FINE-GRAINED PROVENANCE AND APPLICATIONS TO DATA ANALYTICS COMPUTATIONS

Nan Zheng

A DISSERTATION

In

Computer and Information Science

Presented to the Faculties of the University of Pennsylvania in Partial Fulfillment of the Requirements for the Degree of Doctor of Philosophy

2021

Zachary G. Ives
Supervisor of Dissertation

Mayur Naik
Graduate Group Chairperson

Dissertation Committee

Susan B. Davidson, Weiss Professor of Computer and Information Science, University of Pennsylvania

Boon Thau Loo, RCA Professor of Computer and Information Science, University of Pennsylvania

Andreas Haeberlen, Associate Professor of Computer and Information Science, University of Pennsylvania

Junhyong Kim, Patricia M. Williams Term Professor of Biology, University of Pennsylvania
FINE-GRAINED PROVENANCE AND APPLICATIONS TO DATA ANALYTICS COMPUTATIONS

© COPYRIGHT

2021

Nan Zheng
To my father Honglin Zheng, mother Qiuyun Zhang, and husband Xinfeng Li.
ACKNOWLEDGEMENT

First and foremost, I would like to thank my advisor, Dr. Zachary Ives. This thesis would not have been possible without his support and guidance. During my PhD career, he was always available and supportive, despite his busy schedule. He consistently kept up to date with current research studies and techniques, had a keen eye for my research ideas, and often provided insightful feedback. Further, I am especially thankful for his understanding and encouragement, especially while I was working from home and raising a baby during the pandemic. It has been an absolutely valuable and incredible experience working with and learning from Dr. Ives.

My thesis committee consisted of Dr. Susan Davidson, Dr. Andreas Haeberlen, Dr. Boon Thau Loo, and Dr. Junhyong Kim. I am grateful to them for serving on my WPE-II and dissertation committee and suggesting numerous improvements to my thesis.

I am also grateful to Dr. Val Tannen of the Penn Database Group, whose office is just to the opposite of mine. He has been a great mentor to me during my PhD studies in both research and life philosophy. In addition, I wish to thank other professors in the Penn Database Group, especially Dr. Susan Davidson and Dr. Boon Thau Loo, as well as students Yi Zhang, Yinjun Wu, and Soonbo Han. This team formed an amazing and lively peer group with which to discuss research problems in regular group seminars.

My office-mates - Salar Moarref, Mukund Raghothaman and Abdu Alawini—have each contributed to the successful completion of this thesis and my PhD journey in unique ways. Mukund especially taught me a great deal of useful knowledge: from writing code, to problem formalization, to general wisdom about life.

Finally, I thank my parents for their unconditional support and love, and especially my husband, without whom I would not even have begun work on this thesis. For them, no words of gratitude will ever be enough.
ABSTRACT

FINE-GRAINED PROVENANCE AND APPLICATIONS TO DATA ANALYTICS COMPUTATIONS

Nan Zheng
Zachary G. Ives

Data provenance tools seek to facilitate reproducible data science and auditable data analyses by capturing the analytics steps used in generating data analysis results. However, analysts must choose among workflow provenance systems, which allow arbitrary code but only track provenance at the granularity of files; provenance APIs, which provide tuple-level provenance, but incur overhead in all computations; and database provenance tools, which track tuple-level provenance through relational operators and support optimization, but support a limited subset of data science tasks. None of these solutions are well suited for tracing errors introduced during common ETL, record alignment, and matching tasks – for data types such as strings, images, etc.

Additionally, we need a provenance archival layer to store and manage the tracked fine-grained provenance that enables future sophisticated reasoning about why individual output results appear or fail to appear. For reproducibility and auditing, the provenance archival system should be tamper-resistant. On the other hand, the provenance collecting over time or within the same query computation tends to be repeated partially (i.e., the same operation with the same input records in the middle computation step). Hence, we desire efficient provenance storage (i.e., it compresses repeated results).

We address these challenges with novel formalisms and algorithms, implemented in the PROVision system, for reconstructing fine-grained provenance for a broad class of ETL-style workflows. We extend database-style provenance techniques to capture equivalences, support optimizations, and enable lazy evaluations. We develop solutions for storing fine-grained provenance in relational storage systems while both compressing and protecting it via cryptographic hashes. We experimentally
validate our proposed solutions using both scientific and OLAP workloads.
TABLE OF CONTENTS

ACKNOWLEDGEMENT ................................................................. iv

ABSTRACT ................................................................. v

LIST OF TABLES ................................................................. ix

LIST OF ILLUSTRATIONS ................................................... xi

CHAPTER 1 : Introduction .................................................. 1
  1.1 Provenance’s Roles in Supporting Data Analysis .......... 4
  1.2 Provenance Management for Modern Data Science Applications .... 7
  1.3 Research Challenges and Contributions ....................... 10
  1.4 Roadmap ............................................................. 13

CHAPTER 2 : Background and Foundations ......................... 14
  2.1 Integrating Provenance in Data Processing Pipeline ........ 15
  2.2 Provenance Life Cycle ....................................... 17
  2.3 Fundamental Prior Work ....................................... 20

CHAPTER 3 : Problems and Overview of the PROVision System .... 27
  3.1 Open Problems Addressed in this Thesis .................... 27
  3.2 PROVision System Capabilities .................................. 33
  3.3 The PROVision System Architecture ....................... 33

CHAPTER 4 : Fine-grained Provenance for ETL Tasks Computation .... 37
  4.1 Background and Problems ....................................... 37
  4.2 Declarative Representation of Queries ....................... 39
  4.3 Provenance Models ............................................. 44
# LIST OF TABLES

<table>
<thead>
<tr>
<th>TABLE</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>TABLE 1</td>
<td>Simplified example tables of sequences.</td>
<td>31</td>
</tr>
<tr>
<td>TABLE 2</td>
<td>UDF operator equivalences</td>
<td>54</td>
</tr>
<tr>
<td>TABLE 3</td>
<td>UDF type/operator provenance equivalences</td>
<td>54</td>
</tr>
<tr>
<td>TABLE 4</td>
<td>The overall complexity of the semantic descriptors.</td>
<td>66</td>
</tr>
<tr>
<td>TABLE 5</td>
<td>Original workflow execution costs.</td>
<td>67</td>
</tr>
<tr>
<td>TABLE 6</td>
<td>Average CPU time for tracing small subsets of output (5-1000 records).</td>
<td>70</td>
</tr>
<tr>
<td>TABLE 7</td>
<td>The proportion of inputs corresponding to the different output generated by different versions.</td>
<td>71</td>
</tr>
<tr>
<td>TABLE 8</td>
<td>Provenance semiring expressions with input, output and query expressions</td>
<td>75</td>
</tr>
<tr>
<td>TABLE 9</td>
<td>Baseline workflow, no provenance computation.</td>
<td>98</td>
</tr>
<tr>
<td>TABLE 10</td>
<td>Theoretical tamper resistance vs space overhead.</td>
<td>101</td>
</tr>
<tr>
<td>TABLE 11</td>
<td>Actual collisions observed in workloads.</td>
<td>101</td>
</tr>
<tr>
<td>TABLE 12</td>
<td>Number of look-ups or edges in the transitive closure with 1000 inputs.</td>
<td>109</td>
</tr>
</tbody>
</table>
LIST OF ILLUSTRATIONS

FIGURE 1: Sequence Alignment ETL-style workflow example. .......... 4
FIGURE 2: Data analysis pipeline stages. ................................. 15
FIGURE 3: The example databases: an online travel portal. ............ 22
FIGURE 4: A query over tables in Figure 3 and its output with semiring provenance. 22
FIGURE 5: An example of gene sequence alignment workflow. .......... 25
FIGURE 6: Gene sequence alignment workflow. Tissue is first run through a sequencing machine, and the output is then normalized, cleaned, matched against a reference sequence, and mapped using a transformation library. 27
FIGURE 7: The last two steps of the sequence alignment workflow .......... 28
FIGURE 8: Common data analyses query template involving extraction, transformation, alignment, and selection or aggregation of matches. Shared by many ETL/data cleaning scenarios, information extraction scenarios, and biomedical informatics scenarios such as gene sequencing. ........ 29
FIGURE 9: PROVision system architecture. ................................. 34
FIGURE 10: An example of module configuration file and descriptor file. .... 40
FIGURE 11: Algebra of the descriptor file in Figure 10 ................... 41
FIGURE 12: Example of converting Python Pandas code into a relational algebra expression. ............................................. 41
FIGURE 13: Extracting data from structured files is modeled as reading a sequence of $(filename,object)$ pairs from the data lake, then joining these results with the records within the objects. ......................... 43
FIGURE 14: Example of tracing differences between outputs of two workflow runs, to debug inconsistencies. ................................. 57
FIGURE 15: The system architecture of provenance capture layer. .......... 60
Provenance-aware computation captures information that helps determine the derivation history of computational results, tracing back to the original data sources. Such information enables not only the reproduction and validation of results while also reasoning about how the data was generated and what the input datasets and parameters were. It also allows a user to determine trustworthiness or validity by propagating the trust or validating information through mappings from the input records to the output. Provenance information facilitates more efficient data analysis computations by supporting the refinement and debugging of analysis queries in different application domains.

Provenance-aware computation has been widely investigated for these applications in scientific and enterprise data management Davidson and Freire (2008); Freire et al. (2008); Hasan et al. (2007). However, provenance techniques fail to address issues in the processes of provenance computation and provenance archiving. We need to solve these new problems in order to usher in state-of-the-art applications, as we will next describe for both scientific and enterprise data management applications.

Both scientific and enterprise application domains rely on fine-grained provenance of identified origins as well as the certifiable information of computation processes. Among the diverse existing applications in science and business, workflow provenance techniques Oinn et al. (2006); Ludäsch et al. (2006); Goecks et al. (2010) are the most prevalent. However, they treat modules as “black boxes”, and only capture the provenance at the file level. This coarse-grained provenance is unable to solve tasks such as debugging or detailed reasoning. Such tasks require record or even sub-record level provenance information, which is fine-grained provenance. Fine-grained provenance has a long history in databases Buneman et al. (2002). A number of models and solutions have been proposed. One of the most influential works is the provenance semirings Green et al. (2007b), which captures fine-grained provenance through relational algebra operators, while preserving the algebraic equivalences used by query optimization. Based on the provenance semiring model, numerous database-style techniques have been developed Cheney et al. (2009); Glavic and Alonso (2009); Karvounarakis et al. (2010) in data analysis computation. As most of these com-
putations are conducted on declarative data processing platforms, such as Spark, Hadoop, Pandas dataframes (Python), and Extract-Transform-Load workflows (ETL), relational operators with some limited functions naturally match many query settings and big data settings. Provenance semiring can be easily extended to capture provenance on such platforms. These workflows more closely resemble Extract, Transform and Load (ETL) processes (e.g., data ingest and cleaning, joins, gene sequence alignment, image feature extraction), and they often combine tools written in different environments.

**Scientific data management.** Data processing and computing at scale have revolutionized science and enabled many important discoveries. However, with large volumes of data being manipulated and complex computations being utilized, it is challenging to reason about the experimental results. Scientists often process data with queries that allow operations defined by arbitrary code. Meanwhile, they often require the provenance at a fine-grained granularity in order to explain how certain data is generated and which inputs are responsible for the data, to reproduce each experiment run or trial. Scientists need new capabilities to identify the sources of errors, find out why different code versions produce different results, and identify which parameter values affect output. Over time, scientists conduct different experiments on the same data. Those experimental runs might contain repeated sub-computations as well as their tracked provenance. They may want to reuse the repeated provenance without re-computing or compare the newly captured provenance with the provenance of a previous computation. This poses the necessity of new requirements on the provenance management system – notably, the identification property of the provenance. By provenance identification, we mean things: First, the origin should be identified in the provenance computation, where the same inputs should have the same ID or representation. Second, the provenance should be identified by the inputs, operations, and parameters. In other words, the provenance should be identified the same with the same inputs, operations, and parameters. On the other hand, data computations in scientific experiments often contain common operators like matching, ranking, and Extract-Transform-Loading type operators. Special mechanisms need to be designed to capture fine-grained provenance for such operators. Further, we need to archive provenance with the efficient verification property, which is important to prevent tampering with scientific computation
and maintain unforgeable results, especially in published works. Finally, it can be worthwhile in scientific fields to maintain a shared provenance data repository that collects published provenance with the corresponding computations for better collaborations.

**Enterprise data management.** Provenance is also increasingly important for business data management, especially those with activities, tasks, or events that involve customer data. Data provenance tools in business computations facilitate auditing, regulatory compliance, accountability, trustworthiness, security, and privacy. With the recent approval of the General Data Protection Regulation (GDPR), new data protection requirements are imposed on data administrators with respect to the processing of private personal data. When an individual invokes the GDPR to protect their data, enterprises need to validate whether their analyses incorporate the individual’s private records, to verify the compliance with the GDPR data protection law. Different approaches to implementing GDPR privacy guarantees are introduced in Stonebraker et al. (2020). Without maintaining the data provenance information, most of these implementations have to make onerous restrictions on the schemas and applications supported Schwarzkopf et al. (2019); Wang et al. (2019); Pasquier et al. (2019), which is not realistic for enterprise data. In reality, the data is stored with unconstrained schema and processed with arbitrary data movement. It needs protection from two aspects: first, the personal data of customers such as telephone numbers, credit card information, account data and customer addresses, must be used and protected by the enterprises under the GDPR laws. Second, the enterprises need to protect against attacks on the integrity of the data or code used in a query. Both of the above requirements need the enterprises to maintain detailed provenance on all the data movement Stonebraker et al. (2020); Kraska et al. (2019). Such provenance for individual records needs to be *self-certifying*, which helps to show whether a query result takes private inputs.

In this thesis, we focus on these problems in data analysis queries, which are widely used to handle data processing tasks in the science and business domain. We propose models and systematic approaches to collect, archive, and query the fine-grained provenance to work better at reproduction, debugging, certification, and data tamper-resistance.
1.1. Provenance’s Roles in Supporting Data Analysis

Provenance is a fundamental part of data analysis computations and participates in different computation stages and use cases. Figure 2 shows the pipeline stages of data analysis queries. Provenance computation begins in the data acquisition, then query set up (composition), and proceeds to execution (data processing). Next, the process follows to the analysis phase, being captured and archived simultaneously. The archived provenance data can be published together with the query and be shared with other queries or communities.

Provenance enables different capabilities in various application scenarios. It furnishes the hidden context information, helping scientists to discover new research opportunities, finding new problems and challenges. It also aids data analysts or developers in both science and business to detect and fix mistakes, acting similarly to a debug tool over the source code. In addition, for any query, provenance analytics is crucial for understanding a computation result and its dissemination (how the inputs are connected with outputs) and repeating a computation on a new dataset. It also helps to certify that all datasets were produced the same way, learning which sources are reliable based on validation of the results of a computation. Furthermore, provenance data can be archived and published, contributing to provenance sharing among different communities. We illustrate the various example scenarios in scientific data management, where provenance can facilitate better data analytics (Example 1.1.1 and 1.1.2). Analogous problems exist in business data management.

Example 1.1.1 (Gene Sequence Alignment)  In a gene sequencing and alignment workflow, the
inputs in Figure 1 are files consisting of a list of text strings, each of which represents a sequence read, generated by the sequencing machine. The workflow takes this sequence and trims the sequence to filter unneeded parts – this process is similar to the extraction stage in an ETL workflow. Then, it seeks to match each sequence read against a reference genome in another file. This is the matching step. The aligner takes two input files (one from the sequencing machine and one from the reference genome). It reads sequence strings one at a time and attempts to find the best matching against a substring of the reference genome. It outputs a list of pairs describing that matching.

We observe that most of the operators in the example query – such as trimming, extraction, and matching, are not standard relational operators. Besides, how these operations work exactly depends on the data type. For example, extraction would operate differently on strings, images, and structural data. Likewise, similar problems exist in the record linking and fuzzy matching for enterprise data. To capture the provenance of each computation record in such scenarios, we need to consider the operations beyond relational algebra and semantics of provenance for different data types.

**Example 1.1.2 (Understanding Alignment Results)** Several problems may occur during the gene sequence alignment.

1. Scientists might want to reason about one or several alignment results due to interest or a potential error. For example, to assess how many trial-and-error paths produced a particular result, how a given result was derived, and which processes led to a given result, etc.

2. A workflow is generally refined in iterations by scientists. In each iteration, the workflow gets changed a little bit compared to the previous trial run. It’s significant to find the associations or differences between different trials, especially how the results change and which part of the inputs leading to such changes.

3. The analysis workflow that gets updated over time is not controlled by the scientists (e.g., the aligner code is updated or the reference genome is changed). After each workflow change, the results of the computation should typically remain the same. However, the results occasionally differ in unexpected ways, thus introducing error to any data analysis that combines data processed...
under old and new workflow versions. Detecting the change in the output between various workflow versions is helpful, but identifying the subsets of inputs that caused the differences would be even more helpful, as it simplifies the debugging of the workflow logic.

This example propounds several new requirements on provenance modeling, computation, and storage. To illustrate, in case (1), the captured provenance needs to be stored and modeled properly to be path-traceable – given the output, provenance can be traced iteratively backward until the inputs, forming paths describing how the output is generated step by step from the inputs. On the other hand, case (2) and case (3) have similar demands – the archived provenance should be comparable. For example, we would like to know whether two outputs, maybe from different experimental runs, are generated from the same subset of input records; or, have overlapping sub-computations. There are analogous questions in enterprise data management. For example, over time, we might change the matching features or methods in record linking; we might use new filter criteria in data cleaning applications. In such cases, we would like to compare the provenance of the new computation output with that of the old ones.

**Example 1.1.3 (Tamper Resistance)** Scientists may want to publish their experiment results, which should include the input data set, the computations as well as parameter settings. Such information allows others to reproduce the experiments. However, there exist cases that the published data and processes are inconsistent and fail to reproduce:

(1) **Data computation results can introduce technical sources of errors into data analysis and storage systems.**

(2) **Users who generate an analysis, and seek to forge an inconsistent result — thus want to modify provenance records to change (a) specific input, output, or intermediate results, (b) specific query operations run, (c) specific versions of the source and/or binary code executed.**

(3) **Attackers who, given a published analysis result, wish to substitute a new result, or provenance for such a result.**
Example 1.1.3 describes the possible scenarios where the published data can not be reproduced. As all research data are susceptible to errors, misrepresentations, or be forged, maintaining the integrity of research data can be important, which ensuring that the data are complete, verified, and undistorted. Such certification property is also vital for business data. Except for the integrity and tamper-resistant requirements of the business data, we also need to consider data protection and privacy, especially for customers’ personal information. In order to avoid paying costly penalties, businesses must comply with GDPR requirements by validating whether their analyses incorporate the individual’s private records.

**Summary.** To serve the above use cases in Example 1.1.2 and Example 1.1.3, a provenance management solution should consider not only the provenance capture but also the provenance archive. To capture the provenance in general scientific and enterprise data management, the system should be able to compute the provenance of operations beyond standard relational algebra such as approximate matching or aggregation with user-defined functions in arbitrary code. Meanwhile, during the computation, the system should also consider that the provenance annotations might be divergent with different data types.

In addition, a provenance management solution should provide provenance archive services to prepare the captured provenance for future uses. First, it should support provenance traceability over the archived provenance data. To reason about the analysis result or debug the data analysis query, the provenance archive system should enable from simple attribute lookup, to iteratively lookup of all the computation steps from the inputs to the outputs. The archived next provenance needs to be comparable across different data analysis queries. Finally, for reproducibility and auditing use cases, we need the provenance archive system to be tamper-resistant.

1.2. Provenance Management for Modern Data Science Applications

Modern data science applications often consist of complex, ever-changing data analysis computations over big data. As data and code change across time, we must be able to facilitate the consistency and reproducibility of the analysis results. Such needs spurred the development of provenance management tools. Ideally, a provenance management system should be composed of the provenance capture layer and the provenance archive layer.
In the *provenance capture layer*, to achieve a rigorous and reproducible data processing, we need to understand specifically how and why records exist, which is **fine-grained provenance**. Fine-grained provenance enables debugging, tracing back to integrated sources, etc. But most of the current studies focused on the standard relational algebra operations: Select, Project, Join, Union, and Group-By queries (SPJU + G), which is inapplicable in many real-world science and business data analyses. The real-world data analysis queries need operators **beyond relational algebra**. Such operators are often user-defined and might be in different programming languages (in C, python, Java, etc.). We should be able to capture provenance for **user-defined operators in different languages**. Besides, in order to annotate the user-defined function, we need to instrument the code in order to capture the provenance. In such cases, a **synthetic instrumenting mechanism** is required to reduce the instrument overhead.

Furthermore, an ideal provenance management solution should provide two ways to return only related intermediate data and processing stages instead of everything. First, the system should be able to recompute the portion of the data analysis query necessary to produce the provenance of a specific user selection. This is a lazy strategy to compute provenance **on-demand**. The lazy approach computes the provenance of data only when needed Cui and Widom (2003); Buneman et al. (2001b). This strategy reduces the big overhead in recording every derivation in recompiling often-large source code bases. Second, the system should support **eager** recomputation and materialization of an entire query’s results and archive the provenance to allow future inquiry about the provenance of any intermediate or output result. An advantage of using an eager method to compute data provenance is that the source databases need not be probed since the provenance can be fully determined by looking at the annotations associated with a piece of output data. A disadvantage is that the eager approach incurs additional overhead to compute the output and store annotations in the output Bhagwat et al. (2005).

In the *provenance archive layer*, a preferred provenance management system should support basic provenance storage and query operations as well as special requirements such as identification and tamper-resistant, for data consistent checking, reproducibility, and auditing use cases. Moreover,
the storage should be efficient and compact, adding less space overhead. To be more specific, the provenance archive layer of the system should first enable the storage of provenance computed either on-demand or eagerly, in one data analysis query or across many queries over time. Such stored provenance then should be efficient and feasible to be queried, serving the analysis of different use cases like debugging, reasoning data, certifying results and propagating trustworthiness, etc. To begin with, it should support attribute lookup with certain criteria and transitive closure of ancestry for debugging and reasoning purposes. Next, the archived upcoming provenance needs to be comparable across different data analysis queries - **identification and self-certifying**, where the same provenance should be annotated with the same provenance expression. On the other hand, it’s not uncommon that we do the same computation or sub-computation over the same data repeatedly in data analysis queries, especially in scientific experimental analysis and enterprise decision making analysis. If the provenance system only stores the repeated provenance once, it can largely reduce the space overhead of the provenance archive - **compression of repeated provenance**. Finally, concerning the integrity requirements of scientific experiments and the audit requirements of business data analysis, the archived provenance should be **tamper-resistant**, providing certification ability of the consistency and correction of results.

In this thesis, we propose PROVision, a provenance-driven data analysis tool that aims to facilitate reproducible data science and auditable data queries. The system comprises the provenance capture layer and provenance archival layer. The provenance capture layer, like provenance APIs, supports tuple and sub-tuple based provenance reconstruction for a wide array of deterministic ETL and matching computations. But at the same time, it extends database-style provenance techniques which capture equivalences, and further supports optimizations with our self-build query engine. In addition, the provenance capture layer enables lazy evaluation to compute provenance on-demand. On the other hand, the provenance archival layer provides provenance storage management that facilitates simple provenance lookup queries and more complex provenance path queries like derivation and transitive closure queries. PROVision allows efficient storage for computation computed over time through derivation-based compression. Further, in order to achieve reproducibility and auditing, the provenance archival layer of PROVision supports tamper-resistant property via crypt-
1.3. Research Challenges and Contributions

The need to facilitate reproducible data science and auditable data analyses has driven the development of many data provenance management tools. Data provenance not only is the key to capture, reproduce, and certify scientific results but also increasingly serves a similar role in audit logging in the enterprise. Ideally, provenance management tools must ultimately provide two types of services: (1) capturing data provenance for common computations in an appropriate provenance data model, and (2) providing a queriable, auditable, tamper-resistant provenance-augmented data archive for analysis results. Full implementation of such a provenance management system should handle real-world diversified computations of data analysis queries and store provenance in a proper way efficiently, which poses several fundamental challenges.

In the provenance capture layer, we need to compute the fine-grained provenance of operators beyond relational algebra. Several challenges exist in provenance modeling and the efficient implementation of such a model. First, every common ETL-style operator might be defined differently and written in different languages. How do we design a systematic way to model and capture the fine-grained provenance of such operators appropriately, avoiding human instrumenting the raw code of every user-defined operator? How do we make our provenance computations generally applicable across data analysis queries in different language and data types? On the other hand, the overhead of provenance computation is often significant. How do we optimize in provenance computation without affecting the provenance and how do we reduce the overhead by only computing the provenance on-demand? Finally, the user-defined operators are oftentimes open-source tools with specific parameter settings, which might introduce errors or troubles. How do we design a mechanism to troubleshoot such novel problems with the help of data provenance?

In the provenance archive layer, except for the basic provenance storage and query requirements, we have two key aspects to achieve. First, we would like to enable low-overhead storage. However, as the repeated provenance of the sub-computations run in different analyses over time, how do we ensure to store the repeated provenance only once? Second, we would like to use the provenance
to certify analysis results and how do we verify the provenance and guarantee the tamper-resistant without re-run the whole computations?

To tackle the above challenges, this thesis develops a full provenance driven data analysis tool – PROVision, aiming to troubleshoot the problems in general ETL-style data analysis and provide efficient storage as well as the certification property for auditable data queries. In particular, to obtain fine-grained provenance, we leverage ideas from relational query processing but develop novel ways to support complex data formats and user-defined functions. Our techniques rely on expert-provided declarative descriptions of the key functionality within modules composing the data analysis queries (including extraction, transformation, and approximate alignment of structured data elements); we automatically compose these to capture the key functionality within complete data analysis queries. We incrementally reconstruct fine-grained provenance for records of interest for many data science and ETL tasks. Further, to support result troubleshooting, PROVision finds differences among outputs between different code versions of the use-defined functions, and minimally reproduces the provenance “traces” to the underlying source data — even through complex matching and blocking algorithm and hierarchical data structures.

For the provenance archive layer, we develop and study techniques for efficiently capturing fine-grained provenance while archiving results, in a way that (1) takes advantage of repeated computation for efficiency, and (2) uses cryptographic hashes that allow the provenance to be self-certifying. We develop encoding schemes that enable low-overhead storage as well as fast provenance queries, and also consider trade-offs between speed and cryptographic security. We target a variety of workloads in the analytics space, largely focused around matching, data wrangling, and data transformation — where tracing of individual results is often useful not only for reproducibility but also for reasoning about the effects of change. To be more specific, we focus on how to create a tamper-resistant archive of a data analysis results and their fine-grained provenance generated over time, to enable reproducibility, querying of individual records’ provenance, and verification of authenticity. We develop techniques for derivation-based compression: different computational analyses will often share computational structure, and may also be computed over shared data sources or over
inputs with shared records. In such scenarios, we exploit structure and repetition across queries and subqueries. (Of course, we assume such compression is accompanied by value-level compression within the storage system Abadi et al. (2006), but we do not focus on this aspect because it is orthogonal to provenance storage.)

Our system, PROVision reproduces fine-grained, record-to-record provenance across a wide variety of ETL and data processing queries. Our techniques generalize ideas from database provenance Green et al. (2007b) to handle ETL-style operations over hierarchical data and text, and to capture provenance for operations that extract embedded content and perform matching or alignment operations Zheng et al. (2019). Our provenance storage techniques to achieve a compact and tamper-resistant storage, though focus on queries within the relational algebra augmented with certain classes of user-defined functions, can be easily extended to other provenance capture systems Zheng and Ives (2021).

Our contributions are as follows:

- We extend the semiring model Green et al. (2007b) to incorporate two classes of user-defined code (often in combination): code that extracts and transforms sub-content from structured attributes (e.g., extraction from structured files; substrings in genes; MapReduce map functions), and code that operates on subsets of records within groups. The model supports for user-defined blocking, transformation, and ranking functions – with datatype-specific optimizations.

- We provide semantic descriptors based on algebraic operators for capturing the unnesting behavior, and for tracking unnesting within provenance, which handles arbitrary ETL data and code. We develop a mechanism whereby an expert specifies any algebraic equivalence for the UDFs when appropriate.

- We implement novel techniques, including optimization strategies that incorporate expert-provided equivalences and do selective recomputation. We do case studies for troubleshooting results, explaining differences, and discovering parameter settings. We argue that our model
and strategies are fast enough for practical use.

• We develop a full implementation and conduct comprehensive experimental evaluations of our techniques’ performance and scalability, versus alternative methods on datasets from different domains.

• Provenance encoding techniques that ensure results computed using the same algebraic expressions over the same input data will be stored exactly once, irrespective of the number of queries; and which are self-certifying (tamper-resistant).

• Algorithms that exploit these encoding techniques to compare provenance to judge equivalence and explain the “earliest” point of divergence between two queries or computations.

• A cost model for provenance storage, as well as an understanding of when the optimal query evaluation plan also results in the optimal storage scheme.

• An implemented tamper-resistant provenance archival layer that extends the open-source PROVision Zheng et al. (2019) system.

• Performance analysis over a variety of workloads taken from ETL and scientific data management.

1.4. Roadmap

The thesis is organized as follows. In Chapter 2, we provide an overview of the background of provenance management and briefly introduce the technical foundation from prior works. Next, we describe the problems that this thesis addresses and outline an overall picture of our PROVision in Chapter 3. And then specify the model and implementation of provenance capture in Chapter 4 as well as provenance archive in Chapter 5. We survey related work in Chapter 6. Finally, we conclude and point out directions for future work in Chapter 7.
CHAPTER 2 : Background and Foundations

The problem of systematically capturing and archiving provenance for data analytic tasks has recently received significant attention because of its relevance and benefits to a wide range of domains and applications. Provenance-aware computations serve important roles in the data processing. It allows the systems to have the ability to answer the questions like how data was created and what the input datasets and parameters were. Without provenance data, it is impossible to answer how data is processed.

Provenance has been extensively studied across the fields of databases, scientific workflows, distributed systems, and networks, etc. Each of these research communities has adopted slightly different formalisms for provenance. In the database community, Buneman et al. (2001b) define data provenance as the description of the origins of data and the process by which it arrived at the database. The work of Green et al. (2007b) has led to the definition of provenance using polynomials, which is a powerful model that applies to various database semantics (relational, incomplete, and probabilistic database) as well as provenance semantics (how, what, and where). This work models provenance as semirings of polynomials and provides a formal definition of the properties that must hold for these values and operations, namely that they satisfy the constraints of a semiring. The operators in the semiring follow associative and communicative properties which enable the potential equivalent provenance as desired in our provenance compact archive. Our PROVision system is built by extending the provenance semiring model, by incorporating general operators such as extraction, approximate matching, and ranking, with user-defined functions. In scientific research, a scientific workflow model Barker and Van Hemert (2007) is widely used to conduct scientific experiments. Most of the scientific workflow provenance management system treat the workflow modules as “black box” and only captures coarse-grained provenance, which is not suitable for reasoning about individual records. Provenance gives credibility to the experiment, proves its results, makes its reproduction possible, and opens discovery opportunities in comparative studies Biton et al. (2008); Davidson and Freire (2008). For business, provenance has been used for regulatory, compliance Hasan et al. (2007), and privacy prospect Townend et al. (2013). Addition-
ally, a number of efforts have integrated provenance into smart transactions in a blockchain Neisse et al. (2017); Ruan et al. (2019) and into storing coarse-grained provenance Liang et al. (2017). We are considering cryptographic hashing, which is similar in blockchain systems, for both security and sharing purposes with fine-grained provenance. For network, provenance has been successfully applied to network debugging Wu et al. (2014); Handigol et al. (2014), root cause Handigol et al. (2014), and network trust problems Shebaro et al. (2012). While recent work on provenance for networks Zhou et al. (2011b) develops self-certifying techniques, where it was used to sign distributed system events in a log. These approaches do not naturally extend to fine-grained provenance.

In this chapter, we review the background of provenance management including the roles (Section 2.1) that provenance played in the data processing pipeline, the life cycle of provenance data management (Section 2.2), and the fundamental prior work in the relational database, scientific workflow, and business data analysis (Section 2.3).

2.1. Integrating Provenance in Data Processing Pipeline

Provenance plays important roles in the data processing pipeline. In this section, we first introduce the concept of data processing pipeline and the stages that it is composed of, then we present how the provenance participates at each stage and highlights the parts that PROVision covers.

Data produced by applications, devices, or humans must be processed before it is consumed. The processing pipeline represents the flow of data between two or more stages, each stage consumes data units from a previous stage and produces output units. The output can be an endpoint or input
for some other processing stage. Given the scale and processing complexity inherent to Big Data, programs are typically organized into a chained series, with the output of one program becoming the input to the next.

Data pipelines are important for data science projects. Data scientists need to find, explore, cleanse, and integrate data before creating or selecting models. They may need to iteratively refine their models or parameters. However, testing and debugging big data processing pipelines is an expensive and time-consuming process due to the extremely large input data. It is clearly infeasible for developers to read through the production data apriori and design test inputs for their application. Finding intermediate data records responsible for a failure corresponds to finding few records in millions, if not billions, of records. The similar problem exists when a user wants to investigate the probable cause of delay in the processing. Finding straggler records are essential for a user to improve the runtime of the application.

A typical data processing pipeline is shown in Figure 2. We highlight the stages that this thesis is going to cover. A data scientist starts by data discovery from a variety of related data sources, like gene sequences from sequencing machines, customer transaction data from the enterprise’s backend database or semi-structured data crawled from the web. Then the query is created with templates from the shared query library or the scripts written by the data scientist. The new setup query takes the data as input from the first stage, usually extracting out some form of structured (possibly hierarchical) records, filtering and transforming them in some desired way, and then matching or aligning them with reference records, concepts, or instances. At the same time, the system computes the provenance and stores the provenance information for future use. After data processed and provenance collected, the data scientist generally observes, certifies the correctness, troubleshoots, or reasons about certain results with the help of provenance information. As the data analytic tasks are usually data-driven, the query is changing and refined after the data scientist analyzes the results. Once the data scientist is happy about her results. She would publish his query to the shared query library as well as the data and its archived provenance. Another data scientist saw the published data and he is interested in the results and would like to validate and reproduce the experiments. He
would need to first find the corresponding archived provenance data and verify the integrity of his provenance and then reason or reproduce the results with help of the archived provenance data.

The data processing stage in Figure 2 generally contains several computation steps, including Extraction-Transformation-Loading plus some approximate matching, ranking, and aggregation, as well as some more complex computations like exploratory data analysis, model construction, and prediction with machine learning. At each of the computation steps, the data scientist might want to do troubleshooting and look at which data is leading to unexpected behavior with querying the provenance archival.

The system PROVision targets the dark orange rectangles in a data processing pipeline. We focus on how to efficiently (1) capture the whole provenance eagerly and archive the provenance for the next debugging or refinement stage, or for publishing the data as well as their provenance for reproducibility by other people; (2) given a subset of result records that the scientist is interested, compute the provenance on-demand; (3) for the purpose of data provenance integrity and verification of authenticity, create a tamper-resistant archive of provenance by cryptographic hash. Note here that the data processing pipeline is sometimes scheduled to run periodically in a distributed environment, where multiple pipelines are executed at the same time. In this thesis, we are not considering the concurrency and conflicts among multiple pipelines, we only focus on the most simple data pipeline chaining pattern.

2.2. Provenance Life Cycle

As data flows through each stage in the processing pipeline, provenance is handled accordingly to aid troubleshooting, certification, and reproducibility of data analysis queries, which forms the provenance lifecycle. Provenance lifecycle consists of three phases, in whatever domain, granularity, or whatever provenance semantics and models are involved, which are (1) provenance capture, (2) provenance archival, (3) provenance query and analysis. Figure 2 shows the roles that provenance plays and the connections between phases in the processing pipeline. Those roles of provenance form the provenance lifecycle, among which the provenance capture phase is the most extensively studied. As we will discuss details on the related work of provenance capture in Section 2.3,
in this section, we mainly focus on the implementation considerations regarding the provenance capture phase.

2.2.1. Provenance Capture

The provenance systems should be minimally intrusive to capture provenance, which can generally be costly Cheah et al. (2013). Work has to be done to capture provenance, especially for fine-grained provenance. In the implementation of provenance systems, provenance can be integrated at the computation operator level, the process level, or the workflow level Freire et al. (2008). Incorporating provenance at different levels may have different impacts on the performance of an application due to intrusiveness. In addition, optimization of the query can be considered to reduce the cost of provenance capture.

2.2.2. Provenance Archive

The captured provenance needs to be stored for future use. Depending on the granularity, data type, and goals of usage, a variety of storage models are available for provenance information archival Gehani and Tariq (2012); Muniswamy-Reddy et al. (2006b). The annotated provenance information can generally be large and minimal storage overhead is considered for provenance archival Xie et al. (2012).

**Storage model** generally depends on the granularity requirements of provenance. It also concerns efficient query support, storage size, and sometimes inference support. The coarse-grained provenance logging the provenance at the file level and the storage model is *file-system* storage Muniswamy-Reddy et al. (2006b). In *file-system* storage, provenance is stored in original data files, which can be images and videos Gehani and Lindqvist (2007); Muniswamy-Reddy et al. (2006b). The *file-system* storage model has the drawback of limited provenance querying capabilities. Retrieval attributes of objects based on user-specified criteria can be time-consuming. Meanwhile, the fine-grained provenance is annotated at the record level, or even at attribute, pixel, and char level. The storage of fine-grained provenance often relies on storing provenance records in SQL databases Sar and Cao (2005). The storage model has support for JDBC-compliant databases like SQLServer, MySQL, and Oracle. Along with SQL databases, SPADE Gehani and Tariq (2012) has the capability for graph storage using Neo4j. Milieu Cheah et al. (2013) framework utilizes a NoSQL database Mon-
goDB. Our PROVision deploys the SQL database storage model for fine-grained provenance, with simple relations to enable direct query or recursive query for tracing ancestors.

**Minimal storage overhead** is another major issue in provenance archival. Traditional compression techniques are not directly queriable, which is inapplicable directly to overcome provenance storage overhead. The efficient provenance-aware system demands the optimized provenance storage mechanism. One of the factors that contribute to the large provenance size is the duplication of information. Our work considering storing the duplicated expression only once to overcome the overhead. We are inspired by recent work of Lee and colleagues Lee et al. (2019), which considered how to rewrite or factor relational algebra computations to more efficiently store provenance for individual queries.

### 2.2.3. Provenance Query

Provenance collection and storage effort is meaningless without the support of query and analysis. There are generally three categories of provenance queries Patni et al. (2010); Hasan et al. (2007):

- **Attribute Lookup**: Query for provenance record matching certain criteria, which has no recursion. Most of the provenance management systems inherently support these types of queries.

- **Transitive Closure of Ancestry**: Query for data objects recursively by specifying constraints on provenance information, like the stopping conditions, tracing depth, etc.

- **Integrity Checking**: Given certain records, query to check whether the data is altered or added forged information by an adversary.

The first two categories of queries are straightforward. Attribute lookup is the most commonly used and it’s a basic query that can form the other two types of queries. For example, in the query to trace from an output until its inputs, the very first step would be an attribute lookup query to find the output’s provenance. Then a recursion query would be posted to recursively trace until there are no ancestors of the returned records. At each step of the recursion, an attribute lookup query would be executed. The last integrity query is in order to achieve trustworthy provenance. The integrity of the data provenance should be tamper-evident. The integrity checking can ensure
that no forged data in the scientific experiments, no added entries into the ownership sequence in a business blockchain. This integrity query Patni et al. (2010) can be accomplished by signatures, checksums, signed hashes, etc.

2.2.4. Summary of the Provenance Lifecycle

This thesis aims to cover the entire three phases of the provenance lifecycle. In the provenance capture phase, it’s necessary to design mechanisms that allow the ability to instrument the users’ code offered by provenance APIs as well as the database-style techniques. The users need to compose the query with declarative descriptions where each computation step refers to some procedural code library. In addition, our solution has its own query engine to optimize the data analysis queries to reduce the cost of provenance capture. In the provenance archive phase, we deploy the SQL database storage model with simple relations for better performance. Further, we exploit to make use of the identification of provenance input data as well as the provenance expressions to avoid redundant provenance storage hence reducing the overhead. The provenance query phase relies on the provenance archive thoroughly. Here, we take advantage of the simplicity of the storage relations and the identification property of the archived provenance. We support all three types of queries: attribute lookup, which is a simple lookup for certain attributes of the provenance storage relation; transitive closure of ancestry, which is enabled with the recursive query in SQL database; and integrity checking with the comparison of identifications.

2.3. Fundamental Prior Work

This thesis is theoretically based on some fundamental prior works in the provenance of relational databases, scientific workflows as well as business data analysis. We present these fundamental prior works in this section. We summarize other related works of provenance in general in Chapter 6.

2.3.1. Provenance in Relational Databases

Provenance capture has been addressed over the past 15 years in database communities and scientific computing. In database communities, A number of notions of provenance in databases have been proposed in the literature Green et al. (2007b); Buneman and Tan (2007); Tan et al. (2007). They have extensively studied fine-grained provenance, starting with the work by Cui and Widom Cui (2001) and Tan et al. Buneman et al. (2002), with the now widely-accepted foundations
being established by Green et al. Green et al. (2007b,a) and more recently being extended to a variety of system implementations Glavic and Alonso (2009); Karvounarakis et al. (2010) and to richer operators Amsterdamer et al. (2011e); Foster et al. (2008a); Yan et al. (2016). The most common forms of database provenance describe relationships between data in the source and in the output, for example, by explaining where output data came from in the input Buneman et al. (2001a) or describing in detail how an output record was produced Green et al. (2007b). To be more specific, the provenance of a result can be obtained by annotating attribute values in the database and then propagating these annotations through the processing of relational queries all the way to the result. Then, a result value can be associated with its original location through the propagated annotation. In this subsection, we present a background introduction to the provenance semiring model for relational algebra and its extension to queries with aggregations.

**Provenance Semiring**

Green et al. Green et al. (2007b); Foster et al. (2008b) formalize a notion of how-provenance for relational algebra in terms of an appropriate “provenance semiring”, and extend their approach to handle recursive datalog. Subsequently, an interesting application of how-provenance appears in the context of ORCHESTRA Green et al. (2010); Ives et al. (2008), a collaborative data sharing system in a network of peers interconnected through schema mappings. An extension of the semiring model of Green et al. Green et al. (2010) to schema mappings is used in ORCHESTRA to efficiently support trust-based filtering of updates, and incremental maintenance of peers’ databases with updates in the system.

The provenance semiring model captures how an output tuple is derived according to the query. To illustrate, for the database in Figure 3 consider the query $Q_2$ in Figure 4 which asks for all cities where tours are offered Cheney et al. (2009). Intuitively, the provenance of the output tuple (San Francisco, 415-1200) is represented as a polynomial, which in this example is $t_1^2 + t_1 \times t_3$. The polynomials for each output tuple are shown on the right of the result of $Q_2$. In this example, the polynomial describes that the output tuple is witnessed in two distinct ways: one using $t_1$ twice, and the other using $t_1$ and $t_3$. 

21
They define a $K$-relational semantics ($K, 0, 1, +, \cdot$) to be an algebraic structure with two binary operations and two distinguished elements. If we assume that $K$-relational semantics satisfies the same equivalence laws as positive relational algebra operators over bags (i.e., union (+) is associative, commutative and has identity $\phi$, join (\cdot) is associative, commutative and distributive over the union, and projection and selection commute with each other, as well as with union and join), Green et al. (2007b) conclude that ($K, 0, 1, +, \cdot$) must be a commutative semiring. Recall that an algebraic structure ($K, 0, 1, +, \cdot$) is a commutative semiring if ($K, 0, +$) and ($K, 1, \cdot$) are commutative monoids, \cdot distributes over + and $0 \cdot a = a \cdot 0 = 0$, $\forall a \in K$. Here, a commutative monoid is an algebraic structure ($K, id, op$), where $op$ is associative and commutative and has identity element $id$.

The semiring operations essentially document how each output tuple is produced from source tuples. Intuitively, if each source tuple in a database $D$ is tagged with a distinct tuple id, the semiring gives us the how-provenance for each output tuple in the form of a polynomial with coefficients from the
set $N$ of natural numbers and indeterminates (or variables) from the set of source tuple ids.

**Extended Semiring for Aggregations**

Amsterdamer et al. Amsterdamer et al. (2011d) investigated the extension to aggregation and group-by of provenance semiring polynomials, and found that capturing their provenance requires annotating with provenance information, not just tuples, but also individual values within tuples. Then they generate the $N[X]$-relations by extending their data domain with aggregated values. For example, in the case of $SUM$-aggregation of a set of tuples, such a value is a formal sum $\sum_i t_i \otimes v_i$, where $v_i$ is the value of the aggregated attribute in the $i^{th}$ tuple, while $t_i$ is the provenance of that tuple. We can think of the tensor products $\otimes$ as an operation that associates the semiring annotation with the possible aggregation values. This extended model on set/bag semantics for $min/max/sum$ computes with semiring homomorphisms. The work in Amsterdamer et al. (2011d) also described a precise algebraic treatment of aggregated values and equivalence laws that are based on semimodules and tensor products.

Our work is built upon the provenance semiring polynomials by extending to the user-defined function. The provenance semiring model provides a solid theoretical foundation of the correctness of our fine-grained how-provenance model. Additionally, the associative and commutative property in the semiring polynomial is essential for query rewrite and optimization. Our extended provenance semiring model also carefully preserve the algebraic equivalent rules among computations for our query optimizer (including equivalences that hold for particular data types and UDFs).

**2.3.2. Provenance in Scientific Workflows**

Provenance is also an active topic of research in data analytic tasks, eg. scientific workflow. Workflow management systems, such as Taverna Oinn et al. (2006), Kepler Ludäscher et al. (2006), Vis-Trails Bavoil et al. (2005), and Galaxy Goecks et al. (2010), have become more and more popular as a way of describing and processing data-intensive analysis. In such systems, a workflow can be graphically described by chaining together different tasks (e.g., for aligning biological sequences), where each task may take input data from previous tasks, parameter settings, and data from external sources. In other words, a workflow specification can be modeled as a graph, where each node rep-
resents a module and each edge captures the flow of data between these modules. Here, provenance is typically captured by log files, scripts, and/or scientific workflow systems Oinn et al. (2006); Ludäscher et al. (2006); Goecks et al. (2010), and is thus coarse-grained file-to-file provenance. Techniques for OS logging of low-level provenance have also been investigated Stamatogiannakis et al. (2016); Muniswamy-Reddy et al. (2006a).

The above-mentioned scientific workflow management systems e.g., Taverna Oinn et al. (2006), Kepler Ludäscher et al. (2006), VisTrails Bavoil et al. (2005) and Galaxy Goecks et al. (2010), incorporate coarse-grained provenance. In contrast, our work provides finer-grained analysis than those scientific workflow management systems. We resemble recent attempts to bridge between the two models for data-driven programs, e.g., Pig Amsterdamer et al. (2011b), Spark Interlandi et al. (2015), and custom relational engines Psallidas and Wu (2018). However, PROVision is differentiated by its support for content extraction, handling of nested data, approximate matching, and blocking operations.

The automated tracking and storage of provenance information promise to be a major advantage of scientific workflow systems. As described in Davidson et al. (2007), a workflow can be graphically described by connecting together tasks, where each task may take input data from previous tasks, parameter settings, and data coming from external data sources. In general, a workflow specification can be thought of as a graph, where nodes represent modules of an analysis query and edges capture the flow of data between these modules.

For example, Figure 5 shows workflow specification, which describes common analysis steps in the gene sequence alignment. This workflow first takes the testing biological tissue into the sequencing machine which generates the DNA sequence from the tissue. Those sequences are supplied as input data to the alignment workflow and the workflow scripts perform a multiple sequence alignment over those inputs. The workflow script can be expanded into a detailed workflow that chains different modules: trim, align and verse. Between modules, data objects are represented as nodes that are generated by the previous module and supply to the next module. The result of executing a scientific workflow is called a run. As a workflow executes, data flows between module invocations.
Most of the scientific workflow systems capture the provenance information in a coarse-grained way. For example, Taverna Oinn et al. (2006) records the data flow information at the file level with event logs. Those logs capture the dependencies of the data objects by using analysis of the logical order of the events during the workflow execution. However, modules are treated as “black boxes”. In contrast, some scientific workflow systems treat the models as “white boxes”. One family of such systems takes database-style operators and deploy the provenance analysis techniques in Buneman and Tan (2007). Others use provenance API techniques Wu et al. (2013), which allow programmers to manually instrument arbitrary code with API calls, thus revealing fine-grained provenance. However, such APIs impose notable overhead during standard computations, and they produce provenance that depends on the order of evaluation of operations.

For those scientific workflow systems which treat modules as “black boxes”, another category of provenance is designed to care more about people and activities. They define provenance as information about entities, activities, and people involved in producing a piece of data or thing, which can be used to form assessments about its quality, reliability, or trustworthiness. PROV-DM Moreau et al. (2013) is the conceptual data model that forms a basis for the W3C provenance (PROV) family of specifications. The PROV Ontology (PROV-O) Lebo et al. (2013) expresses the PROV Data Model (PROV-DM) using the OWL2 Web Ontology Language (OWL2). It provides a set of classes,
properties, and restrictions that can be used to represent and interchange provenance information generated in different systems and under different contexts.

2.3.3. Provenance in Business Data Management

Provenance is also essential to the business domain where it can be used to track the creation of intellectual property, and provide an audit trail for regulatory purposes. Traditionally, business stakeholders work with an organized data schema, where the structure and semantics of the data in use are shared across the corporation. This mechanism relies on clearly established schemas for data interchange and usually with trusted partners. Nowadays, business users demand an integrated view of different types of data collected from multiple data sources without pre-defined schemas. Yet, a large proportion of businesses deal with bad quality data, and this is accentuated when they are aggregated from different parts of the enterprise into a data warehouse Hanrahan (2004). Sources of bad data need to be identified and corrected to maintain the data quality and avoid costly errors in business forecasting. Therefore, a mechanism for the normalization of heterogeneous provenance data is an implicit requirement Hammad and Wu (2014). Provenance tracing in data warehouses needs to consider the integrated view of data from multiple sources while retaining the depth of the data and summarized information on it Cui and Widom (2003).

Additionally, to meet the requirements of the identification and trustworthiness in business applications, a number of efforts have integrated provenance into smart transactions in a blockchain Neisse et al. (2017); Ruan et al. (2019) and into storing coarse-grained provenance Liang et al. (2017). A blockchain is a chain of blocks, in which each block contains many transactions and is linked with the previous block via a hash pointer. As a blockchain is decentralized, which allows mutually distrusting parties to manage the data together instead of trusting a single party. Also, it supports integrity protection (tamper evidence) via cryptographic hashing to all transactions recorded in the ledger, which ensures the security of the whole transaction history Ruan et al. (2019). Our work PROVision borrows the idea of cryptographic hashes that allow the provenance to be self-certifying, auditing, and accountability Haeberlen et al. (2007). We can verify that an output result was produced through a particular derivation from a given input, and any tampering with the structure or data would not satisfy the hash.
CHAPTER 3 : Problems and Overview of the PROVision System

In this chapter, we first represent some open issues that this thesis addresses in fine-grained provenance capture and compact archive with tamper-resistant (Section 3.1). Next, we provide an overview of our solution – PROVision system as a provenance management tool (Section 3.2). At last, we focus on the system architecture and the functionality of each module (Section 3.3).

3.1. Open Problems Addressed in this Thesis

Still, there exist some open issues in the prior techniques introduced above for both provenance capture and archive. In this section, we summarize those open issues that the thesis addresses. We start to specify the details of those open issues by continuing the gene sequence alignment examples that have been introduced in Chapter 1.

Example 3.1.1 Here is a brief recap of the gene sequence alignment example. Our life sciences collaborators run a high-throughput gene sequencing workflow. From a more detailed semantic perspective, please see Figure 6. The sequencing machine generates a file consisting of a list of text strings — each of which represents a sequence read from the tissue. The workflow takes this sequence and seeks to match each sequence read against a reference genome in another file. The aligner takes two input files (one from the sequencing machine and one from the reference genome). The aligner takes a list of strings and a very long reference string, and it reads sequence strings one at a time and attempts to find the best matching against a substring of the reference genome. It outputs a list of pairs describing that matching.

A problem that occasionally occurs (and causes significant issues) is that the analysis workflow
gets updated over time (e.g., the aligner code is updated or the reference genome is changed). After each workflow change, the results of the computation should typically remain the same. However, the results occasionally differ in unexpected ways, thus introducing error to any data analysis that combines data processed under old and new workflow versions. Detecting the change in the output between various workflow versions is helpful, but identifying the subsets of inputs that caused the differences would be even more helpful, as it simplifies the debugging of the workflow logic.

Figure 7: The last two steps of the sequence alignment workflow

**Example 3.1.2 (Provenance Capture)** Revisiting our gene sequence alignment workflow in more detail, please see Figure 7. The high-throughput sequencer, when run over a piece of tissue, outputs a list of files containing sequences of DNA fragments (one per line). These fragments first go through a series of trimming steps (not shown in Figure 7) with conditions, generating well-organized and effective sequences. These are fed into the **Alignment**, where they are compared against a complete reference genome (in a separate file), whose individual sequences are actually mapped to known genes. In Figure 7, we can see details of the aligner’s output **Align-n.txt**. **Module Verse** then takes the alignment results and the reference gene expression library as inputs. It first joins each of the files **Align-*.txt** and **Ref-expression.txt** to predict which RNA each input sequence would express, and then determines a count for each RNA expression type. The final output would show the estimated amount of each RNA expression generated by the input tissue. Observe that
Figure 8: Common data analyses query template involving extraction, transformation, alignment, and selection or aggregation of matches. Shared by many ETL/data cleaning scenarios, information extraction scenarios, and biomedical informatics scenarios such as gene sequencing.

from an abstract perspective, the workflow really operates over a series of records — namely, the sequences and positions, RNA expressions, etc. These are stored as textual strings.

Over time, the module binaries are changed in ways the developer believes should not affect output. However, in some cases bugs or race conditions are introduced and have unexpected changes. We may wish to run the two different workflow versions (pre- and post-change) and compare outputs. If these outputs differ, the question is what (minimal) combination of inputs cause this difference.

Example 3.1.2 describes a simplified data analyses task of gene sequence alignment, which shows the basic modules. Masses of such tasks in scientific fields have similar patterns, as well as enterprise data analysis tasks such as record linking, data cleaning, data-driven decision making, etc. Figure 8 illustrates a very common data processing “design pattern” used across enterprise ETL or Master Data Management, in information extraction, in ontology alignment, and in biomedical informatics. The basic pattern involves a sequence of operators including taking data from a file, extracting out some form of structured (possibly hierarchical) records, filtering and transforming them, and then linking or aligning them with reference records, concepts, or instances. Importantly, the code modules defining the operators are typically arbitrary procedural operations and the alignment typically involves some form of approximate matching (e.g., string edit distance) using a technique termed blocking Christen (2012), to limit the number of comparisons performed. For provenance capture, the thesis focuses on data analysis queries with the ETL-style design pattern shown in Figure 8 as well as its variations – for example, we might have no filter and transform step but have multiple extract steps.

29
Next, we summarize some important observations of the data analysis queries in real-world scientific and business applications. Basically, the queries do not run in SQL Psallidas and Wu (2018), Spark Zaharia et al. (2010), or Pig Olston et al. (2008) (they are in C, Python, Java, etc). They also require more than standard Relational Algebra operators like extraction, approximate matching, ranking, which are neither SPJU nor standard aggregation studied in prior work. Prior work requires specific aggregation operators; or considers provenance to be ALL inputs to each aggregate function. Consequently, the data generated in those queries is not in first normal form, but works on content embedded in attributes (images, substrings, text) that are extracted using user-defined functions. The provenance annotations should depend on data types. For instance, in the above Example 3.1.2, the trimming step extracts sub-strings from the input gene sequence strings. Similar examples of alignment or annotation of projected data elements can be found in other domains with other data modalities: for instance, image extraction may take an image file and return a polygon or other shape identifying a region of interest; time series event detection may take time series data and return a time-range projection identifying the samples that were obtained during the detected event; an information extraction task may take a formatted text file and return a mapping between textual substrings and semantic fields. To meet those requirements, a preferable provenance capture system should support not only standard relation algebra operators but also user-defined functions including code that extract, block and aggregate, etc. in a user-customized way.

Besides, our motivation for these problems comes from collaborations with geneticists, neurologists, and radiologists who have difficulty ensuring consistent processing of data acquired at multiple sites. To tackle the problems in the aforementioned techniques, we need tools that can not only capture provenance relationships among data files and code modules (“coarse-grained” or workflow provenance) but also track record-to-record (“fine-grained”) provenance. Fine-grained provenance allows us to take output records that are erroneous or that differ between two different workflow executions — and to trace back to the subsets of the input that yielded those results. Fine-grained provenance also allows us to subset the input and rapidly explore different potential parameter settings, and thus to fill in “missing” provenance. We seek a solution to capture the fine-grained provenance with the optimizability and the potential for on-demand computation provided by database-style techniques,
the ability to instrument user-defined code offered by provenance APIs, and general applicability across languages and environments provided by workflow provenance.

Example 3.1.3 (Provenance Archive) Consider the simplified query in the aligner, where the trimmed sequences match to the reference sequence library. Here the sequence matching condition is simplified to value identical. We can rewrite the relational algebra with the following SQL query (later we extend to consider other computations): 

\[
\text{SELECT * FROM REF NATURAL JOIN INPUT I WHERE seqid = '1'.}
\]

This might be converted into the simple expression:

\[
\sigma_{\text{sequence}=\text{'aat'}}((\text{REF} \bowtie (\text{INPUT})))
\]

<table>
<thead>
<tr>
<th>R( prov</th>
<th>seqid</th>
<th>sequence)</th>
</tr>
</thead>
<tbody>
<tr>
<td>r₁</td>
<td>1</td>
<td>aat</td>
</tr>
<tr>
<td>r₂</td>
<td>2</td>
<td>ccg</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>S( prov</th>
<th>sequence)</th>
</tr>
</thead>
<tbody>
<tr>
<td>s₁</td>
<td>ccg</td>
</tr>
<tr>
<td>s₂</td>
<td>aat</td>
</tr>
</tbody>
</table>

Table 1: Simplified example tables of sequences.

Suppose input relations \( R \) and \( S \) are as in the Table 1 (initially disregard the “prov” column). We get a final tuple \((1, \text{aat})\), whose provenance is based on the tuples whose prov column includes \( r₁ \) and \( s₂ \). More precisely, if \( r₁, s₂ \) are the provenance of the respective input tuples, then the provenance of the output involves joint use of \( r₁, s₂ \) — which we represent as a provenance expression \( r₁ \cdot s₂ \) Green et al. (2007b). We would like to store this tuple and its accompanying provenance in a form that can be consulted for reproducibility.

Now suppose we run a second query over the same input data, this time projecting only the sequence field:

\[
\pi_{\text{sequence}}(\sigma_{\text{sequence}=\text{'aat'}}((\text{REF} \bowtie (\text{INPUT}))))
\]

Since this second query shares a subexpression with the first, its output and its provenance might be derived from the materialized results of the first query — avoiding redundant storage by reusing the provenance derivations from the first query: this is what we mean by derivation-based compression.
Moreover, while our example is based on a real requirement from the sciences, similar tasks are encountered doing data cleaning in the enterprise or from Web information extraction.

**Example 3.1.4 (Provenance Archive Cont.)** Finally, suppose we want to publish the results and their provenance, and want to be sure the source data and the provenance of intermediate steps have not been tampered with. We could share the data, the provenance, and cryptographic hash information establishing authenticity.

In our view, an archival system for provenance should (1) be efficient with respect to its encoding; (2) support efficient querying; (3) be tamper-resistant. Naturally, it should be flexible enough to store fine-grained provenance as results are computed using standard relational operators including aggregation Amsterdamer et al. (2011e), and also enable common user-defined functions (UDFs) Zheng et al. (2019) — to handle the basic kinds of operations used in ETL, scientific data analysis, and OLAP.

To begin with, provenance capture in user-defined operators such as extractors, approximate matching, etc., is often not at fine-grained granularity. Specifically, extraction needs to be at the granularity of sub-record level, for example, a substring extractor needs to record the provenance of which part of the record is extracted. Existing works fail to provide such detailed provenance information (Section 4.2). On the other hand, provenance capture may be incomplete, due to tool limitations. Coarse-grained, file-level provenance in workflow provenance is insufficient to help us understand why a subset of the output is incorrect. A multi-step data analysis query often involves external tools, which if often open-source, as modules. Detailed capture of such queries may not capture enough about individual module versions (Section 4.1) and their functionality (Section 4.3) for us to understand when two queries run at different times (with updated module versions) would produce incomparable results (*Version Inconsistent* problem in Section 4.1). Sometimes, critical settings (e.g. thresholds) may be specified not in the query but in files that are not automatically captured (*Missing Parameters* problem in Section 4.1). Likewise, two key aspects of provenance archive have been unaddressed. First, in an archival system, we may need to store many analyses computed from the same input datasets. Since provenance is often bigger than the analysis
results, compression (Section 5.2.2) of repeated provenance is highly desirable. More importantly, if the provenance is to be used to certify results for an audit record, it must be tamper-resistant (Section 5.2.3), preferably at a fine-grained level.

We focus on provenance management on data analysis computation, including efficiently capturing fine-grained provenance as well as archiving such provenance data. In provenance computation, we study to support the tuple-based provenance for a wide array of ETL and matching operations with user-defined functions (UDFs). We enable lazy evaluation to only capture the desired provenance. We achieve the minimal instrument on code with the shared use of code modules. We allow the user to represent the data analysis queries with declarative representations referring to shared code modules. In the provenance archive, we ensure not only the easy lookup queries and the transitive closure queries but also the derivation-based compression and tamper-resistant property. By creating a tamper-resistant archive of provenance, we enable reproducibility and verification of authenticity.

3.2. PROVision System Capabilities

The PROVision system provides user-driven tools for reconstructing missing provenance details to aid for data consistency. It focuses on the provenance management of data analysis queries, including provenance capture and provenance archive. PROVision supports fine-grained provenance capture for queries beyond relational operators such as extraction and approximate matching with user-defined functions (UDFs) which is potentially in arbitrary code, as well as aggregation with UDFs over a group of records. The system allows simple attribute lookup over archived provenance and transitive closure lookup to retrieve all the provenance ancestors of given records. In addition, it archives provenance with an identity which further supports provenance comparison across data analysis queries over time. We support compact storage of provenance annotations by only storing repeated provenance one time. The system judges the repeated provenance with the identification of the provenance expressions. At last, the archived provenance is tamper-resistant if the provenance is to be used to certify results for an audit record.

3.3. The PROVision System Architecture

PROVision is given coarse-grained provenance from the workflow, including input and intermediate files as well as source modules; and, optionally, a subset of records of interest to the user. Upon
request, it produces record-to-record provenance, test instances that produce differences across workflow versions, and values for missing parameters. It consists of a series of modules shown in Figure 9.

**Module Descriptor Registry.** PROVision looks up the workflow modules in a central module repository to find accompanying *semantic descriptors*, which are declarative representations of the data queries (Section 4.2). Each descriptor, stored as a JSON file, specifies inputs and data formats, a tree of relational algebra operators, and optional user-defined code (or links to code) for the operations. We keep such a hub for popular modules so that scientists can share their semantic descriptors and module code. As a result, we can have a shared library of modules with experts edited semantic descriptors which can be applied to a large number of workflows’ provenance tracing.

**Plan Generator** Given the coarse-grained provenance from a workflow run, as well as semantic descriptors for each module in the workflow, the annotated workflow generator builds a rudimentary *query plan* for recomputing the workflow results. This query plan “replaces” each step in the workflow with an algebraic expression tree that also makes selected calls to user-defined code for
similarity matching, ranking, etc.

**Plan Rewriting Optimizer** A rewrite-based optimizer, written in the style of Volcano Graefe and McKenna (1993), then takes cost information gathered from the original workflow provenance and data, as well as any user selections for results of interest, and generates a more efficient plan. Our optimizer aggressively uses a *semijoin-based optimization* technique to prune intermediate results (Section 4.6).

**Provenance Reconstruction** PROVision executes the query plan using a custom query engine (Section 4.5.3), which works over external files, interfaces with external code, and reproduces workflow results *annotated with provenance* that can be traced.

**Token Maintenance** At each step of provenance computation, the Token Maintenance module returns a canonical node ID (provenance token) based on the operator or versioned user-defined function being executed, the inputs, and any other parameters. We assume if the inputs, parameters, and operations of outputs are the same, the provenances of the outputs are identical. The token maintenance ensures that the tokens of the identical provenance annotations are always the same even though those provenance annotations are generated at different computations.

**Provenance Storage** Given that the provenance is computed during query processing, the provenance storage maps it to persistent storage. We encode the provenance for a running query as a directed acyclic graph, where nodes represent expressions and implicitly have edges to their subexpressions, captured as object references. The token maintenance assigns an identity to the provenance node as a hash of provenance polynomial expressions. The provenance storage stores the acyclic graph as an edge table. PROVision supports the storage of many derived query results, each accompanied by semiring provenance expressions. As the identical computations on the same inputs generate identical provenance tokens, we enable provenance subexpressions sharing as well as compact storage. Besides, PROVision provides means of certification of the results and the inputs.

**Tools for Provenance Analysis** PROVision’s interactive tools (Chapter 4.4) enable the user to trace
the provenance of individual results, isolate sets of records that produce different results across different workflow versions, and rapidly reconstruct missing parameter values.
CHAPTER 4 : Fine-grained Provenance for ETL Tasks Computation

In this chapter, we present the provenance capture layer of the PROVision system. We first review the background of the fine-grained provenance capture problem in Section 4.1. Then we revisit the problem and the PROVision system architecture of the provenance capture layer in Section 4.1. We next present details of declarative representations of data queries – semantic descriptors which is an important component of our systems in Section 4.2. We devote the data model, operators, and provenance model used in our system in Section 4.3 and illustrate three types of user analysis scenarios in Section 4.4. After that, we introduce the implementation of the provenance capture layer and the optimization of provenance reconstructions in Section 4.5. Finally, we experimentally analyze our results in Section 4.7.

4.1. Background and Problems

Many of the computations in the workflows introduced in Section 3.1 require extensive data matching, and have been optimized for high performance. Our collaborators do not want to incur the significant (factor-of-two or greater) overheads required in tracking provenance via instrumentation Wu et al. (2013) when they only occasionally need to debug a few answers.

Workflow modules for data science take many forms. Our focus is on ETL, content extraction, and approximate matching-style computations, where we believe fine-grained provenance is of the highest value in diagnosing issues. Rather than modifying and instrumenting individual workflow modules to get fine-grained provenance, we instead develop methods to later recompute provenance on-demand, for efficiency using declaratively specified implementations of the workflow modules.

We then support the following operations:

1. Explain, via a provenance trace, specific output records produced through extraction, record linking, etc.

2. Given output records that differ across workflow versions, isolate the inputs that lead to the differences.
3. If an existing workflow execution used unknown parameters, rapidly find the appropriate parameter values.

While PROVision requires manual effort by experts to annotate and describe file formats and source modules, we believe that this is feasible in today’s open-source world with standardized repositories like GitHub and standardized dependency managers like pip. While workflows and code versions change frequently as updates are made, the “core” data formats and matching algorithms are much more stable, and could be declaratively specified by community experts. Moreover, many workflow modules implement operations that already closely correspond to relational algebra operations.

Conventional provenance tools do not adequately support detailed reasoning about common ETL-style, matching, and ranking tasks because they are limited to tuple-level operations and they do not support approximate matching or sub-selection. Our study of this problem is motivated by biomedical collaborators who operate a gene sequencing center. Their sequencing machine generates files with lists of text strings representing gene sequence reads. The data is analyzed via a workflow built from open-source tools written in different languages (C, Python, shell scripts). A key stage is sequence alignment: much like a record linking tool, the aligner module reads strings from the sequence machine’s output and compares them against sequences in a reference genome file. It outputs a list of pairs describing the best matching.

This gene sequence alignment workflow is specified via a shell script that executes the modules with appropriate command-line parameters, input files, and outputs. Unfortunately, two novel problems arise as the same workflow script is run at collaborating sites.

**Version Inconsistency:** As workflow modules or reference datasets are updated, input data gets processed slightly differently. Prior and current workflow versions may produce results that differ in subtly different ways — pointing to a likely bug in one or both versions of the software! This problem requires debugging by a human expert — given a small input test case. Changes in output records can be computed using a standard “diff” tool \(^1\) as in data versioning systems and diff tools Bao et al. (2009). Our goal is to identify sets of inputs that can be used to reproduce those

---

\(^1\)“Record” denotes an element in a collection, e.g., a tuple, JSON tree, etc.
different outputs (assuming the tools are deterministic). In order to deterministically reproduce the exact same ranking and choice among potential outputs, for many matching algorithms our input set must include not only those inputs that directly contributed to the outputs, but other “candidate” inputs that were considered but discarded within the same group, block, or ranking computation.

*Missing Parameters:* Many scientific workflows are built from shell scripts, which execute binaries with command-line parameters. It is straightforward to instrument such scripts to capture the majority of provenance information. However, some configuration parameters (e.g. thresholds) are often specified in local configuration files (e.g., in `/etc`), and these are often missing from the data and provenance shared across a data lake or distributed file system. Given output produced by the workflow with unknown parameters, we might be able to reverse engineer which parameter values produced that output. If we understand the operation of the workflow, we can test over a carefully chosen *subset* of the input data.

4.2. Declarative Representation of Queries

We exploit the fact that most data files can be read by existing open-source *extractors* that can be treated as sources of records; and that most scalable data analysis routines can be re-implemented using a combination of select-project-join-aggregate operations and application-specific code that computes new values from records or selects or aggregates values from groups of records. We develop an extended relational algebra for handling complex datatypes and user-defined extraction and aggregation functions and use these as the basis of *semantic descriptors* describing module functionality. Our PROVision system uses the semantic descriptors to selectively reproduce and reason about provenance across compositions of these modules (workflows). PROVision finds differences among outputs between different workflow versions, and minimally reproduces the provenance “traces” to the underlying source data — even through complex matching and blocking algorithm and, hierarchical data structures. It also searches over different parameter settings to recover which values could plausibly generate specific outputs.

Perhaps not surprisingly, the vast majority of data analysis tools today include high-level operations over arrays, matrices, and — especially relevant to the DBMS community — collections of
structured objects. Microsoft’s LINQ, Java’s streams interface, Python and R’s dataframes, and the multitude of map-reduce implementations in all environments, mean that it is possible to perform simple code and dataflow analysis to derive relational algebra-like descriptions of some lines of code within a cell. Given a high-level declarative specification for a piece of code, the DRMS can infer how the code processes and produces individual records. This can be extremely useful in reasoning about differences in results across workflows. It has another advantage, namely that we can easily generate retargeted code, e.g., to move from Python Pandas to Apache Spark, or potentially even convert from a local SciKit-Learn implementation to a more scalable learning platform like TensorFlow.

Workflow modules are arbitrary data-driven programs, invoked with parameter lists, typically operating with structured files as inputs and outputs. Our goal is to describe, using a more tractable specification, the data processing operations being performed within the module — such that we can trace from individual “records” within the output file, back to input “records” in the input file(s). We term this simplified specification a workflow module descriptor.

Each workflow module descriptor is specified as a query plan in an extended relational algebra, in which the operators directly compute and maintain provenance.

Figure 10 shows an example of a module descriptor. With this descriptor, Figure 12 shows how a

---

2Given their familiarity with Pandas, our users were comfortable creating relational expressions with user-defined Python code. PROVision could equivalently use SQL queries.
A segment of Python code (using Pandas) makes a variety of function calls producing a dataframe, which we map into a series of relational operators. Obviously, not all code (especially iterative or recursive code) maps naturally to relational expressions. However, we can generally break the lines of code into a DAG in which nodes represent blocks of relational-equivalent code and “other” code. Sometimes the relational code will include calls to user-defined functions (e.g., a user-defined map operation).

Our main innovation lies in provenance-preserving query operators that invoke user-defined functions to **compute** new attributes and/or **extract** content embedded within composite (possibly binary, free-text, image, or substring) attributes. Of course, our module descriptor algebra, which lacks support for iteration, cannot capture the full generality of Turing-complete programs. However, given that PROVision specifically targets provenance in scalable ETL- and “big data”-style programs, such modules are quite naturally captured using this abstraction.
Our algebra must address several issues. First, building upon the foundations proposed in the fine-grained provenance literature Green et al. (2007b); Karvounarakis et al. (2010), conventional relational operators should produce results annotated with *provenance polynomial* expressions (or their equivalents), using the widely accepted *provenance semiring* model in which equivalent query plans produce algebraically equivalent provenance polynomial expressions. We incorporate the “core” relational algebra operators, namely selection, projection, join, group-by, and union.

Importantly, most ETL and data matching tasks involve *user-defined functions*, whether to extract content from composite data (e.g., a nested object in a structured file, or a feature in an image), to return items within a group (e.g., by ranking items within a set), or to combine inputs to produce a new result (e.g., multiplying an input value by a constant). This requires us to develop a comprehensive treatment for interfacing with *user-defined functions*, such that we can determine not only what their atomic input values are, but also what we term *locations* — datatype-specific specifiers of projections, such as subsets, ranges, and bounding boxes — within specific inputs.

**Modeling extraction in a datatype-independent way.**

Another major source of irregularity in data science occurs in *information extraction and ETL workflows*, where data (or features) are pulled from structured or unstructured files, records are aligned or mapped against a reference dataset, and results are fed into downstream processing (e.g., OLAP, machine learning, data visualization). Sometimes extraction is done incorrectly, or different workflow executions produce different results due to changes in (undocumented) parameters, or due to subtle bugs in code updates that are revealed as outputs are compared for consistency over time. Unfortunately, none of the existing techniques described above, is well-suited for helping troubleshoot such issues. Workflow provenance is too coarse-grained to help troubleshoot issues. Provenance APIs require recompilation of often-large source code bases — but more importantly, incur overhead in recording *every derivation in advance*, and are “brittle” to changes in the execution order that may not affect the output. Database-style techniques hold promise, but do not handle information extraction-style operations over arbitrary datatypes, do not handle user-defined functions, and require that the computation occurs in a DBMS or “big data” engine.
ETL tasks and scientific workflows operate on a wide variety of structured inputs: JSON and XML files, structured and CSV files, images, raw text, and even (e.g., in genetics) substrings of larger strings. Our objective is to represent queries and transforms over any of these formats in a uniform way. Virtually all modern query engines, regardless of the data model, internally process tuple streams. For graph and tree-structured query languages, the tuples may be of bindings to specific parts of an input graph or tree; and a core primitive is path expression evaluation, which takes tuples of input bindings, evaluates path expressions against the portions of the graph pointed to by the input bindings and returns tuples that contain additional bindings (to the matches to the new path expressions). Similarly, in information extraction or image extraction, an extractor takes tuples with bindings to input files, evaluates some internal extraction function against those inputs, and returns tuples with additional bindings to individual extractions.

We can generalize both of the above cases to a “pattern” in which input tuples contain sets of (bindings to) values; an extraction operation takes as input a projection over these values; the extraction returns a set (relation) of extractions from its inputs; and each input tuple is joined with the relation returned by the extraction. This corresponds to how web service calls have long been represented in the data integration literature — as dependent joins Florescu et al. (1999).

**Example 4.2.1** Consider a gene sequence extraction operation over multiple files in Figure 13. An
example of one input item being queried from the data lake might be the structured file File1 containing a series of gene sequences. Within the file, assume each sequence is represented by a separate row, delimited by a CR/LF combination. The gene sequence extraction operation performed by the file format reader takes input sequences of files (as tuples), and for each file returns a set of \( \langle \text{location}, \text{sequence} \rangle \) pairs, such as \( \langle 45, \text{CGGTACCTA} \rangle \) and \( \langle 55, \text{TGACACCTA} \rangle \). In our dependent join abstraction, query results are effectively a join between the input \( \langle \text{file}, \text{object} \rangle \) tuples and the set of extracted \( \langle \text{location}, \text{sequence} \rangle \) tuples, where the location is relative to the file (and thus requires us to compose the provenance expressions of the file and the location).

### 4.3. Provenance Models

In this section, we devote how we encode the composition of provenance expressions as shown in Example 4.2.1. To begin with, we present our full data model, including provenance, and extended relational algebra.

#### 4.3.1. Data Model and Operations

We describe a workflow’s data processing modules using algebraic expression trees that filter, combine, and extract data, starting from raw input data that is stored in files or is remotely accessible via URLs, and resulting in structured outputs.

We assume a tuple-oriented, but non-first-normal-form, bag data model\(^3\). In a typical relational DBMS setting, queries (and their provenance) are derived from a set of 1NF base relations. Given that PROVision operates in a file-based environment and performs ETL and matching-style operations, we instead assume that all of our base data is maintained in a “data lake.” This data lake stores \( \langle \text{URL}, \text{object} \rangle \) pairs in a single relation \( \mathcal{L}(\text{key}, \text{value}) \). Data values are arbitrary, typically composite, binary objects, such as files or objects in a key/value store — so we make no assumption that they are in 1NF. As in Green et al. (2007a), each tuple in the data lake, \( t \), is conceptually annotated with a provenance token \( \text{Prov}[t] \) — a unique, opaque ID with a 1:1 mapping to the key in the data lake.

Tuples are processed using a combination of standard relational algebra operators and extraction or

\(^3\)JSON and XML data are encoded as non-1NF attributes.
computation operations, whose results are (as sketched in the previous section) modeled as dependent joins.

**Basic Relational Algebra Operators**

PROVision adopts standard relational algebra operators, including selection, projection, join, union, and distinct (nesting and grouping are handled specially, in Section 4.3.1). As with other non-1NF query engines, we extend these operators to handle collection-valued attributes. We also add selection and join predicates that test attribute equality-by-value and equality-by-reference.

For each output tuple \( \bar{t} \), each algebra operator creates an annotation, denoted, \( \text{Prov}[\bar{t}] \), that is a provenance polynomial from the semiring model Green et al. (2007b). Briefly, we assume a unique variable or token associated with each base tuple, which represents any provenance metadata “attached” to that tuple. Each time we derive a new tuple via a relational algebra operation, this new tuple will be annotated with an algebraic, polynomial expression derived from the annotation of the input tuple(s). The expressions are computed as follows:

- For a **select** expression \( \sigma_\phi(R) \), for each tuple \( \bar{t} \in R \) satisfying \( \phi(\bar{t}) \), its provenance expression is \( \text{Prov}[\bar{t}] \). (**Provenance is unchanged by selection.**)

- For a (bag) **project** expression \( \Pi_\alpha(R) \), for each tuple \( \bar{t} \in R \), its provenance expression is \( \text{Prov}[\bar{t}(\alpha)] \).

- For each output \( \bar{t}' \) from a **join** expression \( R \bowtie \theta S \), for each tuple pair \( \bar{t}_1 \in R, \bar{t}_2 \in S \) satisfying \( \theta(\bar{t}_1, \bar{t}_2) \), its provenance expression is \( \text{Prov}[\bar{t}_1] \cdot \text{Prov}[\bar{t}_2] \).

- For a (bag) **union** expression \( R \cup S \), for each tuple \( \bar{t} \in R \cup S \), its provenance expression is \( \text{Prov}[\bar{t}] \).

- For a result \( \bar{t}' \) output by a **duplicate removal** expression, **distinct** \( R \), if \( \bar{t}_1, \cdots \bar{t}_m \in R \) and all \( m \) tuples are equal, \( \bar{t}_1 = \cdots = \bar{t}_m \), then \( \bar{t}' \)’s provenance expression is \( \text{Prov}[\bar{t}_1] + \cdots + \text{Prov}[\bar{t}_m] \).

**Example 4.3.1** Suppose we have a relational algebra expression **distinct** \( (\Pi_{a,y}(R \bowtie c=x \phi_{x<5}(S))) \),
applied to schema $\Sigma = \{ R(a, b, c), S(x, y) \}$, and tuples $R(1, 2, 3), R(1, 4, 3), S(3, 4)$ with provenance tokens $p_1, p_2,$ and $p_3$, respectively. The result $t(1, 4)$ would be annotated with the provenance expression $\text{Prov}[t] = p_1 \cdot p_3 + p_2 \cdot p_3$, representing that the derived result is generated twice, from the first-and-third and second-and-third base tuples.

Operators for User-Defined Functions

ETL tasks often invoke non-declarative code to extract embedded content within an input object, or to compute a value using complex logic over some fields of a record. We assume our query plan embeds this logic in the form of a user-defined function (UDF) modeled after the original workflow module, but (as we describe below) that our UDFs additionally provide a limited amount of information about the provenance of any result being computed.

Since some UDFs can be applied to sets of tuples as a result of grouping, and others can be applied to single tuples at a time, we develop separate operators for each (the group-by and compute operations, respectively). We define the operators using the same basic ideas.

UDFs as joins with binding pattern restrictions. Borrowing from the data integration literature Rajaraman et al. (1995), we model the invocation of a UDF, which takes a set of input parameters, as a dependent join with a relation with binding patterns, of the form $R_f(a_1^b, \ldots, a_n^b, b_1^f, \ldots, b_q^f)$. Attributes adorned with $^b$ are bound and those annotated with $^f$ are free. To retrieve tuples in $R_f$, we must parameterize (join on) the bound attributes.

Example 4.3.2 Suppose we have the function $f_n(x, y)$ which returns a set of pairs $(a, b)$. We model this not as a function with a set-valued output, but rather as a relation from inputs to outputs, $R(x^b, y^b, a^f, b^f)$. We can then represent a function call to $f$, based on the contents of relation $S(u, v)$ as a dependent join, $S \bowtie R$, whose results will have the schema $(u, v, a, b)$.

Now we define relational algebra operators for each type of UDF input.

Definition 1 (Scalar UDF operator) The scalar UDF operator, compute, evaluates one tuple $\bar{i}$ at a time, computing a function $f_n$ over the fields $\bar{i}[\bar{\alpha}]$, returning a list of attributes $\bar{\beta}$: compute$_{f_n,\bar{\alpha},\bar{\beta}}(R)$. The input parameters to $f_n$ must match the arity and types of $R[\bar{\alpha}]$. 

46
The scalar UDF operator is extremely useful for building query plans with extraction functions, and for query optimization. However, we will (in the next 2 sections) need to define the provenance for its outputs. Here it is useful to note that the scalar UDF can be modeled using the dependent join (hence, a standard join for which provenance is well understood), as follows. Let us represent function $f_n$ as a relation $R_{f_n}$, whose schema is $\bar{\alpha} \cup \bar{\beta}$, where $\bar{\alpha}$ are all bound and $\bar{\beta}$ are all free. Compute $f_n,\bar{\alpha},\bar{\beta}(R)$ can then be rewritten as a dependent join $R \bowtie_{\bar{\alpha}} R_{f_n}$.

**Definition 2 (Grouping UDF operator)** The grouping UDF operator, `group`, partitions the input relation $R$ into sets of tuples sharing the same values for grouping fields $R[G]$. For each set of tuples, it then applies a series of aggregate functions, $F_N_1$ through $F_N_m$ over projections $\bar{\alpha}_1$ through $\bar{\alpha}_m$, respectively; returning values $\bar{\beta}_1$ through $\bar{\beta}_m$. We denote it as follows:

$$
grup_{G, (F_N_1, \bar{\alpha}_1, \bar{\beta}_1), \ldots, (F_N_m, \bar{\alpha}_m, \bar{\beta}_m)}(R)$$

Unlike with the scalar UDF case, aggregate functions are second-order and we cannot capture the full semantics using select/project-join expressions. However, for each set of tuples $T \subseteq R$ belonging to a group (i.e., sharing the same values for all grouping fields $G$), the output of the grouping operator is a join between the portion of the tuple corresponding to the grouped fields, and the results of applying each function to the set of tuples:

$$\text{distinct}(T[G]) \bowtie_G F_N_1(T) \cdots \bowtie_G F_N_m(T)$$

This is similar to the scalar UDF operator, but results in a bag of tuples (namely, a Cartesian product between the grouping tuple and the outputs of each of the $m$ aggregate functions.) Note that each $\alpha$ term consists only of attributes from $R$ so the order of evaluation of the functions does not matter.

**Example 4.3.3** Suppose we are given two aggregate functions, $\text{min}$, which returns the minimum value among a collection of values (and is modeled as relation $R_{\text{min}}(x^b, m^f)$, and the table-valued function $\text{top2}$, which returns the two largest values among a collection of values (modeled as rela-
tion $R_{\min}(y^b, t_f)$. Given an SQL query:

```sql
SELECT id, average(x), top2(y)
FROM r GROUP BY id
```

and a table $r$ with values $r(1, 2, 3)$, $r(1, 3, 4)$, $r(1, 4, 2)$ and $r(2, 3, 4)$. The group with $id = 1$ has
three tuples $r(1, 2, 3)$, $r(1, 3, 4)$, and $r(1, 4, 2)$. The grouping tuple will simply be comprised of the
grouping attribute: $(1)$. The function `average` will be called on the values of $x$, $\{2, 3, 4\}$ and will
return a single unary tuple $(3)$. The function `top2` will be called on the values of $y$, $\{3, 4, 2\}$ and will
return unary relation $\{(4), (3)\}$. The ultimate output for this group will be the Cartesian product of
these three intermediate relations, which will result in the two tuples $(1, 3, 4)$ and $(1, 3, 3)$.

The above formulations specify the tuples output by the UDF operators, but additionally we need to
specify their provenance expressions. We first describe what we mean by provenance for aggregate
results, and then we consider provenance for user-defined functions, before looking at how the two
aspects compose.

4.3.2. Provenance for Extracted Data

Unlike the standard relational queries studied in much of the prior work on fine-grained provenance,
ETL workloads do not start with records in their fully parsed form. Thus they often take as input a
“BLOB” (Binary Large OBject) or “CLOB” (Character Large OBject) of binary or string data, and
apply an extraction function to the data within that object. For instance, we may extract segments
of comma-separated text into different fields, or we may apply an information extraction function
to find mentions of dates in an HTML file. These are common use cases for PROVision’s **scalar
UDF operator**, which takes a tuple at a time, applies a user-defined function, and returns a set of
tuples representing the extractions. The scalar UDF operator can additionally be useful in allowing
a workflow to apply transformations from tuples to tuples (e.g., converting fields from one unit to
another) or sets of tuples (e.g., extracting words from lines of text).

Recall that every “base tuple” in our data lake has a corresponding globally unique provenance
token. Every derived SPJU tuple has a **provenance semiring polynomial expression** in terms of
these base tokens, as described at the start of this section. More broadly, we capture the provenance of each tuple in terms of an expression over the provenance of its source tuples.

Here, we can exploit the observation in the previous section that the scalar UDF operator is a form of a (dependent) join. However, the extraction UDF itself adds a wrinkle: indeed a UDF takes zero or more arguments and produces results based on those arguments (which, in turn, come from the input tuple). However, this is often highly imprecise, as the UDF may only uses a portion of the data in each input tuple’s fields (consider extraction of a single column from a CSV).

To truly capture the fine-grained provenance in this setting, we need a more general way — yet a datatype- and UDF-specific way — of capturing the subsets of data used within attributes.

**Type- and UDF-specific Provenance**

Let us assume the presence of a location specifier and value extractor for a given attribute \( x \) and function \( f_n \).

**Definition 3 (Location specifier)** A location specifier \( L_{a,f_n} \), is a datatype- and operation-specific token — typically a range, bounding box, or predicate — for use in extracting a value from a subset of an attribute value \( a \).

In essence, a location specifier is a special case of a provenance token, which applies to non-relational data and operations.

**Example 4.3.4** For a CSV string CATGGCCG, alpha, a location specifier might be the interval \([0, 7]\).

Now we factor function \( f_n \) into the composition of two functions, \( f_n = f_n' \circ v \) where \( v \) is a value extractor function that takes a series of location specifiers (one per input argument to \( f_n \)).

**Definition 4 (Value extractor)** A value extractor for function \( f \), \( v_{f_n,x}(\vec{t}, \vec{L}) \) is an operation that, given a tuple \( \vec{t} \) and a vector of location specifiers for each attribute in \( \vec{x} \), \( \vec{L} \), returns a list of subsets of \( \vec{t}[\vec{x}] \) from which \( f_n(\vec{x}) \) can be computed.
The value extractor is akin to a selection operation in the relational algebra: it returns a subset of the input data, which is used by the transformational or computational aspects of UDF $f$.

**Example 4.3.5** Following the previous example, the value extractor may simply be the substring function, which takes a string from the CSV file (e.g., CATGGCCG, alpha) and an interval (e.g., [0, 7]), and returns all characters within that interval (CATGGCCG).

We assume our value selector is defined in a way that is independent of any specific input record. Given this, and the ability to compare location specifiers according to a partial ordering on restrictiveness, we can also define a minimal location specifier to be the most restrictive location specifier $L_{a_{\text{min}}}$ for a given value $a$, which still returns the same output $f'(v(a)) = f(a)$. For instance, the minimal location specifier may represent the smallest substring from which a value is computed, or the minimum bounding box.

**Composing Provenance**

We also want the provenance of the output of our UDF operators to be the composition of each input tuple’s provenance, along with its location specifiers. Given function $fn$ which takes parameters $a_1, \ldots, a_m$ and returns a set of (zero or more) $R_{fn}(b_1, \ldots, b_q)$ tuples:

$$fn(a_1, \ldots, a_m) \mapsto R_{fnout}(b_1, \ldots, b_q)$$

we define the provenance of each output tuple $\tilde{t}$ as a provenance function composed from the provenance of the base tuple, plus the UDF-specific provenance of the prior section:

$$P_{fn}(\text{Prov}[^{\tilde{t}}], L_1, L_2, \ldots, L_m)$$

where $P_{fn}$ represents a function symbol in the provenance semiring specific to our function $fn$. (We later allow for specific algebraic equivalence to be associated with the provenance functions, for query optimization purposes.)
Finally, each output of the scalar UDF function represents the (dependent) join of the input tuple with each output tuple returned from the function, i.e., it is the provenance expression:

\[
\text{Prov}[t] \cdot P_{fn}(\text{Prov}[t], L_1, L_2, \ldots, L_m)
\]

**Example 4.3.6 (Blocking)** A key operation in record linking Elmagarmid et al. (2007) (as well as string and gene sequence alignment Mount (2007)) is known as blocking. Given the cost of performing a full comparison between all pairs of tuples, blocking is used to prune the set of comparisons to those with common features. Each tuple is associated with one or more blocks, and all tuples within a block are combined for a similarity comparison. A common blocking function is the n-gram, where all subsets of up to n tokens are returned as candidate blocks. (Observe that each tuple may have multiple blocks, in contrast to a hashing function.)

Given a tuple (‘smith’, 123) with provenance token p₀, and a scalar UDF returning all trigrams, \( fn_{3\text{gram}} \), applied to the first attribute, we will get the results and provenance:

<table>
<thead>
<tr>
<th>block</th>
<th>name</th>
<th>id</th>
<th>provenance</th>
</tr>
</thead>
<tbody>
<tr>
<td>_s</td>
<td>smith</td>
<td>123</td>
<td>( p_0 \cdot P_{3\text{gram}}(p_0, [-2, 0]) )</td>
</tr>
<tr>
<td>_sm</td>
<td>smith</td>
<td>123</td>
<td>( p_0 \cdot P_{3\text{gram}}(p_0, [-1, 1]) )</td>
</tr>
<tr>
<td>smi</td>
<td>smith</td>
<td>123</td>
<td>( p_0 \cdot P_{3\text{gram}}(p_0, [0, 2]) )</td>
</tr>
<tr>
<td>mit</td>
<td>smith</td>
<td>123</td>
<td>( p_0 \cdot P_{3\text{gram}}(p_0, [1, 3]) )</td>
</tr>
<tr>
<td>:</td>
<td>:</td>
<td>:</td>
<td>:</td>
</tr>
</tbody>
</table>

Observe that the provenance column represents the product of the input tuple with a provenance function (for \( fn_{3\text{gram}} \)) and a location specifier representing the index positions of a substring. We assume here that index positions that are out of string bounds are filled in with blank ‘\_’ characters.

**4.3.3. Provenance for Aggregates**

We now consider another type of user-defined function, which takes a set of tuples as its input. Classically, this is an aggregate function in SQL. However, many types of matching, ranking, and approximate join operations, such as record linking Oracle (2013); Elmagarmid et al. (2007); Ananthakrishna et al. (2002), can be captured using a combination of (1) computing, via the scalar UDF
function, a set of one or more blocks for each input record, as in our prior example, (2) joining tuples within blocks, forming a Cartesian product among these, (3) and then performing a ranking or thresholding function over the collection of joint tuples within the block to find the most promising matches. The grouping UDF operator is critical to this third step.

To define provenance for each output from the grouping UDF operator, we note that aggregate functions are generally divided into exemplars — input tuples whose output appears in the output — and summaries — where all of the input tuples are combined to produce an output. For summaries, the provenance should clearly be based on the provenance of all of the input tuples. For exemplars, there is a choice between capturing the provenance of all tuples whose values affect the output, and all tuples whose values were considered in producing the output. In either case, we can define a notion of relative provenance, similar to that in Section 4.3.2. This will represent a combination of the provenance of the input group (e.g., the semiring sum of the provenance expressions of the input tuples) with a notion of type- and operation-specific provenance.

For each aggregate function \( FN(a_1, \ldots, a_q) \) applied to a group of tuples \( T \), we get a result tuple whose provenance is:

\[
P_{FN}(\sum_{t \in T} Prov[t], \sum_{t_i \in T} \langle L_{i,1}(t_i[a_1]), \cdots, L_{i,m}(t_i[a_q]) \rangle)
\]

Recall from Section 4.3.1 that we can express the computation done by the grouping UDF operator for each group of tuples \( T \subseteq R \), with multiple functions \( FN_1 \ldots FN_m \), as a series of joins:

\[
distinct(T[G]) \bowtie_G FN_1(T) \cdots \bowtie_G FN_m(T)
\]

Thus, the output provenance for each aggregate tuple, based on a group of tuples \( T \subseteq R \), is a
product of the form:

$$\sum_{\bar{t} \in T} \text{Prov}[\bar{t}]$$

$$\cdot P_{FN_1}(\sum_{\bar{t} \in T} \text{Prov}[\bar{t}], \sum_{\bar{t}_i \in T} \langle L_{i,1}(t_i[a_{y_{1,1}}]), \cdots, L_{i,m}(t_i[a_{y_{m,1}}]) \rangle)$$

$$\cdots$$

$$\cdot P_{FN_m}(\sum_{\bar{t} \in T} \text{Prov}[\bar{t}], \sum_{\bar{t}_i \in T} \langle L_{i,1}(t_i[a_{y_{1,1}}]), \cdots, L_{i,m}(t_i[a_{y_{m,1}}]) \rangle)$$

**Example 4.3.7 (Aggregation)** Suppose we are matching tuples in two relations: \(A('\text{smith}', 123), B('\text{smythe}', 345), B('\text{simpson}', 456)\) with provenance tokens \(p_0, p_1, p_2\), respectively. We use \(f_{3\text{gram}}\) to compute a block for each tuple, and we join candidate matches on the block ID.

<table>
<thead>
<tr>
<th>block</th>
<th>name(_1)</th>
<th>name(_2)</th>
<th>id(_1)</th>
<th>id(_2)</th>
<th>provenance</th>
</tr>
</thead>
<tbody>
<tr>
<td>_s</td>
<td>smith</td>
<td>smythe</td>
<td>123</td>
<td>345</td>
<td>(p_0 \cdot p_1 \cdot P_{3\text{gram}}(p_0 \cdot p_1, \langle [-2, 0] \rangle + \langle [-2, 0] \rangle))</td>
</tr>
<tr>
<td>_s</td>
<td>smith</td>
<td>simpson</td>
<td>123</td>
<td>456</td>
<td>(p_0 \cdot p_1 \cdot P_{3\text{gram}}(p_0 \cdot p_2, \langle [-2, 0] \rangle + \langle [-2, 0] \rangle))</td>
</tr>
<tr>
<td>_sm</td>
<td>smith</td>
<td>smythe</td>
<td>123</td>
<td>345</td>
<td>(p_0 \cdot p_1 \cdot P_{3\text{gram}}(p_0 \cdot p_1, \langle [-1, 1] \rangle + \langle [-1, 1] \rangle))</td>
</tr>
<tr>
<td>:</td>
<td>:</td>
<td>:</td>
<td>:</td>
<td>:</td>
<td>:</td>
</tr>
</tbody>
</table>

Finally, for each block, we return the highest-scoring pairwise match (top1). We can visualize an intermediate point in the computation. For instance, for block _s, the result would be ('smith', 'smythe', 123, 345) given that its string edit distance is the lowest in this block. Note that the provenance of the output result would be:

\[
(p_0 \cdot p_1 + p_0 \cdot p_2) \cdot P_{\text{top1}}((p_0 \cdot p_1 + p_0 \cdot p_2), \langle p_0 \cdot p_1 \cdot P_{3\text{gram}}(p_0 \cdot p_1, \langle [-1, 1] \rangle + \langle [-1, 1] \rangle)\))
\]
Commutativity

\[
\text{group}_{G,g,\alpha}(\text{group}_{G,f,\alpha_f,\beta_f}(R)) \equiv \\
\text{group}_{G,f,\alpha_f,\beta_f}(\text{group}_{G,g,\alpha,\beta}(R))
\]

if \( \alpha_g \cap \beta_f = \emptyset \wedge \alpha_f \cap \beta_g = \emptyset \)

Compute/Group

\[
\text{compute}_{g,\alpha,\beta}(\text{compute}_{f,\alpha_f,\beta_f}(R)) \equiv \\
\text{compute}_{f,\alpha_f,\beta_f}(\text{compute}_{g,\alpha,\beta}(R))
\]

if \( \alpha_g \cap \beta_f = \emptyset \wedge \alpha_f \cap \beta_g = \emptyset \)

Table 2: UDF operator equivalences

Strings and substrings

\[
\text{Prov}[\text{substring}_{c,d}(\text{substring}_{a,b}(S))] \equiv \\
\text{Prov}[\text{substring}_{a+c,a+d}(S)]
\]

if \( c, d \leq b - a \)

Images and cropping

\[
\text{Prov}[\text{crop}_{x_3,y_3},(x_4,y_4)](I) \equiv \\
\text{Prov}[\text{crop}_{x_1,y_1},(x_2,y_2)](I)
\]

if \( x_3 \leq x_4 - x_1 \) \& \( y_3 \leq y_4 - y_1 \)

Trees and simple path expressions

\[
\text{Prov}[\text{patheval}_{x}(\text{patheval}_{y}(T))] \equiv \\
\text{Prov}[\text{pathstep}_{x/y}(T)]
\]

Table 3: UDF type/operator provenance equivalences

4.3.4. Equivalence Rules for Component Provenance

Our extended algebra exhibits all of the standard equivalences for the relational algebra: join associativity and commutativity, selection pushdown, distributivity of join through union, and group-by/join pushdown Chaudhuri and Shim (1994). Moreover, our UDF operators show certain equivalences, shown in Table 2. (Our assumption here is that UDFs, while effectively black-box, are deterministic.)

A virtue of the provenance semiring model is that algebraic equivalences used in query optimization (e.g., commutativity of joins) result in equivalent provenance. The \( \cdot \) operator in the provenance semiring model follows the commutativity property such that \( p_1 \cdot p_2 = p_2 \cdot p_1 \). Similarly, certain equivalences hold over extractions. Suppose we have two extractions function \( f, g \) operated on value with provenance \( p \). In general, \( f(g(p)) \neq g(f(p)) \). But there exist some cases that \( f(g(p)) = \)
We rewrite the nested UDF with provenance and value pairs. With input tuple \((p, v)\),

\[
f : (p_{x_1}, x_1) \mapsto (f(p_{x_1}, x_1), y_1) \quad g : (p_{x_2}, x_2) \mapsto (g(p_{x_2}, x_2), y_2)
\]

We are interested in arbitrary datatypes and UDFs, for which equivalences may or may not hold. Our PROVision system allows an expert to provide type-and-operator-specific equivalence rules. We describe here an important class of datatypes for which we pre-encode equivalence rules for provenance expressions: types with hierarchical containment and operators that project locations. Table 3 shows some properties that hold for several common cases: namely, strings, images, and trees.

4.4. User Analysis Tools

Building upon the provenance reconstruction techniques described in the previous sections, PROVision addresses the three problems introduced in Section 4.1.

Our PROVision system uses the semantic descriptors to selectively reproduce and reason about provenance across compositions of these modules (workflows). PROVision finds differences among outputs between different workflow versions, and minimally reproduces the provenance “traces” to the underlying source data — even through complex matching and blocking algorithm and, hierarchical data structures. It also searches over different parameter settings to recover which values could plausibly generate specific outputs.

Tracing from output records back to the original data. To support result troubleshooting, PROVision finds differences among outputs between different workflow versions, and minimally reproduces the provenance “traces” to the underlying source data — even through complex matching and blocking algorithm with hierarchical data structures.

4.4.1. Tracing Fine-grained Provenance

Through PROVision’s user interface, the user may select records from the output results, and trace their provenance back to their inputs. This use case is not unique to our system — being shared with several databases and big data provenance systems Chiticariu and Tan (2006); Green et al. (2007a);
Glavic and Alonso (2009); Kermarrec et al. (2001) — but its fine-grained capability is not supported in other systems that allow for arbitrary code and scripts. Algorithm 1 is called with a workflow and intermediate results, $WF$. It computes either the complete provenance for the workflow (if selected outputs $O_{sel} = \emptyset$) or the provenance for a subset of output tuples $O_{sel}$ (if this is non-empty).

**Algorithm 1** TraceProv($WF$, $O_{sel}$)

1: Load semantic descriptors of modules in $W$.
2: Build execution plan $Q$ from semantic descriptors.
3: **if** $O_{sel} == \emptyset$ **then**
4: $Q' = Q$
5: **else**
6: $Q' = Q \bowtie O_{sel}$
7: **end if**
8: Optimize and execute $Q'$, recording provenance.
9: **return** prov($Q'$)

Building upon TraceProv, we can address the more complex problems of version inconsistency, where with the same input data two versions produce different outputs, and finding missing parameters for the workflow, where the captured provenance is missing parameters necessary to ensure the same output is produced.

### 4.4.2. Version Inconsistency

Like all software, workflow modules are constantly updated by developers. Sometimes these changes are expected to produce different results, e.g., when a reference library or a pruning heuristic are changed. The more challenging part is the unexpected cases, where the developers inadvertently introduced a change that resulted in the new behavior. Here, it is vital that the developers be given test instances to determine what the correct output should be.

We assume a trivial pre-processing step that executes each version of a module in the workflow, and does a **diff** between the versions — this needs to be part of the scientist’s general workflow process to test for consistency. Then, given a discovered difference and such a workflow execution, PROVision finds the earliest module that results in a difference: the branching module. The tool then identifies the output records that differ between the versions, allows the user to sub-select from these (see Figure 14), and then reconstructs provenance from these. Finally, it outputs the set of all
Figure 14: Example of tracing differences between outputs of two workflow runs, to debug inconsistencies.

Here, we require that the abstract workflow specification be the same across versions, with variations at the module and file level. We define for the workflow specification a workflow template: this is a DAG $T = (V, E)$ where nodes $V = V^m \cup V^f$ represent modules ($V^m$) and input or output files ($V^f$); directed edges $E$ connect from file nodes to module nodes (representing inputs to a program) or from module nodes to file nodes (representing outputs produced by a program). The template $T$ is instantiated each time the workflow is executed. Execution maps each node in $T$ to actual execution instances as follows: in execution run $X_j$, for every module node $m_i \in V^m$ is mapped to an executable program ($M_j : m_i \rightarrow Program_{i,j}$); for every file node $f_i \in V^f$, $f_i$ is mapped to a set of data files used in the execution ($F_j : f_i \rightarrow \{File_{i,j,k}\}$, where $k = 1, \ldots, q$ represent multiple(q) files as input or output of a module).

We take the template $T$ and mappings of two executions $Map_1 = \{M_1, F_1\}$ and $Map_2 = \{M_2, F_2\}$, and trace the execution instances with Algorithm 2. Here the two executions share the same input files, with an overall input set of records $I$, leading to different output sets of records $O_1$ and $O_2$ in
the output files. Our goal is to find the responsible subset of records \( I_{\text{resp}} \in I \) which leads to the different output results. The algorithm traverses the file nodes in workflow template \( T \) and compares the associated files using a function \( \text{isEqual} \). At the earliest point of divergence, i.e., the branching module, we trace provenance back to the input records, for each of the two executions.

Algorithm 2 WorkflowDebugInstance(\( Map_1, Map_2, T \))

1: for each \( f_i \in V \) in topological sort order do
2: search mappings \( F_1 \in Map_1 \) and \( F_2 \in Map_2 \), find \( \{File_{i,1,k}\} \) and \( \{File_{i,2,k}\} \)
3: if not \( \text{isEqual}(\{File_{i,1,k}\}, \{File_{i,2,k}\}) \) then
4: Let \( T' := (V', E') \) where \( V', E' \) represents the transitive closure of all edges and nodes connecting to \( f_i \).
5: Let \( WF'_1 \) and \( WF'_2 \) represent the subset of workflow and module nodes mapped from \( T' \) for executions 1 and 2, respectively
6: Let \( O_{sel_1} := O_1 - (O_1 \cup O_2) \) and \( O_{sel_2} := O_2 - (O_1 \cup O_2) \).
7: \( prov_1 := \text{TraceProv}(WF'_1, O_{sel_1}) \)
8: \( I_{resp_1} := \text{the input records within } prov_1 \)
9: \( prov_2 := \text{TraceProv}(WF'_2, O_{sel_2}) \)
10: \( I_{resp_2} := \text{the input records within } prov_2 \)
11: return \( I_{resp_1} \cup I_{resp_2} \)
12: end if
13: end for

4.4.3. Missing Parameters

A different type of inconsistency occurs when we are given a workflow and its provenance but are lacking key parameters necessary for reproducibility in the provenance. Fortunately, PROVision “knows” the semantics of the modules (thanks to semantic descriptors): this allows us to run the workflow under different parameter settings, but test over a carefully chosen subset of the data, to quickly isolate “plausible” parameter values. Algorithm 3 illustrates the approach to searching for the missing parameter values. Here, in addition to original workflow \( WF \), we take a search space \( S = (X_1, \ldots, X_m) \) where each set of values \( X_i \) represents the possible values for parameter \( p_i \). (The overall search space is \( |S| = \Pi_{i=1}^m |X_i| \)). A possible setting for missing parameters is \( P^{(i)} = < p_1^{(i)}, \ldots, p_m^{(i)} > \), where \( P^{(i)} \in S \).

Now we enumerate possible settings for the \( m \) parameters and test (on a subset of the input data) whether these produce results consistent with the original workflow output. In Algorithm 3, we first pick a subset of inputs \( T \in I \), which is generally a set of items that are mapped by the algorithm to
the same hash bucket or block. Then we reconstruct the workflow output over input subset $I_i$ with a particular set of parameter values, producing outputs $O_i$ with provenance $\text{prov}(O_i)$. Next, in Lines 4-6, we validate whether such records are part of the provenance of the original workflow execution — if not, we prune this set of parameter values.

Once we have removed all parameter settings that fail on a subset of the input, we take the (much smaller) set of parameter settings and verify that they produce the correct output over the full input dataset (Lines 8-12). Whichever combinations remain are plausible parameter settings. (Note that more than one combination of parameter values could potentially produce the workflow’s output results, but we argue this is a useful conclusion!)

**Algorithm 3** SearchParamValues($WF, S$)

1: choose subset $T \in I$.
2: set $S = \text{set of all parameter combinations from } X_1, ..., X_m$
3: for $\bar{s} \in S$ do
4: if not $TestWF(WF, \bar{s}, T, O)$ then
5: remove $\bar{s}$ from $S$
   {Not all $\bar{s}$ passed the test over $T$.}
6: end if
7: end for
8: for $\bar{s} \in S$ do
9: if not $TestWF(WF, \bar{s}, I, O)$ then
10: remove $\bar{s}$ from $S$
   {Not all $\bar{s}$ passed the test over $T$.}
11: end if
12: end for
13: return $S$

4.5. Implementation of PROVision Engine

Section 4.3 detailed the core algebra of PROVision, including its relationship with provenance. We now describe how we implement a query processor for efficiently reconstructing fine-grained provenance and optimization. We then report preliminary experimental results in Section 4.7.

The PROVision system provides tools for reconstructing provenance to improve data consistency. PROVision is given a workflow, input and intermediate files, and records selected by the user. It selectively produces record-level provenance for outputs, subsets of data that produce differences across workflow versions, and values for missing parameters. It is comprised of the modules shown
Figure 15: The system architecture of provenance capture layer.

4.5.1. Module Registry

PROVision looks up the workflow modules in a central repository to find accompanying semantic descriptors. Each descriptor, stored as a JSON file, specifies inputs and data formats, a tree of relational algebra operators (a semantic descriptor, Section 4.2), and optional user-defined code (or links to code) for the operations. Figure 10 shows an example of the trim module in genome workflow: the left figure is the configuration file of trim and the right one is the algebra tree descriptor in JSON format. We keep such a hub for popular modules so that scientists can share their semantic descriptors and module code. As a result, we can have a library of modules with experts edited semantic descriptors which can be applied to a large number of workflows’ provenance tracing.

The semantic descriptors for modules (module descriptors) are declarative specifications (in the form of relational algebra trees specifying the operations, and schemas of the inputs and outputs) of the structured data extraction and record-to-record transformations being done in each program. Such descriptors are important for composing workflows and queries as they manage to “lift” procedural code to declarative representations. Observing that most data analysis code heavily emphasizes arrays, matrices, and collections of structured objects (or tabular data), the module descriptors make use of “bulk operations” such as matrix transformations, bulk arithmetic operations, and re-
lational algebra (filter, join) or map-reduce operations, as well as calls to high-level libraries and toolkits.

4.5.2. Generating the Initial Execution Plan from Descriptor

As alluded to in Figure 9, the first step in PROVision is to take the various modules executed in the workflow and to look up each of these in the module registry to retrieve its semantic descriptor. The semantic descriptor specifies the schema and file formats for input and output results, but most importantly it specifies a tree of algebraic operators (as in Section 4.2) for the module — as well as links to any external files and code that must be retrieved to execute any associated user-defined functions. Our implementation supports code modules written in Python, Java, and C.

As the whole annotated workflow consists of several annotated modules, we describe how the forward recomputation of provenance of each module, and the provenance computation for the whole workflow will be straightforward.

Assume we first fetch a descriptor for a module, which is normally in a JSON format describing several tree-structured data processing operators. Then we parse the JSON file and generating an in-memory query (plan) tree with our queryGeneration engine. This in-memory query tree is an algebra description, we next transform it into a declarative and calculus-style representation of SPJ followed by GroupBy, generating a QueryGraph which satisfying the following conditions:

- **Node < T >** in the QueryGraph is either: (1) a source (which is essentially a pointer to the physical operator for scanning the source); or (2) a field (which includes both a source and a field name).

- Each **HyperEdge < T >** in the QueryGraph is a predicate. Typically we have either binary or unary hyperedges. Each field or value tested by a predicate becomes a node. For example, “\( R.x = S.y \) AND \( R.z < 10 \)” becomes two hyperedges, one (bidirectional, equality) from node \((R.x)\) to \((S.y)\) and one on \(R.z\) (think of it as a cycle) with “\(< 10\)”.

The QueryGraph is the initial execution plan. Following the initial execution plan, we search for the best query plan to execute the query and get the provenance expression for each output. The
searching for the best plans is part of the optimization which we will talk about later in Section 4.6.

4.5.3. Engine for Provenance Reconstruction

In our early explorations of the design space for PROVision, we considered building over or extending existing open-source query processors. However, most of the use cases for PROVision are based on data in files, we needed to support user-defined functions in several different languages, and we needed the operators to compute provenance. Hence PROVision has its own custom query engine implemented in Java, with support for pipelined execution over “batches” of tuples, as well as the ability to execute Python, C, and Java code from within UDF operators. (Our engine was in Java but we used JNI and Jython to interface with external code.)

The query engine is built using an iterator model, in which tuples are recursively requested from root to child operators. Every tuple carries a unique provenance polynomial expression, itself stored as an in-memory expression (tree) with references. As input tuples are composed within an operator (e.g., a join) to produce output tuples, the output tuples are annotated with polynomial expression trees composed from the inputs (linked by reference, thus avoiding copying). We found that “carrying” the polynomials along with the tuples, produced the best performance within a pipelined query engine; methods for storing the provenance in a separate subsystem added significant overhead due to value copying.

4.6. Optimizing Provenance Reconstruction

The initial execution plan is likely to be suboptimal because the query plans for the workflow module descriptors do not consider costs and potential rewrites. Thus, we introduce a query optimization step, based on logical and physical algebra rewrite rules Graefe (1993). The query optimizer is built in the style of Volcano Graefe (1993) and its successors: it consists of logical-to-logical algebraic transformation rules with optional constraints, as well as logical-to-physical transformation rules with constraints and costs. The search space is internally encoded as an AND/OR DAG, where each node has an associated signature that is the same for logically equivalent expressions (e.g., it accounts for commutativity and associativity). The PROVision framework operates over files, and thus does not have a DBMSs’ sophisticated mechanisms for computing histograms and performing rich cost estimation. However, in fact we have access to the inputs and outputs of each workflow
module, since it was previously run and its output materialized, so for many expressions we can directly use the cardinality of the results. Additionally, in our experience most workflow plans have only a few join and aggregation steps, which limits the error that accumulates through cost estimation. As with the Volcano optimizer, we leverage branch-and-bound pruning to avoid searching plans that are more expensive than the best-known alternative.

**Algebraic rewrites** We implement transformation rules for the standard relational algebra equivalences, e.g., join commutativity and associativity, distributivity of join through union, and predicate and projection pushdown. The optimizer treats the scalar UDF operator as a join with an input binding restriction Florescu et al. (1999), where one of the inputs must have bound values for the UDF’s parameters. Given this framework, it is able to select an optimal ordering among joins and UDF calls. Similarly, as described in Section 4.2, we can re-order the aggregate functions called by a single grouping operator.

**Pruning with semijoins** An optimization unique to PROVision exploits the fact that the user typically only wants to reconstruct the provenance for a *subset of the output tuples* $S \subseteq W$, where $W$ is the output of the workflow. We initially model this as computing the query plan for $W \bowtie S$, i.e., $W$ semijoined with the selected tuples. We then introduce transformation rules for *pushing* the semi-join before selections, projections, joins, and the other operators Ives and Taylor (2008). With the semijoin optimization and the input set of selected tuples, PROVision can heavily prune the results it uses during reconstruction, as we shall see in Section 4.7.

**Provenance expression rewrites** The provenance semiring model has become widely accepted because it preserves the equivalences of the core relational algebra. Our focus in this thesis has been on developing extensions to support user-defined functions (over scalars and groups). Not surprisingly, in the general case we can get very large provenance expressions under this model.

In the general case, this model may have very few equivalence properties. However, in specific cases a variety of equivalences may hold (Section 4.3.4), especially over the composition of tuple-level provenance and location specifiers. However, we note that the location specifiers for many datatypes
and their associated extraction functions are \(n\)-dimensional bounding boxes. Given a composition of extractions applied to a value, \(f(g(a))\), we may be able to simplify the location specifier for \(f\). Suppose \(f\) and \(g\) both represent the substring function (we are doing two levels of string extraction). Then if the location specifier for \(g\) with respect to value \(a\), \(L_g(a)\), is some range \([s_1, e_1]\) representing the substring bounds, and \(L_f(g(a))\) is a subrange of \(g\)'s output from \(a\), \([s_2, e_2]\) — we can replace the overall expression with \(L' = [s_1 + s_2, s_1 + e_2]\). Our query optimizer takes rules such as these and embeds them into provenance expression simplification rules executed by the scalar UDF and grouping UDF operators. Fortunately, for many real-world datatypes and UDFs, we can define simplification rules that allow us to reduce (and find equivalences among) the provenance. We can use such rules both heuristically and as part of query optimization (see the next section) to generate the most efficient means of reconstructing provenance.

We do an in-memory optimization by searching all subsets of the possible query plans. We first build a memo table, which is a map, from “signatures” (canonical IDs for expressions regardless of evaluation order) to nodes in an AND/OR Graph. The AND/OR Graph consists of PlanSubgraphs — which in turn are AlternativePlanSubgraphs (OR) or JointPlanSubgraph (AND). Each is \(n\)-ary. Signatures are BitSets, matching the vector of sources. We first enumerate a physical subplan satisfying the JointPlanSubgraph — then it calls rewrite rules to find alternatives. It will keep (skyline-style) all candidate plans whose cost isn’t strictly worse than the ones it has seen.

**Overall strategies** The above optimizations can be incorporated into two over-arching strategies to reconstruct provenance. PROVision may eagerly recompute and materialize an entire workflow’s results and provenance, in order to allow future inquiry about the provenance of any intermediate or output result. The recompute strategy takes a forward tracing technique which is straightforward. It starts from inputs of the workflow and generating the new workflow based on the annotation. The generation is automatically done by our in-build query engine. The new workflow is built like a query plan which can be executed at our query engine. This execution can process all the inputs of the original workflow and generating the same or a superset of the outputs while generating the provenance information of each output.
Alternatively, we may adopt an on-demand strategy where we only recompute the portion of the workflow necessary to produce the provenance of a specific user selection (and exploit pruning techniques such as semijoin pushdown). Section 4.7 studies the trade-offs between these approaches.

### 4.7. Experimental Evaluations

We evaluate the performance of provenance reconstruction and of PROVision’s ability to address the version inconsistency and missing-parameter reconstruction problems. We consider the impact of different optimization strategies, as well as the space and time overheads of provenance reconstruction. PROVision targets ETL-style data cleaning, integration, and matching workloads; while we ultimately plan to investigate machine learning workflows, they are out of scope for this thesis.

**Experimental Workloads**

We selected workflows and datasets for three common ETL and scientific tasks.

**Gene sequence alignment (Genome).** Scientists often perform DNA sequencing on tissue, and seek to quantify the genes and related proteins. A workflow cleans the sequence records (trim), aligns trimmed sequences against a reference “library” of genes, and finally looks up the genes to determine which proteins are coded. This workflow comes from our biologist collaborators, who use modules from the STAR Dobin et al. (2013) open source alignment toolkit. Our experiments use 145.5M sequences and three versions of STAR (2.3.0, 2.3.1, and 2.4.0), which each produced subtly different results.

**Entity matching (Magellan).** The Magellan Konda et al. (2016) entity matching toolkit provides a suite of blocking, alignment, and ranking algorithms. Magellan workflows include stages for blocking (subsetting pairs of records that are compared to find an alignment) and matching (determining which pairs match above a predefined threshold). Building on example workflows provided with Magellan, we seek to link entities between the ACM Digital Library (1813 records) and DBLP (1780 records).

**Data cleaning (DuDe).** Another common ETL task involves cleaning records within a data set. The Duplicate Detection (DuDe) toolkit Draisbach and Naumann (2010) is a data cleaning frame-
work, which supports the search for tuples that represent the same real-world object in a variety of
data sources (deduplication). Our experiments use a standard DuDe workflow over a compact disc
dataset, with 9763 records comprised of 107 (possibly null) attributes.

The three workflows above have a relatively simple module structure. Moreover, the non-declarative
portions of the code — the call directly to the UDF plus any referenced functions — are proportion-
ally very small. Those referenced functions in general call other libraries and functions which we do
not care about in our descriptor. We summarize the overall complexity of the semantic descriptors
below:

Our experiments were conducted on an Intel Xeon E5-2630 processor running at 2.20GHz with 24
cores and 64GB of RAM. Our implementation used the Java OpenJDK 1.8.0. Results are averaged
over 5 runs and we present 95% confidence intervals.

We address the following questions:

• How much does it cost to recompute provenance (Section 4.7.1)?

• What are the points at which it makes sense to transition from completely recomputing the
  provenance, to selectively recomputing on demand, and can we use cost estimation to get the
  optimal strategy (Section 4.7.1)?

• How quickly can we trace from outputs to inputs to determine debugging instances when we
  have a version mismatch problem (Section 4.7.2)?

• How effectively can we recover missing parameter values (Section 4.7.2)?

<table>
<thead>
<tr>
<th></th>
<th>Genome</th>
<th>Magellan</th>
<th>Dude</th>
</tr>
</thead>
<tbody>
<tr>
<td>Num. of modules</td>
<td>3</td>
<td>3</td>
<td>2</td>
</tr>
<tr>
<td>Num. of operators</td>
<td>12</td>
<td>8</td>
<td>3</td>
</tr>
<tr>
<td>Code refd. by UDF</td>
<td>0.3%</td>
<td>2.2%</td>
<td>0.8%</td>
</tr>
</tbody>
</table>

Table 4: The overall complexity of the semantic descriptors.
<table>
<thead>
<tr>
<th></th>
<th>input size</th>
<th>exec time</th>
<th>space</th>
</tr>
</thead>
<tbody>
<tr>
<td>Genome</td>
<td>3.5GB</td>
<td>13.4hr</td>
<td>127GB</td>
</tr>
<tr>
<td>Magellan</td>
<td>615KB</td>
<td>4.3min</td>
<td>615KB</td>
</tr>
<tr>
<td>DuDe</td>
<td>4.6MB</td>
<td>25.3min</td>
<td>116GB</td>
</tr>
</tbody>
</table>

Table 5: Original workflow execution costs.

4.7.1. Overhead of Provenance Reconstruction

Unlike in many data provenance settings, our task with PROVision is not to instrument an existing workflow system, but rather to later re-execute certain operations in a workflow (using declarative modules instead of the original binaries) to derive record-to-record provenance. This means there is no overhead on the “normal” execution path, but we must consider how expensive it is to reconstruct provenance when needed for analyzing results. We consider this in terms of execution time as well as space, and then we study how particular heuristics and optimizations can reduce the overhead.

We consider three methods for precomputing a complete provenance trace and compare these with a technique for selectively recomputing provenance on-demand based on output tuples of interest to the user. The naive method simply recomputes all data and its provenance, materializing the outputs of each module in the workflow. The materialized results are self-contained, comprising both output and data. The recomp-k method recomputes the provenance as annotations for each tuple, using foreign keys to link the provenance to the data. The recomp-cs method further composes and simplifies provenance expressions, reducing space at the cost of some additional work. Finally, the on-demand method starts with user-selected tuples, and recomputes only the subset of provenance that it needs in order to trace the provenance of those tuples.

Baseline: original workflow running time and space The baseline costs, in terms of space and time, are shown for the original data workflows in Table 5. Naturally, the Genome workflow is the most intensive in terms of space and time; the Web entity resolution workflow is much smaller but requires a fair amount of computation per record; the product data cleaning workflow has a good deal of data but is reasonably fast on a per-record basis.

Full provenance reconstruction We first measure the overhead involved in recomputing prove-
Figure 16: Normalized space overhead.

Figure 17: Normalized execution time.
nance information, using different techniques. Figure 16 validates that the space overhead is quite low, in the range of 7-17%. The overhead depends on the number of extractions, joins, and aggregations: DuDe only contains two such operators, so adds little overhead; Genome and Magellan have multiple join and aggregation steps so they are slightly larger. The provenance, especially when it only contains foreign keys, is substantially larger than the actual data, however, so the overhead is below 20%. We note that simplification of provenance expressions yields a small space benefit. If we look at the execution times in Figure 17, we see that they are essentially identical, hence we conclude that the recomp-es method is the best choice in all circumstances.

We note from Figure 17 that CPU overhead varies significantly across the different workloads, based on (1) how many tuple combinations are being considered in the computation and (2) how many calls are made to the Python UDF code. Our declarative implementation of DuDe essentially performs a complete Cartesian product of all inputs, hence it adds more than 4x overhead. This motivates us to consider on-demand approaches for computing only provenance for particular results.

**Composition and Simplification** We do provenance composition and simplification with the rewriting rules to save space. However, from table 17 column R-CS it will cost slightly more computation time. In table 16, we can see that it saves 2-3% of the provenance size compared with no simplifications. This statistic varies depending on the workflow and data. 2–3% is relatively low which means that the possible simplification we can make in the provenance expression is limited.

**Selective (on-demand) reconstruction.** The on-demand approach generally starts with a set of user-selected output records (see Figure 14), and PROVision uses semi-join pushdown to limit its computation to relevant results. Table 6 shows that for small numbers of outputs, greedily using the semijoin (greedy) results in very efficient provenance computations (between 0.7 and 5 sec). In fact, taking costs into effect (cost-based) results in the same query plan, hence the same execution times. Figure 21 shows how the performance speedups diminish as we select larger subsets of the output.


```
<table>
<thead>
<tr>
<th>Workflow</th>
<th>Full</th>
<th>Greedy</th>
<th>Cost-based</th>
</tr>
</thead>
<tbody>
<tr>
<td>Genome</td>
<td>13.4hr</td>
<td>4.71s</td>
<td>4.71s</td>
</tr>
<tr>
<td>Magellan</td>
<td>4.3min</td>
<td>0.995s</td>
<td>0.995s</td>
</tr>
<tr>
<td>DuDe</td>
<td>25.3min</td>
<td>0.695s</td>
<td>0.695s</td>
</tr>
</tbody>
</table>
```

Table 6: Average CPU time for tracing small subsets of output (5-1000 records).

Figure 18: Execution time vs proportion of output selected, for Genome workflow.

**Choosing full vs. selective reconstruction.** A question is *how much* of the output needs to be of interest before it makes sense to precompute all of the provenance. We investigate this for the Genome, Magellan, and DuDe workflows in Figures 18, 19, and 20, respectively. In each figure, we plot the greedy strategy’s reconstruction time (blue line) versus the cost-based strategy (red line) versus the baseline (compute all results ahead of time, dashed line). We observe that the switch-over point between strategies is typically at around 45-90% of the output results, depending on the workloads. Our cost-based strategy proves to be very effective at choosing the better approach in each case.

4.7.2. Enabling Consistency & Reproducibility

PROVision uses provenance to (1) find input test sets that yield inconsistent results across workflow module versions (debugging “version inconsistency”); and (2) find missing parameter values from workflow runs (“parameter finding”).

**Version Inconsistency**

Our bioinformatics collaborators often face versioning issues. If two versions of a workflow are run over an input record, and their results differ according to `diff3`, PROVision is called. It can
“trace back” from the differing outputs, to find an input data subset useful for testing. We took three versions of our collaborators’ workflow modules (v1-v3), compared the outputs to find differences, and then traced back to the input records that contributed to those outputs.

Relatively few outputs differ between any pair of workflow versions. PROVision can trace back to the specific input records that contributed to those differences — yielding an input set of 3.4-12.7%

<table>
<thead>
<tr>
<th></th>
<th>v1-v2</th>
<th>v2-v3</th>
<th>v1-v3</th>
</tr>
</thead>
<tbody>
<tr>
<td>Prop. of outputs differing</td>
<td>0.3%</td>
<td>2.1%</td>
<td>2.1%</td>
</tr>
<tr>
<td>Prop. of inputs contributing</td>
<td>3.4%</td>
<td>11.6%</td>
<td>12.7%</td>
</tr>
</tbody>
</table>

Table 7: The proportion of inputs corresponding to the different output generated by different versions.
of the original input set. (Execution costs are identical to Figure 18 and thus not reproduced.) This shows that PROVision helps the user focus on a relatively small set of inputs that directly contribute to differences in answers. In fact, the user can generate even smaller test sets by selecting a few outputs of interest from the “diff” and tracing those in a few seconds (as in Table 6).

**Missing Parameter Discovery**

The Magellan entity matching workflow includes stages for blocking, feature selection, and matching. The blocking stage reduces the number of comparisons needed, whereas the feature selection and matching determine the alignment results. We study how PROVision can recover missing information about the features used in a prior workflow execution.

Given a fixed schema, the space of possible features is fixed. The ACM and DBLP tables have 21 candidate features. About half of these are “obvious” features that will always be used, and about half are “tuning” features that need to be adjusted by an expert. Figure 22 shows the cost of exploring the feature space for Magellan to validate which features were used. The cost is exponential in the number of features, but feasible due to PROVision’s ability to test on a subset of the data: It takes between half a minute to about 8 minutes for PROVision to find the set of features used to perform entity matching.
Figure 22: Provenance computation times to fill in unknown feature values, vs number of missing features.
CHAPTER 5 : Data and Provenance Archival

In this section, we focus on how to create a tamper-resistant archive of the data analysis results and their fine-grained provenance generated over time, to enable reproducibility, querying of individual records’ provenance, and verification of authenticity. We develop techniques for derivation-based compression: different computational analyses will often share computational structure, and may also be computed over shared data sources or over inputs with shared records. In such scenarios, we exploit structure and repetition across queries and subqueries. (Of course, we assume such compression is accompanied by value-level compression within the storage system Abadi et al. (2006), but we do not focus on this aspect because it is orthogonal to provenance storage.)

Our focus is on queries that can be expressed within the relational algebra augmented with certain classes of user-defined functions. This includes traditional OLAP-style queries Psallidas and Wu (2018), as well as the types of user-defined functions used for ETL and gene sequence matching Zheng et al. (2019): extraction based on patterns, approximate match, top-k selection. Several custom query engines Psallidas and Wu (2018); Zheng et al. (2019) target such tasks and automatically track and record provenance within their algebraic operator implementations.

5.1. Background and Approaches

Fine-grained provenance capture Green et al. (2007b) involves recording, for each tuple derivation, the relationship between the inputs and the output. Briefly, we associate a provenance token, a unique identifier, with each input or derived tuple. As a query engine derives a tuple from one or more inputs, the fine-grained provenance of this derived result is specified using a polynomial expression over the input provenance tokens, in a way that corresponds to the relational algebra operators performed over the data.

For provenance archival, our goals are twofold: (1) create provenance tokens in a way that ensures the same expressions result in the same tokens, thus allowing for repeated computations to receive the same token values; (2) create provenance tokens in a way that prevents forgery or tampering. In this section, we provide the foundations for our approach, and in Section 5.2 we discuss an
algorithmic strategy for implementing our approach in a relational query engine. We assume the query processor computes and records provenance as it derives results Psallidas and Wu (2018); Zheng et al. (2019).

we will use a custom query engine with query operators that automatically store provenance (we use PROVision Zheng et al. (2019), but Smoke Psallidas and Wu (2018) or instrumented versions of Apache Spark Zaharia et al. (2010); Interlandi et al. (2015) could alternatively be used). More precisely, each operator will store information about each specific derivation. It is making of an output tuple, from one or more input tuples and their provenance. We next consider how to store this derivation, in a way that is suitable for using traditional DBMS storage techniques.

5.1.1. Provenance Semiring Expressions

To capture fine-grained provenance, we adopt the standard provenance semiring model Green et al. (2007a,b), which annotates tuples with provenance polynomial expressions satisfying the properties of an algebraic semiring, defining how the tuples were produced. Relational algebra expressions that are equivalent under bag semantics have provenance polynomials that are equivalent. In the model, base tuples are annotated with unique IDs called provenance tokens. As relational algebra operators are applied, we annotate each derived tuple with an expression over the provenance of the input tuples. This derived tuple incorporates one of two standard operators: (·) representing joint use of inputs to produce a result, and the sum operator (+) representing alternate derivations of the same value. If we let \( P[t] \) designate the provenance expression for tuple \( t \), and look at inputs and outputs as paired tuples and their provenance — then the rules are:

<table>
<thead>
<tr>
<th>Expr.</th>
<th>Inputs</th>
<th>Outputs</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \sigma_\theta(T) )</td>
<td>((t, P[t]))</td>
<td>((t, P[t]))</td>
</tr>
<tr>
<td>( \Pi_a(T) )</td>
<td>((t, P[t]))</td>
<td>((t, P[t]))</td>
</tr>
<tr>
<td>( T_1 \cup T_2 )</td>
<td>((t_1, P[t_1]), (t_2, P[t_2])), ( t_1 \in T_1 = t_2 \in T_2 )</td>
<td>((t_1, P[t_1] + P[t_2]))</td>
</tr>
<tr>
<td>( T_1 \bowtie T_2 )</td>
<td>((t_1, P[t_1]), (t_2, P[t_2])), ( t_1 \in T_1, t_2 \in T_2 )</td>
<td>((t_1t_2), P[t_1 \cdot P[t_2]])</td>
</tr>
</tbody>
</table>

Table 8: Provenance semiring expressions with input, output and query expressions

Provenance management systems Glavic and Alonso (2009); Green et al. (2007a); Karvounarakis
et al. (2010) have developed methods for encoding the polynomials in relational form (in effect as expression trees or graphs). Under this model, while multiple provenance expressions are equivalent, the order of evaluation of the relational algebra expressions notably affects their overall size.

Extensions to the semiring model support aggregation functions Amsterdamer et al. (2011c), as well as user-defined aggregate and extraction functions Zheng et al. (2019). Since this chapter is focused more on the structure of provenance expressions rather than semantics, we briefly sketch how this works here, and refer the reader to Zheng et al. (2019) for details.

For each UDF, we introduce a function symbol in the provenance polynomials. A given UDF, e.g., information extractor, typically extracts data from within some datatype-specific location specifier within a field (e.g., a substring within a string attribute). Finally, a series of function- and datatype-specific equivalence rules may define how different provenance expressions are equivalent.

**Example 5.1.1** Consider a gene sequence matching workflow from computational biology Dobin et al. (2013), specified using the relational algebra and UDFs. This workflow can be captured using physical query operators, as follows: (1) file-scan operators over input text files, reading from them source and reference genome relations $S$ and $R$; (2) a UDF $pfx$ that extracts a prefix from the sequences in $S$ and $R$; (3) a join matching the prefixes extracted from the sources; (4) a filter expression $\theta$ which returns results exceeding some overall similarity criterion. We might encode this as:

$$
\sigma_\theta(\text{udf}_{pfx}(\text{refseq})(\text{scan}(R))) \\
\times \text{udf}_{pfx}(\text{seq})(\text{scan}(S)))
$$

where the UDF calls function $pfx$ over each tuple. Suppose input relations $R$ and $S$ are as follows:

Initially, ignore the prov field representing the provenance. Suppose function $pref$ returns the first 4 characters of the sequence; then for tuples $r_1$ and $s_1$ we will get substring $ccca$ and for tuples $r_2$ and $s_2$...
and $s_2$ we will get prefix `aaae`. The two pairs of tuples will join. Assuming the threshold is met, we will get two final tuples (`'abcetd'`, `'cccatg'`) and (`'edceeq'`, `'aaaedf'`, `'aaaeij'`).

Here, join operators correspond to product operations ($\cdot$) in the provenance polynomials. Following Zheng et al. (2019), user-defined functions are a kind of joint use (combination or Cartesian product) of the input source tuple values from $s$ combined with the results of the UDF, $f(s)$: 

$s \bowtie f(s)$\textsuperscript{1}. Thus, each result tuple is annotated with a new provenance expression containing function $f'$, the input provenance, and additional information (a location specifier Zheng et al. (2019)) identifying the data used by $f$ to return its result.

For example, if we take tuple $R(\text{edceeq}, \text{aattg})$ with provenance token $r_2$ and apply the UDF $pfx$ over it, the result will be a tuple $(\text{edceeq}, \text{aattg}, \text{aaat})$ with provenance expression $r_2 \cdot pfx(r_2)$ where $pfx'$ represents a predicate extracting a new token (“location specifier”) based on $r_2$ and the semantics of function $pfx$.

The semantics of the extraction function $f$ can be broken into (1) the logic that identifies the gene sequence prefix to be extracted, followed by (2) the actual sub-operation (extract-substring) that computes the returned value from the input data. The location specifier specifies which data was used as input to the latter sub-operation. For our example, this would be:

$$P_{\text{substring(\text{refseq})}}(r_1, [0, 3]) \cdot P_{\text{substring(\text{seq})}}(s_1, [0, 3])$$

where the interval $[0, 3]$ is a location specifier for the first 4 characters from the sequence, and the provenance function $P_{\text{extract}}$ represents the function.

\textsuperscript{1}Note $\bowtie$ represents a dependent join Florescu et al. (1999) used to pass parameters when modeling function calls as joins.
Finally, we can factor any provenance polynomial

\[ p_{\text{out}} = e_1(e_2(p_1, p_2, \ldots)) \]

into the composition of two named expressions, \( p_{\text{new}} = e_2(p_1, p_2, \ldots) \) and \( p_{\text{out}} = e_1(p_{\text{new}}) \), where new variable \( p_{\text{new}} \) can be treated indistinguishably from a token in the new expression for \( p_{\text{out}} \).

### 5.1.2. Computing Query Results with Provenance

Section 5.3 describes our implementation, but at a high level, PROVision’s relational query operators not only compute a stream of output tuples from the input tuples; but annotate such tuples with provenance expressions derived from the input tuples’ provenance Zheng et al. (2019). As query results are computed in memory, provenance expressions are encoded as objects passed by reference, so internally each tuple’s associated provenance annotation is an object representing a rooted expression tree (consisting of semiring operations, function symbols, and provenance tokens). Any intermediate node links to the provenance annotation nodes on the input tuples, and so on, back to the provenance over the source data. We illustrate a simple provenance computation with UDF operator in Example 5.1.2.

Figure 23: Provenance graph for Example 5.1.1 query.

Finally, we can factor any provenance polynomial

\[ p_{\text{out}} = e_1(e_2(p_1, p_2, \ldots)) \]

into the composition of two named expressions, \( p_{\text{new}} = e_2(p_1, p_2, \ldots) \) and \( p_{\text{out}} = e_1(p_{\text{new}}) \), where new variable \( p_{\text{new}} \) can be treated indistinguishably from a token in the new expression for \( p_{\text{out}} \).

### 5.1.2. Computing Query Results with Provenance

Section 5.3 describes our implementation, but at a high level, PROVision’s relational query operators not only compute a stream of output tuples from the input tuples; but annotate such tuples with provenance expressions derived from the input tuples’ provenance Zheng et al. (2019). As query results are computed in memory, provenance expressions are encoded as objects passed by reference, so internally each tuple’s associated provenance annotation is an object representing a rooted expression tree (consisting of semiring operations, function symbols, and provenance tokens). Any intermediate node links to the provenance annotation nodes on the input tuples, and so on, back to the provenance over the source data. We illustrate a simple provenance computation with UDF operator in Example 5.1.2.

\[ p_{\text{out}} = e_1(e_2(p_1, p_2, \ldots)) \]
Example 5.1.2 See Figure 23 for the provenance graph corresponding to Example 5.1.1. Each base tuple \((r_1, r_2, s_1, s_2)\) is fed to the UDF operator, which runs function \(pfx\), producing a result (orange) that is combined with the input tuple. The results are joined (\(\cdot\)) to produce the outputs indicated in boldface.

5.1.3. Blockchain and Merkle tree

Blockchain technology is one of the most popular issues in recent years. It is not just only single one technique, but contains Cryptography, mathematics, Algorithm and economic model, combining peer-to-peer networks and using distributed consensus algorithm to solve traditional distributed database synchronize problem. It seems complicated but its core concept is really quite simple. A blockchain is a specific type of database. Blockchains keeps a growing list of record, called blocks that are chained using cryptography. Each block contains a cryptographic hash of the previous block, a timestamp, and transaction data (generally represented as a Merkle tree) Narayanan et al. (2016). Different types of information can be stored on a blockchain but the most common use so far has been as a ledger for transactions. A blockchain system implements a tamper-evident ledger for recording transactions that modify some global states. The system captures entire evolution history of the states. By design, a blockchain is resistant to modification of its data. This is because once recorded, the data in any given block cannot be altered retroactively without the alteration of all subsequent blocks. Blocks in a blockchain hold batches of valid data or transactions that are hashed and encoded into a Merkle tree. Each block in the Merkle tree includes the cryptographic hash of the prior block in the blockchain. This iterative process confirms the integrity of the previous block, all the way back to the initial block.

Figure 24 shows an example Merkle tree. It is a tree in which every leaf node is labelled with the cryptographic hash of a data block, and every non-leaf node is labelled with the cryptographic hash of the labels of its child nodes. To be more specific, a Merkle tree has been defined to be a complete binary tree with a k-bit value associated with each node such that each interior node value is a one-way function of the node values of its children. Hash trees allow efficient and secure verification of the contents of large data structures. Demonstrating that a leaf node is a part of a given binary hash tree requires computing a number of hashes proportional to the logarithm of the number of leaf
nodes of the tree.

This chapter is focused on how to persist the in-memory structure in relational storage — in the process (1) allowing sharing of graph nodes across queries and query subresults, (2) providing support for rapid lookups of subexpressions, and (3) cryptographically certifying the provenance. To do this, we must develop a strategy for assigning node IDs, such that two provenance subgraphs have the same node if they are isomorphic, and the node IDs include a cryptographic hash of the subgraph contents.

5.2. Encoding Provenance Graphs

We now consider how to take provenance that is computed during query processing, and map it to persistent storage. As described in the previous section, the provenance for a running query is encoded in-memory as a directed acyclic graph, where nodes represent expressions and implicitly have edges to their subexpressions, captured as object references. The basic query processing model of Section 5.1.2 creates the graph on a per-query basis.

As a provenance graph node is being mapped to disk, intuitively we assign its identity as a hash of its provenance polynomial expression. This allows us to take a provenance polynomial expression, and quickly look it up (create a new node if it does not exist). Edges are simply binary relations between hashes.

We provide an outline of our goals and approach to provenance archival in Section 5.2.1. We then describe our relational storage scheme in Section 5.2.2, then discuss how we make the stored structure tamper-resistant in Section 5.2.3. We conclude the section by looking at the conditions under which we expect the provenance graph to be space-efficient if the query optimizer is minimizing execution cost.

5.2.1. Provenance Encoding for Archival

Our goal is to support the storage of many derived query results, each accompanied by semiring provenance expressions. We assert that such an archival system should support subexpression sharing for efficiency, and, as a means of enabling auditing, it must also enable certification of authenticity that results and their inputs have not been forged or tampered with.
Cross-query sharing. If we define a test for semantic equivalence of source (Section 5.2.1), then if two queries perform the same algebraic query expression over the same source tuples, we can consider the derivation tree (the results and the provenance) to be the same. Under those assumptions, we should store the subtree for such an expression (a derivation) once, and include it in the provenance of both queries (derivation-based compression). We can maintain a single provenance directed acyclic graph that shares common subexpressions — resulting in more efficient storage. This requires a compositional encoding scheme for provenance expressions, in which a provenance expression can be composed from other provenance subexpressions through factoring and substitution.

Tamper resistance and threat model. Our goal is to have the provenance structure be self-certifying, protecting against attacks on the integrity of the data or code used in a result. We assume a trusted query execution environment, and that standard cryptographic signatures can be used to prevent forged results. We assume a model in which the source data and binary operations (UDFs) will remain available in archival form on the Internet, accompanied by signatures — e.g., through GitHub, public repositories, or archived storage — but that a derived result and its associated provenance structure (a) might be tampered with in order to manipulate the record, or (b) might be alleged to have been derived from a different source or using different code. It’s OK to let the potentially malicious analyst declare equivalences in the query set up, as long as the equivalences are linked to the queries and can be inspected by third parties along with the queries themselves. If the attacker declares the equivalences that are not actually true and causes the provenance to be wrong, an auditor can still detect the reason. We do not assume that standard cryptographic hash functions are collision-free, but assume that we will use a combination of cryptographic hashes that makes it highly improbable that an aspect of the provenance can be modified without affecting a hash value in a tamper-evident fashion.

We address both problems through the careful choice of provenance tokens and cryptographic hashing.
Encoding Source Tuple Provenance

A first question is how to ensure the provenance structure “certifies” the value of the inputs — to do this, we assign values to provenance tokens for the base data with cryptographic signature. In fact, there are a number of subtleties in how we might formalize this, all having to do with the context of a data record: should provenance consider where a record appears (considering, e.g., the file’s full contents, the file’s name or semantic description, the record’s position) or just what its value is? We consider three formulations which preserve increasing amounts of context:

1. **Value-equivalent**: only the data matters, so we create a token by hashing the record.

2. **Origin-equivalent**: here we consider data to be different if it appears in a different place (within the file, if there are duplicate records; or across files). We create a token by hashing the file URI (or other global name) and timestamp, plus the location of the data.

3. **Content-and-origin-equivalent**: the provenance of a tuple further depends on the specific file version in which it appears, as well as where it appears. Here, we create a token by hashing a digest of the file contents, file URI, and the location of the data.

Other definitions are possible, but we feel these three definitions represent the majority of use cases encountered in data science.

Note that the choice of contextual information not only provides different levels of detail about the source, but affects when two provenance derivations are considered equivalent. As we include more contextual information in the token, there are fewer cases when source results are considered the same (which, in turn, reduces the amount of potential subexpression sharing).

Encoding Derived Expressions

Next we consider provenance expressions for derived results. Generally, one would need to reason about algebraic equivalence to determine if two provenance expressions are equivalent. However, a sound-but-incomplete scheme for testing for algebraic equivalence is to look at whether the provenance expressions are structurally equivalent (i.e., the same relational algebra expression over the
same inputs) with the same inputs. We develop a scheme that derives the provenance token in a deterministic way that ensures the *same subexpression* evaluated over the *same input data* always has the same value. This property allows us to (1) store the provenance token and its derivation exactly once in memory, and (2) quickly determine that two expressions are identical if their tokens are identical. We can achieve this through the use of hash functions, applied to the provenance expressions, that are strongly collision resistant — such as cryptographic hash functions.

A natural question is when two operations are equivalent. For relational algebra operations built into the query processor, various physical operators that implement the same logical operator (e.g., different join implementations) are considered equivalent. For user-defined functions, this gets more complex: our implementation assumes that there exists a digest for each function, and that functions are identical if their digests are identical. Of course, if the functions have dependencies to external libraries, or if different runtime conditions affect operation, one would need to generalize this approach. Such approaches require substantial engineering but would fit naturally into our approach (the digest would include all relevant dependencies).

Using cryptographic hash functions as the basis of provenance expression identification provides us with a second opportunity. We can produce an encoding that is self-certifying, in the sense that we can verify that an output was produced through a particular derivation from a given input, and any tampering with the structure or data would not satisfy the hash. To do this, we adopt the Merkle tree Merkle (1987); Ruan et al. (2019), a data structure that has recently been popularized in blockchain protocols Ruan et al. (2019) and used for auditing and accountability Haeberlen et al. (2007). (Our Merkle tree implementation supports variable arity for intermediate nodes.) A Merkle tree in effect uses recursive hashing: an intermediate node is assigned the hash of its children’s hashes, and so on to the leaf level.

For a provenance expression, we derive a provenance token as the *cryptographic hash* of the root operator and any inputs (which, for intermediate nodes, are themselves recursively computed as cryptographic hashes of their inputs, all the way back to the leaf nodes, whose tokens are computed as cryptographic hashes of the original input tuples as in Section 5.2.1). This both ensures the
provenance is tamper-resistant, and also guarantees that a provenance token derived from the same expression and inputs will always have the same value\(^2\).

**Example 5.2.1** Figure 24 shows the Merkle tree of Example 5.1.1. The leaves are the records from the reference genome file \(f_1\) and source genome file \(f_2\). Each line of file \(f_1\) encodes a tuple from relation \(R\), and similarly for \(f_2\) and relation \(S\). Each such input will have associated with it a provenance token, here computed as the hash of the filename and position (origin-equivalent). The first operator, extract prefix, will take the input tuples and generate a token (e.g., \(H_{2,1}\) as an example) for each output, by hashing the operator name (extract\_pref) and input token (\(H_{1,1}\)). Each operator assigns a provenance token to its output, generated by hashing the unique value of the operator type, parameters and input tokens. The token for each output takes the hash of its child hash values (augmented with additional information), forming a Merkle tree structure of the hash tokens.

\(^2\)Note by “same expression” here, we refer to an algebraic expression with the same ordering and structure.

---

**Figure 24:** Merkle tree for Example 5.1.1 query.
In PROVision, we consider two ways of protecting provenance with Merkle trees. The simplest approach is to use large enough cryptographic hashes to make it effectively impossible to find a hash collision and forge a result. Here provenance can be verified without access to the input data. A second approach is to use a smaller hash function in the Merkle tree, where collisions are incredibly improbable but not computationally intractable to find — but then to restrict the range of possible values an attacker can use, by using input data which is permanently available and signed.

5.2.2. Storing the Provenance Graph

Our goal is to encode the graph on disk in relations, in part because this is broadly applicable across a wide range of DBMSes and applications. Our on-disk representation is based on a generalization of a labeled, ordered edge relation for a directed graph. We focus on compression at the logical-level, relying on existing DBMS physical-level data compression to complement our work. Here, we have an n-ary relationship between input expressions (by token), an operator, and a result. Conceptually, the schema of the $PTable$ ("provenance table") relation is:

\[
\langle \text{input}, \text{label}, \text{index}, \text{derived} \rangle
\]

where $\text{input}$ is the ID of an input node, $\text{label}$ defines the algebraic operation or versioned UDF, $\text{index}$ represents the index of the ordered edge with respect to the derived node, and $\text{derived}$ represents the ID of the newly derived node.

The $\text{derived}$ value is generated by hashing the derived tuple’s key fields, the operator that produced the tuple, and the list of input tuples’ tokens. This means $\text{derived}$ can be deterministically computed based on the tuple value and provenance polynomial expression. However, in the worst case, even strong cryptographic hash functions may produce collisions. Thus, the actual, unique node identifier we use is the RID (unique row ID in the storage system) of the first edge (index position 0) stored for this derived node. Each of the $\text{input}$ IDs in the record above is in fact the RID of the record defining the input tuple’s derivation, as opposed to the actual hash value for that tuple’s derivation.
Basic Relational Algebra

As described in Section 5.2.1, the provenance nodes for raw input tuples are given node IDs derived from the tuple values and the appropriate level of context. The tuple keys and context are stored as the node identifier, and the remaining fields can be stored as node content. Selection and projection do not create new nodes in the provenance graph.

Both join ($\cdot$) and union (+), as binary operators, yield two provenance derivation edges. We illustrate with a join from Example 1:

**Example 5.2.2** The join output of Example 1 is represented as two records, one for each input to the join ($\cdot$) operation:

$$PTable(r_{1RID}, \cdot, 0, h(r_1, r_2, 'Join'))$$

$$PTable(s_{2RID}, \cdot, 1, h(r_1, r_2, 'Join'))$$

where $h(r_1, r_2, 'Join')$ represents the cryptographic hash of the expression, $r_{1RID}$ represents the unique row ID of the 0th derivation record for $r_1$, and $s_{2RID}$ represents the unique row ID of the 0th derivation record for $s_2$.

Union has an analogous structure.

Aggregate & User-Defined Functions

Grouping and aggregation (whether through standard or user-defined functions) are encoded analogously to join, but an aggregation may be computed over an arbitrary number of rows. Here, given
$n$ inputs to some aggregate function $f$, we will have $n$ rows of the form:

$$PTable(r_i, f', i, h(r_1, \ldots, r_i, \ldots, r_n, f'))$$

where $1 \leq i \leq n, f'$ represents the name of the function, and as before $h()$ represents the function responsible for generating a deterministic node hash.

**Table-valued functions.** PROVision supports *table-valued* functions, which return multiple result rows that must be joined with the grouping keys. To incorporate this, we add an additional field to the schema, to capture the output row index.

**Equivalent-but-not-identical query expressions.** Our hash-based strategy achieves compression of results produced via the same provenance polynomial expression. This means they must be produced by query subexpressions that (other than selection and projection conditions), follow the same evaluation order. A question we studied was whether we could *rewrite* the subexpressions to equivalent expressions in order to increase sharing (if an existing result used a different evaluation order). Unfortunately, because query expressions are computed bottom-up, sub-expressions are produced before super-expressions, making techniques for rewriting the provenance polynomial from one expression to another ineffective in reducing storage overhead. We thus rely on the fact that the optimal evaluation plan for the same subexpression will be the same, even under minor variations to the selection and projection conditions. In Section 5.2.4 we discuss how the query optimizer’s choice of plans affects the size of the provenance graph.

**Lookup and Tracing**

The $PTable$ table is indexed in two ways: (1) by row ID, allowing direct random-access lookups to the first derivation of a node, as well as sequential lookups of the remaining derivations; (2) by node hash, allowing lookups by hashing the provenance expressions.

The latter is used (1) to determine if there are multiple colliding nodes with the same hash value, (2) to determine if a subexpression has been previously computed, in which case we can *share the*
For provenance querying, we will typically want to trace from a subset of the PTable relation \( P \), namely those output nodes that satisfy predicate \( \phi \), via transitive closure until we get to the input data. Algorithm 4 traverses the edges in \( P \) to return those that are part of the subgraph.

**Algorithm 4 RetrieveProvenance(\( P, \phi \))**

1. \( \text{Ret} = \{ e \mid e \in P \land \phi(e) \} \) \{Edges satisfying \( \phi \}\}
2. \( \text{Parents} = \{ \text{edge.input} \mid \text{edge} \in P \} \)
3. if \( \text{Parents} \neq \emptyset \) then
4. \( \text{Ret} = \text{Ret} \cup \text{ProvSubgraph}(P, \lambda e : e.RID \in \text{Parents}) \) \{ New \( \phi \): edge \( e \)'s RID appears in \( \text{Parents} \) \}
5. end if
6. return \( \text{Ret} \)

In addition to tracing of provenance, another common type of provenance query involves looking for overlap or differences in results (either within or across queries). Here, we can immediately detect commonality by checking the provenance token values as we trace.

5.2.3. Tamper Resistance

Our discussion in this section has largely focused on the use of cryptographic hashing of provenance expressions as a way of generating a (nearly) unique signature for each expression. However, recall from Section 5.2.1 that we actually propose not to precisely hash each provenance expression, but rather to build a Merkle tree to assign a cryptographic hash to it — making it tamper-resistant, and thus useful for maintaining an audit trail, as is needed for a provenance archive.

As in our example of Figure 24, provenance is generated by the PROVision query engine bottom-up, starting with the leaf-level table scan operators. As records are read (and filtered with any pushed-down predicates), their values are cryptographically hashed and added as nodes to our provenance graph edge relation (if an identical node does not already exist). Each node \( n \) has a (nearly impossible to forge, but not fully collision-free) cryptographic hash \( n.\text{derived} \) and upon insertion receives a unique ID \( n.RID \).

At the next relational operator, we take the derived tuple \( n' \) and create a set of derivation edges, each linking to the input tuple(s) (via the unique RID). The cryptographic hash \( n'.\text{derived} \) is formulated
as the hash of the list of inputs’ hash values and any additional parameters, e.g., the derivation operation. We do not rely on the integrity of the rows in the provenance graph storage system, which could in fact be vulnerable to tampering. Instead, we use the Merkle tree to ensure integrity.

Each time PROVision makes a derivation, it looks in the stored provenance graph to see if the derivation has previously been performed — if so, we simply reuse the node, sharing the representation of the subexpression in the graph. This repeats recursively all the way until the root of the tree (the provenance expression of the tuple output by the query) is computed and stored.

We may want to ensure the integrity not only of individual output records, but of an entire output table, i.e., to certify that no records have been added, deleted, or replaced. Here, we can either attach a digest of the result (as is sometimes done when sharing files), or, more commonly we add one more level to the Merkle tree (hashing the hashes of all records) and use that as a digest.

5.2.4. Query Optimization and Provenance Size

Prior work Lee et al. (2019) has studied how to use factorized representations Olteanu and Závodný (2015) to encode provenance, in order to try to reduce the overall size of the provenance. Our scenario is somewhat more complex: PROVision attempts to compute results and provenance efficiently, and thus includes a cost-based query optimizer. We opportunistically exploit shared subexpressions as they result.

The size of the provenance graph is determined by the specific relational algebra expression used to compute the query. It is fairly straightforward to develop a dynamic programming-style cost optimizer to minimize the size of the provenance graph, since space overhead is at least as easy to estimate as running time. This has some drawbacks, e.g., it would have to rely on heuristics to determine which physical operators to use for each logical step (since each results in the same physical storage cost). However, we observe that the provenance graph size is proportional to the amount of intermediate state generated during the query. In fact, query execution time is heavily dependent on intermediate state size, so the two optimization criteria are closely related. In this section we seek to understand this relationship, by considering under which assumptions would a minimum-work query plan, as produced by an optimizer, also produce a minimum-size provenance.
Intuitively, if we ignore access path selection and choosing between alternate algorithms — the query optimizer cost model emphasizes *minimizing the number of intermediate results*, because that minimizes the amount of I/O and work done. Under certain assumptions, the number of “expensive” computations in the query optimizer closely mirrors the number of provenance nodes and edges in the provenance graph.

- Assume all selection and projection operations are maximally pushed down. Suppose we have an index such that *only* those tuples satisfying the selection conditions are retrieved, and we must retrieve an entire tuple to project the desired fields. For a relation $R$ with $n$ tuples, this results in the loading and parsing of $n$ tuples, at some cost $c_{load}$. If in the provenance graph we only store those nodes satisfying the pushed-down selection conditions, this correspondingly results in the creation of $n$ nodes (each receiving an ID that is a cryptographic hash of the data) at storage cost $c_{source}$.

- Assume we have only a single join implementation, a hash join\(^3\) with adequate RAM. Given tables $R, S$ with sizes $m$ and $n$, respectively, a hash join performs $O(m + n)$ hashes during the build and probe stages. It performs $|R \bowtie T|$ Cartesian product operations ($c_{join}$, which typically are the dominant factor in the cost) to produce the join output tuples. The provenance graph will include $|R \bowtie T|$ new derivations from source tuples with storage cost $c_{product}$.

- Assume only one group-by implementation that calls an aggregate function expressed as a UDF. Then for $g$ groups averaging size $n_g$, the group-by will call the UDF $g$ times, at cost $c_{udf}$, each over an average of $n_g$ inputs, resulting in $g$ outputs. The provenance graph will include $g$ nodes representing the groups, linking to $n_g$ input nodes, and producing $g$ result nodes; assume these links take storage cost $c_{agg}$.

If the above assumptions are met, and we can find a single weight $w$ relating query processing time vs provenance storage time, such that $c_{load} = w \cdot c_{source}$, $c_{join} = w \cdot c_{product}$, and $c_{udf} = w \cdot c_{agg}$.

\(^3\)Equivalent performance would be observed with a merge join, but the merge join requires inputs to be sorted on the join keys.
\(w \cdot c_{agg}\), then minimizing the query cost also minimizes the size of the provenance graph. Of course, some of these assumptions may not hold in practice, but nonetheless it is clear that, with a single implementation for each operator, optimizing for query running time is a reasonable heuristic for optimizing provenance graph size.

5.3. Implementation of PROVision Storage

The previous sections established the basic model and core algorithmic approaches to building a tamper-resistant store that exploits common subexpressions to “compress” its data. We now describe the concrete details of implementing this strategy within our custom provenance management and in-memory query/workflow engine, PROVision Zheng et al. (2019). PROVision is an execution engine for processing structured data stored in files, serving a similar role to Python with Pandas, Apache Spark, etc. Users will typically write scripts that execute multiple queries in steps, and/or they may compose queries over views. We allow queries to be written both in SQL and in a lower-level, manually-defined, physical algebra form. Both SQL and algebraic queries can call user-defined functions in Java and Python, which is essential for supporting many kinds of operations in scientific data processing, including gene sequence matching as well as ETL-style data cleaning and record linking.

Our query engine Zheng et al. (2019) supports standard relational algebra (SPJGU) operations, as well as UDFs, and automatically annotates each tuple with provenance. PROVision allows “eager” computation of a query’s output results with provenance, as well as “lazy” computation in which the user may select specific outputs for which he or she wants provenance. This chapter focuses on eagerly computing all provenance and storing it for archival.

We first describe the main components of PROVision, and then conclude the section with a brief discussion of the inherent tradeoffs we will study experimentally, in terms of provenance space overhead versus security guarantees.

5.3.1. System Architecture

Figure 9 depicts the major modules to compute and archive results and their provenance in PROVision. Given a query containing user-defined functions, PROVision looks up any references to view
definitions or user-defined functions in a centralized Registry. The Plan Generator takes any combination of SQL queries, programmer-defined relational algebra expressions, and (Java and Python) UDFs, and generates a single unified query plan.

Query optimization. Next PROVision uses a Volcano-style rewrite-based query Optimizer Graefe and McKenna (1993) to find a more efficient rewriting of the plan to execute. It consults a data catalog which includes statistics about sources (cardinalities, numbers of unique values) that are cached across query runs. The optimizer’s cost estimator uses information about key-foreign key dependencies, as well as input cardinalities, to estimate intermediate result sizes and costs. User-defined functions (which are allowed to return table-valued results, in addition to single values or tuples) are included in the search space.

Query execution. The Query-Provenance Engine implements a standard pipelined relational query processor in Java. User-defined functions may be written in Java or Python (via Jython); UDFs may use an API to provide detailed provenance information to PROVision, described in prior work Zheng et al. (2019). Also, PROVision maintains its own internal representation of provenance as expres-
sions annotating each tuple: As described in Section 5.1.2, each provenance expression is rooted as an in-memory object, which links by reference to the provenance expressions associated with input tuples, forming a recursive data structure across tuples.

Our focus: external provenance storage. The main focus of this chapter is on the rightmost two modules. We incorporate the BerkeleyDB storage system as our provenance storage layer within PROVision\(^4\). To enable tuples and provenance to be stored externally, PROVision uses a Token Maintenance system to generate globally unique IDs for provenance nodes as they are derived (described next), and it stores corresponding provenance graph edge information in a Provenance Storage system (Section 5.2.2).

Token Maintenance

At each step of provenance computation, the Token Maintenance module returns a canonical node ID (provenance token) based on the operator or versioned user-defined function being executed, the inputs, and any other parameters (see Section 5.2.2). Generally, we will directly use a cryptographic hash of the corresponding provenance derivation step, i.e., the root operation list of hashes of the input tokens, and the parameters, as per the Merkle tree structure of Section 5.2.3). Suppose this is \(D\), in which case the cryptographic hash is \(h(D)\). Unfortunately, at scale and for strict correctness, we may (rarely\(^5\)) have to deal with a situation in which two values of \(D\) have the same \(h(D)\).

Detecting Collisions. To determine if we have a collision, we maintain a hash-input map (as a B-Tree with a large in-memory cache) from \(h(D)\) to the set of possible \(D\)s that hash to the same value. Every time we receive (or wish to look up) a new derivation \(D\), we compute \(h(D)\), then look up any matching entries in the hash-input map. If there is no entry, or if \(h(D)\) maps to \(D\), then we are collision-free, so \(h(D)\) can be used to look up all tuples related to derivation \(D\) in the \(PTable\) relation encoding the graph. We use the Provenance Storage module (described below) to look up the first tuple matching \(h(D)\) and return its node ID as a “pointer” to the node entry in the table.

\(^4\)We could use any relational storage system that supports indexing by hash values and a means of retrieving row IDs, but BerkeleyDB allows fast lookups without going through SQL.

\(^5\)This probability is increased if we use fewer bits for the hash value, which may be desirable for speed.
Resolving Collisions. In the event of a collision, \( h(D) \) is not a unique key, thus cannot be used to look up or update the hash node. If this is the first occurrence of derivation \( D \), we call the Provenance Storage module to add all tuples related to this derivation to the \( PTable \), and retrieve the row ID of the first derivation. In a collision map structure, we record a mapping from \( D \) to this row ID. We can look up \( D \) in the collision map to retrieve and return the row ID, as the identity of the node.

We expect the number of collisions to be small, so the overhead of token maintenance should be as low as possible. The collision map (while we persist it to disk) should easily be small enough to fit into memory. We study the overhead in Section 5.4.

Provenance Storage

The \( PTable \) structure described in Section 5.2.2 is mapped to a B+ Tree in our storage manager. It writes each derivation, which may involve several edges, transactionally and it retrieves the row ID of the first derivation edge, as a unique location for the node. Additionally, the B+ Tree for the \( PTable \) relation is indexed by the derivation hash, such that it is possible to look for provenance derivations based on the cryptographic hash at the root of their Merkle trees.

Additionally, we store the persistent state of the Token Maintenance module in B+ Trees. For each of these structures, we rely on buffering and batching to queue up writes, and we rely on buffer pool management to cache frequently accessed values.

5.3.2. Tamper-Resistance vs. Speed/Space

A common concern about cryptographic hashes is that while they are nearly collision-free, they impose significant overhead — not just computationally, but also in terms of space. Many cryptographic hash functions output 256 or more bits — which not only increases storage overhead, but makes operations such as joining on a key more expensive.

For PROVision, we looked at the space of cryptographic hash functions and identified the Blake2 algorithm BLAKE2 as having a good trade-off of computational speed (it is faster than MD5 and the SHA family) and security (its security analysis indicates it is as secure as SHA3 with a 512-
Moreover, Blake2, while slower than Java’s built-in hashCode(), in practice added negligible overhead to our overall running times.

In terms of space, Blake2 is “tunable” to different sizes for its hash values — so we can pick hash digests anywhere from 1 byte to 64 bytes, with stronger security guarantees and fewer collisions as we increase the size of the digest. Note here, hash functions with shorter digest size are enough to get the performance benefits (derivation-based compression, reuse, etc.), but to get the security benefits, we still need a cryptographic hash function with security parameters that are deemed to be safe. In our implementation, Blake2 should generate the token with a size of 256 or 512 bits. Security trade-offs for the different sizes can be considered based on the probability of a collision BLAKE2; in the next section we empirically look at the performance trade-offs (I/O cost, cost of resolving collisions) of different hash sizes.

5.4. Experimental Evaluations

Our evaluation of PROVision as a provenance archival system studies its overhead versus its ability to compress provenance and to certify the integrity of results.

**Tasks and queries.** Fine-grained provenance is useful across a range of applications, extending across loading, wrangling, querying, and analysis. We consider simple OLAP queries based on TPC-H, scientific applications such as Genome sequence matching, and common ETL tasks such as schema matching (Magellan Konda et al. (2016)) and data cleaning (DuDe Draisbach and Naumann (2010)).

**TPC-H:** We use the TPC-H data generator with Scale Factor 0.1 (107MB) to generate 8 input tables with a total of 867K rows. We pick single-block SQL queries (Q1, Q3, Q5), avoiding negation. PROVision executes these queries directly over the CSV files, since its engine focuses on outside-the-database tasks. Q1 does a single aggregation; Q3 joins three tables; Q5 joins six tables.

**Gene sequence alignment (Genome).** Scientists often seek to quantify the genes and related proteins from DNA-sequenced tissue. A query cleans the sequence records (trim), aligns trimmed sequences against a reference “library” of genes, and finally looks up the genes to determine which
proteins are coded. We use components from the STAR alignment toolkit as Python UDFs in PROVision. Our experiments use 145.5M sequences.

Entity matching (Magellan). The Magellan Konda et al. (2016) entity matching toolkit provides algorithms for linking records. Magellan includes stages for blocking (comparing subsets of record pairs to find an alignment) and matching (determining which pairs match above a threshold). Building on example queries provided with Magellan, we link entities between the ACM Digital Library (1813 records) and DBLP (1780 records).

Data cleaning (DuDe). Another common ETL task involves cleaning records within a data set. The DuDe Draisbach and Naumann (2010) data cleaning framework searches for pairs of tuples that represent the same real-world object (deduplication). Our experiments use a standard DuDe example, cleaning a compact disc dataset with 9763 records comprised of 107 (possibly null) attributes.

Workload generation and data. We generate workloads as sequences of queries over subsets of data. For the TPC-H queries, we use the standard query generator to create and parameterize queries Q1, Q3, and Q5. We directly query the tables from the data generator.

For our scientific and ETL tasks, we simulate the execution of similar data analysis tasks across time: subsequent queries are similar but may use different UDFs or selection predicates, and some may be over the same input tables whereas others will be over new data. We define two configuration parameters: the proportion of repeated sub-segments of the input data within a query or across queries (see below for details on data generation) and the similarity between queries in a workload. Each query is generated by parameterizing a template with randomly chosen values, as follows:

- **Genome.** The first stage of gene sequence alignment contains up to 8 trimming conditions. A given query may take any subset of these trimming conditions. For the second step, we choose one of three versions (2.3.0, 2.3.1 and 2.4.0) of STAR.

- **Magellan.** Magellan templates are divided into two record linking steps: blocking and matching. The template for blocking can test for attribute equivalence, or instead use rules. In the
matching stage, we can choose among different string similarity measurement functions.

- **DuDe.** DuDe performs duplication detection tasks. As with Magellan, the template randomly chooses among different similarity functions.

In each step of the workload, we may apply the query over a different set of input relations sampled from the original dataset. We control both the sizes of the inputs and the amount of duplication (which is important for measuring the benefits of subexpression sharing). Given a source relation with \( n \) records, we sample \( k \) records uniformly at random. Then starting from each chosen record, we include the next \( m \) consecutive rows. From this set, if we wish to introduce repetition, we randomly sample with replacement until we reach our target of \( n' \) records. This gives us a set of records with a duplicate-record ratio of \( (k \times m)/n \).

**Environment.** Experiments were conducted on an Intel Xeon E5-2630 running at 2.20GHz with 24 cores and 64GB of RAM. Our implementation used the Java OpenJDK 1.11.0, Python 3.6, PostgreSQL 10.12, BerkeleyDB JE 7.5.11, and the BLAKE2b implementation of the BLAKE2 cryptographic hash. Results are averaged over 5 runs and we present 95% confidence intervals.

In subsequent sections, we experimentally answer the following questions:

1. What is the overhead (in space and time) of cryptographic hashing, and of maintaining our token management structures? If we are willing to trade off probabilistic security guarantees vs. space, do we see benefits? (Section 5.4.1.)

2. How effectively does our hashing-based scheme compress the overall data, when there is sharing of input records, both within and across queries? (Section 5.4.2.)

3. How effectively does our provenance encoding scheme support provenance queries, namely tracing of provenance as well as comparison of provenance? (Section 5.4.3.)

**5.4.1. Space and Time Overhead**

A key consideration in choosing cryptographic hashing is the amount of overhead (both computational and space) that is incurred. In some cases, a provenance token may be larger than the input
Table 9: Baseline workflow, no provenance computation.

<table>
<thead>
<tr>
<th>Data Set</th>
<th>input size</th>
<th>outputs</th>
<th>exec time</th>
<th>PostgreSQL load+exec</th>
</tr>
</thead>
<tbody>
<tr>
<td>TPC-H Q1</td>
<td>71MB</td>
<td>735B</td>
<td>5.2s</td>
<td>2.7+1.2s</td>
</tr>
<tr>
<td>TPC-H Q3</td>
<td>90.4MB</td>
<td>747B</td>
<td>4.9s</td>
<td>3.2 + 0.18s</td>
</tr>
<tr>
<td>TPC-H Q5</td>
<td>90.5MB</td>
<td>258B</td>
<td>7.4s</td>
<td>3.3 + 0.14s</td>
</tr>
<tr>
<td>Genome</td>
<td>3.5GB</td>
<td>127GB</td>
<td>13.4hr</td>
<td>-</td>
</tr>
<tr>
<td>Magellan</td>
<td>615KB</td>
<td>615KB</td>
<td>4.3min</td>
<td>-</td>
</tr>
<tr>
<td>DuDe</td>
<td>4.6MB</td>
<td>116GB</td>
<td>25.3min</td>
<td>-</td>
</tr>
</tbody>
</table>

Baseline performance. To establish a baseline for our measurements, we record the performance characteristics of the queries without provenance capture, in Table 5. Note the substantial diversity in input sizes, output state size, and execution time. We do not view our PROVision platform as interchangeable with PostgreSQL, since it runs over unindexed external data and is written in Java.

Figure 26: Relative space overhead for computing and logging provenance.

data records. Moreover, cryptographic techniques are computationally intensive, so there is a question of how much this affects query execution times. Thus, we begin our experimental evaluation by looking at different space/security (and collision frequency) trade-offs. We consider different hashing schemes for creating tokens, ranging from lightweight (non-cryptographic) hashing to successively larger cryptographic hashes.
as opposed to highly-memory-optimized C; nonetheless we include PostgreSQL times for TPC-H queries to give calibrate our performance. For the other queries, we are unaware of a system (even including other provenance systems such as Smoke Psallidas and Wu (2018)) that is capable of running these UDFs and tracing their provenance.

In the remainder of our experiments, we will plot execution times normalized to the baseline in this table, and provenance space overhead normalized to the combination of input and output result sizes. (For workflow-style computations with multiple steps, the output result size includes all intermediate outputs).

We establish a second baseline for provenance capture and storage itself: the ProvCSV method captures provenance in memory as provenance polynomial expressions, and ultimately writes them alongside the output in a CSV file Zheng et al. (2019). To better understand our space efficiency, we additionally consider the Perm system Glavic and Alonso (2009), which extends PostgreSQL with (non-compressed, non-tamper-resistant) provenance, for queries that run in PostgreSQL; and we adapted the provenance compression techniques of Chen et al. Chen et al. (2017b), in consultation with one of the authors, from Network Datalog to work with PROVision’s query model.
Hashing Schemes vs Space

Figure 26 shows the relative space overheads incurred in tracking and storing provenance, for our various workloads. Recall from our description above that the TPC-H queries are centered around joins and simple aggregation, whereas the scientific and ETL tasks involve substring extraction, approximate match, and ranked, thresholded computations. As noted above, we normalize against Baseline space overhead; and against ProvCSV, which computes provenance in-memory, and writes provenance expressions as strings in the resulting CSV file.

For the remaining bars, we compare different tradeoffs between token size (which affects I/O and even computation costs) versus collisions and probabilistic security guarantees: we explore full cryptographic hashing and compare with options that fit within the storage system’s highly efficient long and int datatypes.

The cryptographically secure method (the PROVision-F bar, which uses full 512b Blake2 hashes) adds 40–90% space overhead versus the baseline input + output result sizes: the overhead of each cryptographic hash (8 bytes) on every intermediate result tuple is nontrivial. For applications that need strong security guarantees, this overhead is nontrivial, but certainly not prohibitive.

In some settings, e.g., if the source data is cryptographic signed, users may be able to tolerate reduced tamper resistance in the provenance structure — so we consider several alternatives. The PROVision-I configuration uses the computationally efficient Java String hashCode(), which offers limited collision resistance. PROVision-32 uses the same space, but replaces the standard hash function with a 32b version of the Blake2 cryptographic hash. Finally, PROVision-64 uses a larger, 64b version of the Blake2 hash, which should have much better collision resistance as well as some (limited) tamper resistance. We see that PROVision-32 does better than PROVision-I with the same efficient 10–20% space overhead, because it uses the same space but has fewer collisions. A reasonable trade-off that has high collision resistance and some tamper resistance is PROVision-64, with 15–39% space overhead. However, we note that the tamper resistance in this case will require we test against a signed copy of the original database, since collisions among hashes in the Merkle
<table>
<thead>
<tr>
<th>Sys. Setting</th>
<th>Nbr. Ops.</th>
<th>Space Overhead</th>
</tr>
</thead>
<tbody>
<tr>
<td>PROVision-F</td>
<td>$2^{256}$</td>
<td>40% to 90%</td>
</tr>
<tr>
<td>PROVision-64</td>
<td>$2^{32}$</td>
<td>15% to 39%</td>
</tr>
<tr>
<td>PROVision-32</td>
<td>$2^{16}$</td>
<td>10% to 20%</td>
</tr>
</tbody>
</table>

Table 10: Theoretical tamper resistance vs space overhead.

<table>
<thead>
<tr>
<th>PROVision</th>
<th>Q1</th>
<th>Q3</th>
<th>Q5</th>
<th>Genome</th>
<th>Magellan</th>
<th>DuDe</th>
</tr>
</thead>
<tbody>
<tr>
<td>PROVision-I</td>
<td>115</td>
<td>123</td>
<td>129</td>
<td>11.4K</td>
<td>1245</td>
<td>241K</td>
</tr>
<tr>
<td>PROVision-32</td>
<td>104</td>
<td>116</td>
<td>119</td>
<td>10.5K</td>
<td>1223</td>
<td>238K</td>
</tr>
<tr>
<td>PROVision-64</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>PROVision-F</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

Table 11: Actual collisions observed in workloads.

tree can be fairly easily generated by an adversary.

Table 10 shows the theoretical trade-off between the security level and the space efficiency: “Nbr. Ops” represents the number of hashing operations required before we expect a collision. An adversary is expected to need at least this many operations to find a value that could be substituted for the actual input, without causing a mismatch with the hash. In principle, \( \text{PROVision-F} \) needs \( 2^{256} \) hash attempts to find a collision. For our actual datasets and query workloads, we did not observe any collisions with 64-bit or higher hashing, as shown in Table 11. For 32-bit hashing, we see that the Java hash function (PROVision-I) is actually reasonably close to the 32-bit cryptographic hash (PROVision-32) in collision resistance, even if it does not provide the same theoretical guarantees.

**Sources of provenance overhead.** PROVision requires space proportional to a query’s intermediate state: each intermediate tuple results in a node in the graph; each provenance expression is stored in multiple \( PTable \) rows (\( n \) rows for an \( n \)-ary operator). We see that TPC-H Q1 (aggregation over a single input table) and the Magellan query — which compute many intermediate results that are aggregated or selected through top-\( k \) computations — have the highest overhead relative to their input + output size. The other queries tend to prune more of the input data and produce proportionally larger output results, so their relative overhead is less. In later experiments, we will show that these intermediate results often can be efficiently shared if they appear repeatedly across queries.
Hashing Schemes vs Execution Time

In our study of execution-time overhead, we seek to identify the individual contributions of query computation, provenance graph I/O, and token management I/O, for the different configurations described in the previous section. In all cases our optimizer produces the same query plan. Figure 27 breaks down query costs into three components: provenance storage I/O (dark lower portions of the bars), query computation (longer, middle segment of the bar), and token management I/O (top of the bar). Provenance storage I/O is a relatively small fraction of the overall running times, since PROVision extensively caches reads in the graph and buffers writes.

Naturally, the query execution overhead depends on the computations being performed within the queries. Significant overhead (approximately 5x versus the baseline) occurs for the DuDe workload: this query computes an internal Cartesian product, requiring many intermediate results and the computation of their provenance, before it prunes to the most promising duplicates. All other queries are notably more efficient, with most adding 10–30% overhead under the different hashing schemes.

We observe that PROVision-32 is slightly (1% to 3%) slower than the other hashing schemes. This is because it is both a fairly expensive hash function to compute, yet it suffers from many collisions. PROVision-64 and PROVision-F are indistinguishable, suggesting that the internal Blake2 algorithm does the same amount of work independent of the output hash size. PROVision-I is slightly faster
due to its low CPU overhead, despite suffering from many collisions (recall Table 11).

Finally, we observe that the provenance token management overhead (largely, I/O to ad from the BerkeleyDB-backed token manager) is on average roughly 5% of the total execution time. The overhead is somewhat higher for Magellan and DuDe because of their large intermediate state.

Figure 28 shows detailed lookup costs incurred for the hash input map, for the Genome query: we can see that on average (i.e., amortized across the query), the BerkeleyDB storage manager (BDB) takes roughly 4 to 10 (for a warm BerkeleyDB cache) 8 to 20 (for a cold cache) $\mu$sec per lookup. As a point of reference, we include (as “inMem”) a plot of the overhead of a fully in-memory implementation of token management.

5.4.2. Provenance Reuse

A major goal of PROVision is to enable sharing of storage for common expressions. We evaluate this by simulating a series of computations, each with small variations in its selection predicates and user-defined functions, that is posed over time, using different input data files. The input data files consist of largely disjoint data, but may have a few similar or identical data records. (See the beginning of Section 5.4 for an overview of the workload generator.) For this experiment, we focus on the scientific and ETL computations, for which we would expect data and computation
to change over time to mimic common patterns in data science. (TPC-H varies the predicates in queries to generate a workload, but it does not vary the data, thus offers limited insights.)

We simulate the evolution of queries and datasets by varying: (1) the percentage of input records in common across different query executions; (2) the amount of overlap in the query plans being executed. For these experiments, we show the PROVision-64 configuration; the same general trends are observed in other configurations.

**Inputs with repeated sub-segments.** To study the effects of repeated input data, for each scientific or ETL application we generate 10 queries $Q_1, \ldots, Q_{10}$ from the associated query template as a workload, and run each successive query over a different input set. For each application, we also generate 10 sets of input files, $D_1, \ldots, D_{10}$. Within these input sets, we control the number of repeated records (measured by the ratio of unique inputs to the total number of inputs). We ultimately run each $Q_i$ over the corresponding $D_i$.

Figure 29, 30 and 31 show, for our three respective query workloads, how the provenance space overhead changes as repetition occurs in the inputs. On the left side of the plot, duplicates are common (the ratio of unique input values is very low); we see that PROVision approaches the storage space of the baseline, because provenance space is amortized across many repetitions. As
we increase the proportion of unique values, we see that PROVision continues to be more efficient than the ProvCSV approach until about 45-75% of the values are unique. The crossover point for Genome (which is aggregation-heavy) is lower than the other two tasks, which are more join-heavy.

Of course, while our experiment measures the effects of repetition within the input file(s), the same benefits will be observed a query is run multiple times across different inputs (or input versions) that share records. This argues that our scheme is effective at “compressing” repeated patterns in provenance expressions.

**Different source token equivalence schemes.** The above experiments adopt a *value-equivalent* formulation to encode the source tokens: records are considered to match based on value-equivalence, independent of input file. Section 5.2.1 proposed several alternative definitions of equivalence, including origin-equivalence (OE) and content-and-origin-equivalence (COE).

To study how stricter definitions of equivalence affect provenance reuse, we simulate the evolution of data and queries over time as follows. We take 10,000 input sequences from the Genome dataset. We generate 10 files, each containing 1K samples from the input, with controlled overlap. We randomly assign a name to each file (potentially giving two files the same name, representing different versions of the file over time), and execute the Genome query over the file. This setup allows us to
duplicated records, where each record may occur in a file with a different name (making it no longer origin-equivalent) and/or a different hash signature (making it no longer content-equivalent).

Figure 32 shows, using a boxplot since every run has slightly different performance, how our different notions of equivalence affect the storage overhead, when compared to value-equivalence (the blue line). The x-axis of the plot shows the ratio of unique records (by record value) across the workload, and the y-axis shows the observed space overhead versus the original data size. Note that when the ratio of unique records is low (left side of the plot), the storage overhead for COE and OE shows high variance and is considerably higher than for value equivalence. As the ratio of repeated records increases, we see the overhead of the different schemes start to become more similar.

**Variations in queries across the workload.** Our previous experiments repeated the same general query (with different selection predicates) over different data — which had the benefit of using the same query plan, and thus producing the same provenance structures over the same input data. Of course data scientists might also change their queries over time, which can result in less commonality among derivations. Nonetheless, even in this setting, the lowest-level query subexpressions are likely to be shared across different query versions.

To study the impact of changing queries, we define a very simple measure of query expression over-
lap. We consider two relational algebra operators to be identical if they have the same parameters, predicates, and (as appropriate) user-defined function code. (Note this is not a recursive definition: it does not consider whether the input subexpressions are identical.)

Given this measure, we can now take a set of queries, and compute the ratio of identical operators as a proportion of the total number of operators; in essence this is a Jaccard similarity between two query expressions. Figure 34 plots the provenance space overhead (normalized against the setting where there are no common subexpressions) versus this ratio of identical operators, for the variations on our three basic queries. The figure clearly shows that as the number of shared operations increases, there are increasing efficiencies in storage as PROVision exploits common subexpressions. The aggregation-heavy Genome query benefits the most from sharing.

---

6The DuDe query template had two operators that could be varied, whereas the others had only one.
5.4.3. Provenance Retrieval and Comparison

Of course, we often need to query provenance after it has been stored — for reproducibility, to determine which inputs were used (as in GDPR), to assess trust, or to compare why two different query outputs’ results are different Karvounarakis et al. (2010); Zheng et al. (2019). Our final set of experiments studies query performance.

**Tracing to intermediate results at fixed depths.** We evaluate the provenance tracing times for different proportions of the overall output, in Figure 33 (a)-(c). For each query, we plot tracing time against the fraction of the output traced, and each line represents the cumulative time to reach a given tracing depth (1 through 3 hops, which was the maximum depth of our individual scientific and ETL queries). Tracing times are roughly linear in the size of the selected output, although for the Genome and Magellan queries at Depth 3, execution times slightly tail off at about 75% of the output size. These queries include an aggregation step that has high fan-in.

**Divergent derivations.** When debugging, sometimes scientists need to compare the outputs of two slightly different queries, to see where their results are identical and where they diverge Zheng et al. (2019). This involves tracing from the two queries’ outputs, determining which paths trace back to common provenance derivations, and which do not (divergent derivations, which may in turn be
composed of common and divergent derivations).

We initially set a frontier set for each query to the selected output nodes. In each tracing iteration, we copy (and remove from the frontier) any nodes that are shared between the two query results: this can be tested by looking at expression equivalence, generally just by comparing token hash values. Such values represent common derivations. We then trace from the remaining nodes on the frontier to their inputs, and repeat the process. Each node we trace represents a derivation that is unique to one of the queries, and is included as a divergent derivation. Figure 35 shows query times through this procedure. DuDe and Magellan run in time approximately linear in the size of the outputs being traced, whereas Genome’s times increase at a significantly higher rate: this is because the Genome data consists of groups ranging in size from 1 to 60,000 tuples.

<table>
<thead>
<tr>
<th>Workflows</th>
<th>Depth</th>
<th>Num. lookups</th>
</tr>
</thead>
<tbody>
<tr>
<td>Genome</td>
<td>8</td>
<td>6834</td>
</tr>
<tr>
<td>Magellan</td>
<td>6</td>
<td>5071</td>
</tr>
<tr>
<td>Dude</td>
<td>3</td>
<td>3500</td>
</tr>
</tbody>
</table>

Table 12: Number of look-ups or edges in the transitive closure with 1000 inputs.

Transitive closure. A third common class of provenance query involves tracing all the way back to inputs via transitive closure — this may be to determine if output records are based on trustwor-
thy inputs, or if they make use of certain records. Our last experiment randomly selects different numbers of inputs as starting nodes, and then computes their the transitive closure in the provenance graph. Figure 36 shows that the query times to find the transitive closure are mostly linear in the size of the given inputs. That is because the query time depends on the number of edges in the transitive closure, i.e., the number of rows retrieved in the PTable (Table 12). We observe that (consistent with our prior query experiments) Genome has the highest rate of increase due to its higher fan-in, whereas DuDe and Magellan are fairly similar.
CHAPTER 6 : Related Work

In today’s era of big data, of complex, ever-changing data analysis computations, we must be able to facilitate the rigor, consistency, and reproducibility of data-driven results, as data and code change across time. Such needs have spurred the development of tools for capturing data provenance Ludäscher et al. (2006); Oinn et al. (2006); Goecks et al. (2010); Bavoil et al. (2005); Cheney et al. (2009); Hellerstein et al. (2017). Provenance lets us understand the history of data, documents, data production steps, and allows us to ensure that different results were processed in a consistent and directly comparable way. Archived provenance can provide better support to query the data history, compare the way data is processed and verify the integrity as well as the persistence of data Chapman et al. (2008b); Xie et al. (2012); Bao et al. (2012).

In this chapter, We first survey related work in application domains other than relational databases, scientific workflows and business computations (Section 6.1). We then summarize different provenance granularity levels considering different requirements in various application domains (Section 6.2). Next, we present works in provenance capture techniques (Section 6.3) and provenance storage (Section 6.4). At last, we discuss how our PROVision distinguishes from the existing works (Section 6.5).

6.1. Provenance in Other Application Domains

We have reviewed provenance models in domains of relational databases, scientific workflows and business data analysis computations in Section 2.3. In this section, We continue to discuss other related work in machine learning and network domains.

6.1.1. Provenance in Linear Algebra

Machine learning and artificial intelligence is ubiquitously used in our daily lives. Whenever a decision or a prediction has been made, it’s natural to ask why and whether the result is trustful, especially for critical decisions like making a diagnosis or taking an action in the auto-drive cars, etc. On the other hand, model training in machine learning are particularly sensitive to erroneous data in training datasets. It is indispensable to understand the explanations of a model’s output and how a decision has been made. In machine learning research, the problem of deriving explanations
of machine learning models is called interpretability, which is trying to achieve a similar goal as data provenance. Both seek to find explanations, at different levels of granularity, for the output of a program or a process Buneman and Tan (2019). However, the interpretability problem is generally harder as the model training process is typically opaque and more complex, especially deep neural network models.

Provenance support for linear algebra operations in the context of machine learning tasks has also been recently studied. The work of Yan et al. Yan et al. (2016) has presented the first steps towards using data provenance for the interpretability of machine learning models. Their work builds on the provenance semiring framework Green et al. (2007b) to include basic linear algebra operations. They capture the fine-grained provenance with semantics preserving for linear algebra operations. Linear algebra operations are heavily used in many machine learning techniques, such as linear regression, latent semantic analysis and deep learning. Based on Yan’s work, Wu et al. Wu et al. (2020) present PrIU to capture provenance in the training phase of linear regression and (binary and multinomial) logistic regression. Such provenance aids to understand and debug machine learning models, as well as in identifying and repairing errors in training datasets. This paper presents an efficient provenance-based approach for incrementally updating model parameters without sacrificing prediction accuracy.

6.1.2. Provenance in Network

Provenance has been successfully applied to a variety of areas: many tools and techniques Wu et al. (2014); Handigol et al. (2014) for network debugging and root cause analysis have been proposed; some works Chen et al. (2016); Wu et al. (2015) deploy provenance to diagnose problems in complex networks. The provenance in networks is very different from the provenance in scientific and ETL workflows. They care more about scalability and distribution Chen et al. (2017b,a). However, the network provenance format is generally fixed and simple Chen et al. (2017b) while the data analytics can be complex. The recursion is very common in network provenance Chen et al. (2017b); Zhou et al. (2012); Wu et al. (2014) but not in the provenance in scientific and ETL workflows. Cryptographic hashing is also applied to provenance in network diagnosis Zhou et al. (2011b), where it was used to sign distributed system events in a log. These approaches do not
naturally extend to fine-grained provenance.

6.2. Provenance Granularity

Provenance finds its use in academic and research organizations, as well as in business establishments. As in various application domains, the data organizations, representation granularity, and analysis purposes are not the same, the corresponding provenance may be defined, modeled, and captured differently. These differences introduce a divergence of instrument levels in the implementation of provenance management systems. For instance, the provenance computation can be at the operator level Psallidas and Wu (2018); Cheney et al. (2009); Zheng et al. (2019), the process level Tariq et al. (2012), workflow level Freire et al. (2008), or system level. Those different instrument levels naturally lead to the ability to capture distinct provenance granularity, namely the fine-grained provenance (eg. tuple operator level instrument) and the coarse-grained provenance (eg. process level and workflow level instrument).

In addition, the instrument levels are directly related to the semantic levels of provenance models. For example, instruments at the computation operator level are required for data tuple provenance Green et al. (2007a); Glavic and Alonso (2009); Lee et al. (2019). Meanwhile, integrating provenance at the workflow module level is enough for most of the workflow provenance Anand et al. (2010). Furthermore, the semantic levels of provenance and the corresponding implementation instrument levels impact on the generality of computation. As we aim to collect more details of the computation, additional restrictions on provenance type apply, which further limit the set of supported processes or computations. To illustrate, some highly instrumented provenance systems do not support arbitrary black-box operations, but rather assume relational-style operators Psallidas and Wu (2018); Cheney et al. (2009); Zheng et al. (2019).

The effectiveness of provenance in a specific domain is linked to the level of granularity at which it is collected. Accordingly, the corresponding annotated provenance can be defined and captured at different granularity Tan et al. (2007). The domain requirements can capture provenance on attributes or tuples Buneman et al. (2001b); Green et al. (2007b) in a database that represent distinct substring, individual pixels Woodruff and Stonebraker (1997) or array elements Marathe (2001), which
is fine-grained provenance. Alternatively, in coarse-grained provenance, files or a collection/group of items run through the same experimental run can have provenance for the files or group of items as a whole Oinn et al. (2004); Ludäscher et al. (2006); Bavoil et al. (2005). In some other cases, it may suffice to store provenance at the collection of all files Plale et al. (2005). Some domains require provenance to be stored at multiple levels of granularity and this requires a flexible approach in the provenance capture and storage. The cost of collecting, representing and archiving provenance can be inverse to its granularity and this will play a role in the granularity needed. Our PROVision solves extraction operators and aims to reason about individual data records as well as certify the authentication of the record, which can be achieved only through fine-grained provenance.

6.3. Provenance Capture Techniques

We have shown the prior works by application domains. Now we summarize those works with respect to the techniques used in the provenance capture. For provenance computation, there are three “families” of provenance techniques Cheney et al. (2009), each making different trade-offs.

**Workflow provenance** techniques Oinn et al. (2006); Ludäscher et al. (2006); Goecks et al. (2010) are highly general, as they accept complex workflows consisting of “black box” modules. While being general, they only capture *coarse-grained* (file-process-file) relationships, which limits their ability to help with tasks such as debugging. **Provenance API** techniques Wu et al. (2013) allow programmers to manually instrument arbitrary code with API calls, thus revealing fine-grained provenance. However, such APIs impose notable overhead during standard computations, and they produce provenance that depends on the *order of evaluation* of operations. **Database-style** techniques Cheney et al. (2009); Glavic and Alonso (2009); Karvounarakis et al. (2010) have a long history Cui (2001); Buneman et al. (2002), but today generally leverage and extend the *provenance semiring* model to capture provenance through standard relational operators. Importantly, (bag-)equivalent query expressions, as produced by a query optimizer, yield equivalent provenance. In recent years, extensions have been made to support aggregation Amstderamer et al. (2011c), a variety of middleware Glavic and Alonso (2009); Karvounarakis et al. (2010) and custom engine-based Wu et al. (2013) solutions have been developed, and database-style techniques have been incorporated into “big data” engines such as Hadoop, Pig, and Spark Ikeda et al. (2011); Interlandi
Provenance capture techniques aim at different granularity levels. As shown in the previous section, file-and-program level ("workflow") provenance capture Oinn et al. (2006); Ludäscher et al. (2006); Bavoil et al. (2005) is sufficient to re-run scientific computations to reproduce files ("coarse-grained" provenance). Conversely, the database community has developed techniques to track provenance at the record level ("fine-grained" provenance) Green et al. (2007b); Cheney et al. (2009); Zheng et al. (2019). But most of the studies are focused on tracing through relational operators. The benefits of the latter case are that data scientists can reason about why specific answers exist Chiticariu and Tan (2006). Besides the database and scientific communities, enterprises also have a requirement to trace fine-grained provenance to verify compliance with the GDPR data protection laws, by showing whether an analysis result actually uses private records. But there are only a few studies on fine-grained provenance in the business community. The work of Ruan and his colleagues Ruan et al. (2019) is the first study on fine-grained provenance capture in a BlockChain system.

Finally, the provenance models adopted in the provenance capture techniques need a way to judge whether two provenance instances are equivalent, especially in the application scenario where query rewrites lead to different provenance instances. In relational databases, Green et al. Green et al. (2007b) address this by annotating the provenance with polynomials. As the operators, union (+) and join (·), are associative and commutative, The provenance instances can be normalized and rewritten to the equivalent forms. In the scientific workflow, the provenance model PROV-DM Belhajjame et al. (2013) defines the constraints to provide a way to normalize PROV instances to forms that can easily be compared in order to determine whether two PROV instances are equivalent.

6.4. Provenance Storage Optimization

Provenance storage has been studied, both in terms of the provenance representation (e.g., PROV-DM Belhajjame et al. (2013), provenance semirings Green et al. (2007b)), as well as encoding in a graph or relational DBMS. Early fine-grained provenance systems developed middleware over relational DBMSs and rewrote queries and updates to automatically capture provenance Green et al.
(2007a); Glavic and Alonso (2009); Lee et al. (2019). Later work investigated how query engines themselves could be augmented to capture *semiring provenance* Green et al. (2007b) as part of the evaluation of relational algebra operators Amsterdamer et al. (2011a); Psallidas and Wu (2018); Zheng et al. (2019), enabling a variety of optimizations. Most recently, the work of Lee and colleagues Lee et al. (2019) considered how to rewrite or *factor* relational algebra computations to more efficiently store provenance for individual queries. Other work on networks Chen et al. (2017b) considers provenance compression with techniques that store the core provenance generated by the same network query. Yet, this work does not apply to frequently querying the provenance of intermediate computation steps as we need to expand the core provenance in each query, adding to overhead significantly. While recent works in business Neisse et al. (2017); Ruan et al. (2019); Liang et al. (2017) and networks Zhou et al. (2011a) domains consider certifying results for audit records, these approaches do not naturally extend to fine-grained provenance.

In addition, it is also imperative to provide an easy-to-query and tamper-resistant provenance archive in provenance management systems, especially in enterprise services. As mentioned above, provenance storage has been studied mostly in representation and encoding formats in earlier work. And later work investigated on optimization of provenance query, including provenance algebra rewriting and factorization Lee et al. (2019). None of them is considering the repeated provenance and the tamper-resistant property of the stored provenance. PROVision considers not only the optimization of provenance storage by storing the repeated provenance only once, but also preserves the unforgeability of provenance data.

In fine-grained provenance, the order of evaluation of results affects provenance expression size. Lee and co-authors Lee et al. (2019) pointed out a natural connection between storing provenance semiring expressions and creating more efficient *factorized* representations of queries Olteanu and Závodný (2015). Their PUG system *factors* provenance into a *d-tree* representation Olteanu and Závodný (2015), and uses this as the basis of storage. Similarly, Bao et al. Bao et al. (2012) develop strategies for factoring out provenance storage for common query expressions. In contrast to

---

1Similar observations about the benefits of factorization were previously made for coarse-grained, workflow provenance Chapman et al. (2008a), where one can factor common subexpressions in workflows.
such work, we focus on minimizing query overhead (the common case) and show that our approach is optimal under certain (simplifying) assumptions about cost; we develop mechanisms for “compressing” and authenticating the storage of outputs; we support a richer set of operations including aggregation.

The encoding of fine-grained provenance has also been studied extensively. Perm Glavic and Alonso (2009) uses witness lists to capture, for each output, the set of input tuples used; it duplicates the output tuple if there are multiple witnesses. It can annotate results with their associated SQL statements. Orchestra Karvounarakis et al. (2010) maintains a derivation graph structure using semiring provenance tokens. Chen et al. Chen et al. (2017b) targets network datalog execution, and develops a hash-based encoding scheme for derivation trees, which eliminates intermediate nodes. Anand et al. Anand et al. (2009) develop techniques for encoding provenance for updates to hierarchical data.

6.5. PROVision in Context

PROVision is built upon the literature in the database provenance space Cheney et al. (2009), particularly provenance semirings Green et al. (2007b,a); Amsterdamer et al. (2011c) that capture fine-grained provenance through relational algebra operators, while preserving the algebraic equivalences used by query optimization. In the provenance semiring model, each tuple produced by a query expression is annotated with expressions comprised of tokens representing the input tuples, the product operator (·) representing joint use of inputs (join, Cartesian product) to produce a result, and the sum operator (+) representing alternate derivations of the same value (union, projection). Extensions consider grouping and user-defined functions (UDFs) Amsterdamer et al. (2011c); Zheng et al. (2019), specifically tracking the extraction of sets of values from within user-defined datatypes, and in supporting functions that perform operations such as blocking, approximate matching, and ranking. Another indispensable property with the provenance semiring model is that it matches the equivalence rules which are most commonly used in query optimization. We would like to preserve this property but with a more flexible notion of equivalence rules.

Like Smoke Psallidas and Wu (2018), PROVision is implemented based on our own query process-
ing engine — as opposed to using a standard DBMS Glavic and Alonso (2009); Karvounarakis et al. (2010) that is ill-suited to external data and structured scientific file formats, or an instrumented “big data” engine based on Hadoop, Spark, or Pig Amsterdamer et al. (2011b). Similarly, provenance is computed alongside results, but we emphasize selective provenance reconstruction. Our implementation enables the UDFs to specify what items in an object or a group were “sub-selected”, while also capturing the relationship to the broader object or group. In contrast to SubZero’s Wu et al. (2013) or to event logging Muniswamy-Reddy et al. (2006a); Stamatogiannakis et al. (2016); Muniswamy-Reddy et al. (2006a), our model captures equivalences among computations (including equivalences that hold for particular datatypes and UDFs). PROVision’s query optimizer exploits these to “trace” provenance and aid in troubleshooting.

PROVision studies finer-grained provenance than scientific workflow management systems such as Taverna Oinn et al. (2006), Kepler Ludäsch et al. (2006), VisTrails Bavoil et al. (2005), and Galaxy Goecks et al. (2010). However, we are limited to relational-style operators augmented by “gray-box” operations, where key functionality is described in tuple- or tuple-group-based user-defined functions.

We apply cryptographic hashing to provenance. This has been considered in network diagnosis Zhou et al. (2011a), where it was used to sign distributed system events in a log. Additionally, a number of efforts have integrated provenance into smart transactions in a blockchain Neisse et al. (2017); Ruan et al. (2019) and into storing coarse-grained provenance Liang et al. (2017). To the best of our knowledge, we are the first to consider cryptographic hashing as both a security and sharing mechanism for fine-grained provenance using extensions of the semiring model.
CHAPTER 7: Conclusion

7.1. Summary
In this thesis, we proposed novel techniques to reconstruct and reason about fine-grained provenance in data science and ETL workflows. We developed a strategy for encoding fine-grained (semiring) provenance, such that repeated computations would be stored only once (derivation-based compression) and that provenance would be tamper-resistant. We built the PROVision system consisting of the provenance capture layer, implementing our novel techniques to reconstruct the provenance, and the provenance archive layer, applying our compression strategy as well as supporting tamper-resistant.

More specifically, in the provenance capture layer, PROVision uses an extended relational algebra with UDFs that produce provenance annotations. It incorporates type-and-operator-specific equivalence rules and a novel query optimizer and engine to selectively recompute provenance. Using real ETL and scientific workflows, we showed that our methods efficiently trace erroneous results, create test sets for debugging differences in workflow module outputs, and reconstruct missing parameters. Our approach efficiently and retrospectively reconstructs the information necessary to aid in debugging or filling in workflow data. On the other hand, in the provenance archive layer, we adopt the cryptographic hashing techniques on encoding the provenance annotations. Cryptographic hashing provides a mechanism for quickly determining if a subexpression has been tampered with, and allows us to compare whether two subexpressions are identical. We developed a storage scheme for this model, including for collision resolution. Using our PROVision system, we studied the trade-offs between probabilistic security guarantees and performance.

To sum up, in this thesis, we study two problems – to capture the fine-grained provenance of the ETL-style operators defined in arbitrary programming languages to assist data analysis troubleshooting and reproducibility; to archive provenance in a compressed way and ensuring the tamper-resistant and certification property. We built the provenance management tool – PROVision system to verify that our provenance capture techniques help with the data analysis queries reason-
ing, comparison, and other troubles such as missing parameters and version inconsistent problems. Moreover, the archive layer of the PROVision system provides queriable, auditable, tamper-resistant provenance-augmented data archive services for analysis results, which further better support the query reasoning, troubleshooting and query reproducibility.

7.2. Future Work

As future work, we are interested in expanding our provenance capture techniques to a broader class of workloads, including machine learning tasks. As it’s common to retrain the models on many different new datasets, it may be worthwhile to cache the provenance information about the training process on different datasets. Such provenance information helps the model training to be more efficient as we can only compute the updates of the model training introduced by the new dataset. Besides, the provenance of the model training also helps in identifying why two models trained on various datasets differ. Such ideas have been used recently Ginart et al. (2019); Wu et al. (2020). However, the existing approaches have various limitations: They only apply to specialized problems such as k-means Ginart et al. (2019) or logistic regression Wu et al. (2020).

Further, we would like to explore integration with platforms sharing code, like GitHub or Jupyter notebooks on Kaggle, and dependency managers for tracking binary and code versions. Recent work Koop and Patel (2017); Carvalho et al. (2017) has considered more sophisticated models for tracking dependencies among cells over notebook platform. We would like to exploit semantic relationships to find and propose relationships among data and code artifacts, and ultimately to promote the effective reuse of the “best” data products and code, especially to find commonalities.

Besides, we would like to study more on storage. As we justify the choice of query syntax dependence of provenance with the online setting where the provenance of the input of an operator has been stored beforehand and, thus, its structure is determined by what past queries have been run. In principle, it would be possible to keep a ”window” of provenance to allow further compression before persisting provenance in its final representation. On top of that, we would like to investigate whether we might store provenance “templates” instead of full trees, to capture patterns more compactly.


A. Chen, Y. Wu, A. Haeberlen, W. Zhou, and B. T. Loo. The good, the bad, and the differences:


J. N. Foster, T. J. Green, and V. Tannen. Annotated xml: queries and provenance. In Proceedings


N. Handigol, B. Heller, V. Jeyakumar, D. Mazières, and N. McKeown. I know what your packet


