t)$ by solving the following optimization problem (Equation 3.3.4):

$$\underset{join}{\delta}(F_r, F_s, F_t) = \arg\max_{k \in K} \left\{ \underset{contain}{\Delta}(SC_r, TC_r) \times \underset{contain}{\Delta}(SC_s, TC_s) \right\} \tag{3.3.4a}$$

$$\text{where } K = (C_{F_r} \cap C_{F_s} \cap C_{F_t})$$

$$J = \left\{ \pi_k V_{F_r} \cap \pi_k V_{F_s} \cap \pi_k V_{F_t} \right\}$$

$$SC_r = \pi_{C_{rt}} \left(\sigma_{j \in J}(V_{F_r}) \right);\ SC_s = \pi_{C_{st}} \left(\sigma_{j \in J}(V_{F_s}) \right) \tag{3.3.4b}$$

$$TC_r = \pi_{C_{rt}} \left(\sigma_{j \in J}(V_{F_t}) \right);\ TC_s = \pi_{C_{st}} \left(\sigma_{j \in J}(V_{F_t}) \right) \tag{3.3.4c}$$

$$\text{s.t.} \quad \pi_k V_{F_r} \cap \pi_k V_{F_s} \supseteq \pi_k V_{F_t} \tag{3.3.4d}$$

where $\underset{contain}{\Delta}(X, Y) = \frac{|X \cap Y|}{|Y|}$, the jaccard containment of the set Y in in set X, and σ and π are the selection and projection operations in relational algebra, respectively.

The optimization problem in Equation 3.3.4 tries to find the best join key $k \in K$ such that the product of the value containment of the source column contributions from either source artifact (SC_r, SC_s) is maximized in the target (TC_r, TC_s). If the final join score is above some threshold ($\epsilon_{join} = 0.9$) and is the triple with the maximum join score at that instant, then the edges (F_r, F_t) and (F_s, F_t) can be added to our lineage graph. Figure 3.4c illustrates the join detector invoked on our running example and scores the ground-truth join with $\delta_{join}(F_2, F_4, F_5) = 1.0$, as it satisfies all the conditions stated above. In case there are ties for the maximum join score, we break the tie as described in Section 3.3.5.

The Groupby Detector

The Groupby detector works by identifying a groupby-aggregate operation that transforms an artifact F_u to produce F_v by selecting some set of columns C_g to form groups and another set of columns C_a to compute some form of aggregate over. We can check for the presence of

a group-by via a few identifying characteristics; The grouped columns C_g will form a key in the destination artifact and should be contained within the destination graph. Our groupby detector works as follows:

1. Check for schema compatibility. Let $C = C_{F_u} \cap C_{F_v} \neq \emptyset$. If $C = \emptyset$, then $\delta_{groupby}(F_u, F_v) = 0$.

2. Assign source (F_s) and destination (F_d) labels to F_u and F_v based on the number of rows in each, such that $R_{F_s} > R_{F_d}$. If $R_{F_s} = R_{F_d}$ then $\delta_{groupby}(F_u, F_v) = 0$, as we assume a groupby/aggregation operation always results in a contraction of the number of rows in a table.

3. Partition C into two disjoint sets, the grouped columns (C_g) and the aggregate columns (C_a). The grouped columns should exhibit high value containment with the corresponding source columns and should form a unique key in the destination artifact, while the aggregate columns should exhibit low containment. The optimal partition can be found by considering the value containment and keyness constraints as described below:

$$\underset{C_g \subset C}{\arg\max} \quad \underset{contain}{\Delta}\left(\pi_{C_g}(F_d), \pi_{C_g}(F_s)\right) \tag{3.3.5a}$$

$$\text{s.t.} \quad \pi_{C_g}(F_s) \supseteq \pi_{C_g}(F_d) \tag{3.3.5b}$$

$$\frac{\pi_{C_g}(V_{F_d})}{\left|R_{F_d}\right|} = 1.0 \tag{3.3.5c}$$

$$\frac{\pi_{C_g}(V_{F_s})}{\left|R_{F_s}\right|} < 1.0 \tag{3.3.5d}$$

Equation 3.3.5a seeks to find the optimal group/aggregate column partitioning by

finding a column or group of columns in the source that is most contained in the destination. This is subject to additional conditions that the destination values for that column must be fully contained in the source (Equation 3.3.5b). The constraints also specify keyness conditions in the destination artifact (Equation 3.3.5c), which are not in the source (Equation 3.3.5d). If any one of these conditions is not met, then the groupby score for this pair $\delta_{groupby}(F_u, F_v) = 0$.

4. If all the conditions above above are met, the group-by score is computed as:

$$\begin{aligned} \underset{groupby}{\delta}(F_s, F_d) = \quad & \Big(|C_g| - \Big|\pi_{C_g}(F_s) - \pi_{C_g}(F_d)\Big|\Big) \\ & - \big(1 - \Delta_{jaccard}(C_{F_s}, C_{F_d})\big) \end{aligned} \tag{3.3.6}$$

where $\underset{jaccard}{\Delta}(X, Y) = \frac{|X \cap Y|}{|X \cup Y|}$, the jaccard index of the set Y in in set X.

Equation 3.3.6 weighs valid multi-column groupbys higher than those using single columns and penalizes for missing group values that should be present in a groupby and accounts for a column label difference between the source and destination artifacts. Any pair that scores above a threshold ($\epsilon_{groupby} = 1.0$) is considered to be a possible groupby operation. In our experiments, we found this threshold to be satisfied a large number of artifact pairs and use a tie-breaker mechanism to decide which edges to include in the final lineage (Section 3.3.5).

Figure 3.4d illustrates an example of the groupby detector invoked on our running example, and scoring the ground-truth groupby operation with $\delta_{groupby}(F_3, F_4) = 1.0$. You may also note that the other artifacts in the workflow score 0 with the groupby detector as it does not satisfy our row contraction and schema compatibility requirements.

The Pivot Detector

In a pivot operation, there are three types of columns used as input parameters; the first is the destination index column (C_i), another the new set of column labels in the destination (C_c), and the third, is the values in the destination (C_v). Our detector needs to identify these three columns individually. Pandas additionally supports pivots with aggregations, allowing us to ignore the value column (C_v) if we do not find a good match.

1. First, verify that the schema are compatible for a pivot between F_u and F_v. The set of row labels of the pivot destination should be in common with values in some column of the source. The artifact whose row set contains such values should be assigned to F_d and the artifact containing the matching column should be assigned F_s.

2. Determine C_i, C_c, C_v as follows:

$$IC = \underset{C_i \in C_{F_s}}{\arg\max} \underset{contain}{\Delta} \left(\pi_{C_i}(V_{F_s}), R_{F_d}\right) \tag{3.3.7a}$$

$$CC = \underset{C_c \in C_{F_s}}{\arg\max} \underset{contain}{\Delta} \left(\pi_{C_c}(V_{F_s}), C_{F_d}\right) \tag{3.3.7b}$$

$$VC = \underset{C_j \in C_{F_s}}{\arg\max} \underset{contain}{\Delta} \left(\pi_{C_v}(V_{F_s}), V_{F_d}\right) \tag{3.3.7c}$$

$$\text{s.t.} \quad \pi_{C_i}(V_{F_s}) \supseteq R_{F_d} \tag{3.3.7d}$$

$$\pi_{C_c}(V_{F_s}) \supseteq C_{F_d} \tag{3.3.7e}$$

$$\pi_{C_v}(V_{F_s}) \supseteq V_{F_d} \tag{3.3.7f}$$

$$C_i \neq C_c \neq C_v \tag{3.3.7g}$$

$$C_i, C_c \neq \emptyset \tag{3.3.7h}$$

3. If all the conditions above are met, and we identified C_i, C_c and, optionally C_v, we

assign a final pivot score δ_{pivot} as follows:

$$\underset{pivot}{\delta}(F_s, F_d) \quad = \quad (IC \times CC) + VC \tag{3.3.8}$$

The optimization problem in Equation 3.3.7 aims to find source column assignments for C_i, C_c and C_v to maximize their individual containments in the target. The final pivot score is then computed as a product of the value containments of C_i and C_c in the target, with the value containment of C_v added to the final score. Note that C_v may be empty, as certain pivot operations may include an aggregation for the value column, leading to little or no containment. Figure 3.4e illustrates an example of the pivot detector invoked on our running example, and scoring the ground-truth pivot operation with $\delta_{pivot}(F_5, F_6) = (1.0{*}1.0){+}1.0 = 2.0$, which is the highest pivot score among all other possibilities. F_3 and F_4 are not possible pivot sources since they do not contain a column that can be compatible for the column label containment C_c requirement (Equation 3.3.7e). $\delta_{pivot}(F_2, F_6) = (1.0 * 1.0) + 0 = 1.0$ since there is no compatible column for the value column C_v.

3.3.5 Breaking Ties when Selecting Edges

If multiple artifact pairs present the same maximum similarity score when building out the lineage graph, we break ties as follows:

- **Cell-level Jaccard Score**: Choose the pair with the maximum number of overlapping cells.
- **Cell-level Jaccard Containment Score**: Choose the pair with the least difference in the number of rows.
- **Join Detector**: Attempt to replay the join using the key identified for the artifact triple. We compute $F_r \bowtie_k F_s = F_t{}'$ and $F_s \bowtie_k F_r = F_t{}''$. The final tie breaker

score for the join detector is the maximum cell-level Jaccard similarity between the replayed joins (F_t' and F_t'') and the join destination artifact F_t, i.e. $\delta_{join\_tb} = \max\left(\delta_{cell}(F_t', F_t), \delta_{cell}(F_t'', F_t)\right)$. Select the triple with the highest replay score.

- **Groupby Detector**: Attempt to replay the group-by using the detected group-columns C_g over a set of aggregate functions[4]. Compute the cell-level Jaccard score between the replayed results and the target artifact and compare the max scores to select one groupby edge to include in the lineage.

- **Pivot Detector**: Replay the pivot operation and compare with the target dataframes and select the edge with the best replay similarity score similar to the groupby and join detectors above.

3.4 Experimental Evaluation

Our evaluation methodology has been designed with the following goals in mind:

- **Overall Accuracy**: How accurate are the inferred lineage graphs are compared to the ground truth?

- **Variation with Workflow Configuration**: How does the accuracy change with workflow configuration (number of artifact files, size of the original artifacts, and number of operations between materialization of artifact files)?

- **Individual Detector Performance**: What were the edge contributions from each detector (cell, containment, join, groupby etc.) and how accurate were they?

- **Runtime Performance**: What is the overall time taken? What part of our technique takes the longest time to run?

4. We used min, max, sum, mean and count as the aggregate functions

3.4.1 Datasets Used

To the best of our knowledge, there are no standardized data analysis benchmarks or workloads that can be used for evaluating our lineage inference technique. Hence, we rely on a combination of workflows derived from Jupyter notebooks published as a corpus [76] and synthetically generated workflows.

Workflows in the Wild

To evaluate our technique on realistic workloads, we use a variety of workflows sourced from a notebook corpus [76], AzureML, and Kaggle. Jupyter notebooks are used extensively for data analysis and exploration, and the linear nature of notebook code allows us to automatically execute code, observe outputs, and construct ground-truth lineage in a semi-automatic fashion [71]. However, as flexible as Jupyter notebooks are, they are also notoriously messy [76], as they primarily contain experimental, ad-hoc, and exploratory code that may be manipulated and executed out of order. Additionally, the notebook corpus does not contain any associated metadata or contextual information. We sifted through the corpus to find a sample of notebooks that primarily use pandas for data preparation, load data from valid public URLs, generate at-least 5 dataframes, and belong to a single workflow that deals with data analysis or ML prep (as opposed to homework assignments or tutorial/example notebooks). These notebooks were then executed, verified, and hand-annotated to produce the artifact files and ground truth lineage graphs. These workflows are outlined in Table 3.3.

Synthetic Workflow Generator

Our synthetic workflow generator can generate pandas dataframes using the generation parameters listed in Table 5.3. To generate realistic datasets, we used the Faker [28] python library to generate values in different types of columns. Columns are categorized into different types, i.e., numeric, string, group-able. Group-able columns typically have lower cardinally

Name	Description	$(\|\mathcal{F}\|, \|R_{F_0}\|, \|C_{F_0}\|)$
agri-mex	Data Analysis Workflow*	(9, 1300, 9)
churn	Computing Customer Churn*	(5, 3333, 21)
githubviz	Github Repository Visualizations*	(6, 8697,5)
london-crime	Analysis of Crime in London*	(11, 446975, 5)
nyc-cab	Analysis of Cab Rides in NYC*	(17, 150000, 21)
nyc-noise	Analysis of 311 noise complaints*	(24, 136080, 51)
nyc-property	Analysis of NYC property taxes*	(15, 13060, 11)
prop-64	California prop-64 donors*	(10, 56379,8)
retail	Bike rental ML prep†	(21, 17379,16)
titanic	Titanic survivor ML prep‡	(13, 891, 12)

Table 3.3: List of workflows obtained from Jupyter corpus(*), Azure ML(†), Kaggle(‡).

(such as country or state), which allow for meaningful group-by operations to be performed. *Base artifacts* are artifacts that do not have any ancestors in the workflow. Based on the statistics of scraped dataframes [76], we vary the cardinality of columns for each of the base tables as a function of the row-size of the table, to capture the properties of tables found "in the wild".

After generating a base artifact with the specified number of rows and columns, a synthetic workflow is generated by perturbing the artifact using a randomly selected pandas operation[5] with random parameters. For example, a random column may be selected and a random value within that column maybe selected to be replaced with a new one. Our generator performs quality checks to keep the operations meaningful (e.g., it does not generate empty tables or tables with NaNs, we also limit the number of pivots and groupbys performed in a chain). The outdegree of each artifact in the synthetic workflows is also carefully controlled in order to generate a tree-like workflow that is roughly between straight line path and a star-like workflow.

5. The operations are point-edits, row sampling, column drops, new derived columns, groupby, merge and pivot

3.4.2 Configurations to be Evaluated

We have tested the following configurations in RELIC:

- **cell**: Constructs a spanning tree consisting of edges in decreasing order of δ_{cell}, with no threshold on the lowest edge score.

- **cell+detectors**: Constructs a spanning tree consisting of δ_{cell} edges followed by δ_{join}, $\delta_{groupby}$, δ_{pivot} edges, each in decreasing order. We use $\epsilon_{cell} = 0.1$, $\epsilon_{join} = 0.9$, $\epsilon_{groupby} = 1.0$ and $\epsilon_{pivot} = 0.99$ to determine the minimum edge inclusion threshold for each detector.

- **RELIC**: Constructs a tree as defined in Algorithm 1. Artifacts are first clustered by schema, then δ_{cell} edges are added within each cluster. This is followed by δ_{join}, $\delta_{containment}$, $\delta_{groupby}$, δ_{pivot} edges. We use $\epsilon_{intra\_cell} = \epsilon_{inter\_cell} = 0.1$, $\epsilon_{contain} = 0.99$, $\epsilon_{join} = 0.9$, $\epsilon_{groupby} = 1.0$ and $\epsilon_{pivot} = 1.0$. This is the finalized configuration for RELIC.

- **column**: Constructs a spanning tree consisting of edges in decreasing order of column-level jaccard similarity (δ_{column}) edges (as defined in Equation 3.4.1), with no threshold on lowest edge score.

$$\delta_{column}(F_i, F_j) = \frac{\sum\limits_{k \in C_{F_i} \cap C_{F_j}} \left(\Delta\limits_{jaccard} \left(\pi_{C_k}(V_{F_i}), \pi_{C_k}(V_{F_j}) \right) \right)}{|C_{F_i} \cup C_{F_j}|} \tag{3.4.1}$$

The **column** configuration is adapted from the most closely related work at the time of writing, Aurum [30], which is used for dataset similarity search. δ_{column} is a column-oriented similarity function that returns the average column jaccard similarity between two artifacts. Aurum uses approximate similarity search techniques such as LSH [56], trading accuracy for

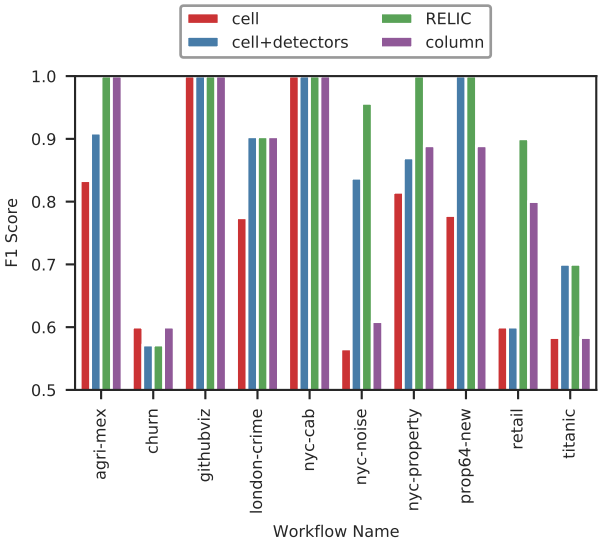
a reduction of search space, which we have not implemented in the interest of fairness to the **column** configuration.

3.4.3 Overall Accuracy

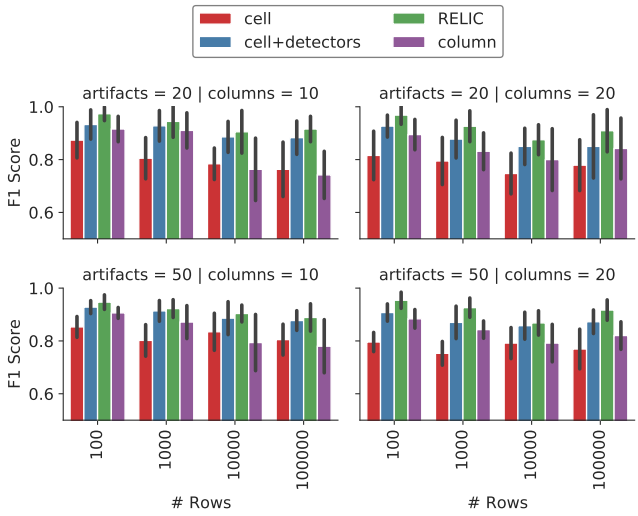
We first evaluate how effective RELIC and other configurations are at recreating lineage for a set of artifacts. Figure 3.6a shows our results for the 10 real workflows described in Section 3.4.1. The average F1 score of our technique RELIC across all the real workflows is around 0.90. We notice that the **column** configuration performs quite well in a number of workflows (`agri-mex, githubviz, nyc-cab`), but fails to produce sufficiently good results for other workflows. Specifically, `nyc-noise` contains multiple `groupby` and `pivot` operations that are better handled in RELIC via the respective detectors than the generalized column baseline. Similarly, `nyc-property` contains `sample` and `concat` operations that were handled by the clustering and containment scoring in RELIC well. RELIC's performance on `retail` highlights the importance of clustering; the `retail` workflow consists of multiple feature selection and sampling, as well as test/train splits, causing erroneous edges to be inferred between feature splits in the baseline configurations (which did not perform the initial clustering).

Finally, `churn` and `titanic` are the primary outliers in terms of accuracy; this can be attributed to a few operations that are not supported, with RELIC such as `crosstab`, and value scaling/normalization. RELIC still infers the lineage correctly for the rest of the edges in those workflows.

Figure 3.6b illustrates the accuracy of our method on synthetic workflows with varying number of artifacts, rows, and columns. For each configuration, we generated ten random workflows. The F1 score RELIC is favorable with an average score of ~ 0.91, and is the best performer on average for all the workflow configurations. Overall, RELIC is able to recover lineage for a wide variety of real and synthetic workflows with reasonable accuracy. The



(a) Real Workflows



(b) Synthetic Workflows. Each bar represents the mean F1 score of 10 separate workflows generated using the (artifact, column, row) configuration. Error bars indicate standard deviation.

Figure 3.6: Lineage recovery accuracy (F1 Score) for the workflows described in Section 3.4.1

addition of clustering and specialized detectors allows RELIC to infer edges more accurately than the other methods.

3.4.4 Individual Detector Performance

We now assess the contributions of the individual detectors towards our overall accuracy. Since each stage of RELIC uses a different similarity score or detector, we label edges as being inferred at that specific stage, using a specific detector. Grouping the inferred edges by detector, we can compute the detector-specific precision and recall, presented in Table 3.4. Note that the edges under consideration at each stage decrease as the graph is built, since we add edges between pairs (or triples) of disconnected components. We see the precision of the individual stages are quite good, with groupby and pivot detectors having the lowest precision. We also see that cell level detectors pick up a majority of the edges with a recall rate of around 0.65. The recall scores for $\delta_{contain}$ are low as well, as this detector is invoked after δ_{cell} and δ_{join} detectors to capture `sample` and `concat`-style operations, which is a

small fraction of the total edges in the workflow.

Score / Detector	real		synthetic	
	precision	**recall**	**precision**	**recall**
δ_{cell}	0.96	0.64	0.95	0.65
δ_{join}	1.00	0.86*	0.99	0.99*
$\delta_{contain}$	1.00	0.05	0.67	0.02
$\delta_{groupby}$	0.83	0.83*	0.69	0.37*
δ_{pivot}	1.00	0.67*	0.38	0.32*
Total	**0.92**	**0.89**	**0.87**	**0.87**

Table 3.4: Individual Stage Performance in RELIC. The ground truth edge space is all edges in the workflow, except for (*), which indicates the recall score specific to the edges for which that specific detector was designed for.

Additionally, we found that the groupby and pivot detectors often produce an "equivalent edge" for that operation, which is the same operation applied to a different source dataframe to generate the same result. This is plausible since we found that for every synthetically generated artifact produced by a groupby or pivot in the ground truth, there are, on average, approximately 2 alternate groupby sources and 6 alternate pivot sources that could have produced the same result. If these "equivalent-edges" are taken into consideration, our synthetic workflow (precision, recall) numbers improve significantly to (0.92,0.50) for the groupby detector and (0.99,0.84) for the pivot detector. Thus, the stage ordering and thresholds used in RELIC provides good opportunities for each of the detectors to find the respective edges, and is reflected in the stage-wise performance metrics presented.

3.4.5 Time to Infer Workflows

We empirically show how RELIC scales in terms of wall-clock time. *Time-to-infer* in our context includes the wall-clock time to load all the artifact tables to memory, perform clustering, compute pair-wise similarity scores and run the individual detectors. All of the experiments were run on a desktop with a Intel Core i7-8700K CPU, 16 GB of RAM and SSD storage. Figure 3.7 breaks down the time-to-infer using RELIC for the real and synthetic workflows

under consideration. Our method scales in time linear to the table size and quadratic to the number of artifacts, due to the increase in the number of pairwise similarities that need to be computed. The join detector forms the bulk of the timing breakdown, especially for larger workflows, as the Join detector looks at artifact triples and is thus cubic in time complexity. The time taken by individual detectors is dependent on the number of components that remain to be connected at the instant the detector is invoked. Small workflows such as `agri-mex`, `chrun`, `githubviz`, `nyc-property`, `pro64-new` and `titanic` have their workflows inferred within 15 seconds, making interactive usage of RELIC plausible. `london-crime`, `nyc-cab`, `nyc-noise` and `retail` all have large artifacts and multiple tied joins and group-bys, which contributes to the long time taken by those detectors. We note that sampling and approximate matching techniques (such as LSH [30, 56]) may be useful in reducing artifact scan sizes- and pairwise comparison overheads, and is something we plan to explore in future work.

3.5 Conclusions and Future Work

In this Chapter, we introduced the problem of retrospective lineage inference and outlined a technique that allows us to recover the lineage of artifact files produced from data analysis workflows. RELIC produces a lineage graph that closely represents the ground truth lineage; all while relying solely on the data present in the files. Our approach differentiates between two types of transformations (PPT and NPPT), and we use different similarity scores and detection techniques for each. Our technique was evaluated against real workflows derived from Jupyter notebooks, as well as synthetically generated workflows. We can recover lineage with an F1 score of approximately 0.90 and 0.91 for real and synthetically generated workflow respectively.

Apart from pandas, we believe that RELIC can be extended to alternatives such as SQL and R. To improve accuracy and support a wider variety of operations, new detectors can

be constructed (such as normalization, scaling, flatten, melt, etc.). Once the lineage graph is constructed, directionality may be inferred, and techniques from query synthesis [10] can be used to infer the exact operation used to generate a destination artifact from a source artifact, providing a more complete picture of the workflow lineage.

3.6 Relaxing Relic's Assumptions

As we saw in Chapter 3, the RELIC system is designed to work with a single workflow. In this chapter, we relax this assumption and show how RELIC can be extended to support multiple workflows where the artifacts belonging to multiple workflows are stored in the same location. We also show how RELIC can be extended to support workflows with different materialization rates. Finally, we show how RELIC can be extended to support workflows with unmatched schemas. We evaluate the performance of these extensions using a set of synthetic workloads.

3.6.1 Effect of Materialization Rate on Accuracy

In addition to our overall results, we would like to see what effect the materialization rate has on accuracy for RELIC. Artifacts that are materialized less frequently should exhibit less similarity and may affect many of the conditions that we use in our NPPT detectors, impacting accuracy. Setting a fixed number of operations, rows, and columns (50,1000,20), we vary the materialization rates from 1 (which generates an artifact after every operation) and 8 (which generates an artifact after every eight operations). We refrain from generating join operations in this experiment as it is hard to keep the number of artifacts fixed as a join requires two inputs. Figure 3.8 plots the average F1 score for each materialization rate. We see that as the materialization rates increase, the accuracy of our technique decreases, especially for non-point preserving operations, as the detectors may not correctly capture the containment metrics that we target when several operations are stacked upon each other

before an artifact is materialized. However, even with four operations between materialization, we have a reasonable accuracy range, which suggests that RELIC could be used in more general lineage recovery scenarios.

3.6.2 Unmatched Schema

RELIC's similarity metrics and detectors heavily rely on schema-matched and row-matched artifacts; Within our pandas workflows, we leaned on the availability of consistently named columns and indices that are matched for accurately computing scores such as the δ_{cell}. This is not always true; columns can be arbitrarily renamed, rows can be sorted and indices can be dropped. For RELIC to be useful in arbitrary data-lake environments, we will need to include some form of schema matching to allow for these deviations from our assumptions.

Schema Matching Valentine [53] is the most-up-to-date benchmarking survey of the state of the art methods in schema matching, and an implementation was readily available for experimenting with pandas dataframes[69]. Our initial experiments have shown that schema-based methods (column names, types and relations) are strike a good balance between speed and accuracy for the retrospective lineage inference application. Fig 3.9 shows the mean F1 accuracy of schema matching against the synthetic dataset for 4 different schema matching systems for a sample of synthetically generated workflows, organized by operation type:

- **coma**: COMA[25] combines multiple schema-based matchers. Schemata are represented as rooted directed acyclic graphs, where the associated elements are graph nodes con- nected by edges of different types (e.g. containment). The match result is a set of element pairs and their corresponding similarity score.

- **cupid**: Cupid[58] is a schema-based approach. Schemata are translated into tree structures representing the hierarchy of different elements (relations, attributes etc.). The overall similarity of two elements is the weighted similarity of i) Linguistic Matching

# Rows	Matcher	assign	dropcol	groupby	merge	pivot	point_edit	sample
100	COMA	1.83	2.00	2.01	1.88	1.92	1.89	1.90
	CUPID	915.60	766.37	644.77	688.76	438.85	847.55	1044.25
	JLM	1.25	1.43	0.28	1.82	0.04	1.15	0.77
	SF	1.08	1.00	0.61	1.31	0.41	1.02	1.09
1000	COMA	2.34	2.09	2.10	2.35	2.04	2.26	1.99
	CUPID	DNF	DNF	DNF	DNF	DNF	DNF	DNF
	JLM	11.33	8.49	5.35	17.73	0.06	6.03	5.18
	SF	1.89	0.77	0.60	1.81	0.84	0.79	0.72
10000	COMA	1.78	1.70	1.76	1.85	1.99	1.78	1.63
	CUPID	DNF	DNF	DNF	DNF	DNF	DNF	DNF
	JLM	31.97	15.26	0.77	110.79	0.84	15.93	11.89
	SF	0.45	0.37	0.45	1.47	4.64	0.45	0.47
50000	COMA	2.67	2.45	2.49	3.33	9.59	2.43	2.37
	CUPID	DNF	DNF	DNF	DNF	DNF	DNF	DNF
	JLM	215.28	311.20	77.05	2109.40	11.76	207.48	134.31
	SF	1.66	1.21	0.96	2.40	33.19	1.08	1.75

Table 3.5: Mean run-time (in seconds) to infer the schema among a single pair of tables. Results are organized by table size, schema matcher and operation. `DNF` indicates that the matcher took longer than 3600 seconds to run.

and ii) Structural Matching.

- **jlm**: A naive instance-based schmea matcher that uses jaccard-levenshtien distance of column pairs, as implemented in [69]

- **sf**: Similarity Flooding[59] is a schema- based matching approach that relies on graphs, and outputs correspondence between any kind of elements (relations, attributes, data types) of two given schemata.

Figure 3.9 shows the relative accuracy of the schema matching techniques when applied to source/destination dataframe pairs for different types of operations. Table 3.5 shows the number of seconds elapsed when inferring the schema between a single pair of artifacts organized by table size and operation. Our initial results show that Similarity Flooding (sf) is the most promising in terms of both accuracy and performance for our application.

We can now show the impact of using the Similarity Flooding schema matching technique on the overall Relic accuracy. For this experiment, we have randomly selected 5 workflows from Section 3.4.1, and altered the schema of all the artifacts as follows. First,

we randomly select 10% of column labels that appear in more than one workflow. For each of occurrence of a selected column label in a workflow, we perturb the column label with a 40% probability. For labels that are perturbed, we generate a new label with 10% of the label's characters changed. For example, the column label `zip_code` appears in 20 different artifacts and was chosen to be altered; as a result eight occurrences of this attribute change their label to `bip_coda`. In Figure 3.10, we show the results of RELIC on the original (unperturbed workflow), RELIC run with the similarity flooding schema matcher implemented in Valentine [53], as well as RELIC without the schema matching step. The schema matcher was run on every pair of artifacts prior to running any of the similarity functions and detectors. RELIC manages to perform respectably, indicating the viability of our approach even in challenging lineage reconstruction scenarios. Of note is the impact on runtime; The similarity flooding matcher adds ~ 1 second of run-time for every pair of artifacts that needs to be schema matched. For example, (50,100,20) configuration in Figure 3.10 takes roughly 5 minutes to infer lineage without schema matching, but the runtime increases to 30 minutes with the added schema matching step. Further optimizations such as the implementation of a mismatched column threshold before invoking the schema matcher could potentially reduce the impact of this step.

3.6.3 Multiple Workflows

We evaluate the efficacy of RELIC in inferring lineage in a data-lake style setting with artifacts from multiple workflows mixed together in the same directory. Our first mixed workflow (`real-world`) consists of all 135 artifacts from the workflows in the wild (Section 3.4.1). We found RELIC to perform well on this mixed workflow set with an F1 score of 0.926 (Table 3.6), owing to the diverse nature of the workloads and lack of shared artifacts and schema between them. Thus, evaluating RELIC in data-lake style setting requires generating multiple synthetic workflows with controllable schema overlap, allowing us to emulate differing

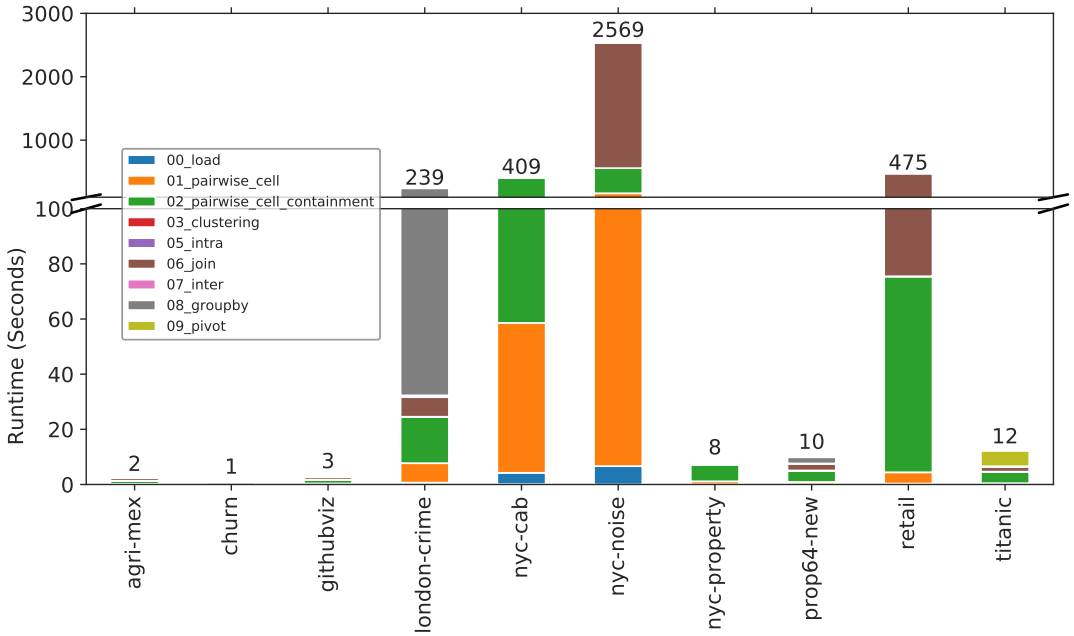
Configuration	#Artifacts	F1 Score	Cross Workflow FPs
real_world	135	0.926	0
$\mu = 0.1$	2156	0.864	0
$\mu = 0.2$	2376	0.846	0
$\mu = 0.5$	2249	0.824	2
$\mu = 0.75$	2253	0.818	5

Table 3.6: F1 score and cross-workflow false positive edges for RELIC on multiple workflows.

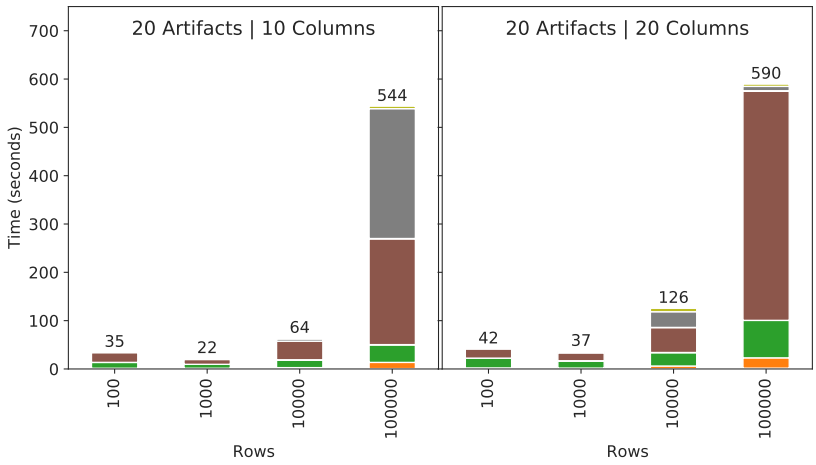
data reuse and similarity levels.

To synthesize a specific data lake with n workflows, we first generate s distinct *seed workflows* using the techniques described in Section 3.4.1. We then generate $n-s$ *overlapping workflows* in which every base artifact has c columns that overlap with some randomly selected seed workflow. c is chosen at random from a Gaussian distribution $\mathcal{N} = (\mu, \sigma^2)$. We set $n = 100$, $\sigma = 0.1$, $s = 25$ and vary μ from 0.1 to 0.75 to generate four different synthetic data lakes with increasing schema overlap.

Table 3.6 shows the performance of RELIC on these multiple workflows. In the synthetic data lakes, we observe that as the column overlap (μ) increases, we find a reduction in accuracy and a slight increase in *cross-workflow false positive edges* (Cross Workflow FPs), (i.e., false-positive edges inferred by RELIC which cross the original ground truth workflow boundaries). Here we find that RELIC's performance degrades as the datasets overlap, but suggests that RELIC can be used in a multi-workflow data-lake environment to help reconstruct the lineage of a data repository or at least help in clustering artifacts by workflow.



(a) Real Workflows



(b) Synthetic Workflows

Figure 3.7: Time to run the lineage inference technique, broken down into individual stages

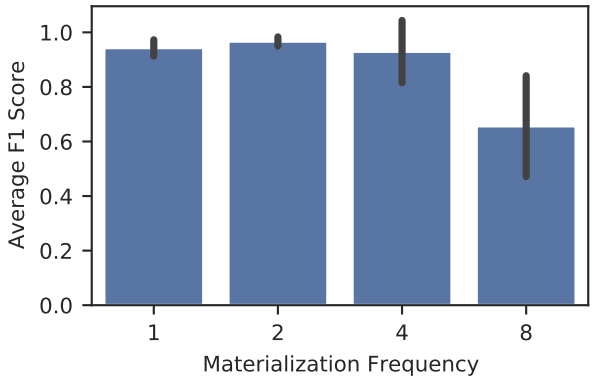


Figure 3.8: Effect on materialization rate on Accuracy. Each bar represents the average of 5 workflows generated using 50 operations with a base table size of 1000x20. Error bars represent the standard deviation.

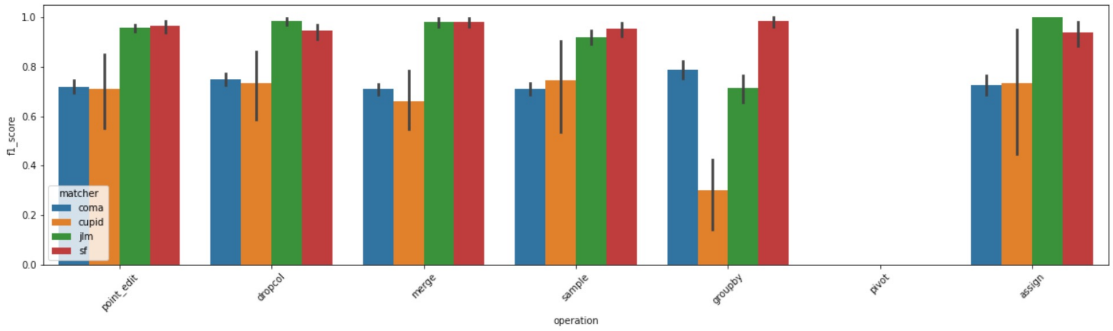


Figure 3.9: Mean F1 score accuracy of various schema matching techniques grouped by operation type for different synthetic workflows. All of the techniques failed to produce a schema match for the `pivot` operation. Error bars indicate standard deviation.

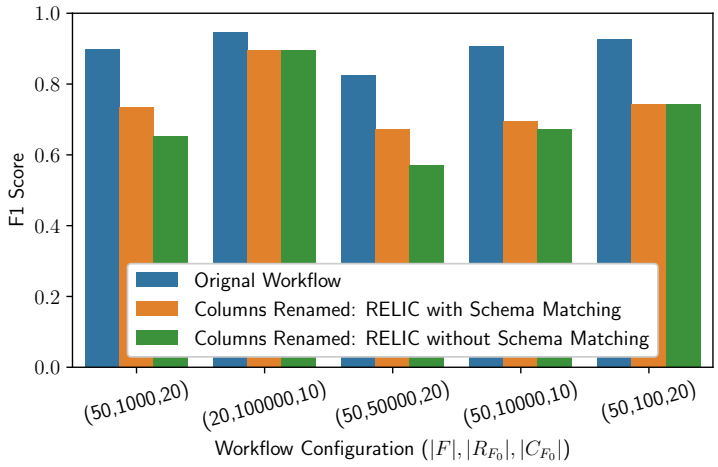


Figure 3.10: Mean F1 score accuracy of RELIC against workflows whose schema have been perturbed.

CHAPTER 4

SCALING RELIC

In Chapter 3, we outlined an initial solution for the *Lineage Inference Problem*, and relied on several simplifying assumptions to present the RELIC technique to infer a lineage graph using similarity metrics. Later in the chapter, we showed the effect of relaxing some of those assumptions and showed that RELIC was about to perform reasonably well, given the circumstances. However, we were only able to show RELIC working on small sets of data, of at most 2,500 individual artifacts, which, in turn, took around 24 hours to complete. For RELIC to perform on larger data lakes, we must implement scalable techniques for inferring lineage.

In this chapter, we will pivot to the problem of answering lineage queries at scale. The work we present in this chapter deals with some of the most time-consuming phases of RELIC, namely the *join*, *groupby* and *pivot* detectors, and shows the ability to use an order-invariant hash along with containment sketch techniques to answer these lineage queries without having to resort to pair-wise similarity computation.

4.1 Motivation

Exploring and finding relevant datasets from a data lake for data analysis, wrangling, or machine learning tasks has often been cited as a time-consuming process [72]. Most work on data lake discovery has focused on helping users tables that are *joinable* [15, 29, 98] (which contain a common key column to augment the query table with additional columns via a join operation) or *unionable* [15] (contain additional rows, with the same set of columns, so that the dataset can be extended). Given the size and complexity of typical data lakes, most prior work relies on *sketching*, a technique by which a compact representation of the data is pre-computed, to be used for storage and searching tasks. *Minhashing*[13] is a popular

sketch used for computing the jaccard similarity of sets without having to compute the full set intersection. Sketches are often used in conjunction with *indexing* techniques to reduce the $O(n^2)$ pair-wise comparison complexity down to a smaller candidate set of tables, which are then ranked and presented to the user as potential joinable or union-able table pairs. For jaccard similarity, the *Locality Sensitive Hash (LSH)* is a popular technique to index minhash sketches. LSHs can then be used to find candidate sets that are similar to a given query set in $O(n)$ time, where n is the number of tables.

In this work, we seek to extend the class of related tables that can be found in previous data lake discovery work [15, 29, 98], which focus on joinable or unionable tables. Instead, we argue that similar sketches and indexes can be constructed to answer a larger set of queries related to table lineage, such as tables that were the source for a join, groupby or pivot operation. As a motivating example, given a data lake that consists of $\mathcal{D}$ tables, consider the following lineage query scenarios:

experience_level	job_title	salary_in_usd	country	remote_ratio
Senior	ML Engineer	192600	US	100
Mid	Data Analyst	50000	Spain	100
Senior	ML Engineer	66265	India	0
Mid	Data Analyst	32974	Australia	100
Senior	Data Engineer	25000	US	100

Table 4.1: Example Table R

country	gdp
Australia	1.3 trillion
India	2.7 trillion
Spain	1.3 trillion
US	21.0 trillion

Table 4.2: Example Table S

Example 1. *GroupBy Source Query: A user is given a query table T_Q, which consists of some unknown source table $T_X \in \mathcal{D}$ that has been grouped by a set of columns C_G,*

experience_level	job_title	salary_in_usd	country	remote_ratio	gdp
Senior	ML Engineer	192600	US	100	21.0 trillion
Mid	Data Analyst	50000	Spain	100	1.3 trillion
Senior	ML Engineer	66265	India	0	2.7 trillion
Mid	Data Analyst	32974	Australia	100	1.3 trillion
Senior	Data Engineer	25000	US	100	21.0 trillion

Table 4.3: Example Table T, constructed by joining R and S on the column *country*

experience_level	job_title	salary_in_usd
Mid	Data Analyst	41487
Senior	Data Engineer	25000
Senior	ML Engineer	129432.5

Table 4.4: Example Table U, constructed by grouping Table S by column experience_level and job_title and applying the average function on the column salary_in_usd

job_title	Australia	India	Spain	US
Data Analyst	32974	–	50000	–
Data Engineer	–	–	–	25000
ML Engineer	–	66265	–	192600

Table 4.5: Example Table V, constructed by pivoting Table S by column job_title as the index and country as the columns and salary_in_usd as the cell values

with another set of columns C_A having an aggregation function $agg(x)$ applied against each group. The user is now tasked to find the T_X in order to explore other groups C'_G and/or apply another aggregate function $agg'(x)$ on possibly another set of aggregate columns C'_A. As an example of this type of query, consider Table U (Illustrated in Table 4.4), and the user is now tasked to find the original table S.

Example 2. *Pivot Source Query: A user is given a query table T_Q, which was generated by applying a pivot function on some source table $T_X \in \mathcal{D}$. The user needs to find T_X in order to explore other pivots that could be applied to T_X or look for other columns in the original table. As an example of this type of query, consider Table V (Illustrated in Table 4.5), and the user is now tasked to find the original table S.*

Example 3. *Join Source(s) Query: A user is given a query table T_Q, which was generated by applying a join function on some source table(s) $\{T_{X_1} \dots T_{X_i}\} \in \mathcal{D}$. The user needs to find each source table T_{X_i} to explore other columns or keys that did not end up in T_Q. As an example of this type of query, consider Table T (Illustrated in Table 4.3), and the user is now tasked to find the original tables R and S.*

Readers will note that this problem is a hybrid combination of data lake discovery as well as Query-Reverse Engineering (QRE) or Query-By-Example (QBE) tasks, which focus on finding the exact query used to generate a specific output view from a set of input tables. Searching for relevant tables within a large table corpus using keywords or an input table is typical in data lake discovery, while QRE/QBE takes input/output table pairs and generates a query that produces the given output from the input(s). In this scenario, we have an output table, as well as an operation (which may not be fully specified), and we would like to find the appropriate input table from the given corpus.

In this work, we explore the application of various sketch techniques that allow us to find a suitable set of candidate Tables T_X for these types of lineage queries. We provide a novel multiple-column coupled-containment sketching scheme that allows us to narrow down the

set of candidate tables to answer these types of data lake lineage queries efficiently. Following the retrieval of the candidate tables for each query, we use specialized ranking functions to return the top-k matches that are likely to be the input table for a given operation.

4.2 Preliminaries

4.2.1 MinHashing

Minhashing, described by Broder et.al. [13], uses k separate hash functions over a set of values V and stores the minimum hash value for each hash function as a set signature. Broder showed that the probability that two sets having the same minhash for a given hash function is equal to their Jaccard similarity. Increasing the number of hash functions k typically improves the estimation accuracy; most literature uses 128 or 256 hash functions for applications in data lake discovery [30, 96, 99].

Formally, consider two sets c_1 and c_2 with every element in each set coming from a universe of elements E such that $\forall e \in c_i, e \in E$. Consider a random hash function h that maps an element in E to S, the universe of hash values, and obtains the minimum hash value $s \in S$ by applying h on c_i, that is $min(h_j(c_i))$. For a given h, the probability that the minhash of two sets c_1 and c_2 being the same is exactly equal to the Jaccard similarity of the sets (Equation 4.2.1):

$$P\left[min\left(h\left(c_1\right)\right) = min\left(h\left(c_2\right)\right)\right] = \frac{|c_1 \cap c_2|}{|c_1 \cup c_2|} \tag{4.2.1}$$

Using K independent hash functions, $h_1, h_2, \ldots, h_K$, the Jaccard similarity can be estimated as follows (Equation 4.2.2):

$$\hat{J}(c_1, c_2) = \frac{1}{K} \sum_{i=1}^{K} \mathbb{1}\!\!\!\!\mathbb{1} \left(min(h_i(c_1)) = min(h_i(c_2)) \right) \tag{4.2.2}$$

where $\mathbb{1}\!\!\!\!\mathbb{1}$ is the indicator function, which is 1 if the condition is *true* and 0 otherwise.

4.2.2 One Permutation Hashing

One Permutation Hashing (OPH) [57] is a variant of Minhashing that uses a single hash function to generate a set signature instead of using K independent hash functions. Every value in the set c_i is hashed once, and the resulting hashes are partitioned into K buckets. The sketch is then constructed by taking the minimum value per bucket. One issue with OPH is that buckets can be empty, especially with skewed data distributions. Recent methods have been proposed, such as Optimal One Permutation Hashing (OOPH) [77], which attempts to fill empty buckets by *densification*, which introduces randomness in the placement of hash values into the buckets.

4.2.3 K-Minimum Value Sketches

Given a no-collision hash function h which maps elements to range $[0, 1]$, a KMV synopses of a set c_i denoted by $\mathcal{L}_{c_i}$, is to keep k minimum hash values of c_i. Then the number of distinct elements $|c_i$ can be estimated as follows (Equation 4.2.3):

$$\hat{|c_i|} = \frac{k-1}{U_{(k)}} \tag{4.2.3}$$

where $U_{(k)}$ is the kth smallest hash value.

Given two sets c_1 and c_2 with KMV synopses $\mathcal{L}_{c_1}$ and $\mathcal{L}_{c_2}$ respectively. We can construct a composite KMV synopses of $c_1 \cup c_2$ as $\mathcal{L}_{c_1} \bigoplus \mathcal{L}_{c_2}$, which consists of the k smallest values

in $\mathcal{L}_{c_1} \cup \mathcal{L}_{c_2}$, and $k = min(|\mathcal{L}_{c_1}|, |\mathcal{L}_{c_2}|)$. The unbiased estimate of the number of distinct elements in $c_1 \cup c_2$ is then given by (Equation 4.2.4):

$$|c_1 \hat{\cup} c_2| = \frac{k-1}{U_{(k)}} \tag{4.2.4}$$

Similarly, the intersection of c_1 and c_2 can be estimated. Let $K_{cap} = |v \in \mathcal{L} : v \in \mathcal{L}_{c_1} \cap \mathcal{L}_{c_2}|$, i.e. the number of common distinct hash values in $\mathcal{L}_{c_1}$ and $\mathcal{L}_{c_2}$. The the number of distinct elements in $c_1 \cap c_2$ is given by (Equation 4.2.5):

$$|c_1 \hat{\cap} c_2| = \frac{K_{cap}}{k} \times \frac{k-1}{U_{(k)}} \tag{4.2.5}$$

Thus, given the estimates of the union and intersection, the jaccard similarity and containment can also be estimated from the KMV synopses.

4.2.4 Locality Sensitive Hashing

For the minhash and one-permutation hashing techniques described above, locality-sensitive-hashing (LSH) can be used to reduce the number of comparisons required to estimate the similarity between two sets. First, a *signature matrix* can be constructed by organizing the minhashes into a matrix, where each row corresponds to a set and each column corresponds to a hash function. This signature matrix can then be divided into b *LSH bands* of r rows each. Given a query set c_q, the minhash signature of c_q can be compared to the signatures in each band. If the signature of c_q matches the signature of any set in the band, then the set is considered a candidate for similarity estimation. The similarity between the query set and the candidate set can then be estimated using the minhash signatures. Thus an LSH index can be used to reduce the number of comparisons from $O(n^2)$ to $O(n)$ where n is the

number of sets in the index. The number of bands b and rows r in each band can be tuned to achieve the desired threshold for jaccard similarity.

4.3 Coupled Containment Sketches

4.3.1 Sketch Construction

A user inputs a query table T_Q as well as a set of k columns $C_Q = C_{Q_1}, \ldots, C_{Q_k}$ which are a part of T_Q and some threshold ϵ. The *coupled containment query problem* is to find all tables T_X that contain k columns $C_X = C_{X_1}, \ldots, C_{X_k}$, such that the Jaccard containment score of tables T_Q in T_X projected on columns C_Q and C_X respectively is greater than or equal to ϵ (Equation 4.3.1)

$$\frac{\left|\pi_{C_Q}(T_Q) \cap \pi_{C_X}(T_X)\right|}{|\pi_{C_Q}(T_Q)|} \geq \epsilon \tag{4.3.1}$$

Note that the sets of columns C_Q and C_X whose containment is to be estimated are *sets* of columns, i.e., unordered. We, therefore, need to design a containment sketch that is *permutation invariant* to column order. Thus the sketching function $\mathcal{F}$ applied over C_X columns of any given relation T_X must produce the same signature for any permutation $\Pi_j \in S_{C_X}$ (formalized in Equation 4.3.2 below):

$$\mathcal{F}\left(\pi_{\Pi_i(C_X)}(X_Q)\right) = \mathcal{F}\left(\pi_{\Pi_j(C_X)}(X_Q)\right) \quad \forall \quad \Pi_i, \Pi_j \in S_{C_X} \tag{4.3.2}$$

We design this sketch by first designing a permutation-invariant hashing scheme applied on each tuple of values for a given set of k columns. Let row R_i of $\pi_{C_X}(X_Q)$ be the tuple of values represented as $V_{R_i} = (v_1, \ldots, v_k)$. Given a hash function $\mathcal{H}$ that maps an arbitrary

value into a fixed length set of b bytes (such as MD5 [74] or MurmurHash3 [5]), we can construct a permutation invariant hash function $\mathcal{H}'$ by summing of all the hash values, as the sum operation is associative[1]. Therefore the permutation invariant hash of tuple values for V_{R_i} is $\mathcal{H}'(V_{R_i}) = \mathcal{H}(v_1) + \mathcal{H}(v_2) \cdots + \mathcal{H}(v_k)$ maps to the same hash value for all tuple value permutations $\Pi(V_{R_i}) \in S_V$.

4.3.2 Index Construction

Algorithm 3: IndexTables(T, ϵ, k)

Input: Set of Tables $T = \{T_1, \ldots, T_n\}$, Target Containment Threshold ϵ, Column Coupling Size k
Output: Index $\mathcal{I}$

```
1 I := InitilizeIndex(ε, k);
2 foreach T_i ∈ T do
3 |   C_i ← GetColumns(T_i) ;
4 |   foreach index j ∈ [1, k] do
5 |   |   foreach C_l ∈ (C_i choose j) do
6 |   |   |   label ← T_i + C_l ;
7 |   |   |   hash_set ← GetHashes(π_{C_l}(T_i)) ;
8 |   |   |   I ← Insert(I, hash_set, label) ;
9 return I ;
```

In the algorithm described in Algorithm 3, we first initialize an index $\mathcal{I}$ with the target containment threshold ϵ and column coupling size k. We then iterate over each table T_i in the set of tables T, and for each table T_i, we iterate over all possible column sets C_l of size j (where $j \in [1, k]$) and compute the hash set GetHashes($\pi_{C_l}(T_i)$) for each column set C_l. We then insert the hash set and the label $T_i + C_l$ into the index $\mathcal{I}$. Currently, the GetHashes function computes a permutation-invariant hash for each tuple in the relation $\pi_{C_l}(T_i)$ and returns a set of all the hashes. We use the MurmurHash3 [5] hash function to compute the hash values. The Insert function inserts the hash set and the label into the index $\mathcal{I}$, based on the type of Index being used.

1. We also apply a bitmask to ensure that the final hash value is restricted a set number of bits b

4.3.3 Querying the Index for Lineage

In this section, we present various applications for using the coupled containment sketch to query source artifacts used to perform Join, Groupby, and Pivot operations. For each operation type, we construct a specialized query that allows us to quickly find the source artifact from a large corpus of sketched artifacts.

Algorithm 4: QUERYTABLE($\mathcal{I}, C_i$)

Input: Index $\mathcal{I}$, Query Tabke T_q, Query Columns C_i
Output: Matching Column Tables *candidates*

1 $candidates \leftarrow \emptyset$;
2 $hash\_set \leftarrow$ GETHASHES($\pi_{C_i}(T_q)$) ;
3 $candidates \leftarrow$ QUERYINDEX($\mathcal{I}, hash\_set$) ;
4 **return** *candidates*;

Join Lineage Queries

In order to find the probable tables that were used to perform a join with join result table T_J, we could query all of the columns $C \in T_J$ and provide a union of all the results back to the user; however, this will likely result in a large number of false-positive results. On the other hand, if the user can be asked to provide additional information, such as the probable key-column used for the join (C_K), as well as an associated column from each side of the join (C_L and C_R) - the resulting containment query is a lot more selective, allowing for a narrower set of candidates to look at. In practice, we have found that it is best to choose the columns with the highest cardinality from each side to minimize false positive results. Thus, we can query the contained sketches of $\langle C_K, C_L \rangle$, and $\langle C_K, C_R \rangle$ in our index to find the probable join sources.

Groupby Lineage Queries

We can find the source table T_X used to generate the table T_G after performing a group-by operation on columns C_G. The number of group columns is more than one, $|C_G| > 1$, we can query multiple columns using our coupled containment scheme $< C_i | C_i \in C_G >$. In case we have sketched only k-coupled columns in our index, we can query only a subset of the columns in C_G and take the intersection of the results to narrow the space of candidate artifacts responsible for the groupby. For example, suppose we have a groupby operation performed on three columns, while our index contains only 2-column coupled sketches. In that case, we can choose 2-column combinations to probe in the index and return an intersection of the query results to the user.

Pivot Lineage Queries

To find the table T_X on which a pivot operation was applied to generate table T_P, we can select the index values and column names to generate 2-column coupled sketches for each index, column name pair and retrieve all values from the index. This can be achived by enumerating all (index, column)ß pairs that have a valid value in T_P.

4.3.4 Ranking the Results

Typically, approximate similarity indices return an unordered list of candidate sets that meet the required similarity or containment threshold. In this work, we explore candidate ranking using the existing sketches to estimate the containment and rank the candidates in ascending containment scores. We use the existing sketch techniques to additionally compute a jaccard containment score using the existing hash signatures and use these scores to return probable candidates back to the user. Thus a containment score is computed between the query sketch and each of the candidate sketches in the index, using the existing hash or KMV sketch signatures. The resulting scores can then be used to rank the set of candidates

and surface the most likely candidate for the given lineage query.

4.4 Implementation

Now that we have obtained a permutation invariant hash of tuple values for a single row and a given set of columns, we can construct a sketch of all the row values for the set of columns for a given relation. In this section, we discuss the specific sketch techniques that we have evaluated for the problem at hand, followed by a description of the client/server system implemented to run large-scale experiments with this system. We have implemented three different techniques as they allow us to explore multiple tradeoffs between the techniques, in terms of sketch size, indexing time, querying time and index size.

4.4.1 Sketching Techniques Used

In this work, we consider 3 separate sketching schemes designed for set containment queries:

1. LSH-Ensemble Sketch [99]
2. Lazo Containment Sketch [31]
3. GB-KMV Sketch [92]

The LSH-Ensemble sketch, first described by Zhu et al. [99] allows for the computing of Jaccard containment using the standard Minhash and LSH Index used for estimating Jaccard Similarity. LSH-E does this by partitioning the space of all indexed items by set cardinality and creating a separate LSH-index for each partition. Zhu et al. further show that an optimal partitioning exists; however, the data must be repartitioned and re-indexed if more items are added to the set, as the accuracy of the index levels off as the index, is updated with new items [31].

In the Lazo [31] method, the LSH index is constructed using OOPH [77] to provide a first estimate of the jaccard containment, and the sizes of the individual sets are stored. Since the intersection and union of the sets are bounded by the minimum and maximum cardinalities of the sets involved, this estimate is adjusted to account for the actual cardinalities of the sets, providing a faster and more accurate estimate of these cardinalities.

Finally, the GB-KMV method constructs an index using the K-Minimum Value Sketch method. It includes a histogram of the most frequent elements to augment the KMV sketch to provide a more robust estimate of the cardinality of set intersection and, consequently, the Jaccard similarities and the Jaccard containment scores for the individual sets.

4.4.2 Software Implementation

All of the sketch techniques were implemented in a client-server model using Apache Arrow [32]. This allowed us to create a common sketch building and querying client, which in turn allowed us to compare the results using different sketching techniques.

Client-Server Interaction Model

A common server API was built to transport Arrow-encoded columnar streams of data from clients. The server is initialized with a few parameters that are specific to the sketch technique. Similarly, the client can be initialized with a directory containing all the tables that are to be indexed, as well as the maximum number of columns to combine while constructing the containment sketches. The client-Server interaction model is shown in Figure 4.1.

In the sketching phase, the client sends a column set to be stored, and the server uses the specific sketch technique to generate a signature for that column. Once all the columns in a session are sketched, the client can issue an Index operation, which produces the query-able containment index. Lazo allows for an incremental index to be constructed, while GB-KMV[2]

2. GB-KMV can be modified to support incremental indexing, but was not implemented in the code that

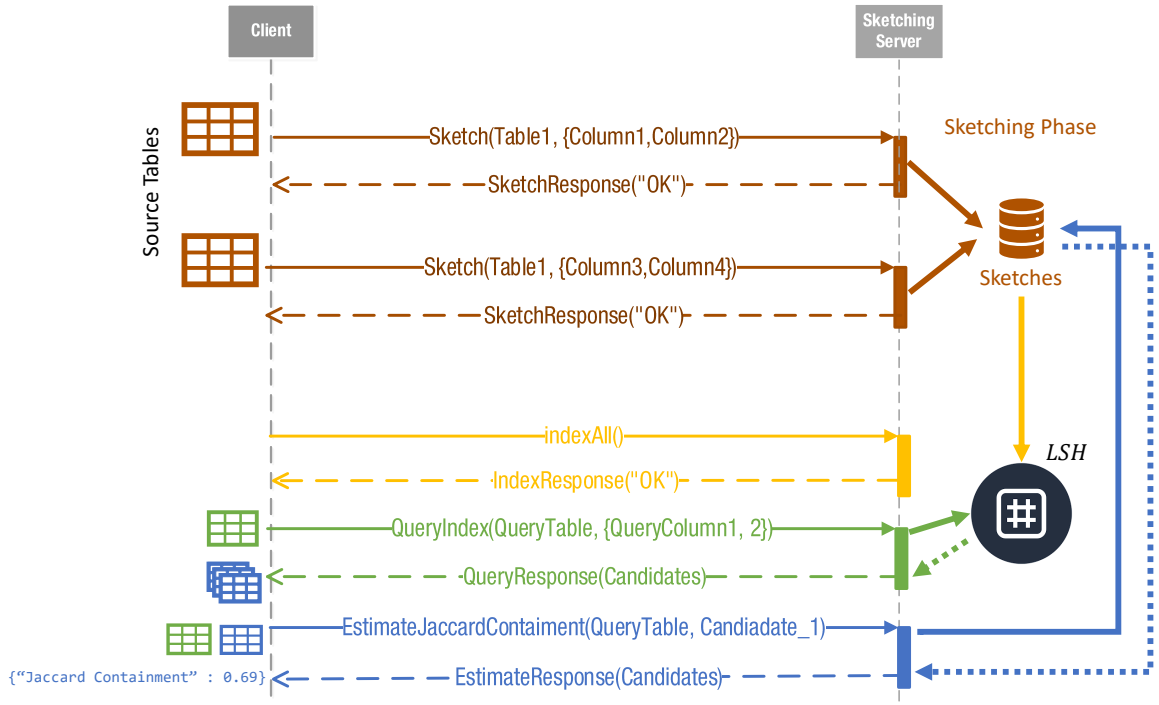


Figure 4.1: Client-Server Interaction Model using Apache Arrow.

and LSH-Ensemble require a fixed domain of sets before they can be indexed. Once an index is constructed, the client can issue query sets to the server in the querying phase, which the server can process by sketching the query column and probing the index for containment matches. The server then returns a set of candidate labels to the client. The client can further query for a containment score estimate from the server for a pair of query/candidate labels in the ranking phase. The client then produces a sorted list of result candidates, which are then presented to the user.

4.5 Evaluation

In this section, we discuss the evaluation of our index implementations and the performance of our proposed system. The evaluation is designed to assess index retrieval and ranking accuracy, as we seek to demonstrate that our system can be used to answer lineage queries

was published

in a real-life data lake scenario. We also want to demonstrate the time and space required to build the index, which is a key consideration for any indexing system, as an offline cost.

4.5.1 Datasets Used

To evaluate the sketching and indexing system, we design an experiment to allow us to evaluate the retrieval and ranking performance of our proposed system in a real-life lineage inference scenario. To this end, we have used two separate datasets.

`gittables`: The `gittables` dataset [49], a collection of tabular datasets that were collected from public GitHub repositories. Our dataset filtering and sampling procedure is summarized in Table 4.6 and outlined below:

Preperation Stage	**Size**
GitTables Full	1,018,649
Filtered	824,457
Sample	5,000

Generated Query Classes

Query Type	**Number Generated**
Two-Column Groupbys	100
Three-Column Groupbys	100
Joins	100
Pivots	100

Table 4.6: GitTables Dataset Filtering and Query Class Generation

1. We filter the GitTables dataset to remove all tables that have less than 10 rows and 3 columns. This leaves us with 824,457 tables.

2. We sample 5,000 tables from the filtered dataset and generate 100 queries for each of the 5 query types: single-column groupbys, two-column groupbys, three-column groupbys, joins, and pivots.

3. For groupbys and pivots, we randomized queries by randomly selecting a source table

and then selecting random columns to perform the groupby aggregation or the pivot operation on.

4. Joins are performed by selecting a random table and random key column and then joining with any random table that has the same column as a key.

5. We then perform the following sanity checks on the generated tables to ensure that they are valid:

   (a) The table must have at least 10 rows.

   (b) At least 45% of the table cells must contain a non-null value

   (c) Generated tables must have a valid schema (non-null, non-empty column labels)

   (d) Groupby operations must collapse the table by at least 50%.

`synthetic`: We have also used a synthetic dataset generated by `fuzzydata`, a randomized dataframe generation system (detailed in Chapter 5). Similar to the `gittables` dataset, we generated 1000 randomized schema descriptions and then generated 5000 random tables of varying row lengths by choosing a schema description uniformly at random. We then generated 100 queries for each of the 5 query types: single-column groupbys, two-column groupbys, three-column groupbys, joins, and pivots. We then performed the same sanity checks on the generated tables as we did for the `gittables` dataset.

4.5.2 Competing Systems

We compare our system to the following baselines:

- **MATE** - At the time of writing, only a single system, MATE [27] was designed to perform search/retrieval using a multi-column sketch. MATE is designed to quickly discover joinable tables using multi-column attributes (for example, a join on First Name, Last Name or a join on City, State). MATE is designed to be used in a data

exploration scenario, where the user provides a table and a set of joinable columns, and wants to find a set of tables that can be joined on that composite key attribute. In contrast, our system is designed to be used in a lineage inference scenario, where the user provides a table, a likely operation and a set of operation parameters, and would like to retrieve tables that are likely the source tables for the given table. MATE uses a custom multi-column hash function, XASH, that is designed to act like a multi-column bloom filter, and is as such, designed to answer set membership queries under very limited constraints. The XASH design constraints include an assumption that string tokens are typically <= 17 characters. In contrast, our system is designed to answer set membership queries under a much wider range of constraints, including the ability to handle string tokens of arbitrary length. In addition, our system is designed to answer set membership queries for a much wider range of operations, including groupbys, joins, pivots, and more.

- **brute-force** - We also compare our system against a non-sketching baseline, which is a brute-force set containment search of the entire GitTables sample tables of 5000 tables. For each query, we compare the query value set against the value sets of all two-column combinations of all the tables in the dataset. This baseline is used to provide a lower bound on query performance and an upper bound on retrieval accuracy, as we compute the exact containment score for each candidate column set.

4.5.3 Evaluation Metrics

We aim to evaluate the accuracy and runtime performance of multiple components of our system: index generation, query/candidate retrieval and ranking. Index generation is the process of generating the sketching index from the GitTables dataset. Query/candidate retrieval is the process of retrieving a set of candidate tables from the GitTables dataset that would be likely to be the source tables for a given table. The ranking is the process of

ranking the candidate tables by their likelihood of being the source tables for a given table.

During the indexing generation phase, we are only interested in the runtime performance of the index generation process. We do not evaluate the accuracy of the index generation process, as the index generation process is designed to be a one-time process and can be amortized over the system's lifetime or done when artifacts enter the data lake.

During the query/candidate retrieval phase, we are interested in both the runtime performance and the accuracy of the retrieval process. For a given class of queries (for example, groupbys), we query the index with the appropriate querying technique 4.3.3, and obtain a set of candidate tables. We report the number of candidate tables returned and if the correct table is present in the set of candidate tables. Since each query class contains 100 queries, we report the ratio of queries that had the correct source table retrieved as the *average recall rate* for the given query class. This experiment is repeated against multiple containment thresholds to determine the optimal containment threshold for each class of queries - the optimal threshold provides the highest accuracy while returning the minimum number of candidate tables. The time to query and retrieve the set of candidate table lists is also reported.

Finally, we evaluate the ranking process by evaluating the quality of ranking using the *recall rate at K* metric. This measure allows us to report the ratio of ranked results where the correct table was present within k ranked results for each query class. We evaluate the recall rate at K for k=1, 5, and 10.

4.6 Results

4.6.1 Index Query Accuracy

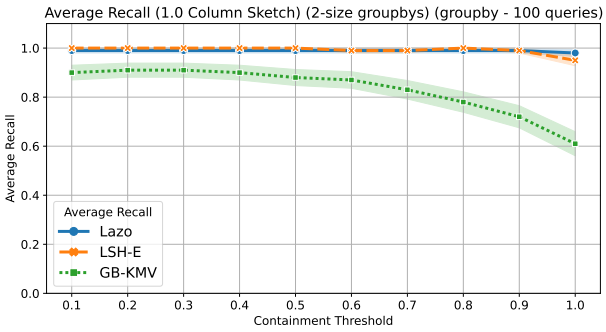
In this section, we present the results of our experiments. We first show a comparison of the three sketch techniques outlined in Section 4.4 for various containment thresholds. We

first focus on the `gittables` dataset. In Figure 4.2a, we see the average recall of the three sketch techniques across containment thresholds for the 100 groupby queries presented in Section 4.5, using only a single-column sketch. We also show the corresponding recall rate using our 2-column coupled sketching scheme in Figure 4.2b. This experiment is repeated for join queries in Figures 4.3 and 4.4, as well as pivot queries in Figure 4.5. From these results, we find that, on average, all three techniques show a high recall rate for containment thresholds of 0.4 and lower, and using extremely high containment thresholds diminishes the recall rate. We also find the number of results returned is higher for lower containment thresholds (as expected), and find that 0.4 is a sweet spot for accuracy in terms of recall across all queries as well as the number of results returned. While all three techniques have favorable average recall rates, overall, we find LSH-E to be quite noisy in terms of the number of results returned, approaching as high as 6000 on average for two-column sketches in groupby queries. We see similar results for the `synthetic` dataset in Figures 4.6,4.7,4.8, and 4.9.

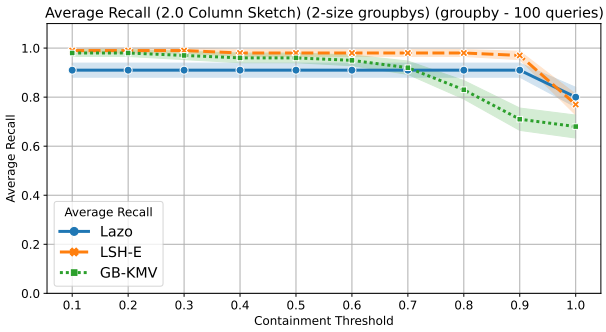
Thus, for all future experiments, we will use 0.4 as the containment threshold as we proceed to assess the ranking performance of the three techniques.

4.6.2 Result Ranking

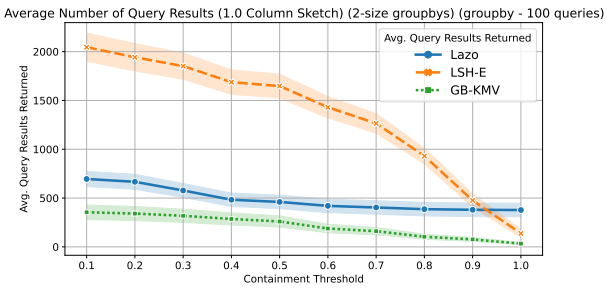
Now that we have shown that the similarity indices can be used to retrieve the relevant tables for each type of lineage query, we can now explore the use of the generated sketches to rank the candidates in order of most likely source for each query. Table 4.7 lists the results of the ranking experiment. We used the target containment threshold of 0.4 as determined in Section 4.6.1. We then retrieved the relevant sketch for each candidate and estimated the containment score of the query set against the candidate sketch, and used the score to rank the results. We report the results for Join separately (as we rank the results separately for each side of the join, left and right). Overall, we see that the use of coupled sketching



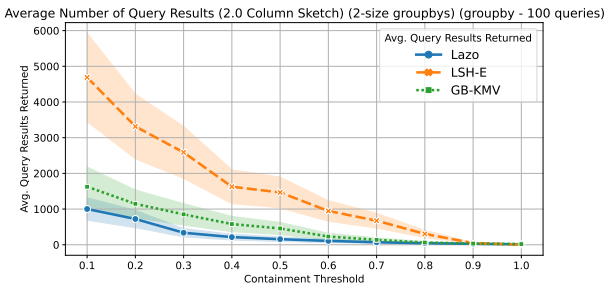
(a) Single Column Sketches (Average Recall)



(b) Two-Column Sketches (Average Recall)

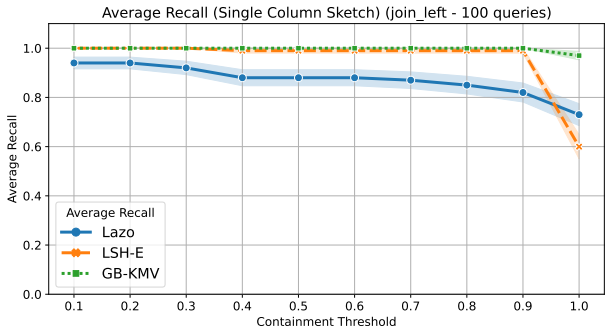


(c) Single Column Sketches (Average Results Returned)

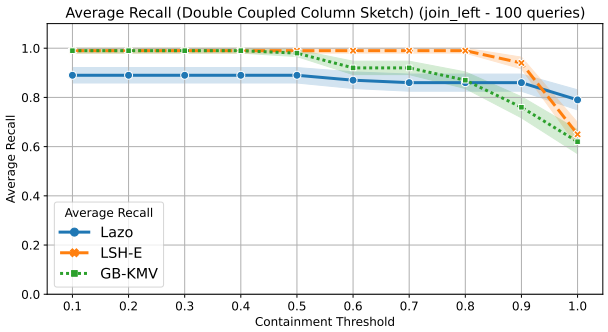


(d) Two-Column Sketches (Average Results Returned)

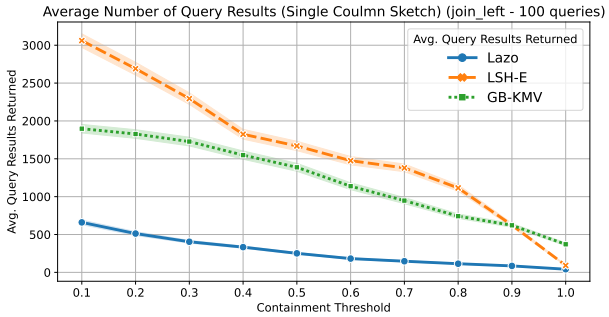
Figure 4.2: Average recall and number of results returned for 2-column groupby queries vs. target containment thresholds for the `gittables` dataset. Error bands indicate the standard error of the mean over 100 queries.



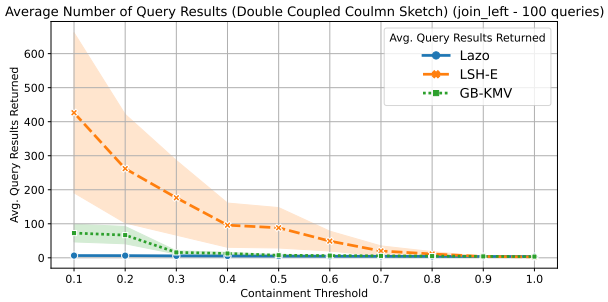
(a) Single Column Sketches (Average Recall)



(b) Two-Column Sketches (Average Recall)

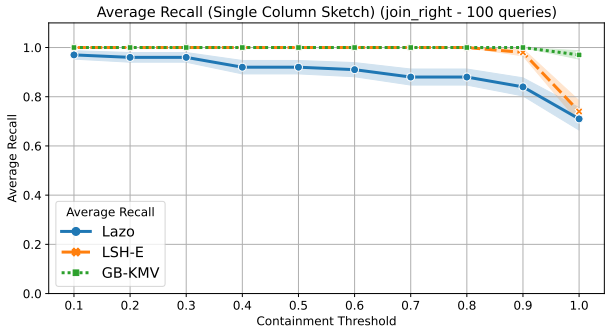


(c) Single Column Sketches (Average Results Returned)

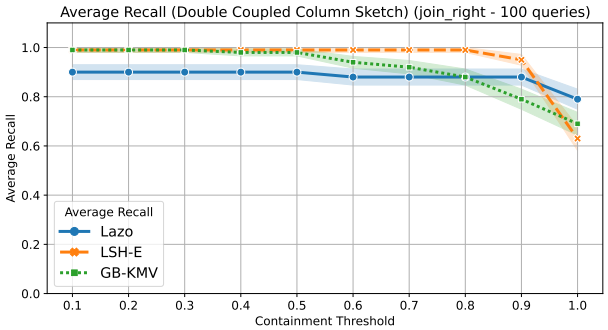


(d) Two-Column Sketches (Average Results Returned)

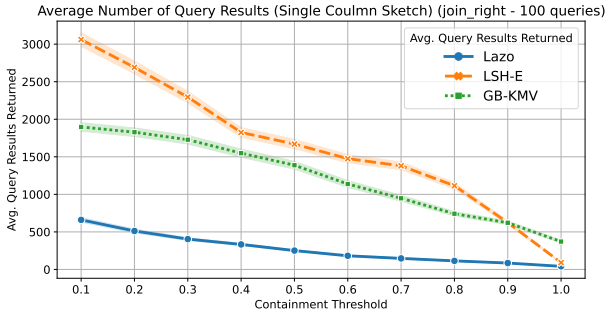
Figure 4.3: Average recall and number of results returned for Join (left side) queries vs. target containment thresholds for the `gittables` dataset. Error bands indicate the standard error of the mean over 100 queries.



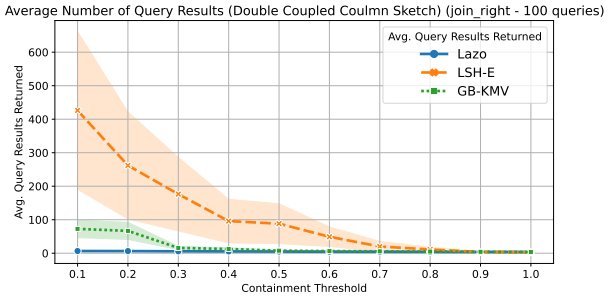
(a) Single Column Sketches (Average Recall)



(b) Two-Column Sketches (Average Recall)

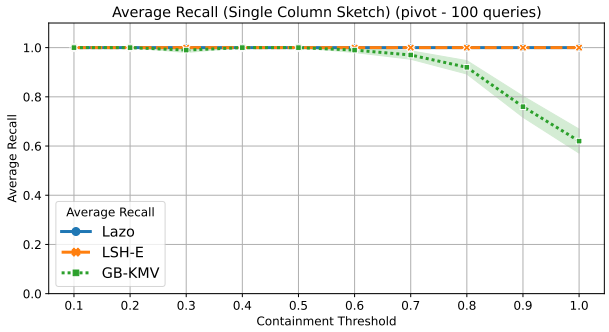


(c) Single Column Sketches (Average Results Returned)

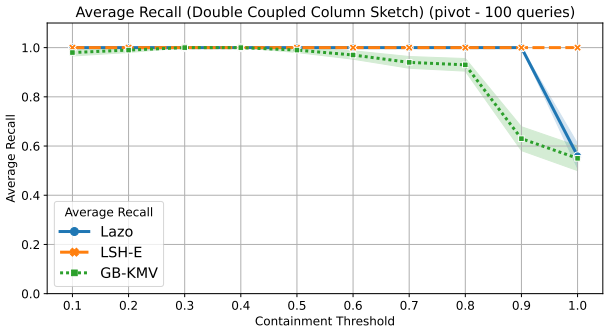


(d) Two-Column Sketches (Average Results Returned)

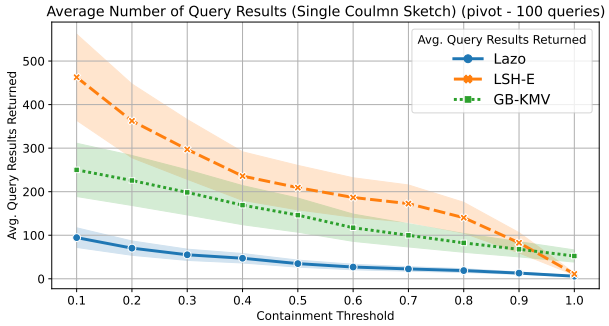
Figure 4.4: Average recall and number of results returned for Join (right side) queries vs. target containment thresholds for the `gittables` dataset. Error bands indicate the standard error of the mean over 100 queries.



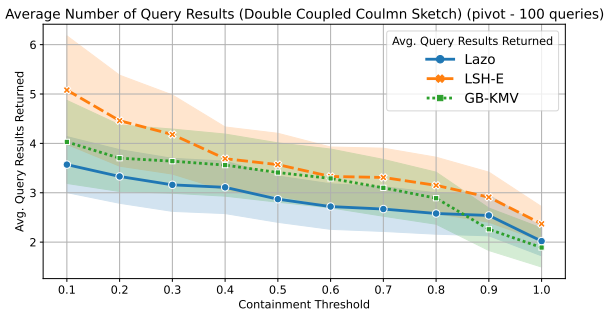
(a) Single Column Sketches (Average Recall)



(b) Two-Column Sketches (Average Recall)

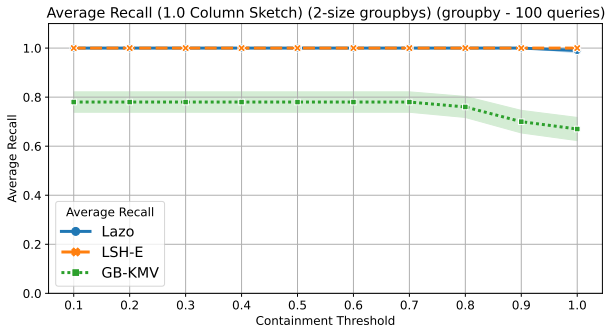


(c) Single Column Sketches (Average Results Returned)

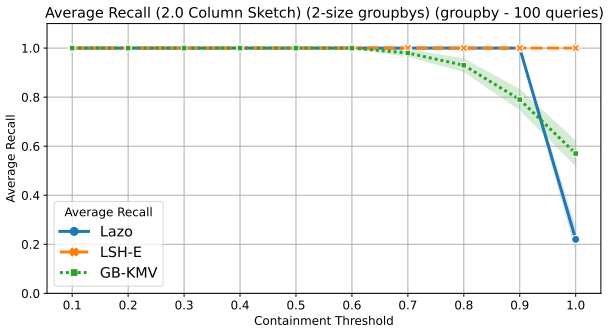


(d) Two-Column Sketches (Average Results Returned)

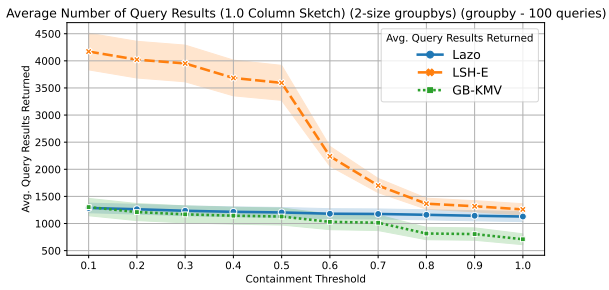
Figure 4.5: Average recall and number of results returned for Pivot queries vs. target containment thresholds for the `gittables` dataset. Error bands indicate the standard error of the mean over 100 queries.



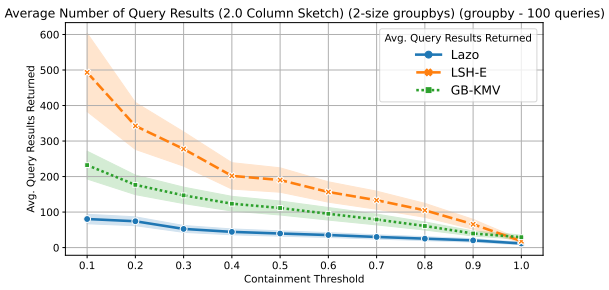
(a) Single Column Sketches (Average Recall)



(b) Two-Column Sketches (Average Recall)

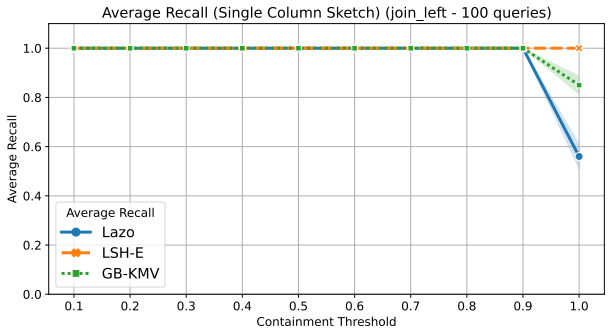


(c) Single Column Sketches (Average Results Returned)

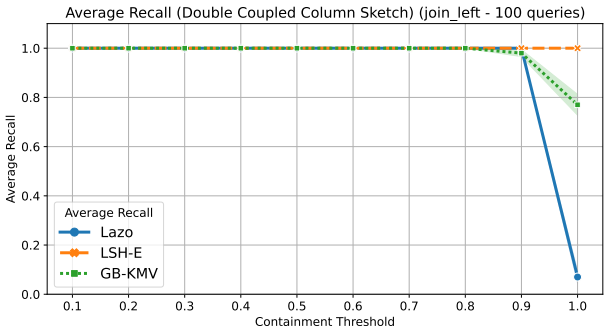


(d) Two-Column Sketches (Average Results Returned)

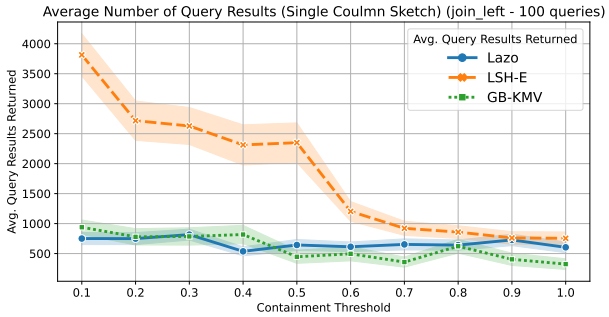
Figure 4.6: Average recall and number of results returned for 2-column groupby queries vs. target containment thresholds for the `synthetic` dataset. Error bands indicate the standard error of the mean over 100 queries.



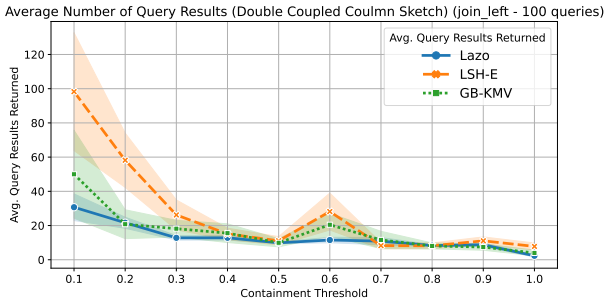
(a) Single Column Sketches (Average Recall)



(b) Two-Column Sketches (Average Recall)

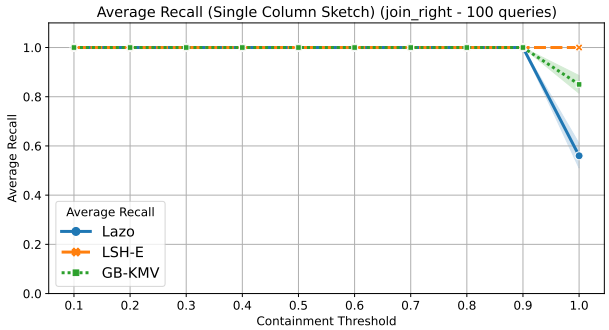


(c) Single Column Sketches (Average Results Returned)

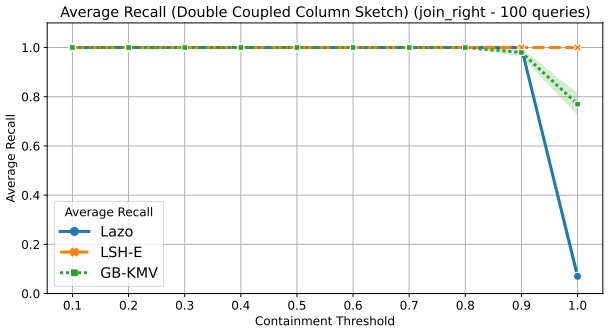


(d) Two-Column Sketches (Average Results Returned)

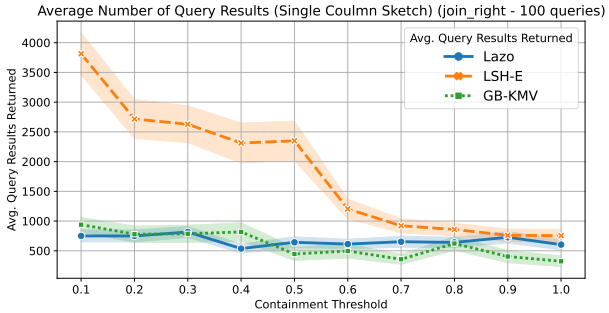
Figure 4.7: Average recall and the number of results returned for Join (left side) queries vs. target containment thresholds for the `synthetic` dataset. Error bands indicate the standard error of the mean over 100 queries.



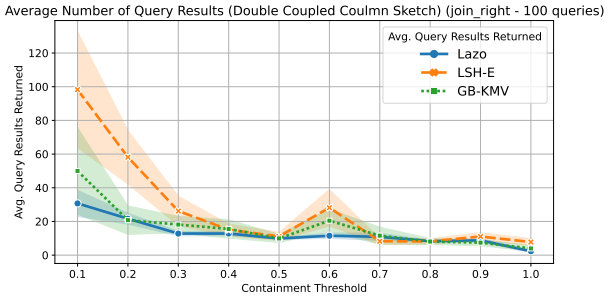
(a) Single Column Sketches (Average Recall)



(b) Two-Column Sketches (Average Recall)

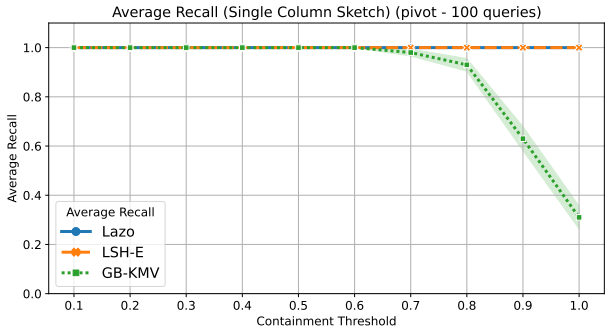


(c) Single Column Sketches (Average Results Returned)

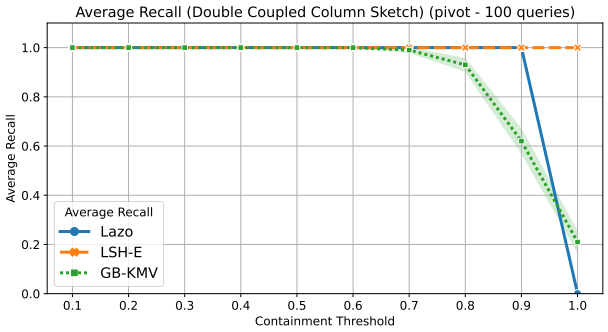


(d) Two-Column Sketches (Average Results Returned)

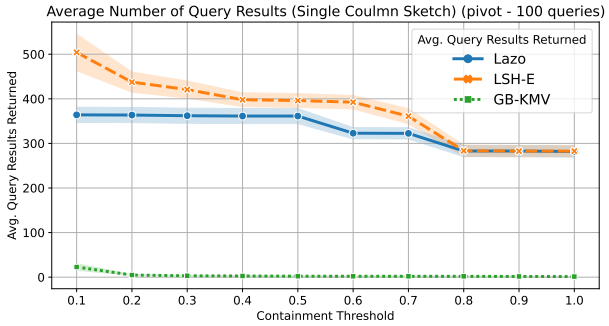
Figure 4.8: Average recall and the number of results returned for Join (right side) queries vs. target containment thresholds for the `synthetic` dataset. Error bands indicate the standard error of the mean over 100 queries.



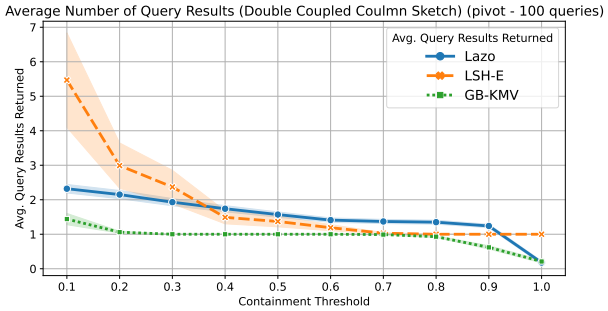
(a) Single Column Sketches (Average Recall)



(b) Two-Column Sketches (Average Recall)



(c) Single Column Sketches (Average Results Returned)



(d) Two-Column Sketches (Average Results Returned)

Figure 4.9: Average recall and the number of results returned for Pivot queries vs. target containment thresholds for the `synthetic` dataset. Error bands indicate the standard error of the mean over 100 queries.

Operation	*Num of Coupled Column Sketches*	*Method*	*Total Results Returned from Index*			*Ranked Recall @ k*		
			Minimum	*Mean*	*Maximum*	*k=1*	*k=5*	*k=10*
GroupBy	Single	gb-kmv	1.00	392.38	8522.00	0.64	0.72	0.76
		lazo	0.00	71.74	5965.00	**0.80**	0.89	0.89
		lsh-e	0.00	3169.93	45886.00	0.78	0.85	0.88
	Double	gb-kmv	0.00	579.95	15087.00	0.75	0.84	0.87
		lazo	0.00	214.26	4537.00	0.79	**0.89**	**0.90**
		lsh-e	0.00	1626.11	22449.00	**0.81**	**0.89**	**0.93**
Join (Left)	Single	gb-kmv	2.00	2695.97	6144.00	0.41	0.46	0.48
		lazo	2.00	849.97	5003.00	0.57	0.67	0.71
		lsh-e	2.00	5644.54	16230.00	0.49	0.61	0.65
	Double	gb-kmv	0.00	966.18	11908.00	**0.75**	**0.88**	**0.90**
		lazo	0.00	309.96	4825.00	0.66	0.74	0.77
		lsh-e	0.00	2056.97	18686.00	0.75	0.84	0.87
Join (Right)	Single	gb-kmv	2.00	2618.48	3765.00	0.33	0.38	0.44
		lazo	2.00	762.78	4995.00	0.50	0.70	0.77
		lsh-e	2.00	5515.43	16170.00	0.54	0.66	0.71
	Double	gb-kmv	1.00	567.89	12630.00	**0.83**	**0.90**	0.93
		lazo	0.00	256.88	6095.00	0.72	0.80	0.81
		lsh-e	1.00	2170.38	18591.00	**0.83**	0.89	**0.90**
Pivot	Single	gb-kmv	1.00	288.48	3111.00	0.69	0.81	0.84
		lazo	1.00	2.61	19.00	**0.89**	0.98	0.98
		lsh-e	1.00	3.99	70.00	0.83	0.98	0.99
	Double	gb-kmv	1.00	3.56	42.00	0.84	0.96	0.98
		lazo	1.00	3.11	40.00	0.88	**0.99**	**1.00**
		lsh-e	1.00	3.69	42.00	**0.89**	0.97	0.99

Table 4.7: Results of using similarity sketches to rank the results of each lineage query for the `gittables` dataset. The index was probed using the target containment threshold of 0.4. We report the minimum, mean and maximum number of results returned from the index for each query before ranking and the ranking recall rate at $k = 1, 5$, and 10.

provides the best ranking accuracy for most of the query types.

Diminishing Returns of 3-Column Sketches

We also explore the use of 3-column sketches to rank the results of the lineage queries. Firstly, we found that only Join queries can be readily expanded to use more than one column when probing the index. We also generated additional 3-column groupby queries so that we can directly probe a 3-column sketch. 3-column sketches do not provide any benefits for pivot lineage queries, since there is no guarantee that the value within a cell of a pivoted dataframe exists in the source dataframe (such cell values may be aggregated in case of multiple values

Operation	*Num of Coupled Column Sketches*	*Method*	*Total Results Returned from Index*			*Ranked Recall Rate @ k*		
			Minimum	*Mean*	*Maximum*	*k=1*	*k=5*	*k=10*
Join (Right)	Double	gb-kmv	1.00	567.89	12630	0.83	**0.90**	**0.93**
		lazo	0.00	256.88	6095	0.72	0.80	0.81
		lsh-e	1.00	2170.38	18591	0.83	0.89	0.9
	Triple	gb-kmv	0.00	970.38	26096	**0.84**	**0.90**	0.92
Join (Left)	Double	gb-kmv	0.00	966.18	11908	0.75	0.88	0.9
		lazo	0.00	309.96	4825	0.66	0.74	0.77
		lsh-e	0.00	2056.97	18686	0.75	0.84	0.87
	Triple	gb-kmv	0.00	692.22	30241	**0.88**	**0.93**	**0.94**
Groupby	Double	gb-kmv	1.00	21056.66	18722520	**0.83**	0.93	**0.94**
		lazo	0.00	14929.75	1200030	0.79	0.87	0.89
		lsh-e	0.00	249076.75	18923372	0.85	0.92	0.92
	Triple	gb-kmv	0.00	2915.05	172776	**0.83**	**0.94**	**0.94**

Table 4.8: Results of using 3-column sketches for Join and Groupby Queries generated on the `gittables` dataset. The index was probed using the target containment threshold of 0.4. We report the minimum, mean and maximum number of results returned from the index for each query before the ranking and the ranking recall rate at $k = 1, 5$, and 10.

that correspond to a specific index/column pair). Finally, of the three systems evaluated, only GB-KMV was able to generate a 3-column index that fit in the memory of our test machine. Our results are in Table 4.8. We find an improvement in ranking performance when using 3-column sketches, but the improvement is not significant enough to justify the additional time to generate the index and the increased memory footprint, as we shall see in Section 4.6.4.

4.6.3 Baseline Performance

As described in Section 4.5.2, we compare the evaluated methods using two existing baselines, MATE and a non-approximate baseline that does a brute-force containment search of all two-column set values. The brute-force method provides an an upper-bound on the accuracy of the evaluated methods. However, the brute-force method is extremely expensive, resulting in each query taking more than 10 minutes to complete, as compared to less than a second of query time for the methods presented in Section 4.6.1. A detailed report of the runtime is presented in Section 4.6.4, and the time taken to index and query the tables combined is

Operation	*Num of Coupled Column Sketches*	*Method*	*Total Results Returned from Index*			*Ranked Recall Rate @ k*		
			Minimum	*Mean*	*Maximum*	*k=1*	*k=5*	*k=10*
Groupby	Double	brute-force	–	–	–	0.89	0.97	0.98
		MATE	1.00	1870.35	3499.00	0.23	0.26	0.29
Join (Left)	Double	brute-force	–	–	–	0.87	0.94	0.94
		MATE	1.00	1317.09	3488.00	0.35	0.38	0.44
Join (Right)	Double	brute-force	–	–	–	0.74	0.83	0.86
		MATE	0.00	1506.66	3464.00	0.27	0.39	0.40
Pivot	Double	brute-force	–	–	–	0.85	0.94	0.97
		MATE	1.00	24.71	1795.00	0.8	0.95	0.95

Table 4.9: Baseline results (using an existing multi-column indexing technique, MATE, and a brute-force containment search) to rank the results of each lineage query using the `gittables` dataset. The index was probed using the target containment threshold of 0.4. We report the minimum, mean, and maximum number of results returned from the index for each query before ranking and the ranking recall rate at $k = 1, 5$, and 10. There are no indexing results for the brute force method, it searches the entire corpus of all two-column combinations of 5000 tables (a total of 529688 sets).

less than the total time taken to run the brute-force method (over 20 hours to complete all the queries).

4.6.4 Index Generation and Query Runtimes and Space Requirements

Table 4.11 lists the total time taken to generate the indices for each of the methods against the 5000 table dataset. Of all the methods, only gb-kmv was able to successfully generate a triple-column containment index. Table 4.12 lists the average time taken to query the indices and retrieve the candidate tables for each of the query types. All queries were completed in less than a second, as opposed to 687 seconds per query under the brute-force method. Table 4.13 list the time taken on average to compute the containment score for each candidate table, and the average time taken to rank a set of results for the double-column index. Table 4.15 shows the serialized size of the indices built by each method for the `gittables` dataset, which is considerably bigger than the dataset size of 200MB. However, in Table 4.16 we show that the size of the index remains static for LSH-E and Lazo methods for a growing

Operation	*Num of Coupled Column Sketches*	*Method*	*Total Results Returned from Index*			*Ranked Recall Rate @ k*		
			Minimum	*Mean*	*Maximum*	*k=1*	*k=5*	*k=10*
Groupby	Double	brute-force	–	–	–	0.55	0.76	0.81
		MATE	150	1376.40	3880.0	0.00	0.01	0.06
Join (Left)	Double	brute-force	–	–	–	1.00	1.00	1.00
		MATE	1.0	1243.92	4304.00	0.25	0.32	0.38
Join (Right)	Double	brute-force	–	–	–	1.00	1.00	1.00
		MATE	1.0	1576.28	4482.00	0.18	0.32	0.37
Pivot	Double	brute-force	–	–	–	1.00	1.00	1.00
		MATE	195	390.07	594.00	0.01	0.01	0.01

Table 4.10: Baseline results (using an existing multi-column indexing technique, MATE, and a brute-force containment search) to rank the results of each lineage query using the `synthetic` dataset. The index was probed using the target containment threshold of 0.4. We report the minimum, mean, and maximum number of results returned from the index for each query before ranking and the ranking recall rate at $k = 1, 5$, and 10. There are no indexing results for the brute force method, it searches the entire corpus of all two-column combinations of 5000 tables (a total of 278840 sets).

Indexing Phase	*Single*			*Double*			*Triple*
	lazo	*Ish-e*	*gb-kmv*	*lazo*	*lsh-e*	*gb-kmv*	*gb-kmv*
Hash Computation	176.95	177.87	311.65	1390.29	1345.42	1355.76	27589.67
Sketching and Indexing Time	103.64	233.14	24.84	68.09	1676.44	68.44	771.80
Total	280.58	411.02	336.49	1458.38	3021.85	1424.19	28361.47

Table 4.11: Total time taken (in Seconds) to generate the indexes using each sketch technique for Single, Double and Triple Column Sketches. Only gb-kmv was able to complete the generation of a 3-column index.

Query Phase	*Single*			*Double*			*Triple*
	lazo	*Ishe*	*gb-kmv*	*lazo*	*Ishe*	*gb-kmv*	*gb-kmv*
Hash Computation	0.00446	0.00464	0.004124	0.002907	0.002903	0.00239	0.002429
Query Upload	0.007112	0.007831	0.000718	0.063101	0.008287	0.000775	0.000762
Query Reply	0.006616	0.021797	0.005911	0.00984	0.016261	0.004511	0.017306
Total	0.018188	0.034268	0.010753	0.075848	0.027451	0.007676	0.020497

Table 4.12: Average time taken (in Seconds) to query the indexes using each sketch technique for Single, Double and Triple Column Sketches. Only gb-kmv was able to complete the generation of a 3-column index.

Method	Mean Time to Estimate JC	Mean Number of Candidates			Mean Time to Rank All Candidates (sec)		
		Join	Groupby	Pivot	Join	Groupby	Pivot
lsh-e	0.000595	2170	1626.11	3.69	2.582	0.968	0.002
lazo	0.000578	309.96	806.59	3.11	0.358	0.466	0.002
gb-kmv	0.000358	966.18	573.95	3.56	0.692	0.205	0.001

Table 4.13: Average time taken (in Seconds) to rank the candidate results using each method, using a containment threshold of 0.4. The mean time to estimate Jaccard Containment (JC) is per candidate pair.

Method	**Index Generation**	**Time Per Query**		**Total Runtime**
		Index Query	**Candidate Ranking Time**	
lsh-e	3021.85	0.027	1.184	363.3
lazo	1458.38	0.076	0.826	270.6
gb-kmv	1424.19	0.008	0.299	92.1
MATE	660	–	242.200	72660.0
brute-force	–	–	687.000	206100.0

Table 4.14: Runtime comparison of our proposed methods with the baseline methods. Total time reported is to run all 300 queries.

Dataset Size	200 MB		
Index Sizes (w/ Double Column Sketch)			
Method	Single	Double	Triple
lsh-e	1019 MB	10 GB	–
lazo	383 MB	3.8 GB	–
gb-kmv	95 MB	905 MB	5.6 GB

Table 4.15: Total size of the index for each method on the GitTables Dataset

Dataset Size (100 Tables)	5.9 MB	34 MB	236 MB
Method	Index Size (w/ Double Column Sketch)		
	100 Rows/Table	1K Rows/Table	10K Rows/Table
lsh-e	813 MB	806 MB	796 MB
lazo	199 MB	195 MB	183 MB
gb-kmv	12 MB	86 MB	860 MB

Table 4.16: Total size of the index for each method on a synthetic 100-table dataset, with a variable number of rows.

4.7 Conclusions

In this chapter, we performed a thorough evaluation of various sketch techniques to answer lineage queries. We showed that we can effectively avoid the expensive pairwise computation of similarity metrics by constructing LSH and KMV-based sketches of columns and showed the benefit of sketching multiple columns at a time. We found that sketching combinations of two columns and indexing them for a target containment threshold of 0.4 allows for an optimal balance of accuracy, while reducing the number of candidates returned by the query. We also showed that we can use the constructed sketches to compute an approximate containment score for each candidate, which can be used to rank the candidates. We then showed that the proposed approach can be used to answer lineage queries in less than one second, while a brute-force search can take over 10 minutes for each query.

CHAPTER 5

THE FUZZYDATA WORKFLOW SPECIFICATION SYSTEM

The rise in popularity of data science and machine learning has catapulted dataframe-based tools such as R [37] and the `pandas` [82] library for Python to the forefront of the data science practice, enabling large and small organizations to extract insight from data quickly. These tools allow for ad-hoc ingestion and transformation of small to moderate amounts of data, with the flexibility of integrating arbitrary code or machine learning workflows to the data analysis workflows.

The dataframe model has thus become a popular programmatic interface to encode data manipulation and transformation operations. Typical dataframe APIs enable both spreadsheet-style manipulation and relational-style operations and on tabular data, allowing for data cleaning and wrangling during the ETL process while retaining most of the powerful querying semantics for analytical and visualization tasks. With the rise in popularity of this model, expectations of such systems have grown, resulting in the growing demand for systems that support these APIs on massive tabular datasets that may not fit a typical desktop or laptop computers' memory. Projects such as dask [75], ray [61], modin [65], SparkDataFrames [6], either provide interfaces to python scientific tools such as NumPy and pandas or implement their own version of the dataframe API to work on parallel/distributed environments with larger datasets.

However, there is an acute lack of workload generation, benchmarking, and testing frameworks that allow for comparisons between these systems or evaluate optimizations within these systems. Prior work encompassing dataframe systems either lacks a comprehensive evaluation or is evaluated against handwritten, static queries run against specific datasets [41, 65, 78, 89]. Existing systems for relational benchmarking (such as TPC [19]), or for NoSQL/key-value stores (such as YCSB [18]) do not adequately represent the workflows typically executed within these dataframe environments. DataFrame systems are a

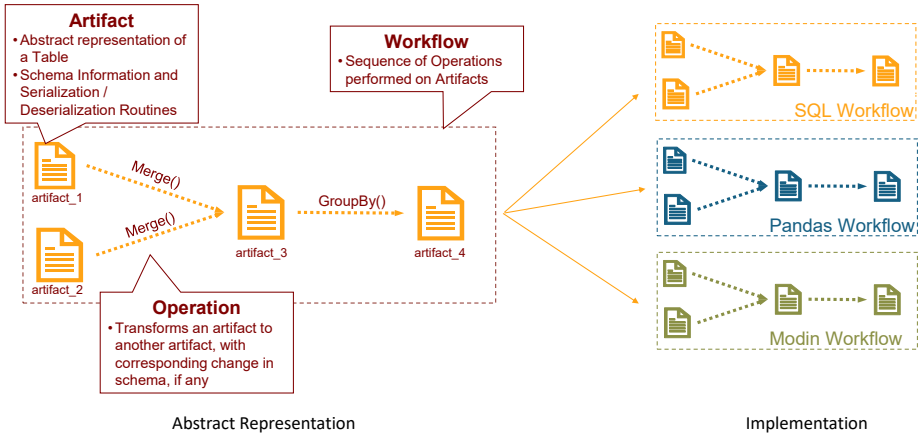


Figure 5.1: The design of FuzzyData.

kind of middle-ground between relational systems and raw key-value stores; they do not require strict schema definitions or DDL steps like the former but are not entirely schema-less and support a vastly broader range of queries and transformations, unlike the latter.

Correctness guarantees and performance bottlenecks are harder to pinpoint with dataframe-based systems because of the diversity in query language and semantics, supported data types, serialization formats, memory layouts and execution engines. For example, the `modin` project advertises performance advantages over `pandas` by parallelizing common dataframe operations over multiple cores, ideally offering speedups of Nx where N is the number of cores available [65]. However, during the time of writing, there were $\sim$ 45 open performance issues on the `modin` GitHub page [66], with users providing empirical evidence of `modin`'s performance being worse than `pandas` for the same operation. A more robust workload generation suite can supplement regression testing practices to catch performance and correctness bugs during the development lifecycle. Thus, there is a need for a framework that allows for synthetic workload generation, allowing for reproducibility, scalability, and stress testing of dataframe systems while using some of the best design practices and patterns from earlier benchmark and workload generator suites designed for relational [19] and NoSQL systems [18].

Our contribution, FuzzyData system [1], allows for data analysis workflows to be manually specified or randomly generated, scaled, and replayed on various dataframe system clients, allowing for "fuzzy" testing and direct performance comparisons between dataframe systems. Our contributions are as follows:

- An implementation-agnostic data model to represent data artifacts, operations, and workflows that can be used to express common data workflow operations in JSON (a human-parsable format).
- A module that generates realistic and diverse tabular data artifacts and workflows consisting of typical data transformations.
- Plug-in clients for FuzzyData that enable workflows to be generated and replayed on `SQLite`, `pandas`, `modin-dask`, and `modin`-ray. We show how, with a few lines of code, FuzzyData can be extended to more data operations or clients by writing simple plug-ins.
- We demonstrate the use of FuzzyData in typical testing/ workload generation and replay scenarios and show the types of insights that can be derived from its use. FuzzyData was used to find one correctness bug (Section 5.2.1) which was reported to the `modin` developers, as well as performance issues in `modin`.

5.1 FuzzyData Design

This section discusses the abstract design of the FuzzyData system and its implementation, along with the random table/workflow generators and the three clients we have implemented using the FuzzyData abstract model.

1. Source Code is available at `https://github.com/suhailrehman/fuzzydata`

Transformation	Description	Generation Constraints	pandas	SQLite	modin
`load`	Load (de-serialize) an artifact from a filesystem location	–	✓	✓	✓
`select`	Selects rows based on a filter condition	`numeric` ≥ 1	✓	✓	✓
`apply`	Create a new derived column as a scalar function applied to an existing numeric column	`numeric` > 1	✓	✓	✓
`project`	Project a set of columns	–	✓	✓	✓
`sample`	Randomly select rows from the artifact	–	✓	✓	✓
`join`	Inner Join with another artifact based on a key column	`joinable` ≥ 1	✓	✓	✓
`pivot`	Pivot the artifact by index, column and values	`groupable` ≥ 2 and `numeric` ≥ 1	✓	–	✓
`groupby`	Groupby set of `group_columns` and apply an aggregate `function` on another set of `agg_columns`	`groupable` ≥ 1	✓	✓	✓
`fill`	Replace an `old` value in a column with another value	–	✓	–	✓
`materialize`	Execute stacked transformations to produce a new artifact	–	✓	✓	✓
`serialize`	Dump the contents of an artifact to disk	–	✓	✓	✓

Table 5.1: Transformation Implementation Matrix. The *Generation Constraints* column lists the minimum number of columns of each type required to generate each transformation using the random workflow generator.

5.1.1 Data Model

The abstract model that we have created allows us to specify *artifacts*, *operations*, and *workflows* in an implementation-agnostic manner:

Artifact

An *artifact* is an abstract representation of a dataframe in FuzzyData. The data model representing an artifact stores the metadata about an artifact, such as:

- Schema related information: A mapping of column label $\rightarrow$ column type
- Artifact representation: Information about the artifact representation in memory or as a table/view in a database
- Serialization routines: File system paths to store serialized versions of artifacts, as well as function pointers to serialization routines necessary to load/store the artifact from disk

The abstract model for an artifact gives FuzzyData the flexibility to support various data access and manipulation APIs. For example, a `pandas` implementation of the artifact class would typically contain the dataframe label, and a filesystem path to (de)-serialize the dataframe artifact from/to disk. On the other hand, a SQL implementation of the artifact class will have much of the same information as the `pandas`' implementation with additional fields such as the database table/view names, schema, and the specific SQL query needed to retrieve a view of the artifact from a database.

Operation

An *operation* is the abstract representation of one or more transformation(s) that takes one or more artifact(s) as input and transforms them into a new artifact. The *operation* model consists of metadata such as the source artifact labels, a list of transformations and their arguments, and the label to be assigned to the new destination artifact created as a result of the operation. The *operation* interface consists of abstract specifications of each of the transformations with their arguments and how each of the transformations change the schema of the artifact. Note that the actual query or code required to execute the transformations for a particular operation is defined in the client implementation and left out the abstract model. Table 5.1 has a listing of all the transformations that are currently implemented in FuzzyData. The operator abstraction is designed to be extensible, allowing for additional transformations or UDFs to be added to FuzzyData to support even more diverse workload types in the future. Transformations can be defined and implemented for data processing steps such as one-hot-encoding or normalization.

Workflow

Formally, a *workflow* is a directed-acyclic-graph (DAG) $\mathcal{W} = (V, E)$, where the set of vertices V is the set of artifacts and the set of edges E are the operations that are used to transform

an artifact $v_1 \in V$ to another artifact $v_2 \in V$. Artifacts that have no incoming edges are *source artifacts*, which are either loaded from disk, or randomly generated. Note that the set of edges E in FuzzyData is *ordered*, and is the sequence of operations that is implicit when encoding an existing workflow (Section 5.1.2), or is the order of operations that is randomly generated by FuzzyData (Section 5.1.4).

In FuzzyData, the *workflow* interface encapsulates information about the workflow, sequence of operations, and the final workflow directed acyclic graph, with enough information for FuzzyData to load and replay specific workflows on different clients (Section 5.1.6). As a workflow is replayed, the wall-clock time taken to load artifacts from the disk and replay the individual operations are recorded for analysis and visualization (Section 5.1.5).

5.1.2 Implementation

FuzzyData is implemented in Python, with `pandas` [82] used as the internal dataframe representation used to represent artifacts during generation (Section 5.1.3) and `networkx` [42] used to represent the workflow graph. Users of FuzzyData have the option to specify a workflow or have FuzzyData randomly generate a workflow (Section 5.1.3) with generation parameters listed in Table 5.3.

FuzzyData supports workflow specifications to be provided in a JSON file. The workflow specification includes a name for the workflow and a list of operations. Each operation consists of a list of source artifacts, the operation type, and the arguments for the operation. When a workflow specification and the source artifacts are loaded into a FuzzyData client, the following actions are performed: (a) The operation list is loaded, (b) the operations are performed in the order specified in the file, and (c) any artifact that is currently not in memory and is needed for the next operation will be deserialized from disk.

index	iso8601	cryptocurrency_code	pyint	rn
0	2004-09-21T09:46:38	NEO	1453	10
1	2016-04-07T21:19:57	BCN	877	7
2	1973-08-09T20:35:50	USDT	8198	8
3	1985-08-24T17:07:41	EOS	7492	10
4	1979-06-16T21:01:12	NEM	157	6
5	2020-12-30T03:19:01	IOTA	5439	12
6	1995-05-03T04:56:00	BCH	2348	13
7	1972-09-02T20:03:53	XRP	1244	13
8	1990-12-27T10:33:05	ETC	8354	11
9	2017-07-03T20:19:32	WAVES	9717	11

Table 5.2: An example artifact generated using FuzzyData. Column labels indicate the faker provider used to generate the values for the column, except for `rn`, which is shorthand for `random_number`. In this table, `pyint` and `rn` are *numeric* columns, while `cryptocurrency_code` is a *groupable, joinable* and *string* column.

5.1.3 Generating Random Artifacts

FuzzyData provides a table generation system that leverages the Python faker library [28]. The faker integration allows FuzzyData to generate diverse tables with a wide range of columns and data, simulating real-world datasets with realistic data values. A user can supply the number of rows (r) and columns (c) to be generated, and FuzzyData by default randomly chooses columns types to generate from faker's portfolio and generates a mix of string and numeric columns with various levels of cardinality. These column types are labeled as one or more of the following types: *numeric, string, groupable* and *joinable*; these labels are essential when enumerating the space of possible operations that can be performed on a given artifact (enumerated in Table 5.1. Table 5.2 shows an example table generated with parameters ($r = 10, c = 4$) with a mix of column types and labels. This table can now be used to generate additional artifacts based on the rules described below in Section 5.1.4.

5.1.4 Generating and Replaying Random Workflows

FuzzyData also supports the generation of entirely random workflows, including randomly

generated source artifacts (using the generator described in Section 5.1.3) and random operations. Users have the option of supplying the following parameters (Table 5.3):

Parameter	Description
n	Number of Artifacts
r	Base Artifact Number of Rows
c	Base Artifact Number of Columns
b	Workflow Branching Factor
T	Set of Allowed Transformations
m	Materialization Rate

Table 5.3: Parameters for generating synthetic workflows.

Users can specify the total number of artifacts to be generated (n) and the number of rows (r) and columns (c) of the base artifact, and a few schema options, if any. The schema options in the generator can provide specific column types to be generated or an exclusion list of the types of columns to avoid. If no schema options are provided, the generator will try to evenly distribute the types of columns in order to maximize the available operations that can be performed (Table 5.1). Once the base artifact is generated, subsequent artifacts are generated as follows:

1. A random artifact is first selected from the set of artifacts already generated in the workflow. The *workflow branch factor* parameter (b) can be used to control the structure of the final workflow graph by biasing the selection probability towards artifacts that were more recently generated. Formally, given we have a list of n' artifacts that have been generated so far, FuzzyData selects the next artifact from the list with index i to be modified with probability (Equation 5.1.1)

$$P[i] = \left(\frac{b}{e^{bn'} - 1}\right) e^{bi} \tag{5.1.1}$$

Thus, with $b = 1.0$, the probability is always skewed towards the newest artifact that is generated, resulting in workflow that is more linear, and with $b = 0.01$, there is

a uniform probability, resulting in a more branched workflow (such as the example workflow generated in Figure 5.2).

2. Once an artifact has been selected as the source artifact for an operation, FuzzyData generates a set of possible transformations that can be performed on the artifact, subject to the set of allowed transformations T. The source artifact schema map is inspected, and the number of columns of each type is enumerated. It then follows the rules listed in Table 5.1 and generates a set of transformations with randomized arguments for each transformation.

3. From the set of possible transformations that can now be performed on a source artifact, a random option is selected and added to the operation chain. The expected schema map that results from the transformation is updated, ensuring any future stacked transformations have accurate schema representation, even if the operation is not materialized.

4. Steps 2 and 3 are repeated until we have m transformations in the current operation chain, i.e. we have reached the materialization rate.

5. The operation chain is materialized (executed) to generate the next artifact. This means that FuzzyData will use an attached client (Section 5.1.6) to execute the chain of operations and generate the resulting artifact. If the generator selects a merge operation to be performed, an additional random artifact that contains the *joinable* column (from the merge arguments) is generated and added to the workflow to simulate an inner PK-FK join.

6. The steps above are repeated until we have generated n artifacts and there are no more artifacts that remain to be generated.

7. Once the workflow is generated, it can be written to disk, which serializes all generated

artifacts to disk, writes the workflow graph, and generates a JSON specification of the workflow, which can be loaded and replayed by FuzzyData clients in the future.

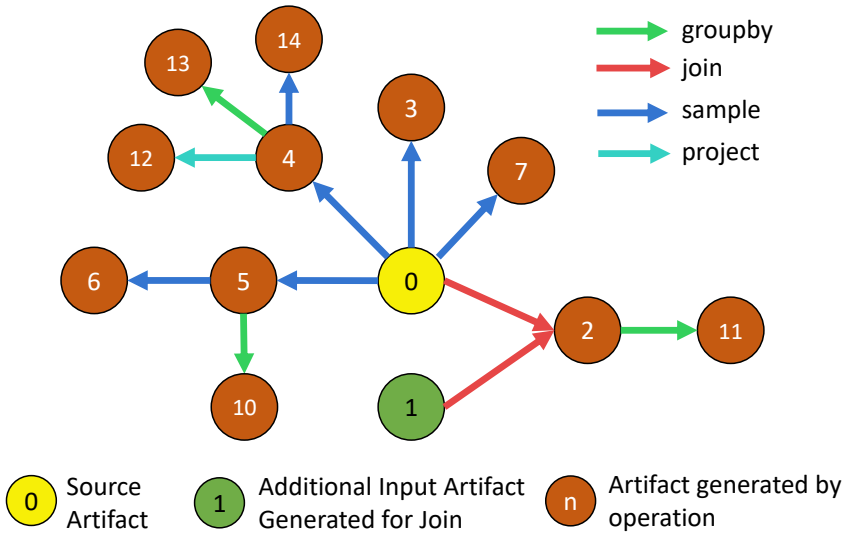


Figure 5.2: DAG of a randomly generated workflow using the generator described in Section 5.1.4. The generation parameters used were $(n = 15, r = 1000, c = 20, b = 0.01, m = 1)$.

Figure 5.2 is an example of the DAG associated with a workflow generated in FuzzyData. While step 2 in our workflow generator currently implements a simple, randomized, rule-based approach for generating operations on artifacts, we foresee the ability to use intelligent, ML-based approaches to automatically generate meaningful operations given the source artifact, similar to AutoSuggest [91], implemented via the plug-in architecture of FuzzyData.

5.1.5 Instrumentation

FuzzyData includes timing hooks implemented into the *workflow* interface for artifact generation, loading, and operations. These hooks can be used to collect runtime performance information for the execution of the workflow. This feature makes FuzzyData valuable for system testing and performance analysis and evaluation of potential bottlenecks in the systems being evaluated.

5.1.6 Clients Implemented

Implementing a FuzzyData client requires implementing each of the abstract interfaces described in Section 5.1. The *artifact* interface requires implementing load/store routines and a hook to the random artifact generation function. The *operation* interface requires implementing each of the operations listed in Table 5.1, using the specific syntax of the client's query language or DSL. The *workflow* interface provides space for setting up parameters used by all the artifacts in the workflow, like filesystem paths or database/execution engine parameters. Thus, with a few lines of code, a client can be implemented in FuzzyData that can generate and replay workflows. Three clients have been implemented in FuzzyData:

`pandas`: The `pandas` client is a dataframe-based implementation of the three abstract interfaces described in Section 5.1. The *operation* implementation generates dataframe transformation code as a string of chained dataframe function calls. To materialize an artifact, the string containing all the dataframe transformations is evaluated and run against the source dataframe artifact.

We show a compact client implementation of FuzzyData on `pandas` in Listing 5.1, which implements the interfaces defined in Section 5.1, along with the operations listed in Section 5.1.

Listing 5.1: `fuzzydata` implementation of a pandas client

```
1  import logging
2  from typing import List
3
4  import pandas
5
6  from fuzzydata.core.artifact import Artifact
7  from fuzzydata.core.generator import generate_table
8  from fuzzydata.core.operation import Operation, T
9  from fuzzydata.core.workflow import Workflow
10
11 logger = logging.getLogger(__name__)
12
```

```
13
14  class DataFrameArtifact(Artifact):
15
16      def __init__(self, *args, **kwargs):
17          self.pd = kwargs.pop("pd", pandas)
18          from_df = kwargs.pop("from_df", None)
19          super(DataFrameArtifact, self).__init__(*args, **kwargs)
20          self._deserialization_function = {
21              'csv': self.pd.read_csv
22          }
23          self._serialization_function = {
24              'csv': 'to_csv'
25          }
26
27          self.operation_class = DataFrameOperation
28          self.table = None
29          self.in_memory = False
30
31          if from_df is not None:
32              self.from_df(from_df)
33
34      def generate(self, num_rows, schema):
35          self.table = generate_table(num_rows, column_dict=schema, pd=self.pd)
36          self.schema_map = schema
37          self.in_memory = True
38
39      def from_df(self, df):
40          self.table = self.pd.DataFrame(df)
41          self.in_memory = True
42
43      def deserialize(self, filename=None):
44          if not filename:
45              filename = self.filename
46
47          self.table = self._deserialization_function[self.file_format](filename)
48          self.in_memory = True
49
50      def serialize(self, filename=None):
51          if not filename:
52              filename = self.filename
```

```
53
54          if self.in_memory:
55              serialization_method = getattr(self.table, self._serialization_function[self
    .file_format])
56              serialization_method(filename)
57
58      def destroy(self):
59          del self.table
60
61      def to_df(self) -> pandas.DataFrame:
62          return self.table
63
64      def __len__(self):
65          if self.in_memory:
66              return len(self.table.index)
67
68
69  class DataFrameOperation(Operation['DataFrameArtifact']):
70      def __init__(self, *args, **kwargs):
71          self.artifact_class = kwargs.pop('artifact_class', DataFrameArtifact)
72          super(DataFrameOperation, self).__init__(*args, **kwargs)
73          self.code = 'self.sources[0].table' # Starting point for chained code generation
    .
74
75      def apply(self, numeric_col: str, a: float, b: float) -> DataFrameArtifact:
76          super(DataFrameOperation, self).apply(numeric_col, a, b)
77          new_col_name = f"{numeric_col}__{int(a)}x_{int(b)}"
78          return f'.assign({new_col_name} = lambda x: x.{numeric_col}*{a}+{b})'
79
80      def sample(self, frac: float) -> DataFrameArtifact:
81          super(DataFrameOperation, self).sample(frac)
82          return f'.sample(frac={frac})'
83
84      def groupby(self, group_columns: List[str], agg_columns: List[str], agg_function:
    str) -> T:
85          super(DataFrameOperation, self).groupby(group_columns, agg_columns, agg_function
    )
86          return f'[{group_columns+agg_columns}].groupby({group_columns}).{agg_function}()
    .reset_index()'
87
```

```
88      def project(self, output_cols: List[str]) -> T:
89          super(DataFrameOperation, self).project(output_cols)
90          return f'[{output_cols}]'
91
92      def select(self, condition: str) -> T:
93          super(DataFrameOperation, self).select(condition)
94          return f'.query("{condition}")'
95
96      def merge(self, key_col: List[str]) -> T:
97          super(DataFrameOperation, self).merge(key_col)
98          return f'.merge(self.sources[1].table, on="{key_col}")'
99
100     def pivot(self, index_cols: List[str], columns: List[str], value_col: List[str],
        agg_func: str) -> T:
101         super(DataFrameOperation, self).pivot(index_cols, columns, value_col, agg_func)
102         return f'.pivot_table(index={index_cols}, columns={columns},values={value_col},
        aggfunc={agg_func})'
103
104     def fill(self, col_name: str, old_value, new_value):
105         super(DataFrameOperation, self).fill(col_name, old_value, new_value)
106         return f'.replace({{ "{col_name}": {old_value} }}, {new_value})'
107
108     def chain_operation(self, op, args):
109         self.code += getattr(self, op)(**args)
110         super(DataFrameOperation, self).chain_operation(op, args)
111
112     def materialize(self, new_label):
113         new_df = eval(self.code)
114         super(DataFrameOperation, self).materialize(new_label)
115         return self.artifact_class(label=self.new_label,
116                                    from_df=new_df,
117                                    schema_map=self.current_schema_map)
118
119
120 class DataFrameWorkflow(Workflow):
121     def __init__(self, *args, **kwargs):
122         super(DataFrameWorkflow, self).__init__(*args, **kwargs)
123         self.artifact_class = DataFrameArtifact
124         self.operator_class = DataFrameOperation
125
```

```
126     def initialize_new_artifact(self, label=None, filename=None, schema_map=None):
127         return DataFrameArtifact(label, filename=filename, schema_map=schema_map)
```

`modin`: The `modin` client (Listing 5.2) is an extension of the `pandas`' client, in which all `pandas` operations are simply routed through the `modin` library [65]. The client provides the user an option of specifying either a `dask` or `ray` execution engine, along with initialization parameters such as number of workers, which can distribute the dataframe operations on parallel hardware.

Listing 5.2: `fuzzydata` implementation of a modin client

```
1  import modin.pandas as mpd
2  from modin.config import Engine
3
4  from fuzzydata.clients.pandas import DataFrameArtifact, DataFrameOperation
5  from fuzzydata.core.workflow import Workflow
6
7
8  class ModinArtifact(DataFrameArtifact):
9
10     def __init__(self, *args, **kwargs):
11         kwargs.update({'pd': mpd})  # Force loading of the modin pandas library
12         super(ModinArtifact, self).__init__(*args, **kwargs)
13         self._deserialization_function = {
14             'csv': self.pd.read_csv
15         }
16         self._serialization_function = {
17             'csv': 'to_csv'
18         }
19
20         self.operation_class = DataFrameOperation
21
22
23 class ModinWorkflow(Workflow):
24     def __init__(self, *args, **kwargs):
25         self.modin_engine = kwargs.pop('modin_engine', 'dask')
26         super(ModinWorkflow, self).__init__(*args, **kwargs)
27         self.artifact_class = ModinArtifact
```

```
28          self.operator_class = DataFrameOperation
29
30          if self.modin_engine == 'dask':
31              from dask.distributed import Client
32              processes = kwargs.pop('processes', True)
33              Client(processes=processes)
34          else:
35              import ray
36              ray.init(ignore_reinit_error=True)
37
38          Engine.put(self.modin_engine)
39
40      def initialize_new_artifact(self, label=None, filename=None, schema_map=None):
41          return ModinArtifact(label, filename=filename, schema_map=schema_map)
```

`SQLite`: The `SQLite` client (Listing 5.3) creates a file-based embedded database on disk and uses the generator described in Section 5.1.3 to generate base table artifacts. The base table artifacts are then operated upon by SQL queries constructed for each operation to generate other artifacts as views in the database. We use nested sub-queries to chain all of the transformations into an operation. All table views are serialized to disk as CSV files at the end of the workflow. A notable exclusion from the `SQLite` client is the `pivot` operation since generic pivots in a generic SQL dialect are quite complex to generate. The client initialization parameters include an SQL connection string, so this client can be used with other SQL databases as well. In case of a SQL dialect mismatch for other database systems, this client could be extended to re-implement any of the incompatible operations.

Listing 5.3: `fuzzydata` implementation of a `SQLite` client

```
1  import math
2  from typing import List
3
4  import pandas
5  import sqlalchemy
6  import logging
7
8  from fuzzydata.core.artifact import Artifact
```

```
from fuzzydata.core.generator import generate_table
from fuzzydata.core.operation import Operation, T
from fuzzydata.core.workflow import Workflow

logger = logging.getLogger(__name__)

class SQLArtifact(Artifact):

    def __init__(self, *args, **kwargs):
        self.sql_engine = kwargs.pop("sql_engine")
        self.from_sql = kwargs.pop("from_sql", None)
        self.sync_df = kwargs.pop("sync_df", False)
        from_df = kwargs.pop("from_df", None)

        super(SQLArtifact, self).__init__(*args, **kwargs)

        self.operation_class = SQLOperation
        self.pd = pandas

        self._deserialization_function = {
            'csv': self.pd.read_csv
        }
        self._serialization_function = {
            'csv': 'to_csv'
        }

        self._get_table = f'SELECT * FROM `{self.label}`'
        self._del_table = f'DROP TABLE IF EXISTS `{self.label}`'
        self._num_rows = f'SELECT COUNT(*) FROM `{self.label}`'

        if self.from_sql:
            self.sql_engine.execute(self.from_sql)
            if self.sync_df:
                self.table = self.pd.read_sql(self._get_table, con=self.sql_engine)

        elif from_df is not None:
            self.from_df(from_df)

    def generate(self, num_rows, schema):
        df = generate_table(num_rows, column_dict=schema)
```

```
49          df.to_sql(self.label, con=self.sql_engine, if_exists='replace')
50          self.schema_map = schema
51          if self.sync_df:
52              self.table = df
53          # self.in_memory = True
54
55      def from_df(self, df):
56          df.to_sql(self.label, con=self.sql_engine, if_exists='replace', index=False)
57          if self.sync_df:
58              self.table = df
59
60      def deserialize(self, filename=None):
61          if not filename:
62              filename = self.filename
63
64          df = self._deserialization_function[self.file_format](filename)
65          df.to_sql(self.label, con=self.sql_engine, if_exists='replace')
66          if self.sync_df:
67              self.table = df
68          # self.in_memory = True
69
70      def serialize(self, filename=None):
71          if not filename:
72              filename = self.filename
73
74          df = self.pd.read_sql(self._get_table, con=self.sql_engine)
75          serialization_method = getattr(df, self._serialization_function[self.file_format
        ])
76          serialization_method(filename)
77
78      def destroy(self):
79          if self.sync_df:
80              del self.table
81          self.sql_engine.execute(self._del_table)
82
83      def to_df(self):
84          return self.pd.read_sql(self._get_table, con=self.sql_engine)
85
86      def __len__(self):
87          return self.sql_engine.execute(self._num_rows).first()[0]
```

```
88
89
90  class SQLOperation(Operation['SQLArtifact']):
91
92      def __init__(self, *args, **kwargs):
93          self.artifact_class = kwargs.pop('artifact_class', SQLArtifact)
94          super(SQLOperation, self).__init__(*args, **kwargs)
95          self.agg_function_dict = {
96              'mean': 'AVG'
97          }
98          self.code = f"SELECT * FROM `{self.sources[0].label}`"
99
100     def sample(self, frac: float) -> SQLArtifact:
101         super(SQLOperation, self).sample(frac)
102         num_rows = len(self.sources[0])
103         sample_rows = math.ceil(num_rows*frac)
104         sql_sample_stmt = f"SELECT * FROM {{source}} ORDER BY RANDOM() " \
105                           f"LIMIT {sample_rows} "
106         return sql_sample_stmt
107
108     def apply(self, numeric_col: str, a: float, b: float) -> SQLArtifact:
109         super(SQLOperation, self).apply(numeric_col, a, b)
110         new_col_name = f"{numeric_col}__{a}x_{b}"
111         sql_apply_stmt = f"SELECT *, (`{numeric_col}` * {a}) + {b} AS `{new_col_name}` "
     \
112                          f"FROM {{source}}"
113         return sql_apply_stmt
114
115     def groupby(self, group_columns: List[str], agg_columns: List[str], agg_function:
    str) -> SQLArtifact:
116         super(SQLOperation, self).groupby(group_columns, agg_columns, agg_function)
117         group_cols_str = ', '.join([f"`{x}`" for x in group_columns])
118
119         # Translate the aggregate function string if required
120         if agg_function in self.agg_function_dict:
121             agg_function = self.agg_function_dict[agg_function]
122
123         agg_cols_str = f"{','.join([f'{agg_function}(`{x}`) AS `{x}`' for x in
    agg_columns])}"
124         sql_groupby_stmt = f"SELECT {group_cols_str}, {agg_cols_str} " \
```

```
125                             f"FROM {{source}} " \
126                             f"GROUP BY {group_cols_str} "
127         return sql_groupby_stmt
128
129     def project(self, output_cols: List[str]) -> T:
130         super(SQLOperation, self).project(output_cols)
131
132         project_predicate = ','.join([f"`{x}`" for x in output_cols])
133
134         sql_project_stmt = f"SELECT {project_predicate} FROM {{source}} "
135         return sql_project_stmt
136
137     def select(self, condition: str) -> T:
138         super(SQLOperation, self).select(condition)
139         sql_select_stmt = f"SELECT * FROM {{source}} " \
140                           f"WHERE {condition}"
141         return sql_select_stmt
142
143     def merge(self, key_col: List[str]) -> T:
144         super(SQLOperation, self).merge(key_col)
145         sql_select_stmt = f"SELECT * FROM {{source}} " \
146                           f"INNER JOIN `{self.sources[1].label}` " \
147                           f"USING (`{key_col}`)"
148         return sql_select_stmt
149
150     def pivot(self, index_cols: List[str], columns: List[str], value_col: List[str],
    agg_func: str) -> T:
151         raise NotImplementedError('Generic Pivots in SQL are Hard!')
152
153     def fill(self, col_name: str, old_value, new_value):
154         super(SQLOperation, self).fill(col_name, old_value, new_value)
155         other_columns = ','.join([f"`{x}`" for x in list(set(self.current_schema_map.
    keys()) - set(col_name))])
156         sql_fill_stmt = f"SELECT {other_columns}, " \
157                         f"CASE WHEN `{col_name}` = '{old_value}' THEN '{new_value}' ELSE
     `{col_name}` END " \
158                         f"AS `{col_name}` FROM {{source}}"
159         return sql_fill_stmt
160
161     def chain_operation(self, op, args):
```

```
162             new_code = getattr(self, op)(**args)
163             logger.debug(f'Code before chaining: {self.code}')
164             self.code = new_code.replace('{source}', f'({self.code})')
165             logger.debug(f'Code after chaining: {self.code}')
166 
167         def materialize(self, new_label):
168             super(SQLOperation, self).materialize(new_label)
169             logger.debug(f'Executing SQL code: {self.code}')
170             self.code = f'CREATE VIEW `{self.new_label}` AS {self.code}'
171             return self.artifact_class(label=self.new_label,
172                                        sql_engine=self.sources[0].sql_engine,
173                                        from_sql=self.code,
174                                        schema_map=self.current_schema_map)
175 
176 
177     class SQLWorkflow(Workflow):
178         def __init__(self, *args, **kwargs):
179             sql_string = kwargs.pop('sql_string', None)
180             super(SQLWorkflow, self).__init__(*args, **kwargs)
181             self.artifact_class = SQLArtifact
182             self.operator_class = SQLOperation
183             if not sql_string:
184                 sql_string = f"sqlite:///{self.out_dir}/{self.name}.db"
185             self.sql_engine = sqlalchemy.create_engine(sql_string)
186 
187         def initialize_new_artifact(self, label=None, filename=None, schema_map=None):
188             return SQLArtifact(label, filename=filename, sql_engine=self.sql_engine,
        schema_map=schema_map)
```

5.2 Use Cases

In this section, we demonstrate the various uses of FuzzyData. We first recreate a simple workflow using a publicly available Jupyter notebook. We then generate a randomized workflow, scale the workflow size and show the runtime performance on the three clients. All of our experiments were run on a server running Ubuntu 18.04, with an Intel Xeon Silver 4416 CPU, 196 GB RAM . All of our code was executed using the Python 3.8.5 interpreter

with `pandas` v1.4.0, `SQLite` v3.33.0, `modin` v0.13.2, `dask` v2022.2.0, and `ray` v1.10.0. The `dask` engine was configured using default settings, which resulted in 8 processes and 48 threads being spawned for each bechmarking session. `ray` was also configured similarly, resulting in upto 48 worker threads being spawned in each session.

5.2.1 Fuzzy Testing Suite for Dataframe Systems

Any client implemented in the FuzzyData library can use the built-in test suite, generating many workflow test cases with variable number of artifacts, rows, columns and operations[2]. These tests can be used to check API and result equivalences and corner cases for operations run using diverse column types. Using FuzzyData's test suite, we uncovered an API corner case in `modin`, wherein a dataframe generated in memory fails to execute consecutive groupbys, and reported the issue to the `modin` developers[3]. The issue stems from lazy metadata propagation in `modin`.

5.2.2 Encoding and Replaying an Existing Workflow

In this experiment, we encode an existing workflow from the `modin` examples page [67]. The primary artifact that is loaded into this workflow is a 1.8GB CSV file, following which we execute three different group-by operations on the same artifact. The resulting execution timeline is depicted in Figure 5.3. We can see that the `modin-ray` client completes the workflow the fastest at approximately 2.8x faster than `pandas` and 21.5x faster than `SQLite`, with the runtime being dominated by the CSV loading process.

The following JSON file (Listing 5.4) is a manually encoded list of operations derived from the workflow example in [67]. This JSON input file and source artifact can be used as input to replay the workflow on all FuzzyData clients. The runtime result of this workflow

2. `https://github.com/suhailrehman/fuzzydata/tree/main/tests`

3. `https://github.com/modin-project/modin/issues/4287`

on all clients is in Section 5.2.2.

Listing 5.4: JSON encoding of a real workflow

```
{
  "name": "nyc-cab",
  "operation_list": [{
    "sources": [
      "yellow_tripdata_2015-01"
    ],
    "new_label": "gb_vendor_id_amount_sum",
    "operation_list": [{
      "op": "groupby",
      "args": {
        "group_columns": [
          "VendorID"
        ],
        "agg_columns": [
          "total_amount"
        ],
        "agg_function": "sum"
    }}]},
    {
      "sources": [
        "yellow_tripdata_2015-01"
      ],
      "new_label": "gb_pcount_amount_mean",
      "operation_list": [{
      "op": "groupby",
      "args": {
        "group_columns": [
          "passenger_count"
        ],
        "agg_columns": [
          "total_amount"
        ],
        "agg_function": "mean"
      }}]
    },
    {
      "sources": [
```

```
38          "yellow_tripdata_2015-01"
39        ],
40        "new_label": "gb_pcount_vendor_total_mean",
41        "operation_list": [{
42        "op": "groupby",
43        "args": {
44          "group_columns": [
45            "passenger_count",
46            "VendorID"
47          ],
48          "agg_columns": [
49            "total_amount"
50          ],
51          "agg_function": "mean"
52        }}]
53      }
54    ]
55  }
```

5.2.3 Scaling and Replaying a Generated Workflow

Figure 5.2 represents a workflow generated by FuzzyData with parameters ($n = 15, r = 1000, c = 20, b = 0.01, m = 1$), with all operations permitted except `pivots`. The workflow was loaded and replayed in the three FuzzyData clients with the results shown in Figures 5.4a.

For the workflow generated in Section 5.1.4, we scaled the base artifact up from 1000 rows up to 5 million and re-ran the workload on all the clients. Figure 5.4a shows the results of our scaling experiment. This specific example shows that `pandas` outperforms all the other clients, even at 5M rows (2.1 GB). Figure 5.4b illustrates the runtime breakdown grouped by operation. Despite the improvement in loading times, the total time to draw the six random samples from the distributed dataframes is much slower than the corresponding operation in `pandas`, slowing down the total runtime in `modin` compared to `pandas`, indicating a potential avenue for improvement in `modin`.

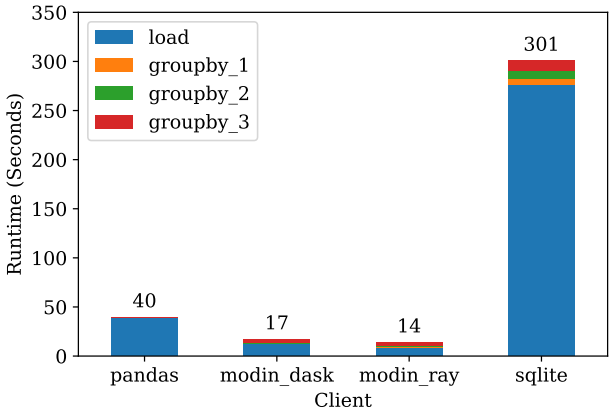
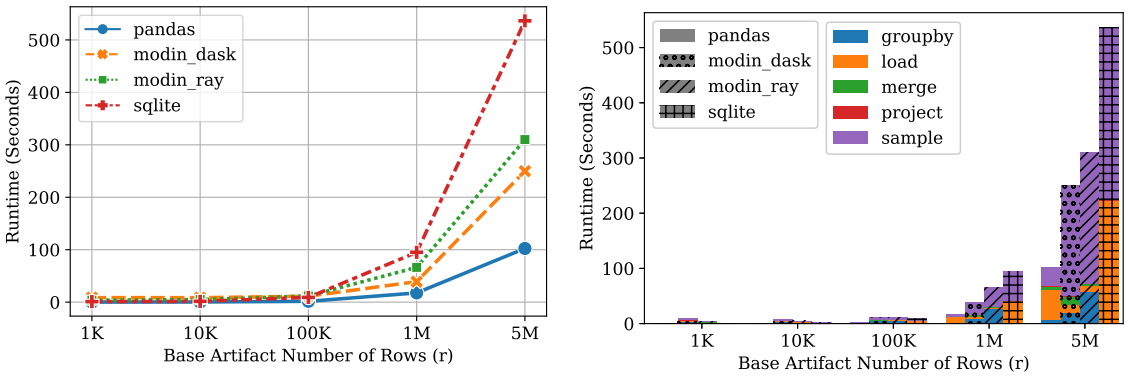


Figure 5.3: Real-World Workflow Replay



(a) Total Runtime

(b) Breakdown of runtime.

Figure 5.4: Results of the Scaling Experiments

5.3 Deployment on Modin

At the time of writing, FuzzyData has been integrated into the CI suite for `modin`, which resulted in it being downloaded over 100k times and being run the main repository 1600 times, generating approximately 3200 randomized workflows. 145 runs resulted in failures, which resulted in 3 issues being filed in the `modin` repository. FuzzyData's CI integration is planned with addtional performance testing features, allowing the developers to track the performance of `modin` over time, and compare it against pandas and other dataframe systems.

5.4 Conclusions and Future Work

We have shown FuzzyData to be a useful workflow generation system that can be used to generate workflows and test/evaluate dataframe-style systems. Due to the pluggable nature of our implementation, many extensions can be considered for future work:

- Further enhancement of the randomized table generator, allowing users to express inter-column functional dependencies and expected cardinalities for columns.
- The rule-based operation/workflow generator can be extended to use learned features to automatically generate even more realistic operations and arguments, based on artifact features, such as those described in [91].
- Integrate additional serialization formats such as `parquet` to enable comparisons across serialization formats.
- Additional clients can be implemented in FuzzyData to provide even more points of comparison.

CHAPTER 6

CONCLUSIONS

In this thesis, we have outlined the retrospective lineage inference (RLI) problem and discussed its relevance in the contemporary data lake setting. With the rapid adoption of data lake style-architectures, and the ever-increasing complexity of data analysis and machine learning pipelines, lineage and provenance management at scale becomes ever more important for sustainable data management. At the time of writing this thesis no clear standard for lineage management has emerged, making RLI a pressing problem for data lakes. This thesis advances a novel approach to RLI by leveraging similarity metrics to infer lineage, in the absence of code or artifact metadata.

In Chapter 3 we presented the Relic system, which is a novel system that addresses the RLI problem. Relic is designed to work with a single workflow, and we have shown in Section 3.6 that it can be extended to support multiple workflows, workflows with different materialization rates, and workflows with unmatched schemas. We showed that we were able to recover the lineage for small workflows in a reasonable amount of time with an average accuracy of 0.9 for a set of real-world workflows. We then showed in Chapter 4 that Relic can be scaled using sampling and sketch techniques, which offloads much of the computation to a preprocessing step, allowing for rapid lineage queries to be issued to a LSH-based index. Our primary contribution was the design and implementation of a coupled-column jaccard containment index that can be used for querying the lineage of several types of operations. Our evaluation also showed that the appropriate index can answer lineage queries in less than one second, while a brute-force search can take over 10 minutes per query.

In Chapter 5, we have shown FuzzyData to be a useful workflow generation system that can be used to generate workflows and test/evaluate dataframe-style systems. FuzzyData was used to generate workflows for dataframe-based systems and is currently being used by modin to perform regression testing.

REFERENCES

[1] AirBnB. 2020. Dataportal Project. https://medium.com/airbnb-engineering/democratizing-data-at-airbnb-852d76c51770. [Online; accessed 12/21/2020].

[2] Abdussalam Alawini, David Maier, Kristin Tufte, and Bill Howe. 2014. Helping scientists reconnect their datasets. In *Proceedings of the 26th International Conference on Scientific and Statistical Database Management (SSDBM '14)*. Association for Computing Machinery, New York, NY, USA, 1–12. https://doi.org/10.1145/2618243.2618263

[3] Abdussalam Alawini, David Maier, Kristin Tufte, Bill Howe, and Rashmi Nandikur. 2015. Towards automated prediction of relationships among scientific datasets. In *Proceedings of the 27th International Conference on Scientific and Statistical Database Management - SSDBM '15*. ACM Press, La Jolla, California, 1–5. https://doi.org/10.1145/2791347.2791358

[4] Ilkay Altintas, Chad Berkley, Efrat Jaeger, Matthew Jones, Bertram Ludascher, and Steve Mock. 2004. Kepler: an extensible system for design and execution of scientific workflows. In *Proceedings. 16th International Conference on Scientific and Statistical Database Management, 2004.* IEEE, 423–424.

[5] Austin Appleby. 2016. MurmurHash3. https://github.com/aappleby/smhasher/wiki/MurmurHash3. Accessed: 2023-01-14.

[6] Michael Armbrust, Tathagata Das, Aaron Davidson, Ali Ghodsi, Andrew Or, Josh Rosen, Ion Stoica, Patrick Wendell, Reynold Xin, and Matei Zaharia. 2015. Scaling spark in the real world: performance and usability. *Proceedings of the VLDB Endowment* 8, 12 (Aug. 2015), 1840–1843. https://doi.org/10.14778/2824032.2824080

[7] Michael Armbrust, Tathagata Das, Liwen Sun, Burak Yavuz, Shixiong Zhu, Mukul Murthy, Joseph Torres, Herman van Hovell, Adrian Ionescu, Alicja Łuszczak, Michał Świtakowski, Michał Szafrański, Xiao Li, Takuya Ueshin, Mostafa Mokhtar, Peter Boncz, Ali Ghodsi, Sameer Paranjpye, Pieter Senster, Reynold Xin, and Matei Zaharia. 2020. Delta lake: high-performance ACID table storage over cloud object stores. *Proceedings of the VLDB Endowment* 13, 12 (Aug. 2020), 3411–3424. https://doi.org/10.14778/3415478.3415560

[8] M. M. Astrahan, M. W. Blasgen, D. D. Chamberlin, K. P. Eswaran, J. N. Gray, P. P. Griffiths, W. F. King, R. A. Lorie, P. R. McJones, J. W. Mehl, G. R. Putzolu, I. L. Traiger, B. W. Wade, and V. Watson. 1976. System R: relational approach to database management. *ACM Transactions on Database Systems* 1, 2 (June 1976), 97–137. https://doi.org/10.1145/320455.320457

[9] Peter Bailis, Joseph M Hellerstein, and Michael Stonebraker. 2015. Readings in Database Systems, 5th Edition. http://www.redbook.io/

[10] Rohan Bavishi, Caroline Lemieux, Roy Fox, Koushik Sen, and Ion Stoica. 2019. AutoPandas: Neural-Backed Generators for Program Synthesis. *Proc. ACM Program. Lang.* 3, OOPSLA, Article 168 (Oct. 2019), 27 pages. https://doi.org/10.1145/3360594

[11] Anant Bhardwaj, Amol Deshpande, Aaron J. Elmore, David Karger, Sam Madden, Aditya Parameswaran, Harihar Subramanyam, Eugene Wu, and Rebecca Zhang. 2015. Collaborative Data Analytics with DataHub. *Proc. VLDB Endow.* 8, 12 (aug 2015), 1916–1919. https://doi.org/10.14778/2824032.2824100

[12] Anant P. Bhardwaj, Souvik Bhattacherjee, Amit Chavan, Amol Deshpande, Aaron J. Elmore, Samuel Madden, and Aditya G. Parameswaran. 2015. DataHub: Collaborative Data Science & Dataset Version Management at Scale. http://cidrdb.org/cidr2015/Papers/CIDR15_Paper18.pdf

[13] A. Broder. 1997. On the Resemblance and Containment of Documents. In *Proceedings of the Compression and Complexity of Sequences 1997 (SEQUENCES '97)*. IEEE Computer Society, USA, 21.

[14] Stanislav Böhm and Jakub Beránek. 2020. Runtime vs Scheduler: Analyzing Dask's Overheads. In *2020 IEEE/ACM Workflows in Support of Large-Scale Science (WORKS)*. 1–8. https://doi.org/10.1109/WORKS51914.2020.00006

[15] Raul Castro Fernandez, Ziawasch Abedjan, Famien Koko, Gina Yuan, Samuel Madden, and Michael Stonebraker. 2018. Aurum: A Data Discovery System. In *2018 IEEE 34th International Conference on Data Engineering (ICDE)*. IEEE, Paris, 1001–1012. https://doi.org/10.1109/ICDE.2018.00094

[16] James Cheney, Laura Chiticariu, and Wang-Chiew Tan. 2009. *Provenance in databases: Why, how, and where*. Now Publishers Inc.

[17] P. Christen. 2012. A Survey of Indexing Techniques for Scalable Record Linkage and Deduplication. *IEEE Transactions on Knowledge and Data Engineering* 24, 9 (Sep. 2012), 1537–1555. https://doi.org/10.1109/TKDE.2011.127

[18] Brian F. Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. 2010. Benchmarking cloud serving systems with YCSB. In *Proceedings of the 1st ACM symposium on Cloud computing - SoCC '10*. ACM Press, Indianapolis, Indiana, USA, 143. https://doi.org/10.1145/1807128.1807152

[19] TPC Council. 2022. TPC-Homepage. https://tpc.org/default5.asp

[20] Y. Cui and J. Widom. 2003. Lineage tracing for general data warehouse transformations. *The VLDB Journal The International Journal on Very Large Data Bases* 12, 1 (May 2003), 41–58. https://doi.org/10.1007/s00778-002-0083-8

[21] Anish Das Sarma, Aditya Parameswaran, Hector Garcia-Molina, and Jennifer Widom. 2010. Synthesizing view definitions from data. In *Proceedings of the 13th International Conference on Database Theory - ICDT '10*. ACM Press, Lausanne, Switzerland, 89. https://doi.org/10.1145/1804669.1804683

[22] Tom De Nies, Sam Coppens, Davy Van Deursen, Erik Mannens, and Rik Van de Walle. 2012. Automatic Discovery of High-Level Provenance Using Semantic Similarity. In *Provenance and Annotation of Data and Processes*, David Hutchison, Takeo Kanade, Josef Kittler, Jon M. Kleinberg, Friedemann Mattern, John C. Mitchell, Moni Naor, Oscar Nierstrasz, C. Pandu Rangan, Bernhard Steffen, Madhu Sudan, Demetri Terzopoulos, Doug Tygar, Moshe Y. Vardi, Gerhard Weikum, Paul Groth, and James Frew (Eds.). Vol. 7525. Springer Berlin Heidelberg, Berlin, Heidelberg, 97–110. https://doi.org/10.1007/978-3-642-34222-6_8 Series Title: Lecture Notes in Computer Science.

[23] Tom De Nies, Erik Mannens, and Rik Van de Walle. 2016. Reconstructing Human-Generated Provenance Through Similarity-Based Clustering. In *Provenance and Annotation of Data and Processes (Lecture Notes in Computer Science)*, Marta Mattoso and Boris Glavic (Eds.). Springer International Publishing, Cham, 191–194. https://doi.org/10.1007/978-3-319-40593-3_19

[24] Vinay Deolalikar and Hernan Laffitte. 2009. Provenance as Data Mining: Combining File System Metadata with Content Analysis. In *First Workshop on on Theory and Practice of Provenance* (San Francisco, CA) *(TAPP'09)*. USENIX Association, USA, Article 10, 10 pages.

[25] Hong-Hai Do and Erhard Rahm. 2002. COMA: a system for flexible combination of schema matching approaches. In *Proceedings of the 28th international conference on Very Large Data Bases (VLDB '02)*. VLDB Endowment, Hong Kong, China, 610–621.

[26] LinkedIn Engineering. 2020. DataHub Project. https://github.com/linkedin/datahub. [Online; accessed 12/21/2020].

[27] Mahdi Esmailoghli, Jorge-Arnulfo Quiané-Ruiz, and Ziawasch Abedjan. 2022. MATE: Multi-Attribute Table Extraction. *Proc. VLDB Endow.* 15, 8 (jun 2022), 1684–1696. https://doi.org/10.14778/3529337.3529353

[28] Daniele Faraglia. 2020. Faker Python Package. https://github.com/joke2k/faker. [Online; accessed 12/21/2020].

[29] Anna Fariha and Alexandra Meliou. 2019. Example-driven Query Intent Discovery: Abductive Reasoning Using Semantic Similarity. *Proc. VLDB Endow.* 12, 11 (July 2019), 1262–1275. https://doi.org/10.14778/3342263.3342266

[30] R. Castro Fernandez, Z. Abedjan, F. Koko, G. Yuan, S. Madden, and M. Stonebraker. 2018. Aurum: A Data Discovery System. In *2018 IEEE 34th International Conference*

on Data Engineering (ICDE). IEEE Computer Society, Los Alamitos, CA, USA, 1001–1012. https://doi.org/10.1109/ICDE.2018.00094

[31] R. Castro Fernandez, J. Min, D. Nava, and S. Madden. 2019. Lazo: A Cardinality-Based Method for Coupled Estimation of Jaccard Similarity and Containment. In *2019 IEEE 35th International Conference on Data Engineering (ICDE)*. IEEE Computer Society, Los Alamitos, CA, USA, 1190–1201. https://doi.org/10.1109/ICDE.2019.00109

[32] Apache Software Foundation. 2020. Apache Arrow. https://arrow.apache.org/

[33] Apache Software Foundation. 2020. Apache Atlas. https://atlas.apache.org/. [Online; accessed 12/21/2020].

[34] LF AI & Data Foundation. 2020. Amundsen Project. https://github.com/amundsen-io/amundsen. [Online; accessed 12/21/2020].

[35] LF AI & Data Foundation. 2020. Egeria Project. https://egeria.odpi.org/. [Online; accessed 12/21/2020].

[36] LF AI & Data Foundation. 2020. Marquez Project. https://github.com/MarquezProject/marquez. [Online; accessed 12/21/2020].

[37] The R Foundation. 2022. R: The R Project for Statistical Computing. https://www.r-project.org/

[38] Jim Gray (Ed.). 1994. *The Benchmark handbook: for database and transaction processing systems* (2. ed., 2. [print.] ed.). Morgan Kaufmann, San Francisco, Calif.

[39] Sumit Gulwani. 2011. Automating String Processing in Spreadsheets Using Input-Output Examples. In *Proceedings of the 38th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (Austin, Texas, USA) *(POPL '11)*. Association for Computing Machinery, New York, NY, USA, 317–330. https://doi.org/10.1145/1926385.1926423

[40] Philip J. Guo and Margo Seltzer. 2012. BURRITO: Wrapping Your Lab Notebook in Computational Infrastructure. In *Proceedings of the 4th USENIX Conference on Theory and Practice of Provenance* (Boston, MA) *(TaPP'12)*. USENIX Association, USA, 7.

[41] H20.ai. 2022. Database-like ops benchmark. https://h2oai.github.io/db-benchmark/

[42] Aric Hagberg, Pieter Swart, and Daniel S Chult. 2008. *Exploring network structure, dynamics, and function using networkx*. Technical Report LA-UR-08-05495; LA-UR-08-5495. Los Alamos National Lab. (LANL), Los Alamos, NM (United States). https://www.osti.gov/biblio/960616

[43] Rihan Hai, Sandra Geisler, and Christoph Quix. 2016. Constance: An intelligent data lake system. In *SIGMOD'16*. ACM, 2097–2100.

[44] Alon Halevy, Flip Korn, Natalya F. Noy, Christopher Olston, Neoklis Polyzotis, Sudip Roy, and Steven Euijong Whang. 2016. Goods: Organizing Google's Datasets. In *Proceedings of the 2016 International Conference on Management of Data*. ACM, San Francisco California USA, 795–806. https://doi.org/10.1145/2882903.2903730

[45] Arvid Heise, Jorge-Arnulfo Quiané-Ruiz, Ziawasch Abedjan, Anja Jentzsch, and Felix Naumann. 2013. Scalable discovery of unique column combinations. *Proceedings of the VLDB Endowment* 7, 4 (2013), 301–312.

[46] Melanie Herschel, Ralf Diestelkämper, and Houssem Ben Lahmar. 2017. A Survey on Provenance: What for? What Form? What From? *The VLDB Journal* 26, 6 (dec 2017), 881–906. https://doi.org/10.1007/s00778-017-0486-1

[47] Dongxu Huang, Qi Liu, Qiu Cui, Zhuhe Fang, Xiaoyu Ma, Fei Xu, Li Shen, Liu Tang, Yuxing Zhou, Menglong Huang, Wan Wei, Cong Liu, Jian Zhang, Jianjun Li, Xuelian Wu, Lingyu Song, Ruoxi Sun, Shuaipeng Yu, Lei Zhao, Nicholas Cameron, Liquan Pei, and Xin Tang. 2020. TiDB: a Raft-based HTAP database. *Proceedings of the VLDB Endowment* 13, 12 (Aug. 2020), 3072–3084. https://doi.org/10.14778/3415478.3415535

[48] Silu Huang, Liqi Xu, Jialin Liu, Aaron J Elmore, and Aditya Parameswaran. 2017. Orpheus DB: bolt-on versioning for relational databases. *Proceedings of the VLDB Endowment* 10, 10 (2017), 1130–1141. Publisher: VLDB Endowment.

[49] Madelon Hulsebos. 2022. *GitTables dataset for SemTab 2022*. https://doi.org/10.5281/zenodo.7091019

[50] Robert Ikeda and Jennifer Widom. 2009. *Data lineage: A survey*. Technical Report. Stanford InfoLab.

[51] Jiyong Jang, Maverick Woo, and David Brumley. 2013. Towards Automatic Software Lineage Inference. In *22nd USENIX Security Symposium (USENIX Security 13)*. USENIX Association, Washington, D.C., 81–96. https://www.usenix.org/conference/usenixsecurity13/technical-sessions/papers/jang

[52] Dmitri V. Kalashnikov, Laks V.S. Lakshmanan, and Divesh Srivastava. 2018. FastQRE: Fast Query Reverse Engineering. In *Proceedings of the 2018 International Conference on Management of Data (SIGMOD '18)*. Association for Computing Machinery, New York, NY, USA, 337–350. https://doi.org/10.1145/3183713.3183727

[53] Christos Koutras, Kyriakos Psarakis, George Siachamis, Andra Ionescu, Marios Fragkoulis, Angela Bonifati, and Asterios Katsifodimos. 2021. Valentine in Action: Matching Tabular Data at Scale. *Proc. VLDB Endow.* 14, 12 (oct 2021), 2871–2874. https://doi.org/10.14778/3476311.3476366

[54] Andreas Kunft, Lukas Stadler, Daniele Bonetta, Cosmin Basca, Jens Meiners, Sebastian Breß, Tilmann Rabl, Juan Fumero, and Volker Markl. 2018. ScootR: Scaling R

Dataframes on Dataflow Systems. In *Proceedings of the ACM Symposium on Cloud Computing (SoCC '18)*. Association for Computing Machinery, New York, NY, USA, 288–300. https://doi.org/10.1145/3267809.3267813

[55] Doris Jung-Lin Lee, Dixin Tang, Kunal Agarwal, Thyne Boonmark, Caitlyn Chen, Jake Kang, Ujjaini Mukhopadhyay, Jerry Song, Micah Yong, Marti A. Hearst, and Aditya G. Parameswaran. 2021. Lux: always-on visualization recommendations for exploratory dataframe workflows. *Proceedings of the VLDB Endowment* 15, 3 (Nov. 2021), 727–738. https://doi.org/10.14778/3494124.3494151

[56] Jure Leskovec, Anand Rajaraman, and Jeffrey David Ullman. 2014. *Mining of Massive Datasets* (2nd ed.). Cambridge University Press, USA.

[57] Ping Li, Art B Owen, and Cun-Hui Zhang. 2012. One Permutation Hashing. In *Proceedings of the 25th International Conference on Neural Information Processing Systems - Volume 2* (Lake Tahoe, Nevada) *(NIPS'12)*. Curran Associates Inc., Red Hook, NY, USA, 3113–3121.

[58] Jayant Madhavan, Philip A. Bernstein, and Erhard Rahm. 2001. Generic Schema Matching with Cupid. In *Proceedings of the 27th International Conference on Very Large Data Bases (VLDB '01)*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 49–58.

[59] S. Melnik, H. Garcia-Molina, and E. Rahm. 2002. Similarity flooding: a versatile graph matching algorithm and its application to schema matching. In *Proceedings 18th International Conference on Data Engineering*. 117–128. https://doi.org/10.1109/ICDE.2002.994702 ISSN: 1063-6382.

[60] Hui Miao, Amit Chavan, and Amol Deshpande. 2017. ProvDB: Lifecycle Management of Collaborative Analysis Workflows. In *Proceedings of the 2nd Workshop on Human-In-the-Loop Data Analytics (HILDA'17)*. Association for Computing Machinery, New York, NY, USA, 1–6. https://doi.org/10.1145/3077257.3077267

[61] Philipp Moritz, Robert Nishihara, Stephanie Wang, Alexey Tumanov, Richard Liaw, Eric Liang, Melih Elibol, Zongheng Yang, William Paul, Michael I. Jordan, and Ion Stoica. 2018. Ray: A Distributed Framework for Emerging AI Applications. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*. USENIX Association, Carlsbad, CA, 561–577. https://www.usenix.org/conference/osdi18/presentation/moritz

[62] Fatemeh Nargesian, Erkang Zhu, Renée J. Miller, Ken Q. Pu, and Patricia C. Arocena. 2019. Data lake management: challenges and opportunities. *Proceedings of the VLDB Endowment* 12, 12 (Aug. 2019), 1986–1989. https://doi.org/10.14778/3352063.3352116

[63] Netflix. 2020. Metacat Project. https://github.com/Netflix/metacat. [Online; accessed 12/21/2020].

[64] Thanh Tam Nguyen, Quoc Viet Hung Nguyen, Matthias Weidlich, and Karl Aberer. 2015. Result selection and summarization for Web Table search. In *2015 IEEE 31st International Conference on Data Engineering*. 231–242. https://doi.org/10.1109/ICDE.2015.7113287 ISSN: 2375-026X.

[65] Devin Petersohn, Stephen Macke, Doris Xin, William Ma, Doris Lee, Xiangxi Mo, Joseph E. Gonzalez, Joseph M. Hellerstein, Anthony D. Joseph, and Aditya Parameswaran. 2020. Towards scalable dataframe systems. *Proceedings of the VLDB Endowment* 13, 12 (Aug. 2020), 2033–2046. https://doi.org/10.14778/3407790.3407807

[66] Modin Project. 2022. Issues · modin-project/modin. https://github.com/modin-project/modin

[67] Modin Project. 2022. Modin NYC Taxi Example Notebook. https://github.com/modin-project/modin/blob/dd9beee3a599d3a91036cbaeef8b8499ba9cc4c1/examples/jupyter/NYC_Taxi.ipynb original-date: 2018-06-21T21:35:05Z.

[68] Fotis Psallidas and Eugene Wu. 2018. Smoke: Fine-Grained Lineage at Interactive Speed. *Proc. VLDB Endow.* 11, 6 (Feb. 2018), 719–732. https://doi.org/10.14778/3184470.3184475

[69] Kyriakos Psarakis. 2022. Valentine: Matching DataFrames Easily. https://github.com/delftdata/valentine original-date: 2019-06-28T13:50:14Z.

[70] Erhard Rahm and Philip A Bernstein. 2001. A survey of approaches to automatic schema matching. *the VLDB Journal* 10, 4 (2001), 334–350.

[71] Mohammed Suhail Rehman. 2019. Towards Understanding Data Analysis Workflows Using a Large Notebook Corpus. In *Proceedings of the 2019 International Conference on Management of Data* (Amsterdam, Netherlands) *(SIGMOD '19)*. Association for Computing Machinery, New York, NY, USA, 1841–1843. https://doi.org/10.1145/3299869.3300107

[72] El Kindi Rezig, Lei Cao, Michael Stonebraker, Giovanni Simonini, Wenbo Tao, Samuel Madden, Mourad Ouzzani, Nan Tang, and Ahmed K. Elmagarmid. 2019. Data Civilizer 2.0: A Holistic Framework for Data Preparation and Analytics. *Proc. VLDB Endow.* 12, 12 (aug 2019), 1954–1957. https://doi.org/10.14778/3352063.3352108

[73] Janessa Rivera and Rob van der Meulen. 2014. Gartner Says Beware of the Data Lake Fallacy. http://www.gartner.com/newsroom/id/2809117. Accessed: 2023-01-14.

[74] Ronald Rivest. 1992. *The MD5 Message-Digest Algorithm.* RFC 1321. RFC Editor. 1–21 pages. https://www.rfc-editor.org/rfc/rfc1321

[75] Matthew Rocklin. 2015. Dask: Parallel Computation with Blocked algorithms and Task Scheduling. In *Proceedings of the 14th Python in Science Conference*. Austin, TX, 130–136.

[76] Adam Rule, Aurélien Tabard, and James D Hollan. 2018. Exploration and explanation in computational notebooks. In *Proceedings of the 2018 CHI Conference on Human Factors in Computing Systems*. ACM, New York, NY, USA, 32.

[77] Anshumali Shrivastava. 2017. Optimal Densification for Fast and Accurate Minwise Hashing. In *Proceedings of the 34th International Conference on Machine Learning - Volume 70* (Sydney, NSW, Australia) *(ICML'17)*. JMLR.org, 3154–3163.

[78] Phanwadee Sinthong and Michael J. Carey. 2019. AFrame: Extending DataFrames for Large-Scale Modern Data Analysis. In *2019 IEEE International Conference on Big Data (Big Data)*. IEEE, 359–371. https://doi.org/10.1109/BigData47090.2019.9006303

[79] Brian Stein and Alan Morrison. 2014. The enterprise data lake: Better integration and deeper analytics. *PwC Technology Forecast: Rethinking integration* 1 (2014).

[80] Wei Chit Tan, Meihui Zhang, Hazem Elmeleegy, and Divesh Srivastava. 2017. Reverse engineering aggregation queries. *Proceedings of the VLDB Endowment* 10, 11 (Aug. 2017), 1394–1405. https://doi.org/10.14778/3137628.3137648

[81] RAPIDS Development Team. 2018. RAPIDS: Collection of Libraries for End to End GPU Data Science. https://rapids.ai

[82] The pandas development team. 2020. pandas-dev/pandas: Pandas. https://doi.org/10.5281/zenodo.3509134

[83] Ignacio G. Terrizzano, Peter M. Schwarz, Mary Roth, and John E. Colino. 2015. Data Wrangling: The Challenging Journey from the Wild to the Lake. In *CIDR 2015, Seventh Biennial Conference on Innovative Data Systems Research, Asilomar, CA, USA, January 4-7, 2015, Online Proceedings*. www.cidrdb.org. http://cidrdb.org/cidr2015/Papers/CIDR15_Paper2.pdf

[84] Quoc Trung Tran, Chee-Yong Chan, and Srinivasan Parthasarathy. 2009. Query by output. In *Proceedings of the 2009 ACM SIGMOD International Conference on Management of data*. ACM, New York, NY, USA, 535–548.

[85] Uber. 2020. DataBook Project. https://eng.uber.com/databook/. [Online; accessed 12/21/2020].

[86] Petros Venetis, Alon Halevy, Jayant Madhavan, Marius Paşca, Warren Shen, Fei Wu, Gengxin Miao, and Chung Wu. 2011. Recovering semantics of tables on the web. *Proceedings of the VLDB Endowment* 4, 9 (June 2011), 528–538. https://doi.org/10.14778/2002938.2002939

[87] Ritche Vink. 2021. Polars. https://github.com/pola-rs/polars original-date: 2020-05-13T19:45:33Z.

[88] Adrian Vogelsgesang, Michael Haubenschild, Jan Finis, Alfons Kemper, Viktor Leis, Tobias Muehlbauer, Thomas Neumann, and Manuel Then. 2018. Get Real: How Benchmarks Fail to Represent the Real World. In *Proceedings of the Workshop on Testing Database Systems (DBTest'18)*. Association for Computing Machinery, New York, NY, USA, 1–6. https://doi.org/10.1145/3209950.3209952

[89] Alex Watson, Deepigha Shree Vittal Babu, and Suprio Ray. 2017. Sanzu: A data science benchmark. In *2017 IEEE International Conference on Big Data (Big Data)*. 263–272. https://doi.org/10.1109/BigData.2017.8257934

[90] Eugene Wu, Samuel Madden, and Michael Stonebraker. 2013. SubZero: A Fine-Grained Lineage System for Scientific Databases. In *Proceedings of the 2013 IEEE International Conference on Data Engineering (ICDE 2013) (ICDE '13)*. IEEE Computer Society, USA, 865–876. https://doi.org/10.1109/ICDE.2013.6544881

[91] Cong Yan and Yeye He. 2020. Auto-Suggest: Learning-to-Recommend Data Preparation Steps Using Data Science Notebooks. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data (SIGMOD '20)*. Association for Computing Machinery, Portland, OR, USA, 1539–1554. https://doi.org/10.1145/3318464.3389738

[92] Y. Yang, Y. Zhang, W. Zhang, and Z. Huang. 2019. GB-KMV: An Augmented KMV Sketch for Approximate Containment Similarity Search. In *2019 IEEE 35th International Conference on Data Engineering (ICDE)*. IEEE Computer Society, Los Alamitos, CA, USA, 458–469. https://doi.org/10.1109/ICDE.2019.00048

[93] Gunce Su Yilmaz, Tana Wattanawaroon, Liqi Xu, Abhishek Nigam, Aaron J. Elmore, and Aditya Parameswaran. 2018. DataDiff: User-Interpretable Data Transformation Summaries for Collaborative Data Analysis. In *Proceedings of the 2018 International Conference on Management of Data (SIGMOD '18)*. Association for Computing Machinery, New York, NY, USA, 1769–1772. https://doi.org/10.1145/3183713.3193564

[94] Matei Zaharia, Andrew Chen, Aaron Davidson, Ali Ghodsi, Sue Ann Hong, Andy Konwinski, Siddharth Murching, Tomas Nykodym, Paul Ogilvie, Mani Parkhe, Fen Xie, and Corey Zumar. 2018. Accelerating the Machine Learning Lifecycle with MLflow. *IEEE Data Engineering Bulletin* 41 (2018), 39–45.

[95] Matei Zaharia, Mosharaf Chowdhury, Michael J. Franklin, Scott Shenker, and Ion Stoica. 2010. Spark: Cluster Computing with Working Sets. In *Proceedings of the 2nd USENIX Conference on Hot Topics in Cloud Computing* (Boston, MA) *(HotCloud'10)*. USENIX Association, USA, 10.

[96] Yi Zhang and Zachary G. Ives. 2020. Finding Related Tables in Data Lakes for Interactive Data Science. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data (SIGMOD '20)*. Association for Computing Machinery, New York, NY, USA, 1951–1966. https://doi.org/10.1145/3318464.3389726

[97] Jun Zhao, Carole Goble, Robert Stevens, and Daniele Turi. 2008. Mining Taverna's semantic web of provenance. *Concurrency and Computation: Practice and Experience* 20, 5 (2008), 463–472. Publisher: Wiley Online Library.

[98] Erkang Zhu, Dong Deng, Fatemeh Nargesian, and Renée J. Miller. 2019. JOSIE: Overlap Set Similarity Search for Finding Joinable Tables in Data Lakes. In *Proceedings of the 2019 International Conference on Management of Data (SIGMOD '19)*. Association for Computing Machinery, New York, NY, USA, 847–864. https://doi.org/10.1145/3299869.3300065

[99] Erkang Zhu, Fatemeh Nargesian, Ken Q. Pu, and Renée J. Miller. 2016. LSH Ensemble: Internet-Scale Domain Search. *Proc. VLDB Endow.* 9, 12 (aug 2016), 1185–1196. https://doi.org/10.14778/2994509.2994534