ABSTRACT

Title of dissertation: PROVENANCE MANAGEMENT FOR COLLABORATIVE DATA SCIENCE WORKFLOWS

Hui Miao
Doctor of Philosophy, 2018

Dissertation directed by: Professor Amol Deshpande
Department of Computer Science

Collaborative data science activities are becoming pervasive in a variety of communities, and are often conducted in teams, with people of different expertise performing back-and-forth modeling and analysis on time-evolving datasets. Current data science systems mainly focus on specific steps in the process such as training machine learning models, scaling to large data volumes, or serving the data or the models, while the issues of end-to-end data science lifecycle management are largely ignored. Such issues include, for example, tracking provenance and derivation history of models, identifying data processing pipelines and keeping track of their evolution, analyzing unexpected behaviors and monitoring the project health, and providing the ability to reason about specific analysis results. We address these challenges by ingesting, managing, and analyzing rich provenance information generated during data science projects, and using it to enable users to easily publish, share, and discover data analytics projects.

We first describe the design of our unified provenance and metadata manage-

ment system, called ProvDB. We adopt a schema-later approach and use a flexible graph-based provenance representation model that combines the core concepts in version control and provenance management. We describe several ingestion mechanisms for this provenance model and show how heterogeneous data analysis environments can be served with natural extensions to this framework. We also describe a set of novel features of the system including graph queries for retrospective provenance, fileviews for data transformations, introspective queries for debugging, and continuous monitoring queries for anomaly detection.

We then illustrate how to support deep learning modeling lifecycle via the extensibility mechanism in ProvDB. We describe techniques to compactly store and efficiently query the rich set of data artifacts generated during deep learning modeling lifecycle. We also describe a high-level domain specific language that helps raise the abstraction level during model exploration and enumeration and accelerate the modeling process.

Lastly, we propose graph query operators and develop efficient evaluation techniques to address the verbose and evolving nature of such provenance graphs. First, we introduce a graph segmentation operator, which queries the provenance of a collection of user-given vertices (e.g., versioned files, author names) via flexible boundary criteria. Second, we propose a graph summarization operator to aggregate the results of multiple segmentation operations, and allow multi-resolution interaction with the aggregation result to understand similar and abnormal behaviors in those segments.

PROVENANCE MANAGEMENT FOR COLLABORATIVE DATA SCIENCE WORKFLOWS

by

Hui Miao

Dissertation submitted to the Faculty of the Graduate School of the
University of Maryland, College Park in partial fulfillment
of the requirements for the degree of
Doctor of Philosophy
2018

Advisory Committee:
Professor Amol Deshpande, Chair/Advisor
Professor Larry S. Davis
Professor Alan Sussman
Professor Richard Marciano
Professor Héctor Corrada Bravo

© Copyright by
Hui Miao
2018

Acknowledgments

It has been a long journey to complete and defense this dissertation. I owe my gratitude to all the people who help me and walk beside me in the fulfilling journey.

First and foremost, I wish like to express my greatest appreciation to my advisor, Professor Amol Deshpande, a great advisor and mentor, who trains me as a researcher with the ability to identify important systems issues and work on challenging problems. Amol gave me freedom to explore new topics, propose ideas and pick the ones that I was interested in but may have high risks, while trusted in me when entering into new fields and helped me stay on a right track that lead to concrete results. He taught me how to elaborate problem formulations and showed me how to present ideas to the research community. I am very grateful for his invaluable time spent on numerous discussions and manuscripts editing even during late nights and weekends. I feel very fortunate to work with him as my advisor.

I would also like to thank other committee members Professor Larry S. Davis, Professor Alan Sussman, Professor Richard Marciano and Professor Héctor Corrada Bravo for their insightful comments and critiques on this dissertation. My special thanks are given to Larry for supporting the collaboration with his computer vision lab on modeling lifecycle and repository discoveries. The discussion with computer vision practitioners was very fruitful and lead to parts of this dissertation.

I would like to thank Professor Lise Getoor, who was my co-advisor during her time at the University of Maryland. I was fortunate to be mentored by her and part of her LINQS group. I am very grateful to her for her invaluable time and

constant support for my graduate study in the first three years. The projects and collaboration in LINQS gave me in-depth understanding of large-scale inference in relational learning and knowledge graph construction.

I had wonderful summer internship experience in Google Research during my graduate study. I would like to thank my mentor Christopher Olston, and other team members, Alon Halevy, Steven Whang, Neoklis Polyzotis, Sunita Sarawagi, Natalya Noy, Sudip Roy and Xiao Yu. Christopher taught me to select impactful problems, avoid premature optimizations and deliver high quality projects. The GOODS experience on searching enterprise datasets broadened my vision and influenced the area that I worked on later in the dissertation.

Looking backwards, the time spent in Maryland was enriching and enjoyable due to many friends and collaborators. I wish to thank many members in the database group, Souvik Bhattacherjee, Amit Chavan, Udayan Khurana, K. Ashwin Kumar, Jayanta Mondal, Walaa Eldin Moustafa, Abdul Quamar, Theodoros Rekatsinas, Virinchi Srinivas, and Konstantinos Xirogiannopoulos for all the joyful time spent and interesting discussions had together. I would also like to thank many friends and collaborators worked together in A.V. Williams, Stephen Bach, Ruofei Du, Shobeir Fakhraei, Shi Feng, Peixin Gao, Hua He, Bert Huang, Ang Li, Hao Li, Xiangyang Liu, Ben London, Alex Memory, Shangfu Peng, Arti Ramesh, Jay Pujara, Ramakrishna Padmanabhan, Sheng Yang, Chen Zhao. I have immensely enjoyed interacting and collaborating with them throughout my graduate study.

I owe a great debt of gratitude to my family, Xiaolin Xi, Xiaoqin Zhan and Xiran Miao. This dissertation would not have been finished without their constant

love, trust and support during the journey. Words cannot express the gratitude. All of them have sacrificed a lot in order to let me pursue my interest.

Portions of this work were supported by NSF grants 1513972 and 1513443.

Table of Contents

Acknowledgements ii

List of Tables ix

List of Figures x

1 Introduction 1
- 1.1 Motivation . . . 1
- 1.2 Challenges & Opportunities . . . 3
  - 1.2.1 Provenance Representation . . . 4
  - 1.2.2 Ingestion Mechanism . . . 6
  - 1.2.3 Designing Query Facilities . . . 7
  - 1.2.4 System Efficiency Issues . . . 9
  - 1.2.5 Discovering Repositories & Learning From Others . . . 11
- 1.3 Approach & Organization . . . 12

2 Related Work 16
- 2.1 Provenance Systems . . . 16
  - 2.1.1 Workflow Provenance . . . 16
  - 2.1.2 Data Provenance . . . 18
  - 2.1.3 Our Contribution . . . 19
- 2.2 Collaborative Data Science Systems . . . 20
  - 2.2.1 Data Management for Collaborative Analytics . . . 20
  - 2.2.2 Version Control Systems for Data Science . . . 22
  - 2.2.3 Our Contribution . . . 23
- 2.3 Machine Learning Systems . . . 24
  - 2.3.1 Modern Machine Learning Software Systems . . . 24
  - 2.3.2 Modeling Pipeline & Lifecycle Management Systems . . . 25
  - 2.3.3 Our Contribution . . . 26

3 ProvDB System Design for Collaborative Analytics 28
3.1 Unified Data Model . . . 28
3.1.1 Conceptual Data Model . . . 29
3.1.2 Physical Property Graph Data Model . . . 32
3.2 Provenance Ingestion . . . 33
3.2.1 Shell command-based Ingestion Framework . . . 34
3.2.2 User Annotations . . . 35
3.2.3 File Views . . . 36
3.2.4 Extension Modules . . . 37
3.3 Query Facilities . . . 38
3.3.1 Queries over Version/Workflow Graph and Properties . . . 39
3.3.2 Reasoning about Pipelines . . . 40
3.3.3 Introspective Diffs: Shallow vs Deep "Diff" Queries . . . 40
3.3.4 Continuous Monitoring or Anomaly Detection . . . 41
3.4 Case Study . . . 42
3.5 Conclusion . . . 44

4 ModelHub: Managing Deep Learning Projects on ProvDB 46
4.1 Motivation & Approach . . . 47
4.1.1 DNN Modeling Lifecycle and Challenges . . . 47
4.1.2 ModelHub Approach . . . 50
4.2 Background . . . 52
4.2.1 Deep Neural Networks . . . 52
4.2.2 Modeling Data Artifacts . . . 55
4.2.3 Model Adjustment . . . 56
4.2.4 Model Sharing . . . 56
4.3 ModelHub System Overview . . . 57
4.3.1 Data Model . . . 59
4.3.1.1 DNN Model . . . 59
4.3.1.2 VCS Data Model . . . 60
4.3.2 Query Facilities . . . 61
4.3.2.1 Model Exploration Queries . . . 61
4.3.2.2 Model Enumeration Queries . . . 63
4.3.3 ModelHub Implementation . . . 66
4.4 Parameter archival storage (PAS) . . . 67
4.4.1 Weight Parameters & Query Types of Interest . . . 67
4.4.2 Parameters As Segmented Float Matrices . . . 70
4.4.2.1 Float Data Type Schemes . . . 70
4.4.2.2 Bytewise Segmentation for Float Matrices . . . 72
4.4.2.3 Delta Encoding Across Snapshots . . . 73
4.4.3 Optimal Parameter Archival Storage . . . 73
4.4.3.1 Constrained Spanning Tree Problem . . . 80
4.4.3.2 PAS-MT . . . 81
4.4.3.3 PAS-PT . . . 83
4.4.4 Model Evaluation Scheme in PAS . . . 86

4.5 Evaluation Study . . . 88
4.5.1 Dataset Description . . . 89
4.5.1.1 Real World Dataset . . . 89
4.5.1.2 Synthetic Datasets . . . 90
4.5.2 Evaluation Results . . . 91
4.5.2.1 Float Representation & Accuracy . . . 91
4.5.2.2 Delta Encoding & Compression Ratio Gain . . . 92
4.5.2.3 Optimal Parameter Archival Storage . . . 94
4.5.2.4 Retrieval Performance . . . 96
4.5.2.5 Progressive Query Evaluation . . . 97
4.6 Conclusion . . . 98

5 Querying Collaborative Analytics Lifecycle Provenance 99
5.1 Introduction . . . 100
5.2 Challenges & Desiderata . . . 103
5.2.1 Motivating Example . . . 103
5.2.2 Provenance Model . . . 105
5.2.3 Provenance Queries & Challenges . . . 111
5.2.3.1 Segmentation . . . 112
5.2.3.2 Summarization . . . 114
5.3 Segmentation Operation . . . 116
5.3.1 Semantics of Segmentation (PgSeg) . . . 117
5.3.1.1 Source ($\mathcal{V}_{src}$) and Destination Entities ($\mathcal{V}_{dst}$) . . . 118
5.3.1.2 Induced Vertices $\mathcal{V}_{ind}$ . . . 119
5.3.1.3 Boundary Criteria $\mathcal{B}$ . . . 124
5.3.1.4 Discussion . . . 125
5.3.2 Query Evaluation . . . 126
5.3.2.1 Overview: Two-Step Approach . . . 126
5.3.2.2 Induce Step . . . 126
5.3.2.3 Adjust Step . . . 135
5.3.2.4 Discussion . . . 135
5.4 Summarization Operation . . . 136
5.4.1 Semantics of Summarization (PgSum) . . . 137
5.4.1.1 Property Aggregations & Provenance Types . . . 137
5.4.1.2 Provenance Summary Graph (Psg) . . . 140
5.4.1.3 Discussion . . . 142
5.4.2 Query Evaluation . . . 142
5.5 System Implementation . . . 145
5.6 Evaluation Study . . . 147
5.6.1 Dataset Description . . . 148
5.6.1.1 Provenance Graphs & PgSeg Queries . . . 148
5.6.1.2 Similar Segments & PgSum Queries . . . 150
5.6.2 Evaluation Results . . . 151
5.6.2.1 Segmentation Operator . . . 151
5.6.2.2 Summarization Operator . . . 156

5.7 Conclusion . . . 160

6 Discovering Hosted Analytics Projects 162
6.1 Motivation . . . 163
6.1.1 Model Discovery . . . 164
6.1.2 Enriched Project Repository . . . 166
6.2 Model Discovery System Vision . . . 167
6.2.1 System Overview . . . 168
6.2.2 Compare & Rank Models . . . 171
6.2.3 Process & Ensemble Returned Models . . . 175
6.3 Evaluation Study . . . 178
6.3.1 Dataset Description . . . 178
6.3.2 Evaluation Result . . . 179
6.4 Conclusion . . . 182

7 Conclusions 183

Bibliography 186

List of Tables

3.1 ProvDB Ingestion Methods . . . 34

4.1 Popular CNN Models for Object Recognition . . . 54
4.2 A list of key `dlv` utilities. . . . 58
4.3 Float Representation Scheme Trade-offs . . . 71
4.4 Recreation Cost of a Snapshot s_i $\mathcal{C}_r(\mathcal{P}_V, s_i)$ in a plan $\mathcal{P}_V$ . . . 77
4.5 Real World DNN Models used in the Experiment Study . . . 90
4.6 Delta Performance For Lossless & Lossy Schemes, 32-bits . . . 94
4.7 Recreation Performance Comparison of Storage Plans . . . 96

5.1 Summary of Provenance Graph Datasets (PG) . . . 149

List of Figures

1.1 ProvDB High-level System Architecture 13

3.1 Conceptual Data Model 29
3.2 Example Workflow 32
3.3 Provenance Property Graph 33
3.4 Diff Artifacts (Result logging files for two deep neural networks) . . . 43
3.5 Cypher Query to Find Related Changes via Derivations 44

4.1 Deep Learning Modeling Lifecycle 47
4.2 Anatomy of A DNN Model (LeNet) 53
4.3 ModelHub System Architecture 57
4.4 Relationships of Model Versions and Snapshots 68
4.5 A Matrix Storage Graph Example 75
4.6 Optimal Matrix Storage Plan without Constraints 75
4.7 Optimal Matrix Storage Plan Constrained by ${\mathcal{C}_r}^{\psi_i}(s_1) \leq 3 \wedge {\mathcal{C}_r}^{\psi_i}(s_2) \leq 6$ 76
4.8 Compression-Accuracy Tradeoff for Float Representation Schemes . . 92
4.9 Compression Performance for Different Delta Schemes & Models . . . 93
4.10 Comparing PAS Archival Storage Algorithms for SD 95
4.11 Progressive Evaluation Query Processing Using High-Order Bytes . . 97

5.1 A Data Analytics Project Lifecycle Example & Associated Provenance 104
5.2 Illustration of the W3C PROV Data Model 106
5.3 Provenance Graph for the Lifecycle Example (Some edges and properties are not shown) 109
5.4 Segmentation Query Examples 113
5.5 Summarization Query Examples 115
5.6 SimProv Normal Form, SimProv → Re. Lg $\subseteq \mathcal{A}\times\mathcal{E}$; Rg, La, Ra $\subseteq \mathcal{A}\times\mathcal{A}$; Lu $\subseteq \mathcal{E}\times\mathcal{A}$; Ru, Le, Re, Qd $\subseteq \mathcal{E}\times\mathcal{E}$. 129
5.7 Proposed SimProv Rewriting, SimProv → Ee. Aa $\subseteq \mathcal{A}\times\mathcal{A}$; Ee $\subseteq \mathcal{E}\times\mathcal{E}$. 132
5.8 Position of Proposed Graph Query Operators in ProvDB 145
5.9 Comparing Cyper Query 5.1, CflrB, SimProvAlg and SimProvTst Efficiency by Varying Graph Size N 152
5.10 Evaluate Performance for Different Project Stereotypes by Varying λ_i 154

5.11 Impact of Input Size λ_i on CFLRB, SIMPROVALG and SIMPROVTST 154
5.12 Effectiveness of SIMPROVALG and SIMPROVTST with Early Stopping 155
5.13 Studying PGSUM on Segments at Different Stages of a Project by Varying Concentration α . . . 157
5.14 Studying PGSUM on Segments with Different Complexity by Varying Activity Types k . . . 158
5.15 Studying PGSUM on Segments with Different Size by Varying Number of Activities n . . . 159
5.16 Studying PGSUM Effectiveness by Varying $|\mathbb{S}|$ . . . 159

6.1 Growth of Github Repositories using iPyhont Notebooks (based on Github Weekly Dump Dataset on Google Big Query) . . . 164
6.2 Enriched Information for Data Science Projects . . . 166
6.3 Overview of ModelHub Discovery Pipeline . . . 170
6.4 Finetuned VGG Models . . . 180
6.5 Correlation Between Aligned Distance and Prediction Results . . . 181

Chapter 1: Introduction

1.1 Motivation

Collaborative data science activities are becoming pervasive in a variety of communities, and are often conducted in teams, with people of different expertise performing back-and-forth modeling and analysis on time-evolving datasets. Current data science systems mainly focus on specific steps in the process such as supporting and accelerating training machine learning models in different data processing frameworks and management systems, scaling to large data volumes by exploiting distributed optimization schemes and system architectures, or serving the data or the models to satisfy demanding service level agreements, while the issues of end-to-end data science lifecycle management are largely ignored. Such issues include, for example, tracking provenance and derivation history of models, identifying data processing pipelines and keeping track of their evolution, analyzing unexpected behaviors and monitoring the project health, and providing the ability to reason about specific analysis results. Addressing these issues in a collaborative data science environment is especially challenging because the process of collaborative data science is often ad hoc, typically featuring highly unstructured datasets, an amalgamation of different tools and techniques, significant back-and-forth among

the members of a team, and trial-and-error to identify the right analysis tools, algorithms, models, and parameters.

More specifically, many modern data analytics activities are ad hoc and reactive to emerging day-to-day problems rather than locking in to a stable platform on well-established business models or scientific workflows. There is no easy way to capture and reason about ad hoc data science pipelines, many of which are often spread across a collection of cleaning and modeling efforts in analysis scripts and the back and forth steps in transient command line histories. Metadata or provenance information about how datasets were generated, including the programs or scripts used for generating them and/or values of any crucial parameters, is often lost. Similarly, it is hard to keep track of any dependencies between the datasets. As most datasets and analysis scripts evolve over time, there is also a need to keep track of their versions over time; using version control systems like `git` can help to some extent, but those do not provide sufficiently rich introspection capabilities to deal with unexpected situations. Moreover, even in a managed version control repository, learning curve for new team members is high, and understanding and reproducing others' work is challenging, e.g., many issues may occur and often one has to contact the author when reproducing results using an open source repository, as the derivation history is not present.

We argue that provenance and metadata about the versioned artifacts and derivations during the analysis process are the key to solve the challenges that are pervasive in the ad hoc data analytics activities. Due to the lack of platform support for capturing and analyzing such provenance and metadata information, data

scientists are required to manually keep track of, and act upon, such information, which is not only tedious, but error-prone. For example, data scientists must manually keep track of which derived datasets need to be updated when a source dataset changes. They often use spreadsheets to list which parameter combinations have been tried out during the development of a machine learning model. Debugging becomes significantly harder; e.g., a small change in an analysis script may have a significant impact on the final result, but identifying that change may be non-trivial, especially in a collaborative setting. It is similarly challenging to identify which input records are most relevant to a particular output record. Repeatability can often be very difficult, even for the same researcher, because of an amalgamation of constantly evolving tools and datasets being used, and because of a lack of easy-to-use mechanism to keep track of the parameter values used during analysis or modeling. Critical errors may be hidden in the mess of datasets and analysis scripts that cannot be easily identified; e.g., a data scientist may erroneously be training on the test dataset due to an inadvertent mistake while creating the testing and training datasets.

1.2 Challenges & Opportunities

In this dissertation, we explore the challenges and the opportunities in unified management of versioned artifacts and all kinds of provenance and metadata about collaborative data analytics activities that gets generated during the analysis process; this includes the information about the different versions of the datasets

that are created, derivation metadata about how derived artifacts were generated, fine-granularity provenance information, dependencies across datasets, and so on. Our hypothesis is that by combining all this information into one place, and making it easy to analyze or query this information, we can enable a rich set of functionality that can simplify the lives of data scientists, make it easier to identify and eliminate errors, decrease the time to obtain actionable insights, and accelerate the process to get into other data analytics activities.

This is hardly a new observation, and there has been much prior work on capturing and analyzing provenance in a variety of communities. However, there is still a lack of practical systems that treat different kinds of provenance and metadata information in a unified manner, and that can be easily integrated in the workflow of a data science project. At the same time, the widespread use of data science has brought to the forefront several important and crucial challenges, such as ethics, transparency, reproducibility, etc., and we posit that fine-granularity provenance is key to addressing those challenges.

There are however several crucial systems requirements and conceptual challenges that must be addressed to fully exploit those opportunities.

1.2.1 Provenance Representation

It is hard to define a unified *schema* for multiple types of information at various granularity a priori to support very diverse analytics processes, as different users of different analytics workflows may wish to capture and analyze different types of such

data. In a collaborative analytics project, such information includes:

- The raw datasets stored in files, which range from structured datasets (e.g., database files, CSV) to semi-structured (e.g., graphs, JSON, Jupyter notebooks) to highly unstructured (e.g., text datasets, audio, images);
- The derivation metadata that captures versioning information about how files within or across versions were created and dependencies among them – a program or a script normally changes files and generates versions, however, such programs and scripts themselves may have different versions;
- Record-level provenance information that is used to connect records across different versions, and structured results of running a data analysis pipeline or an experiment;
- The details about work pipelines of a team, such as task assignments among the team members, project conventions about certain tasks, which often evolve during the process and exist implicitly in the team activities.

Existing solutions can not represent all the information. On one hand, there are data model standardization efforts for general purpose provenance capturing, but they do not treat project artifacts and their versions as first class citizens. On the other hand, there are popular version control systems, such as `git`. Although they can manage the artifact versions and derivations, metadata and provenance about the files are out of their scope; even within the versions, what activities occurred and the derivation dependencies among files are not captured.

1.2.2 Ingestion Mechanism

Even if a data model can be designed, how to ingest the rich information with as little effort as possible from the team is particularly challenging.

In practice, collaborative analytics teams use an amalgamation of software libraries, tools and distributed systems. There are many important aspects of information about how a piece of data was derived that is typically lost by the tools. An ingestion mechanism should be able to capture the information in different circumstances, including:

- The practitioners often tune model with different hyperparameter settings to improve the results, and try different dataset transformations for the model input. The dataset transformations and the model configuration scripts are two pieces of continuously evolving artifacts, and it can be hard to keep track of the relationships between them;
- For the cases when analysis scripts take user-specified parameters as command line options, which are rarely carefully recorded, it is very difficult to repeat an analysis or to understand the origins of a specific dataset, even by the task owner herself;
- Fine-grained provenance information, i.e., keeping track of which input records generated which output records, is also usually difficult to capture without significant effort on the users' part.

Moreover, unless specifically required (e.g., auditing), capturing metadata and

provenance is far from the main project interests of a typical data science team. The captured information has very high long-term value, but less short-term value compared with working on tasks related to the main project objective. Both of the observations lead to the following requirements for the ingestion mechanism:

- We must be able to capture the information with minimal involvement from the users, otherwise the system is unlikely to be used in practice;
- The types of tools that the user can use in the pipeline should not be constrained, due to the rapid evolving nature of modeling paradigms;
- We should have extensible *provenance ingestion* mechanisms, in order to adapt to different analytics processes and tool environments. For instance, a feature engineering modeling practice is very different from an end-to-end learning practice using deep neural networks, as they have different modeling artifacts, tuning best practices, and corresponding practitioners' work style and conventions. Different extensions would be used for different workloads.

1.2.3 Designing Query Facilities

Perhaps conceptually the most difficult challenge is to develop query facilities and/or declarative abstractions to make it easy and powerful to exploit this data.

First, there is a wide range of general queries and provenance analysis tasks that are of interest. For example, such queries include:

- Identifying or retrieving versions of project artifacts;

- Asking lineage of a particular artifact version or a record in that version;
- Comparing multiple lineages to show their differences;
- Finding all project artifact versions associated with a specific metadata.

These queries could be written in a general query language (e.g., SQL, SPARQL, Cypher), once the information is captured in a corresponding structured data model that is well understood by the query issuers. However, such queries often involve versions and path expressions which, written in general query language, tend to be verbose and difficult to compose. More importantly, the assumption that the user is an expert with full understanding of the underlying workflow is hardly true in practice, due to the complex derivations, and the ad-hoc and collaborative nature of such projects.

Second, different development environments have their own modeling processes and the users may ask questions at different levels; for example, a modeler using deep learning may ask about network architecture and hyperparmeters, while a feature engineering modeler may ask about selected features and regularization method. At the same time, there are many potential advanced tasks that would be modeled over such data including:

- Explanation queries where we are looking for origins of a piece of data or a holistic view of a modeling artifact;
- Introspection queries that attempt to identify flaws with the data science process (e.g., *p-value hacking*);

- Continuous monitoring to identify issues during deployment of a data science pipeline (e.g., *concept drifts* where a learned model doesn't fit new data; changes to input data formats).

Last but not least, analytics process is evolving, and there is often no workflow skeleton as seen in other provenance systems, such as scientific workflows and business processes. The data science workflow consists of repetitive trails which may have subtle differences, some of which include potential errors. The captured provenance and metadata tend to be verbose, and the users who are the team members or outsiders often only have partial knowledge of the project artifacts and work steps. In that situation, designing an easy-to-use query facility to identify the precise and concise information needed by the users is very challenging; some example include: **a)** identifying the most relevant part in the lineage by just specifying the artifacts that the user knows about; **b)** identifying the common and abnormal pipelines among a set of artifact derivations. We should be able to induce contributing artifacts that otherwise may not be specified by the users having only partial understanding of a project.

1.2.4 System Efficiency Issues

We expect many efficiency and optimization issues will arise as the variety of the captured metadata and the volume of the captured data increase. This is especially expected to be an issue in the following aspects: **a)** storing the versions of different types of modeling artifacts, e.g., float numbers and matrices; **b)** answering

graph queries on a progressively evolving workflows in the collaborative team.

In the analytics workflow, due to many repetitive steps, there are often very similar artifacts that have a large storage footprint. One example is the dataset, which is often split into training and testing splits and created as copies; this not only wastes storage space, but also requires careful attention to make sure that the two resultant datasets are disjoint. This can be even trickier if k-fold cross-validation is being used. Another example is the artifact of trained model parameters. Popular modeling methods such as deep learning use billions of float parameters, resulting in model artifacts taking hundreds of GB to store. As the model is trained many times with different network architectures and hyperparameters, many model artifact versions are generated. How to deal with the storage footprint of repetitive trails is a challenge when designing new archiving algorithms. Existing systems (e.g., `git` and GitHub) use a single versioning algorithm for all types of files and discourage versioning important analytics artifacts, such as datasets and trained weights.

Moreover, the derivations among artifacts form graphs, and the evolving nature of analytics workflow lead to verbose and large graphs. Query types of interest in workflows typically involve expensive path queries on the collected graphs, which still has limited support in modern graph databases. Designing efficient query evaluation algorithms for important query types of interest is a challenge and a very important issue.

1.2.5 Discovering Repositories & Learning From Others

In the "Big Data" era, datasets are collected blindly in different domains and industries by logging user and system behavior or labeling data via crowd-sourcing, and there is a large demand to conduct data science projects to find value from it. With more experienced practitioners and better system support, more and more data science projects are built and shared online. For example, there are tens of thousands of hosted Github projects using Jupyter notebooks; deep learning models have been shared by the community starting with publishing models on authors' websites. Now training systems tend to have models hosted at a central portal for practitioners, e.g., Caffe Model Zoo.

However, in current hosting services, identifying a repository which is related to an analytics task is very difficult. For example, for a data science practitioner who is working on her own project and willing to find references, it is not feasible to use available systems to answer the queries such as:

- 'What projects used a similar dataset like mine on a classification problem? (e.g., US census data, 256x256 images)';
- 'Show me a set of diverse projects which explore this specific dataset or use this particular modeling method (e.g., random forest)';
- 'Find or ensemble a model from projects having high recall but reasonable accuracy on a given validation dataset'.

The main reason that the current systems can not answer such queries is

because they view an analytics repository as a collection of files, but not as the unified data and provenance model we discussed in this dissertation. By hosting the enriched repository with the provenance and derivations among project artifacts, we can open up the exciting opportunity to answer those discovery queries, facilitate learning from others, and accelerate the data science project lifecycle.

1.3 Approach & Organization

In this dissertation, we address the aforementioned challenges in a systematic manner. We take a system building approach and design a tool-agnostic system (ProvDB) to ingest, manage and query provenance information, and to allow the users to publish, share, and discover data analytics projects.

Collaborative systems are typically centered around the concept of *versions*, and supported by distributed version control system (DVCS). Recent research extends version control systems (e.g., `git`) originally proposed for collaborative software development to support large datasets [1]. In these types of systems, a version is often immutable and any update to a version conceptually results in a new version with a different version, ID.

ProvDB is a stand-alone system (Figure 1.1), designed to be used in conjunction with a version control system (DVCS) like `git` or DataHub [2]. ProvDB provides a workflow-aware version control commandline toolkit that integrates with DVCS, which handles the actual version management tasks, including supporting the standard *checkout*, *commit*, *merge*, etc., functionality. Using ProvDB, we en-

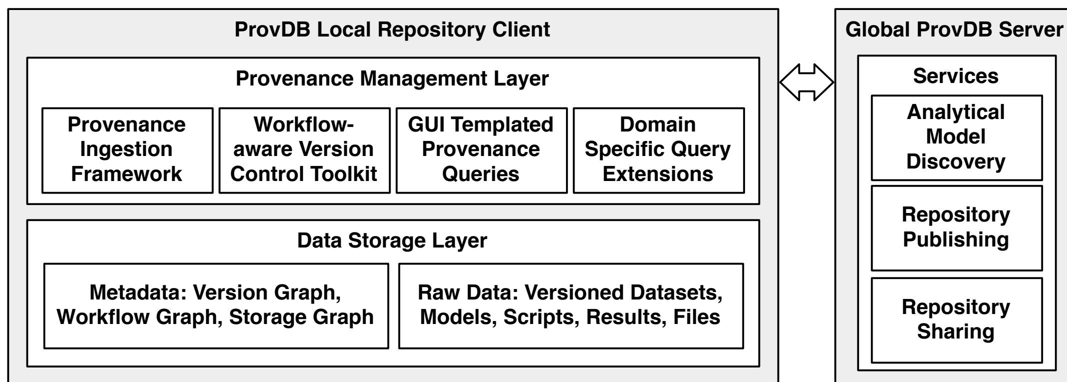


Figure 1.1: ProvDB High-level System Architecture

vision a number of local DVCS "repositories", each corresponding to a team of researchers collaborating closely together. The contents of each repository will typically be replicated across a number of machines as different researchers "check out" the repository contents to work on them. Since we leverage a concrete DVCS (e.g., `git`) for keeping these in sync, the repository contents are available as files for the users to operate upon; the users can run tools as before on those after checking them out, including distributed toolkits like Hadoop or Spark.

In ProvDB, broadly, the data maintained across the system can be categorized into:

- Raw data that the users can directly access and analyze including the datasets, analysis scripts, and any derived artifacts such as trained models;
- Metadata or provenance information transparently maintained by the system, and used for answering queries over the versioning or provenance information.

To capture provenance information, ProvDB organizes metedata (version graphs, workflow graphs, storage graphs) and raw data (versioned datasets, models,

scripts, result files, etc.) in a unified graph data model and provides a set of ingestion mechanisms to capture provenance. On the managed provenance data, ProvDB supports a rich set of query facilities, including asking questions on data lineages, and introspective queries on comparing two versions of a model and explaining the differences. We describe the detailed design of these components in Chapter 3.

In order to better serve users who use specific modeling approaches, we design an extension mechanism in ProvDB. With a concrete modeling paradigm, such as feature engineering and deep learning, a clearer lifecycle can be defined, and we can identify the steps that can be automated to accelerate the modeling process. In Chapter 4, we illustrate our study on extending ProvDB for deep learning modelers on computer vision tasks. We build a deep learning lifecycle management system, ModelHub, to manage artifact versions and provenance of modeling steps in a deep learning lifecycle. In the chapter, we show what kind of provenance information is ingested, how a domain-specific query facility can be used for accelerating the modeling process, and when general version control storage module is inappropriate.

Once the provenance is managed by a system like ProvDB and ModelHub, in Chapter 5, we study how to fully exploit the ingested provenance and help data science project teams. As collaborative analytics projects often have *unstable* lifecycles resulting in evolving and verbose provenance graphs, it is common that team members only have partial knowledge of the provenance graph. Without a predefined workflow skeleton and full understanding of contributing artifacts and steps, it is difficult to write queries and explore the provenance graph using modern graph databases and query languages. In the chapter, we formulate graph query op-

erators for collaborative analytics workflows and propose efficient query evaluation techniques on a modern property graph backend.

In Chapter 6, we examine the opportunities on the server side of lifecycle management systems, and envision a model discovery service. Given the presence of large collections of data science projects uploaded by different groups of data scientists and hosted centrally by a system, model discovery is the problem of identifying relevant projects for a data science practitioner who is working on her own project and willing to find reference. In this chapter, we propose a system around an information retrieval approach, and decompose the discovery task into three major steps: project query and matching, model comparison and ranking, and processing and building ensembles with returned models. Then we describe our system vision, and present opportunities, challenges and techniques for each step.

Chapter 2: Related Work

This chapter surveys the state of the art in provenance systems, collaborative systems as well as model management efforts in the database community. We begin with prior work on provenance systems towards the goal of supporting scientific workflow systems as well as database systems, followed by a review of recent solutions in the version control systems space. Thereafter, we summarize work done towards building systems for data analytics and associated lifecycles.

2.1 Provenance Systems

Provenance information is important in computer-aided tasks. The goal of a provenance system is to capture and manage the origin and history of various objects in its scope, and derivation and passage of an object through its various owners. The prior work can be roughly categorized in two types: workflow (or coarse-grained) provenance [3,4] and data (or fine-grained) provenance [5].

2.1.1 Workflow Provenance

Many workflow provenance systems have been proposed in the scientific applications domain over the years, with some of the prominent systems being Ke-

pler [6], Taverna [7], Galaxy [8], iPlant [9], VisTrails [10], Chimera [11], and Pegasus [12]. A scientific workflow is typically defined for complex data processing tasks, and includes a precise definition of interactions among a collection of computation steps and human-machine interaction steps.. Workflow provenance information includes [13]:

- Prospective information about the definition of the workflow,
- Retrospective information captured during the execution of the workflow,
- Metadata about steps and datasets in a workflow,
- Input/output lineages among steps.

The provenance information captured varies in different systems; it may include a complete record of the sequence of steps taken in a workflow to produce all data points, while in other cases, it may only entail a record of the versions of software used, as well as the models and makes of hardware equipment used in the workflow.

Similar to other data management systems, considering the data being managed is provenance information mentioned earlier, the workflow system consists of key three components: **a)** ingestion mechanism for capturing the provenance data, **b)** a representation schema for the provenance information, **c)** a storage and query system infrastructure [3]. The workflow provenance systems can be distinguished from their scope and design in these components. First, the ingestion mechanism either explicitly requires workflow itself to use callbacks to input prospective and retrospective provenance, or is implicitly implemented at lower levels (e.g., compilers,

OS) to capture the retrospective provenance (e.g., [14–17]). Second, the representation schema is either workflow-specific, or generic, covering partial or thorough prospective and retrospective aspects; there are modern provenance data models standardized over a long period of time as a result of consolidation for scientific workflows [18] and the Web [19]. Third, the infrastructure are dependent on the representation language, ranging from a specialized ontology language (OWL) to XML dialects stored as files to tuples stored in relational database tables.

There are lines of research to improve different aspects of the workflow systems, including proposing query languages to utilize provenance [20–22], operators to transform provenance for ease of use [23–26], storing large provenance graphs efficiently [27,28], and publishing provenance without disclosing important information [29,30].

2.1.2 Data Provenance

On the other hand, fine-grained provenance is often not discussed in workflow systems, but has been rather a topic of focus in the database community. In dataflow systems where the operators are written in a declarative language (e.g., SQL, Pig Latin, Spark), data provenance at record level can be captured if needed [5,31–35]. The fine-grained data provenance system is often built as an auxiliary system component inside a dataflow engine (e.g., SQL database, Spark system). As the dataflow program (e.g., SQL query, Spark program) itself is essentially a clear definition of the prospective provenance, data provenance focuses on retrospective provenance

at the level of individual items in query results. In this setting, verbose logging of derivations of query results is often not affordable due to its large size. A line of work [5] has focused on asking why, how, and where questions about the query results when the detailed execution log is not available.

2.1.3 Our Contribution

Comparing with workflow provenance systems, they often center around creating, automating, and monitoring a well-defined workflow or data analysis pipeline. However they cannot easily handle fast-changing pipelines, and typically are not suitable for ad hoc collaborative data science workflows where clear established pipelines may not exist except in the final, stable versions. Moreover, these systems typically do not support the entire range of tools or systems that the users may want to use, they impose a high overhead on the user time and can substantially increase the development time, and often require using specific computational environments. Further, many of these systems require centralized storage and computation, which may not be an option for large modeling artifacts such as datasets and trained weights.

For collaborative data science workflows, our work aims to combine versioned modeling artifacts and their provenance together on top of a version control system, to provide a uniform platform for collaborative data science workflows. First, we address the challenges in developing a unified data model and tool-agnostic ingestion mechanism for versioned modeling artifacts and their metadata and provenance

during a collaborative data science lifecycle (Chapter 3). Second, given a concrete modeling paradigm (e.g., deep learning), we advance the version storage techniques and provenance query facilities (Chapter 4). Third, as there is no stable skeleton and the users have partial knowledge of the underlying workflow, we develop new provenance query operators (Chapter 5).

In terms of data provenance systems, our work is complementary to, and can utilize, those prior techniques to capture the provenance information itself. In many situations, analysis is not performed using dataflow programs (e.g., SQL), therefore similar problem settings cannot be established in collaborative data scientific workflows without strong assumptions on tool preferences. In our work (Chapter 3), we provide dataflow-based tools to let users write dataflow programs instead of using scripts that may lose prospective provenance, and thus enable prior data provenance work to be used in data science workflows. More importantly, we focus primarily on how to exploit that information for providing richer introspection capabilities on entities in data science workflows, which consists of heterogeneous artifacts at different granularities in contrast to low-granularity tuples output by dataflow programs.

2.2 Collaborative Data Science Systems

2.2.1 Data Management for Collaborative Analytics

Alongside the emergence of big data in many domains, many data management systems have been designed focusing on specific aspects of collaborative analytics. These include building public collaborative systems to share datasets

and analysis findings among non-expert users (Google Fusion tables [36]), addressing data integration issues during sharing of datasets (Orchestra [37]), managing shared SQL queries (CQMS [38], SQLShare [39]) and literate programming notebooks (LabBook [40]), organizing hosted data repositories and domain-specific datasets to be accessible by applications (CKAN [41], Quandl [42], Factual [43]), providing hosted data science platforms (OpenML [44], Domino [45], Amazon SageMaker [46], Google Datalab [47]) and data publishing tools (DataMarket [48]), and enterprise dataset and metadata management systems (GOODS [49], Ground [50], Collibra [51], Apache Atlas [52], Alation [53]).

Most of them do not focus on artifacts and their provenance in the context of data science activities, while there are two projects sharing similar views that aim to improve collaborative data science workflows by reducing the cost of metadata collection and management. LabBook [40], a social data science notebook, uses a property graph to manage metadata captured during collaborative analytics and features a web-based application architecture for analyzing the metadata. However, LabBook does not treat versioning as a first-class construct, and does not focus on developing passive provenance ingestion mechanisms or sophisticated querying abstractions as we do here. Ground [50] is a data context service to manage all the information that informs the use of data. It has a general data model and architecture to import from and export to other systems. However, metadata ingestion and useful high-level query facilities are left to the users.

2.2.2 Version Control Systems for Data Science

For data science, a wide range of analytic packages like SAS [54], Excel [55], R [56], Matlab [57], and Mahout [58], or data science toolkits such as IPython [59], Scikit-Learn [60], and Pandas [61], are frequently used for performing analysis itself; however, those lack comprehensive data management or collaboration capabilities.

In practice, version control systems (VCS), such as `git`, `svn`, were originally proposed for collaborative open source software development. Now VCS and hosted platforms (e.g., GitHub, GitLab, Bitbucket) have become the de facto collaboration platforms for data scientists and many other collaborative projects, e.g., for sharing datasets and IPython/Jupyter notebooks, writing articles, publishing open course materials, etc. In particular, VCS provide transparent support for versioning and sharing, while imposing no constraints on what types of tools can be used for the data processing itself. Thus VCS and hosted platforms built around them are much more appropriate for day-to-day needs in a collaborative data science team.

In terms of the underlying provenance data model and provenance ingestion mechanism, though these systems keep version lineage among committed artifacts and use commit messages to log derivation details, they are typically too "low-level", and have very little support for capturing higher-level workflows or for keeping track of the operations being performed or any kind of provenance information. The versioning APIs supported by these systems are based on a notion of files, and are not capable of allowing data scientists to reason about that data contained within versions and the relationships between the versions in a holistic manner.

Storage Problem & Query Language in VCS

Recently, in a series of efforts [1, 2, 62–64], systems and techniques are proposed to improve the storage infrastructure as well as the query interfaces of the VCS. First, dataset versioning and sharing are identified as an important problem in the context of collaborative data science [2]. The storage problem in VCS is later formally studied in [1], which shows that popular VCS (e.g., `git`) use fairly simple algorithms underneath that are optimized to work with modest-sized source code and have significant limitations when handling large files (e.g., datasets) and large numbers of versions. The storage-retrieval tradeoff of the problem under the delta-based versioning strategy is explored and connected to the balanced minimum spanning tree problem [65]. Next, on a VCS data model, a query language [62] for retrieving versioned data items is proposed, and query evaluation techniques on versioned datasets [63] and versioned relational tables [64] are studied.

2.2.3 Our Contribution

Our system can be seen as a new type of workflow-aware version control system with provenance management capabilities to aid collaborative activities during a data science project lifecycle. Comparing with VCS and prior work (e.g., DataHub, `git`), we enhance versioning data model with workflow provenance and artifacts' metadata information in a unified manner (Chapter 3), and provide rich query facilities spanning different stages of a lifecycle that are tailored for data scientists (e.g., introspection, monitoring, artifact understanding) (Chapter 3). Within a concrete

modeling lifecycle (e.g., deep learning), we explore how to build a specialized system by unifying data and provenance management in order to aid data scientists and accelerate the lifecycle (Chapter 4). Moreover, with the unified model, we open the opportunity to explore new research problems, e.g., discovering relevant data science repositories shared in hosted services (Chapter 6).

In terms of VCS storage and query models, our work shows that the rich set of artifacts (e.g., learned weights) in collaborative data analytics require more general formulation of the VCS storage problems, as well as refined domain-specific language and query facilities over managed modeling artifacts (e.g., explore existing models, and enumerate new models) compared with those in prior work in this aspect (Chapter 4).

2.3 Machine Learning Systems

2.3.1 Modern Machine Learning Software Systems

Nowadays, heterogeneous datasets (e.g., logs, text, images, graphs) are collected in many different domains by logging user and system behavior or labeling data via crowd-sourcing. The sizes of the datasets are increasing (e.g., web clicks of an e-commerce site, graphs representing an entire social network), the complexity of the models is much higher (e.g., millions of features, billions of dimension float weights), and the data-driven algorithmic design-making scenarios have become more common. Traditional analytics packages like SAS [54], Excel [55], R [56], and Matlab [57], or business intelligence and data mining support in commercial rela-

tional database systems, are not sufficient for the needs of modern analytics in terms of **a)** the size of the input data and the complexity of the machine learning models, **b)** the heterogeneous data types and the accessibility from expert and non-expert users performing data analytics.

In the industry and academia, there has been increasing work on developing general-purpose machine learning software systems. On one hand, to address the scalability w.r.t. the increasing model complexity and dataset sizes, there are lines of work proposing software frameworks [58, 66] on top of popular distributed dataflow platforms (e.g., Hadoop, Spark), or novel computation abstractions [67–69] to support training phases of machine learning models. On the other hand, there are single-machine software toolkits developed by supporting multi-dimensional arrays and scientific computing operations [70, 71], integrating data transformation capabilities [61], providing trending modeling methods [72], supporting popular programming languages, and enabling literate programming notebooks [59].

Because many datasets and analytics are conducted in databases, in database community, there are lines of work to support modern machine learning [73], including pushing predictive models and optimization routines into databases [74, 75], and accelerating learning using database join optimization methods [76].

2.3.2 Modeling Pipeline & Lifecycle Management Systems

There is also emerging interest in developing general-purpose systems for handling different aspects of model lifecycle management. Multiple industry tutorials

on machine learning and systems discuss the modeling pipelines in practice [77,78], which are iterative processes consisting of preparing and understanding datasets, training and tuning models, and serving and monitoring model performances.

Accelerating the lifecycle by improving tasks spanning the process, e.g., understanding modeling artifacts and collaborating efficiently with others, will deliver results faster and have drawn significant attention from the database and systems communities.

First, there are lines of work focusing on important aspects spanning the lifecycle, including accelerating iterative updating and training models [79], serving predictive models online [80,81], managing and querying developed models [82–84], organizing provenance and metadata generated in the lifecycle [50,84,85], hosting and discovering reference models [44,86], and assisting collaboration (Section 2.2.1).

Second, there are also platform efforts to support the entire lifecycle [87,88] for expert modelers. Several products and research efforts go further with the goal of using data provided by the user and automatically selecting good models [89,90].

2.3.3 Our Contribution

In terms of machine learning software systems, apart from building or training machine learning models, versioning, collaboration and provenance of the modeling process are largely ignored and left to the users. Our work is complementary to them to a large extent; our focus is on the lifecycle management when a data-driven project team consisting of data analysts/scientists at different skill levels are trying

to collaboratively develop a model. ProvDB can be used as the provenance data management layer for most such systems (Chapter 3, Chapter 5).

Our work overlaps with the model pipeline and lifecycle management systems. In contrast with existing research, our work features tool-neural and extensible provenance ingesting mechanisms without tool or lifecycle assumptions (Chapter 3). We also make contributions in this space by extending our provenance management approach for the deep learning modeling process, which has not been studied in the literature (Chapter 4). Morever, we further discuss important provenance query primitives for iterative and repetitive data science lifecycle, which are not seen in any of the existing work in this space (Chapter 5).

Chapter 3: ProvDB System Design for Collaborative Analytics

In this chapter, we present an in-depth design of the core technical components for the ProvDB system proposed in Chapter 1. First, we adopt a schema later approach and show a flexible data model that combines various elements of data files, model artifacts, versions, workflow derivations, and possible metadata. Next, we explain how the rich provenance data is ingested and how heterogeneous data analysis environments can be served well with natural extensions. Then, a set of general query facilities that are orthogonal to specific environment are described and illustrated in detail. Finally we show the usefulness of the system by a case study on a deep learning project managed by ProvDB.

3.1 Unified Data Model

To encompass a large variety of situations, our goal is to have a flexible data model that reflects versioning and workflow pipelines, and supports addition of arbitrary metadata or provenance information. Though the version control system has a clean model, the analysis steps between the versions are missing in modern systems, and metadata of the versioned files are not addressed.

Based on a versioning data model, we use fixed "base schema" (Figure 3.1)

to capture the information about the versions, the different data artifacts, and so on, while allowing arbitrary properties to be added to the various entities. We map this logical data model to a property graph model (Figure 3.3), which we use as our physical data model to store the information. Comparing with other similar models proposed in the past work [13, 40], our model differs from them primarily in the explicit modeling of versions.

3.1.1 Conceptual Data Model

We view a data science project as a working directory with a set of **artifacts** (files), and a development lifecycle as a series **actions** (shell commands, edits, transformation programs) which perform create/read/update/delete (CRUD) operations in the working directory.

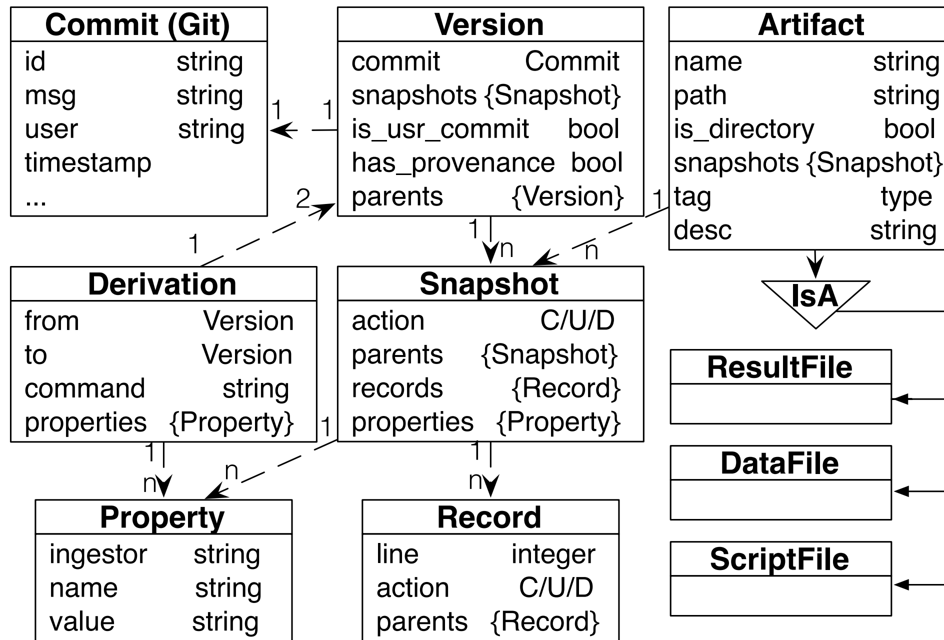


Figure 3.1: Conceptual Data Model

More specifically: an **artifact** is a file, which the user modifies, runs, and talks about with peers. Artifacts can be tagged as belonging to one of three different types: *ResultFile*, *DataFile*, *ScriptFile*, which helps with formulating appropriate queries. A **version** is a checkpoint of the project; in our case, this refers to a physical *commit* created via `git`. ProvDB has *explicit versions* and *implicit versions*; the former are created when a user explicitly issues *commit* command, whereas the latter are created at provenance ingestion time when the user runs commands in the project directory.

Snapshots are checkpointed versions of an artifact, and capture its evolution. ProvDB monitors file changes during the lifecycle, and emits changed (CUD) artifacts as new snapshots, and the previous snapshot of the same artifact before the change is called its *parent*. The content of a snapshot are modeled as **records**, to allow fine-grained record-level provenance (some files, e.g., binary files, would be modeled as having a single record).

Derivations capture the transformation context to the extent possible. If a derivation is performed by running a program or a script, then the information about it is captured along with any arguments. Derivation edges may also be created when the system notices that one or more artifacts have changed (e.g., an edit made using an editor, or a script ran outside the ProvDB context).

Finally, **properties** are used to encode any additional information about the snapshots or the derivations, as *key-value* pairs (where values are often time series or JSON documents themselves). Provenance ingestion tools (discussed in next section) will generate these properties. In addition to the information about programs

or scripts and their arguments, properties may include any information captured by parsing shell scripts or analysis scripts themselves. Properties are also used to extract statistics about the data within the snapshots as well, so that they can be seamlessly queried. This starts blurring the distinction between data and metadata to some extent; we make cleaner distinction for concrete environments, and the users are allowed to annotate and distinguish between these two.

Example 1 *Suppose a user starts an analysis using a script file* script1 *and a data file* datafile1 *by copying them to a repository. She first tries out* script1 *on* datafile1*, and a result file* result1 *with* m *records is generated in the directory. She opens the* result1 *in an IDE and finds a number format issue, after which, she uses* vim *to modify the* script1*, and runs it again on* datafile1*, then each* record *in* result1 *is changed. In Figure 3.2, assuming the system has a commit at the end of each command in the shell, we show the versions, artifacts, snapshots, and derivations. Between the versions, the command is captured as a* Derivation*, whose properties would be the command line arguments (i.e. options, parameters). Each version includes a set of snapshots associated to an artifact. As shown in the figure,* result1 *is an artifact across three versions. It worth pointing out, some of the derivation context happened in the IDEs when opening the* result1 *at the first time may be important as well, which cannot be captured simply by looking at command lines. If there is change before a derivation,* ProvDB *detects it and marks the derivation as missing provenance.*

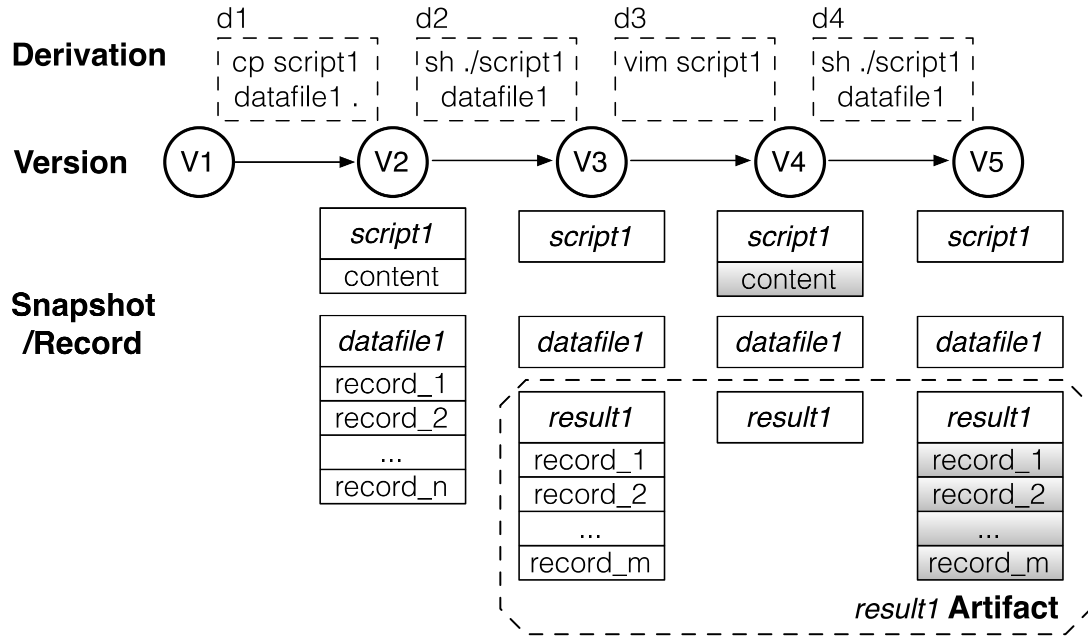


Figure 3.2: Example Workflow

3.1.2 Physical Property Graph Data Model

To actually store the conceptual data model, in addition to relational database, we map the logical data model (with the exception of **Record**) above into a property graph data model, allowing graph traversal queries and visual exploration over the stored information easily. Nodes of the property graph are of types *Version*, *Artifact*, etc., whereas the edges capture the parent-child and composition relationships. Besides *Derivation* is handled as an edge between *Versions*. The edge properties are used to store the detailed parameters and command options.

Example 2 *For the data model instance in Figure 3.2, we show the actual physical property graph in Figure 3.3. The shape of nodes are reflecting the entity types in Figure 3.2. Between artifact and snapshots, (e.g.* result1 *and* result s1*), the edge*

has a composition relationship, while between snapshots, (e.g. result s1 *and* result s2*), the parent-child lineage is stored across versions. For instance, the artifact* result1 *is unchanged in version* V4*, the snapshot parent of* result s2 *in* V5 *is the snapshot* result s1 *in* V3*.*

It is worth pointing out that providing entities across versions, the provenance query can be asked directly from the perspective of artifacts and its changes on snapshots, without knowing the versions. For instance, compare the metadata of top-3 ResultFile snapshots of a classification result artifact.

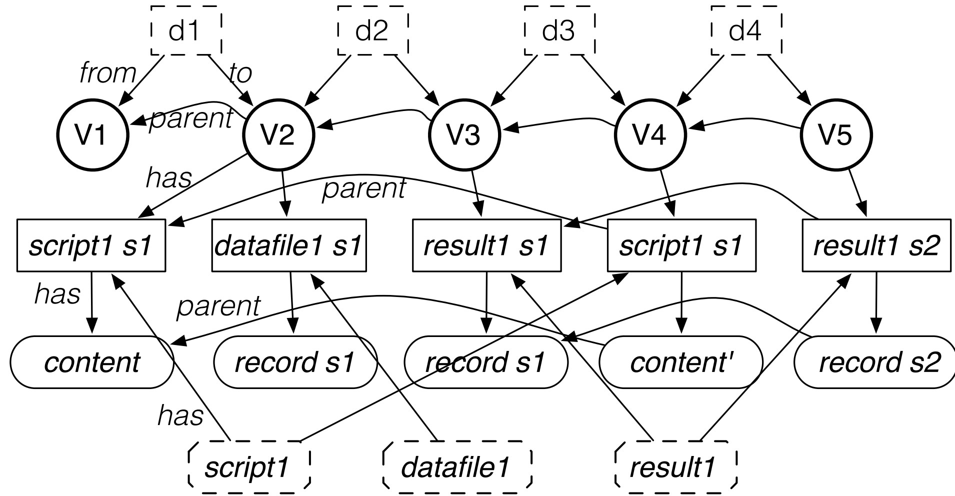


Figure 3.3: Provenance Property Graph

3.2 Provenance Ingestion

The rich provenance information need to be ingested, which is particularly challenging in a situation where the work environment is open world and very diverse. ProvDB captures provenance information or other metadata opportunisti-

cally, and features a suite of mechanisms that can capture provenance for different types of operations. Users can easily add provenance ingestion mechanisms, to both capture more types of information as well as richer information. The list of ingestion mechanisms are listed in Table 3.1

Ingestion Method	Brief Description
Shell-based Ingestion	General framework for command line work environment.
User Annotations	GUI for the users to add important information for peers.
File Views	Capture fine-grained record level provenance.
Extension Modules	Ingestors for popular modeling tool chain.

Table 3.1: ProvDB Ingestion Methods

In this section, we briefly describe the ingestion mechanisms that ProvDB currently supports, which include a general-purpose UNIX shell-based ingestion framework, ingestion of DVCS versioning information, and a mechanism called **file views** which is intended to both simplify workflow and aid in fine-grained provenance capture.

3.2.1 Shell command-based Ingestion Framework

The provenance ingestion framework is centered around the UNIX commandline shell (e.g., bash, zsh, etc). We provide a special command called `provdb ingest` that users can prefix to any other command, and that triggers provenance ingestion. Each run of the command results in creation of a new *implicit* version,

which allows us to capture the changes at a fine granularity. These implicit versions are kept separate from the explicit versions created by a user through use of `git commit`, and are not visible to the users. A collection of *ingestors* is invoked by matching the command against a set of regular expressions, registered a priori along with the ingestors. ProvDB schedules ingestor to run before/during/after execution the user command, and expects the ingestor to return a JSON property graph consisting of a set of key-value pairs denoting properties of the snapshots or derivations. An ingestor can also provide *record-level provenance* information, if it is able to generate such information.

A default ingestor handles abitrary commands by parsing them following POSIX standard (IEEE 1003.1-2001) to annotate utility, options, option arguments and operands. For example, `provdb ingest 'mkdir -p dir'` is parsed as utility *mkdir*, option *p* and operand *dir*. Concatenations of commands are decomposed and ingested separately, while a command with pipes is treated as a single command. If an external tool has been used to make any edits (e.g., a text editor), an implicit version is created next time `provdb` is run, and the derivation information is recorded as missing.

3.2.2 User Annotations

Apart from plugin framework, ProvDB GUI allows users to organize, add, and annotate properties, along with other query facilities. The user can annotate project properties, such as usage descriptions for collaborations on artifacts, or notes

to explain rationale for a particular derivation. A user can also annotate a property as parameter and add range/step to its domains, which turns a derivation into a template and enables batch run of an experiment. For example, a grid search of a template derivation on a start snapshot can be configured directly in the UI. Maintaining such user annotations (and file views discussed next) as the datasets evolve is a complicated issue in itself [91].

3.2.3 File Views

ProvDB provides a functionality called *file views* to assist dataset transformations and to ingest provenance among data files. Analogous to views in relational databases, a file view defines a virtual file as a transformation over an existing file. A file view can be defined either: (a) as a script or a sequence of commands (e.g., `sort | uniq -c`, which is equivalent to an aggregate count view), or (b) as an SQL query where the input files are treated as tables. For instance, the following query counts the rows per label that a classifier predicts wrongly comparing with ground truth.

```
provdb fileview -c -n='results.csv' -q='
  select t._c2 as label, count(*) as err_cnt
  from {testfile.csv} as t, {predfile.csv} as r
  where t._c0 = r._c0 and t._c2 != r._c2 group by t._c2'
```

The SQL feature is implemented by loading the input files into an in-memory `sqlite`

database and executing the query against it. Instead of creating a view, the same syntax can be used for creating a new file instead, saving a user from coding similar functionality.

File views serves as an example of a functionality that can help make the ad hoc process of data science more structured. Aside from making it easier to track dependencies, SQL-based file views also enable capturing record-level provenance by drawing upon techniques developed over the years for provenance in databases.

3.2.4 Extension Modules

ProvDB also supports several specialized ingestion plugins and configurations to cover important data science workflows. In particular, it has an ingestor capable of ingesting provenance information from runs of the *caffe* deep learning framework; it not only ingests the learning hyperparameters from the configuration file, but also the accuracy and loss scores by iteration from the result logging file. Providing parsers for scripts written in popular data science tools such as `Jupyter`, `scikit-learn` and `pandas` is an on-going effort, by building upon prior work [17].

Though ProvDB prototype is designed to be used in a command-line environment, ingesting provenance within other development environments such as different IDEs requires a white box approach, such as providing IDE callbacks and add features to them, which is out of the scope of the prototype. However, the conceptual and physical data models work similarly.

3.3 Query Facilities

Based on the unified data model ingested by various mechanisms ProvDB supports, we identify a set of query workloads and discuss how ProvDB implements them. As an overview, ProvDB queries facilities are as follows:

- Provenance queries about the Version/Workflow Graph and Properties part in the data model.
- Workflow queries on derivations among nodes in the version graph.
- Introspective queries, such as diffing similar artifacts and derivation pipelines in the data science processes.
- *Monitoring* queries on a given property of a series versions, for automatically detecting problems during deployment.
- Apart from general query facilities, using ProvDB extension module, domain-specific high level queries can be answered by extensions, for example, a deep learning extension (Chapter 4) allows to ask the network architectures and hyperparameters used in an experiment, while a feature engineering extension could expose queries on selected feature set.

In the rest of this section, we discuss each type of the query facilities, and show how our current prototype supports them with a case study. In the next chapter, we illustrate the design of a deep learning plugin subsystem in ProvDB.

3.3.1 Queries over Version/Workflow Graph and Properties

In a collaborative workflow, provenance queries to identify what revision and which author last modified a line in an artifact are common (e.g., `git blame`). ProvDB allows such queries at various levels (version, artifact, snapshot, record) and also allows querying the properties associated with the different entities (e.g., details of what parameters have been used, temporal orders of commands, etc). In fact, all the information exposed in the property graph can be directly queried using the Neo4j Cypher query language, which supports graph traversal queries and aggregation queries.

The latter types of queries are primarily limited by the amount of context and properties that can be automatically ingested. ProvDB currently supports ingestors for several popular frameworks, including a program analysis ingestor for *scikit-learn* which extracts the scikit-learn APIs used in a program, and a hyper-parameter and result-table ingestor for *caffe* for deep learning (the hyper-parameter ingestor extracts experiment parameter metadata from *caffe* commands and arguments, while the results-table ingestor extracts optimization errors and accuracy metrics from training logs). Availability of this information allows users to ask more meaningful queries like: what scikit-learn script files contain a specific sequence of commands; what is the learning accuracy curve of a caffe model artifact; enumerate all different parameter combinations that have been tried out for a given learning task, and so on.

Many such queries naturally result in one or more time series of values (e.g.,

properties of an artifact over time as it evolves, results of "diff" queries discussed below); ProvDB supports a uniform visual interface for plotting such time series data, and comparing different time series.

3.3.2 Reasoning about Pipelines

Similar to a workflow management system, we define a pipeline to be a sequence of derivation edges. A pipeline can be annotated by the user by browsing the workflow graph and marking the start and the end edges of the pipeline. Pipelines would also be inferred automatically by the system (e.g., via pattern mining techniques) if the ingested history is large and clean. ProvDB UI allows a user to browse and reuse pipelines present in the system. Being able to reason about pipelines has the potential to hugely simplify the lives of data scientists, by allowing them to learn from others and also helping them avoid mistakes (e.g., omission of a crucial intermediate step). Moreover, it also makes lifecycle automations possible, such as re-invoking an old pipeline on an old artifact to verify the results, or invoking a pipeline on a different snapshot with different parameters, or schedule a cron job.

3.3.3 Introspective Diffs: Shallow vs Deep "Diff" Queries

"Diff" is a first-class operator in ProvDB, and can be used for finding differences at various different levels. Specifically, given a pair of nodes (corresponding to two snapshots) in the property graph, a *shallow* diff operation, by default, focuses on the ingested properties of the two snapshots, which are likely to contain the crucial

differences in most cases. It attempts to "join" the two sets of properties as best as it can, and highlights the differences; in case of time-series properties, it also allows users to generate plots so they can more easily understand the differences. For example, for two *result table artifacts* that may represent the outputs of two different runs of the same script (e.g., model training logs), a line-by-line diff may be useless because of irrelevant and minor numerical differences; however, by plotting the two sets of results against each other, a user can more quickly spot important trends (e.g., that a specific value of parameter leads to quicker convergence). The shallow diff operator also allows differencing the contents of the two files line-by-line if so desired.

A *deep* diff compares the ancestors of the two target snapshots by tracing back their derivations to the common ancestor. It aligns the snapshots along the two paths, and shows the differences between each pair of aligned snapshots. For example, in a prediction workflow, a user may have tried out different prediction models and configurations to identify the best model; using ProvDB, she can start from two result table artifacts, and ask a *deep diff* query to compare how they are derived.

3.3.4 Continuous Monitoring or Anomaly Detection

On ingested properties of artifacts and derivations, ProvDB provides a monitoring and alerting subsystem to aid the user during the development lifecycle. We envision two main use cases for this functionality. (a) First, it can be used to detect

any major changes to the properties of an evolving dataset – e.g., a large change in the distribution of values in a dataset may be cause for taking remedial action. (b) Second, in most applications, there is usually a need to "deploy" an analysis script or a trained model against live incoming data; it is important to keep track of how well the model or the script is behaving and catch any problems as soon as possible (e.g., changing input data properties; higher error rates than expected). ProvDB supports analysis of historical data (as described above) and simple alert queries that can monitor a property of an evolving artifact.

To elaborate, a property belong to an artifact could be registered as a monitoring property. Once there is a new snapshot of the same artifact is committed, the property of the new snapshot is ingested and compared with the alert conditions, which are configured beforehand by the users. Numerical properties of a series of snapshots naturally forms a time series. We support a limited set of time series models, such as moving averages and standard derivation envelopes to be configured as adaptive alert conditions.

3.4 Case Study

Example 3 *We show the* ProvDB *Web GUI using a* `caffe` *deep learning project. In this project, 41 deep neural networks are created for a face classification task. The user tries out models by editing and training models. In Fig 3.4, an introspection query asks how different are two trained models (*model-0 *and* 9*). Using the GUI, the user filters artifacts, and diffs their result logging files. In the right side query*

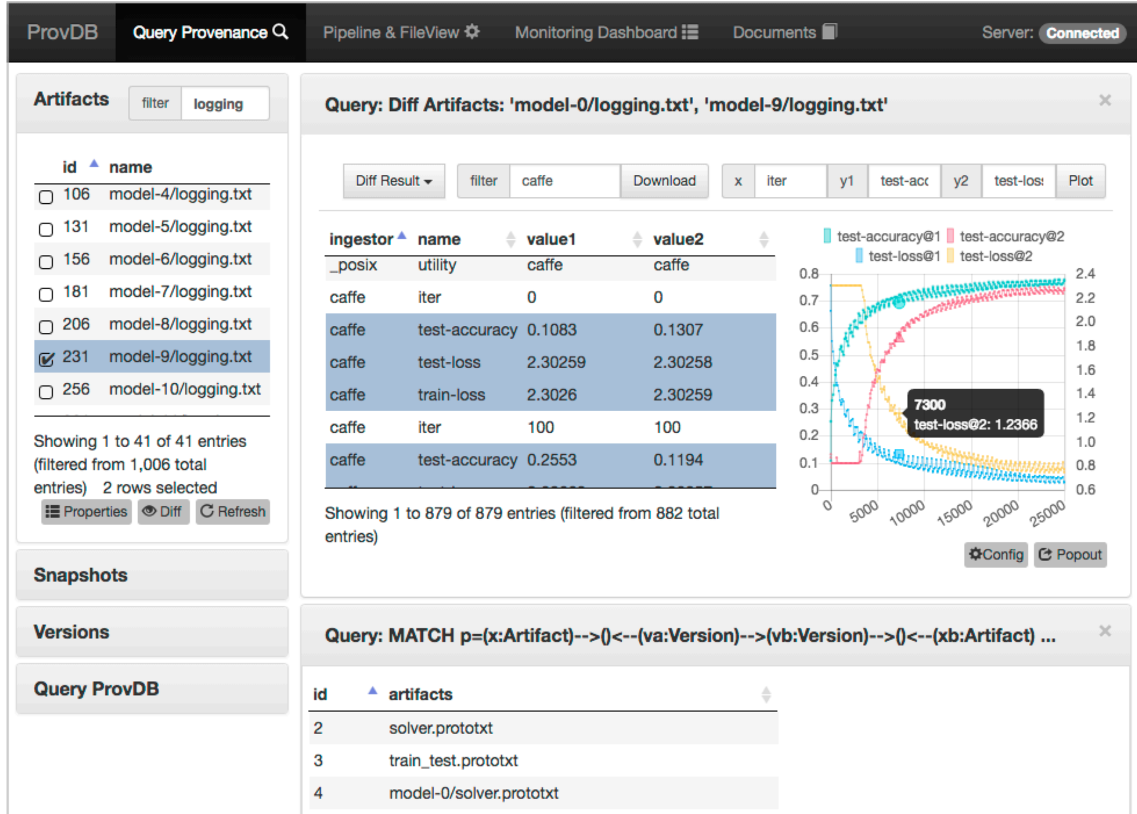


Figure 3.4: Diff Artifacts (Result logging files for two deep neural networks)

result pane, the ingested properties are diffed. The `caffe` *ingestor properties are numerical time series; using the provided charting tool, the user plots the training loss and accuracy against the iteration number. From the results, we can see that* model-9 *does not train well in the beginning, but ends up with similar accuracy. To understand why, a deep diff between the two can be issued in the GUI and complex Cypher queries can be used as well. In Figure 3.5, the query finds previous derivations and shared snapshots, which are training config files; more introspection can be done by finding changed hyperparameters.*

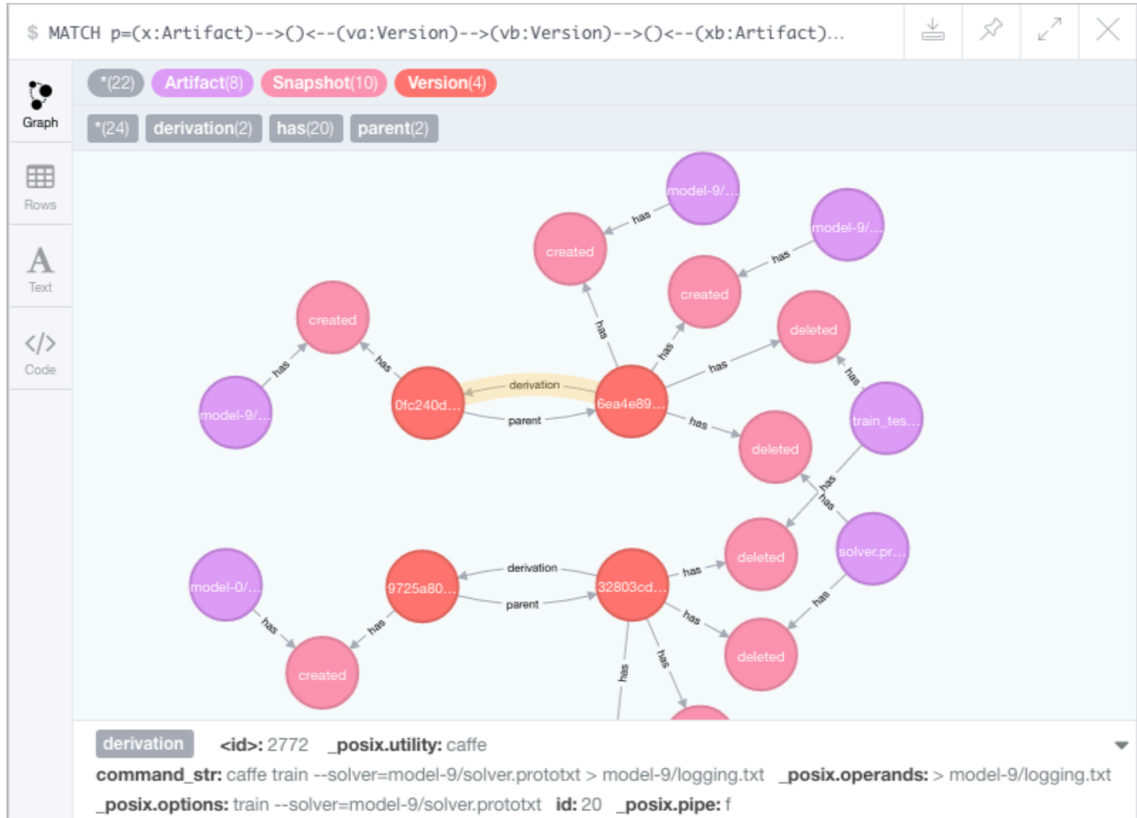


Figure 3.5: Cypher Query to Find Related Changes via Derivations

3.5 Conclusion

In this chapter, we presented ProvDB, its high level system architecture and our major design decisions for simplifying lifecycle management of ad hoc, collaborative analysis workflows that are becoming prevalent in most application domains today. We showed that a large amount of provenance and metadata information can be captured passively, and argue analyzing it in novel ways can immensely simplify the day-to-day processes undertaken by data analysts. Our ProvDB prototype uses `git` and Neo4j, which provides a variety of provenance ingestion mechanisms and the ability to query, analyze, and monitor the captured provenance informa-

tion. Our experience with using this prototype for a deep learning workflow (for a computer vision task) showed that even with limited functionality, it can simplify the bookkeeping tasks and make it easy to compare the effects of different hyperparameters and neural network structures.

Chapter 4: ModelHub: Managing Deep Learning Projects on ProvDB

Deep learning models [92], also called *deep neural networks* (DNN), have improved state-of-the-art results in many important fields, and have been the subject of much research in recent years, leading to the development of several systems for facilitating deep learning. Current systems, however, mainly focus on model building and training phases, while the issues of data management, model sharing, and lifecycle management are largely ignored. Deep learning modeling lifecycle generates a rich set of data artifacts, such as learned parameters and training logs, and comprises of several frequently conducted tasks, e.g., to understand the model behaviors and to try out new models. Dealing with such artifacts and tasks is cumbersome and largely left to the users.

In this chapter, we study the deep learning modeling practice and illustrate the system design of ModelHub, a domain-specific extension of ProvDB, to manage the rich set of modeling artifacts and their provenance over the lifecycle. We first list the challenges of deep learning lifecycle management, derive the system requirements for ModelHub, and illustrate its major components in Section 4.1, followed by introducing background on related topics in DNN modeling lifecycle in Section 4.2. We present an overview of ModelHub, and discuss the declarative interfaces in

Section 4.3. Then we describe the parameter archival store (PAS) in Section 4.4 and present an experimental evaluation in Section 4.5.

4.1 Motivation & Approach

4.1.1 DNN Modeling Lifecycle and Challenges

Compared with the traditional approach of *feature engineering* followed by *model training* [79], deep learning is an end-to-end learning approach, i.e., the features are not given by a human but are learned in an automatic manner from the input data. Moreover, the features are complex and have a hierarchy along with the network representation. This requires less domain expertise and experience from the modeler, but understanding and explaining the learned models is difficult; why even well-studied models work so well is still a mystery and under active research. Thus, when developing new models, changing the learned model (especially its network structure and *hyper-parameters*) becomes an empirical search task.

In Figure 4.1, we show a typical deep learning modeling lifecycle (we present an overview of deep neural networks in the next section). Given a prediction task, a modeler often starts from well-known models that have been successful in similar

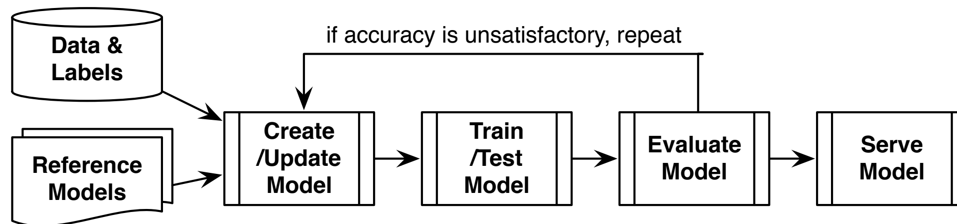


Figure 4.1: Deep Learning Modeling Lifecycle

task domains; she then specifies input training data and output loss functions, and repeatedly adjusts the DNN on operators and connections like Lego bricks, tunes model hyper-parameters, trains and evaluates the model, and repeats this loop until prediction accuracy does not improve. Due to a lack of understanding about why models work, the adjustments and tuning inside the loop are driven by heuristics, e.g., adjusting hyper-parameters that appear to have a significant impact on the learned weights, applying novel layers or tricks seen in recent empirical studies, and so on. Thus, many similar models are trained and compared, and a series of model variants needs to be explored and developed. Due to the expensive learning/training phase, each iteration of the modeling loop takes a long period of time and produces many (checkpointed) snapshots of the model. As we noted above, this is a common workflow across many other ML models as well.

Current systems (Caffe [72], Theano, Torch, TensorFlow [69], etc.) mainly focus on model building and training phases, while the issues of data management, model sharing, and lifecycle management are largely ignored. Modelers are required to write external imperative scripts, edit configurations by hand and manually maintain a manifest of model variations that have been tried out; not only are these tasks irrelevant to the modeling objective, but they are also challenging and nontrivial due to the complexity of the model as well as large footprints of the learned models. More specifically, the tasks and data artifacts in the modeling lifecycle expose several systems and data management challenges, which include:

- *Understanding & Comparing Models*: It is difficult to keep track of the many

models developed and/or understand the differences amongst them. Differences among both the metadata about the model (training sample, hyperparameters, network structure, etc.), as well as the actual learned parameters, are of interest. It is common to see a modeler write all configurations in a spreadsheet to keep track of temporary folders of input, setup scripts, snapshots and logs, which is not only a cumbersome but also an error-prone process.

- *Repetitive Adjusting of Models*: The development lifecycle itself has time-consuming repetitive sub-steps, such as adding a layer at different places to adjust a model, searching through a set of hyper-parameters for the different variations, reusing learned weights to train models, etc., which currently have to be performed manually.

- *Model Versioning*: Similar models are possibly trained and run multiple times, reusing others' weights as initialization (finetuning).Maintaining the different model versions generated over time and their relationships can help with identifying errors and concept drifts, comparing models over new inputs, and potentially reverting back to a previous model. Even for a single learned model, storing the different checkpointed snapshots can help with "warm-start" and can provide important insights into the training processes.

- *Parameter Archiving*: The storage footprint of deep learning models tends to be very large. Recent top-ranked models in the ImageNet task have billions of floating-point parameters and require hundreds of MBs to store one snapshot during training. Due to resource constraints, the modeler has to limit

the number of snapshots, even drop all snapshots of a model at the cost of retraining when needed.

- *Reasoning about Model Results:* Another key data artifact that often needs to be reasoned about is the results of running a learned model on the training or testing dataset. By comparing the results across different models, a modeler can get insights into difficult training examples or understand correlations between specific adjustments and the performance.

4.1.2 ModelHub Approach

We extend ProvDB and build the ModelHub system to address these challenges. The ModelHub system is not meant to replace popular training-focused DNN systems, but rather designed to be used with them to accelerate modeling tasks and manage the rich set of lifecycle artifacts. It consists of three key components:

1. `DLV`: a model versioning system to store, query and aid in understanding the models and their versions. It extends the ProvDB shell command ingestion mechanism by adding deep learning artifact extractors, as well as lifecycle-specific command line suites.

2. `DQL`: In addition to general ProvDB query facilities, we propose a model network adjustment and hyper-parameter tuning domain specific language to serve as an abstraction layer to help modelers focus on the creation of the models.

3. ModelHub: a hosted deep learning model sharing system to exchange `DLV` repositories and enable publishing, discovering and reusing models from others.

Comparing with other deep learning systems and provenance management systems, the key features and innovative design highlights of ModelHub are:

- We use a `git`-like VCS as a familiar UI to let the modeler manage and explore the created models in a repository, and an `SQL`-like model enumeration DSL to aid modelers in making and examining multiple model adjustments easily.

- Behind the declarative constructs, ModelHub manages different artifacts in a split back-end storage: structured data, such as network structure, training logs of a model, lineages of different model versions, output results, are stored in *a relational database*, while learned float-point parameters of a model are viewed as a set of float matrices and managed in *a read-optimized archival storage (*PAS*)*.

- Parameters dominate the storage footprint and floats are well-known at being difficult to compress. We study PAS implementation thoroughly under the context of DNN query workload and advocate a segmented approach to store the learned parameters, where the low-order bytes are stored independently of the high-order bytes. We also develop novel model evaluation schemes to use high order bytes solely and progressively uncompress less-significant chunks if needed to ensure the correctness of an inference query.

- Due to the different utility of developed models, archiving versioned models

using parameter matrix deltas exhibits a new type of dataset versioning problem which not only optimizes between storage and access tradeoff but also has model-level constraints.

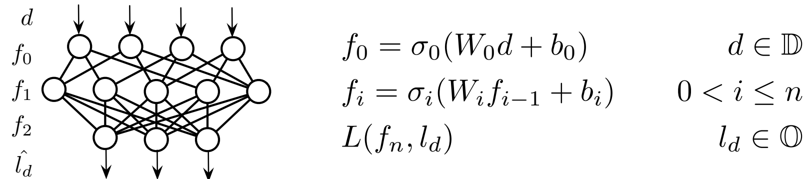
- Finally, the VCS model repository design extends naturally to a collaborative format and online system which contain rich model lineages and enables sharing, reusing, reproducing DNN models.

4.2 Background

To support our design decisions, we overview the artifacts and common task practices in DNN modeling lifecycle.

4.2.1 Deep Neural Networks

A deep learning model is a deep neural network (DNN) consisting of many layers having nonlinear activation functions that are capable of representing complex transformations between input data and desired output. Let $\mathbb{D}$ denote a data domain and $\mathbb{O}$ denote a prediction label domain (e.g., $\mathbb{D}$ may be a set of images; $\mathbb{O}$ may be the names of the set of objects we wish to recognize, i.e, *labels*). As with any prediction model, a DNN is a mapping function $f : \mathbb{D} \rightarrow \mathbb{O}$ that minimizes a certain loss function L, and is of the following form:



$$f_0 = \sigma_0(W_0 d + b_0) \qquad d \in \mathbb{D}$$
$$f_i = \sigma_i(W_i f_{i-1} + b_i) \qquad 0 < i \leq n$$
$$L(f_n, l_d) \qquad l_d \in \mathbb{O}$$

Here i denotes the layer number, (W_i, b_i) are learnable weights and bias parameters in layer i, and σ_i is an activation function that non-linearly transforms the result of the previous layer (common activation functions include sigmoid, ReLU, etc.). Given a learned model and an input d, applying $f_0, f_1, ..., f_n$ in order gives us the prediction label for that input data. In the training phase, the model parameters are learned by minimizing $L(f_n, l_d)$, typically done through iterative methods, such as *stochastic gradient descent*.

Figure 4.2 shows a classic *convolutional DNN*, LeNet. LeNet is proposed to solve a prediction task from handwritten images to digit labels $\{0 \cdots 9\}$. In the figure, a cube represents an intermediate tensor, while the dotted lines are unit transformations between tensors. More formally, a layer, $L_i : (W, H, X) \mapsto Y$, is a function which defines data transformations from tensor X to tensor Y. W are the parameters which are learned from the data, and H are the hyperparameters which are given beforehand. A layer is non-parametric if $W = \emptyset$.

In the computer vision community, the layers defining transformations are considered building blocks of a DNN model, and referred to using a conventional

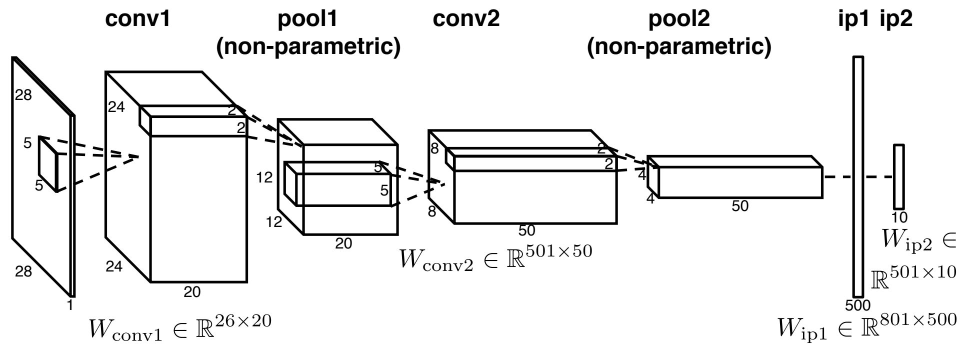


Figure 4.2: Anatomy of A DNN Model (LeNet)

name, such as *full* layer, *convolution* layer, *pool* layer, *normalization* layer, etc. The chain is often called the network *architecture*. The LeNet architecture has two convolution layers, each followed by a pool layer, and two full layers, shown with layer shapes and hyperparameters in Figure 4.2. Moreover, winning models in recent ILSVRC (ImageNet Large Scale Vision Recognition Competitions) are shown in Table 4.1, with their architectures described by a composition of common layers in regular expressions syntax for illustrating the similarities (Note the activation functions and detailed connections are omitted).

DNN models are learned from massive data based on some architecture, and modern successful computer vision DNN architectures consist of a large number of float weight parameters (*flops*) shown in Table 4.1, resulting in large storage footprints (GBs) and long training times (often weeks). Furthermore, the training process is often checkpointed and variations of models need to be explored, leading to many model copies.

Network	**Architecture** (in regular expression)	$\lvert W \rvert$ **(flops)**
LeNet [93]	$(L_{conv}L_{pool})\{2\}L_{ip}\{2\}$	4.31×10^5
AlexNet [94]	$(L_{conv}L_{pool})\{2\}(L_{conv}\{2\}L_{pool})\{2\}L_{ip}\{3\}$	6×10^7
VGG [95]	$(L_{conv}\{2\}L_{pool})\{2\}(L_{conv}\{4\}L_{pool})\{3\}L_{ip}\{3\}$	1.96×10^{10}
ResNet [96]	$(L_{conv}L_{pool})(L_{conv})\{150\}L_{pool}L_{ip}$	1.13×10^{10}

Table 4.1: Popular CNN Models for Object Recognition

4.2.2 Modeling Data Artifacts

Unlike many other prediction methods, DNN modeling results in a very large number of weight parameters, a rich set of hyperparameters, and learning measurements, which are used in unique ways in practice, resulting in a mixture of structured data, files and binary floating number artifacts:

- *Non-convexity & Hyperparameters*: A DNN model is typically non-convex, and $\{W\}$ is a local optimum of the underlying loss-minimization problem. Optimization procedure employs many tricks to reach a solution quickly [97]. The set of hyperparameters (e.g., learning rate, momentum) w.r.t. to the optimization algorithm need to be maintained.

- *Iterations & Measurements*: Models are trained iteratively and checkpointed periodically due to the long running times. A set of learning measurements are collected in various logs, including objective loss values and accuracy scores.

- *Fine-tuning & Snapshots*: Well-known models are often learned from massive real-world data (ImageNet), and require large amounts of resources to train; when prediction tasks do not vary much (e.g., animal recognition vs dog recognition), the model parameters are reused as initializations and adjusted using new data; this is often referred to as fine-tuning. On the other hand, not all snapshots can be simply deleted, as the convergence is not monotonic.

- *Provenance & Arbitrary Files*: Alternate ways to construct architectures or to set hyperparameters lead to human-in-the-loop model adjustments. Initializa-

tion, preprocessing schemes, and hand-crafted scripts are crucial provenance information to explore models and reproduce results.

4.2.3 Model Adjustment

In a modeling lifecycle for a prediction task, the *update-train-evaluate* loop is repeated in daily work, and many model variations are adjusted and trained. In general, once data and loss are determined, model adjustment can be done in two orthogonal steps: a) network architecture adjustments where layers are dropped or added and layer function templates are varied, and b) hyperparameter selections, which affect the behavior of the optimization algorithms. There is much work on search strategies to enumerate and explore both.

4.2.4 Model Sharing

Due to the good generalizability, long training times, and verbose hyperparameters required for large DNN models, there is a need to share the trained models. Jia et al. [72] built an online venue (Caffe Model Zoo) to share models. Briefly, Model Zoo is part of a github repository[1] with a markdown file edited collaboratively. To publish models, modelers add an entry with links to download trained parameters in `caffe` format. Apart from the caffe community, similar initiatives are in place for other training systems.

[1] Caffe Model Zoo: `https://github.com/BVLC/caffe/wiki/Model-Zoo`

4.3 ModelHub System Overview

We show the ModelHub architecture including the key components and their interactions in Figure 4.3. At a high level, the ModelHub functionality is divided among a local component and a remote component. The local functionality includes the integration with popular DNN systems such as `caffe`, `torch`, `tensorflow`, etc., on a local machine or a cluster. The remote functionality includes sharing of models, and their versions, among different groups of users. We primarily focus on the local functionality in this chapter.

On the local system side, `DLV` is a version control system (VCS) implemented as a command-line tool (`dlv`), that serves as an interface to interact with the rest of the local and remote components. Use of a specialized VCS instead of a general-purpose VCS such as `git` or `svn` allows us to better portray and query the internal

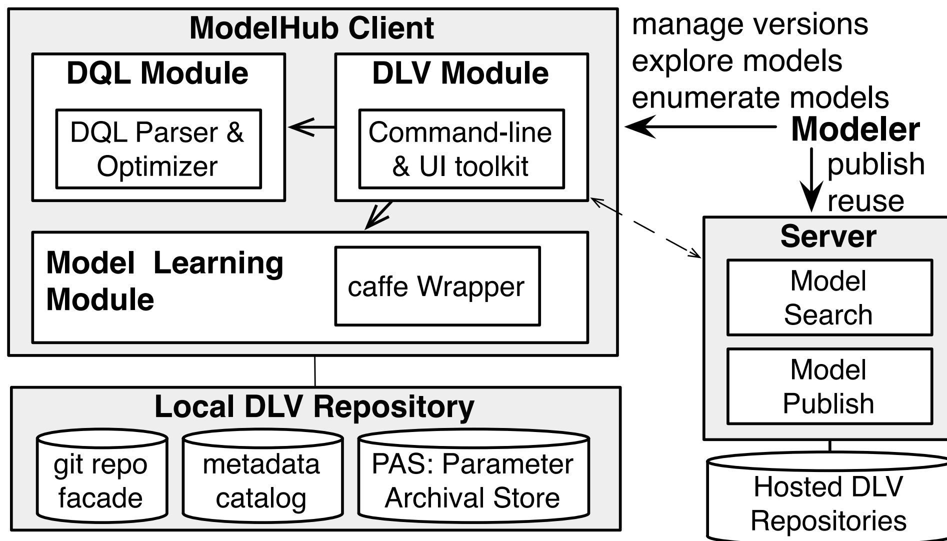


Figure 4.3: ModelHub System Architecture

structure of the artifacts generated in a modeling lifecycle, such as network definitions, training logs, binary weights, and relationships between models. The key utilities of `dlv` are listed in Table 4.2, grouped by their purpose; we explain these in further detail in Section 4.3.2. `DQL` is a DSL we propose to assist modelers in deriving new models; the `DQL` query parser and optimizer components in the figure

Type	Command	Description
model version management	`init`	Initialize a `dlv` repository.
	`add`	Add model files to be committed.
	`commit`	Commit the added files.
	`copy`	Scaffold model from an old one.
	`archive`	Archive models in the repository.
model exploration	`list`	List models and related lineages.
	`desc`	Describe a particular model.
	`diff`	Compare multiple models.
	`eval`	Evaluate a model with given data.
model enumeration	`query`	Run `DQL` clause.
remote interaction	`publish`	Publish a model to ModelHub.
	`search`	Search models in ModelHub.
	`pull`	Download from ModelHub.

Table 4.2: A list of key `dlv` utilities.

are used to support this language. The *model learning module* interacts with external deep learning tools that the modeler uses for training and testing. They are essentially wrappers on specific DNN systems that extract and reproduce modeling artifacts. Finally, the ModelHub service is a hosted toolkit to support publishing, discovering and reusing models, and serves similar role for DNN models as *github* for software development or *DataHub* for data science [2].

4.3.1 Data Model

ModelHub works with two data models: a conceptual DNN model, and a data model for the versions in a DLV repository.

4.3.1.1 DNN Model

A DNN model can be understood in different ways, as one can tell from the different model creation APIs in popular deep learning systems. In the formulation mentioned in Section 4.1, if we view a function f_i as a node and dependency relationship (f_i, f_{i-1}) as an edge, it becomes a directed acyclic graph (DAG). Depending on the granularity of the function in the DAG, either at the tensor operator level (add, multiply), or at a logical composition of those operators (convolution layer, full layer), it forms different DAGs. In ModelHub, we consider a DNN model node as a composition of unit operators (layers), often adopted by computer vision models. The main reason for this decision is that we focus on productivity improvement in the lifecycle, rather than implementation efficiencies of training and testing.

4.3.1.2 VCS Data Model

When managing DNN models in the VCS repository, a *model version* represents the contents in a single version. It consists of a network definition, a collection of weights (each of which is a value assignment for the weight parameters), a set of extracted metadata (such as hyper-parameter, accuracy and loss generated in the training phase), and a collection of files used together with the model instance (e.g., scripts, datasets). In addition, we enforce that a *model version* must be associated with a human readable name for better utility, which reflects the logical groups of a series of improvement efforts over a DNN model in practice.

In the implementation, model versions can be viewed as the following relation *model_version*(name, id, N, W, M, F), where id is part of the primary key of model versions and is auto-generated to distinguish model versions with the same name. In brief, N, W, M, F are the network definition, weight values, extracted metadata and associated files respectively. The DAG, N, is stored as two tables: Node(id, node, A), where A is a list of attributes such as layer name, and Edge(from, to). W is managed in our learned parameter storage (PAS, Section 4.4). M, the metadata, captures the provenance information of training and testing a particular model; it is extracted from training logs by the wrapper module, and includes the hyperparameters when training a model, the loss and accuracy measures at some iterations, as well as dynamic parameters in the optimization process, such as learning rate at some iterations. Finally, F is file list marked to be associated with a model version, including data files, scripts, initial configurations, and etc. Besides a set

of *model versions*, the lineage of the *model versions* are captured using a separate *parent*(base, derived, commit) relation. All of these relations are maintained/updated in a relational backend when the modeler runs the different `dlv` commands that update the repository.

4.3.2 Query Facilities

Once the DNN models and their relationships are managed in `DLV`, the modeler can interact with them easily. The query facilities we provide can be categorized into two types: a) model exploration queries and b) model enumeration queries.

4.3.2.1 Model Exploration Queries

Model exploration queries interact with the models in a repository, and are used to understand a particular model, to query lineages of the models, and to compare several models. For usability, we design it as query templates via `dlv` sub-command, similar to other VCS.

List Models & Related Lineages

By default, the query lists all versions of all models including their commit descriptions and parent versions; it also takes options, such as showing results for a particular model, or limiting the number of versions to be listed.

```
dlv list [--model_name] [--commit_msg] [--last]
```

Describe Model

`dlv desc` shows the extracted metadata from a model version, such as the net-

work definition, learnable parameters, execution footprint (memory and runtime), activations of convolutional DNNs, weight matrices, and evaluation results across iterations. Note the activation is the intermediate output of a DNN model in computer vision and often used as an important tool to understand the model. The output formats are a result of discussions with computer vision modelers to deliver tools that fit their needs. In addition to printing to console, the query supports *HTML output* for displaying the images and visualizing the weight distribution.

```
dlv desc [--model_name | --version] [--output]
```

Compare Models

`dlv diff` takes a list of model names or version ids and allows the modeler to compare the DNN models. Most of `desc` components are aligned and returned in the query result side by side.

```
dlv diff [--model_names | --versions] [--output]
```

Evaluate Model

`dlv eval` runs test phase of the managed models with an optional config specifying different data or changes in the current hyper-parameters. The main usages of exploration query are two-fold: 1) for the users to get familiar with a new model, 2) for the user to test known models on different data or settings. The query returns the accuracy and optionally the activations. It is worth pointing out that complex evaluations can be done via model enumeration queries in `DQL`.

```
dlv eval [--model_name | --versions] [--config]
```

4.3.2.2 Model Enumeration Queries

Model enumeration queries are used to explore variations of currently available models in a repository by changing network structures or tuning hyper-parameters. There are several operations that need to be done in order to derive new models: 1) Select models from the repository to improve; 2) Slice particular models to get reusable components; 3) Construct new models by mutating the existing ones; 4) Try the new models on different hyper-parameters and pick good ones to save and work with. When enumerating models, we also want to stop exploration of bad models early.

To support this rich set of requirements, we propose the `DQL` domain specific language, that can be executed using "`dlv query`". Challenges of designing the language are: a) the data model is a mix of relational and the graph data models and b) the enumeration includes hyper-parameter tuning as well as network structure mutations, which are very different operations. We describe the language and show the key operators and constructs along with a set of examples (Query 4.1∼4.4) to show how requirements are met.

```
select m1
where  m1.name like "alexnet_%" and
       m1.creation_time > "2015-11-22" and
       m1["conv[1,3,5]"].next has POOL("MAX")
```

Query 4.1: `DQL select` query to pick the models.

```
slice m2 from m1
where m1.name like "alexnet-origin%"
mutate m2.input = m1["conv1"] and
       m2.output = m1["fc7"]
```

Query 4.2: `DQL slice` query to get a sub-network.

```
construct m2 from m1
where  m1.name like "alexnet-avgv1%" and
       m1["conv*($1)"].next has POOL("AVG")
mutate m1["conv*($1)"].insert = RELU("relu$1")
```

Query 4.3: `DQL construct` query to derive more models on existing ones.

```
evaluate m
from "query3"
with config = "path to config"
vary config.base_lr in [0.1, 0.01, 0.001] and
     config.net["conv*"].lr auto and
     config.input_data in ["path1", "path2"]
keep top(5, m["loss"], 100)
```

Query 4.4: `DQL evaluate` query to enumerate models with different network architectures, search hyper-parameters, and eliminate models.

Key Operators

We adopt the standard SQL syntax to interact with the repository. `DQL` views the repository as a single model version table. A model version instance is a DAG, which can be viewed as object types in modern SQL conventions. In `DQL`, attributes can be referenced using attribute names (e.g. `m1.name`, `m1.creation_time`, `m2.input`, `m2.output`), while navigating the internal structures of the DAG, i.e. the Node and Edge EDB, we provide a regexp style *selector operator* on a model version to access individual DNN nodes, e.g. `m1["conv[1,3,5]"]` in Query 4.1 filters the nodes in `m1`. Once the selector operator returns a set of nodes, `prev` and `next` attributes of the node allow 1-hop traversal in the DAG. Note that `POOL("MAX")` is one of the standard built-in node templates for condition clauses. Using *SPJ* operators with object type *attribute access* and the *selector operator*, we allow relational queries to be mixed with graph traversal conditions.

To retrieve reusable components in a DAG, and mutate it to get new models, we provide **slice**, **construct** and **mutate** operators. **Slice** originates in programming analysis research; given a start and an end node, it returns a subgraph including all paths from the start to the end and the connections which are needed to produce the output. **Construct** can be found in graph query languages such as SPARQL to create new graphs. We allow **construct** to derive new DAGs by using selected nodes to *insert* nodes by splitting an outgoing edge or to *delete* an outgoing edge connecting to another node. **Mutate** limits the places where *insert* and *delete* can occur. For example, Query 4.2 and 4.3 generate reusable subgraphs and new graphs. Query 4.2 slices a sub-network from matching models between convolution

layer 'conv1' and full layer 'fc7', while Query 4.3 derives new models by appending a ReLU layer after all convolution layers followed by an average pool. All queries can be nested.

Finally, **evaluate** can be used to try out new models, with potential for early out if expectations are not reached. We separate the network enumeration component from the hyper-parameter turning component; while network enumeration can be nested in the *from* clause, we introduce a *with operator* to take an instance of a tuning config template, and a *vary operator* to express the combination of activated multi-dimensional hyper-parameters and search strategies. *auto* is keyword implemented using default search strategies (currently grid search). To stop early and let the user control the stopping logic, we introduce a *keep operator* to take a rule consisting of stopping condition templates, such as top-k of the evaluated models, or accuracy threshold. Query 4.4 evaluates the models constructed and tries combinations of at least three different hyper-parameters, and keeps the top 5 models w.r.t. the loss after 100 iterations.

4.3.3 ModelHub Implementation

On the local side, the implementation of ModelHub maintains the data model in multiple back-ends and utilizes `git` to manage the arbitrary file diffs. Various queries are decomposed and sent to different backends and chained accordingly. On the other hand, as the model repository is standalone, we host the repositories as a whole in a ModelHub service. The modeler can use the `dlv publish` to push

the repository for archiving, collaborating or sharing, and use `dlv search` and `dlv pull` to discover and reuse remote models. We envision such a form of collaboration can facilitate a learning environment, as all versions in the lifecycle are accessible and understandable with ease. The online video[2] highlights the interactions with the prototype and illustrate the described concepts in this section.

4.4 Parameter archival storage (PAS)

Modeling lifecycle for DNNs, and machine learning models in general, is centered around the learned parameters, whose storage footprint can be very large. The goal of PAS is to maintain a large number of learned models as compactly as possible, without compromising on the query performance. Before introducing our design, we first discuss the queries of interest, and some key properties of the model artifacts. We then describe different options to store a single *float matrix*, and to construct *deltas (differences)* between two matrices. We then formulate the optimal version graph storage problem, discuss how it differs from the prior work, and present algorithms for solving it. Finally, we develop a novel approximate model evaluation technique, suitable for the segmented storage technique that PAS uses.

4.4.1 Weight Parameters & Query Types of Interest

We illustrate the key weight parameter artifacts and the relationships among them in Figure 4.4, and also explain some of the notations used in this section.

[2] `https://youtu.be/noEXdahlj_4`

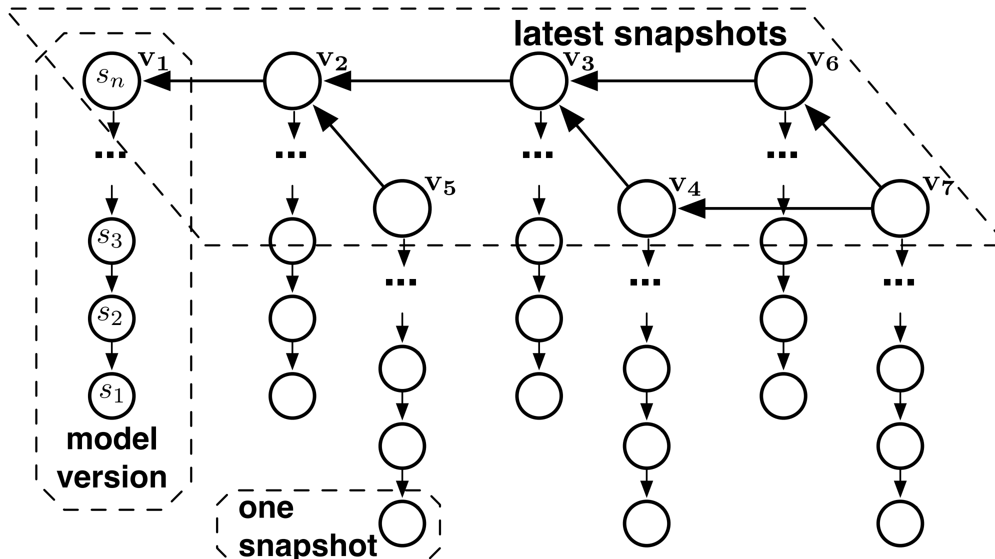


Figure 4.4: Relationships of Model Versions and Snapshots

At a high level, the predecessor-successor relationships between all the developed models is captured as a **version graph**. These relationships are user-specified and conceptual in nature, and the interpretation is left to the user (i.e., an edge $v_i \rightarrow v_j$ indicates that v_j was an updated version of the model that the user checked in after v_i, but the nature of this update is irrelevant for storage purposes). A model version v_i itself consists of a series of snapshots, $s_1, ..., s_n$, which represent checkpoints during the training process (most systems will take such snapshots due to the long running times of the iterations). We refer the last or the best checkpointed snapshot s_n as the **latest snapshot** of v_i, and denote it by s_{v_i}.

One snapshot, in turn, consists of intermediate data X and trained parameters W (e.g., in Figure 4.2, the model has 431080 parameters for W, and $19694 \cdot b$ dimensions for X, where b is the minibatch size). Since X is useful only if training needs to be resumed, only W is stored in PAS. Outside of a few rare exceptions, W

can always be viewed as a collection of float matrices, $\mathbb{R}^{m\times n}, m \geq 1, n \geq 1$, which encode the weights on the edges from outputs of the neurons in one layer to the inputs of the neurons in the next layer. Thus, we treat a *float matrix* as a first class data type in PAS[3].

The retrieval queries of interest are dictated by the operations that are done on these stored models, which include: (a) testing a model, (b) reusing weights to fine-tune other models, (c) comparing parameters of different models, (d) comparing the results of different models on a dataset, and (e) model exploration queries (Section 4.3.2). Most of these operations require execution of **group retrieval** queries, where all the weight matrices in a specific snapshot need to be retrieved. This is different from range queries seen in array databases (e.g., SciDB), and also have unique characteristics that influence the storage and retrieval algorithms.

- *Similarity among Fine-tuned Models*: Although non-convexity of the training algorithm and differences in network architectures across models lead to non-correlated parameters, the widely-used fine-tuning practices (Section 4.2) generate model versions with similar parameters, resulting in efficient delta encoding schemes.

- *Co-usage constraints:* Prior work on versioning and retrieval [1] has focused on retrieving a single artifact stored in its entirety. However, we would like to store the different matrices in a snapshot independently of each other, but

[3] We do not make a distinction about the bias weight; the typical linear transformation $W'x+b$ is treated as $W \cdot (x, 1) = (W', b)^T \cdot (x, 1)$.

we must retrieve them together. These co-usage constraints make the prior algorithms inapplicable as we discuss later.

- *Low Precision Tolerance*: DNNs are well-known for their tolerance to using low-precision floating point numbers, both during training and evaluation. Further, many types of queries (e.g., visualization and comparisons) do not require retrieving the full-precision weights.

- *Unbalanced Access Frequencies*: Not all snapshots are used frequently. The latest snapshots with the best testing accuracy are used in most of the cases. The checkpointed snapshots have limited usages, including debugging and comparisons.

4.4.2 Parameters As Segmented Float Matrices

4.4.2.1 Float Data Type Schemes

Although binary (1/-1) or ternary (1/0/-1) matrices are sometimes used in DNNs, in general PAS handles real number weights. Due to different usages of snapshots, PAS offers a handful of float representations to let the user trade-off storage efficiency with lossyness using `dlv`. In Table 4.3, we list the schemes supported in PAS:

1. *Float Point*: DNNs are typically trained with single precision (32 bit) floats. This scheme uses the standard IEEE 754 floating point encoding to store the weights with sign, exponent, and mantissa bits. IEEE half-precision proposal

(16 bits) and tensorflow truncated 16bits [69] are supported as well and can be used if desired.

2. *Fixed Point*: Fixed point encoding has a global exponent per matrix, and each float number only has sign and mantissa using all k bits. This scheme is a lossy scheme as tail positions are dropped, and a maximum of 2^k different values can be expressed. The entropy of the matrix also drops considerably, aiding in compression.

3. *Quantization*: Similarly, PAS supports quantization using k bits, $k \leq 8$, where 2^k possible values are allowed. The quantization can be done in random manner or uniform manner by analyzing the distribution, and a coding table is used to maintain the integer codes stored in the matrices in PAS. This is most useful for snapshots whose weights are primarily used for fine-tuning or initialization.

Scheme	**Param. Bits**	**Compress**	**Lossyness**	**Usage**
Float Point	64/32/16	Fair	Lossless	Latest
Fixed Point	32/16/8	Good	Good	Latest
Quantization	8/k	Excellent	Poor	Other

Table 4.3: Float Representation Scheme Trade-offs

The float point schemes present here are not new, and are used in DNN systems in practice [98–100]. As a lifecycle management tool, PAS lets experienced users

select schemes rather than deleting snapshots due to resource constraints. Our evaluation shows storage/accuracy tradeoffs of these schemes.

4.4.2.2 Bytewise Segmentation for Float Matrices

One challenge for PAS is the high entropy of float numbers in the float arithmetic representations, which leads to them being very hard to compress. Compression ratio shown in related work for scientific float point datasets, e.g., simulations, is very low. The state of art compression schemes do not work well for DNN parameters either. By exploiting DNN low-precision tolerance, we adopt bytewise decomposition from prior work [101, 102] and extend it to our context to store the float matrices. The basic idea is to separate the high-order and low-order mantissa bits, and so a float matrix is stored in multiple chunks; the first chunk consists of 8 high-order bits, and the rest are segmented one byte per chunk. One major advantage is the high-order bits have low entropy, and standard compression schemes (e.g., *zlib*) are effective for them.

Apart from the simplicity of the approach, the key benefits of segmented approach are two-fold: (a) it allows offloading low-order bytes to remote storage, (b) PAS queries can read high-order bytes only, in exchange for tolerating small errors. Comparison and exploration queries (`dlv desc`, `dlv diff`) can easily tolerate such errors and, as we show later in the chapter, `dlv eval` queries can also be made tolerant to these errors.

4.4.2.3 Delta Encoding Across Snapshots

We observed that, due to the non-convexity in training, even re-training the same model with slightly different initializations results in very different parameters. However, the parameters from checkpoint snapshots for the same or similar models tend to be close to each other. Furthermore, across model versions, fine-tuned models generated using fixed initializations from another model often have similar parameters. The observations naturally suggest use of *delta encoding* between checkpointed snapshots in one model version and latest snapshots across multiple model versions; i.e., instead of storing all matrices in entirety, we can store some in their entirety and others as differences from those. Two possible delta functions (denoted $\ominus$) are *arithmetic subtraction* and *bitwise XOR*. We find the compression footprints when applying the diff $\ominus$ in different directions are similar. We study the delta operators on real models in Section 4.5.

4.4.3 Optimal Parameter Archival Storage

Given the above background, we next address the question of how to best store a collection of model versions, so that the total storage footprint occupied by the large segmented float matrices is minimized while the retrieval performance is not compromised. This recreation/storage tradeoff sits at the core of any version control system. In recent work [1], the authors study six variants of this problem, and show the NP-hardness of most of those variations. However, their techniques cannot be directly applied in PAS, primarily because their approach is not able to

handle the *group retrieval (co-usage)* constraints.

We first introduce the necessary notation, discuss the differences from prior work, and present the new techniques we developed for PAS. In Figure 4.4, a *model version* $v \in V$ consists of time-ordered checkpointed *snapshots*, $S_v = s_1, ..., s_n$. Each *snapshot*, s_i consists of a named list of float matrices $M_{v,i} = \{m_k\}$ representing the learned parameters. All matrices in a repository, $\mathcal{M} = \bigcup_{v \in V} \bigcup_{s_i \in S_v} M_{v,i}$, are the parameter artifacts to archive. Each matrix $m \in \mathcal{M}$ is either stored directly, or is recovered through another matrix $m' \in \mathcal{M}$ via a delta operator $\ominus$, i.e. $m = m' \ominus d$, where d is the delta computed using one of the techniques discussed above. In the latter case, the matrix d is stored instead of m. To unify the two cases, we introduce a empty matrix ν_0, and define $\forall \ominus \forall m \in \mathcal{M}, m \ominus \nu_0 = m$.

Definition 1 (Matrix Storage Graph) *Given a repository of model versions V, let ν_0 be an empty matrix, and $\mathcal{V} = \mathcal{M} \cup \{\nu_0\}$ be the set of all parameter matrices. We denote by $\mathcal{E} = \{m_i \ominus m_j\} \cup \{m_i \ominus \nu_0\}$ the available deltas between all pairs of matrices. Abusing notation somewhat, we also treat $\mathcal{E}$ as the set of all* edges *in a graph where $\mathcal{V}$ are the vertices. Finally, let $\mathcal{G}_V(\mathcal{V}, \mathcal{E}, c_s, c_r)$ denote the* matrix storage graph *of V, where edge weights $c_s, c_r : \mathcal{E} \mapsto \mathbb{R}^+$ are storage cost and recreation cost of an edge respectively.*

Definition 2 (Matrix Storage Plan) *Any connected subgraph of $\mathcal{G}_V(\mathcal{V}, \mathcal{E})$ is called a matrix storage plan for V, and denoted by $\mathcal{P}_V(\mathcal{V}_P, \mathcal{E}_P)$, where $\mathcal{V}_P = \mathcal{V}$ and $\mathcal{E}_P \subseteq \mathcal{E}$.*

Example 4 *In Figure 4.5, we show a matrix storage graph for a repository with two snapshots, $s_1 = \{m_1, m_2\}$ and $s_2 = \{m_3, m_4, m_5\}$. The weights associated with*

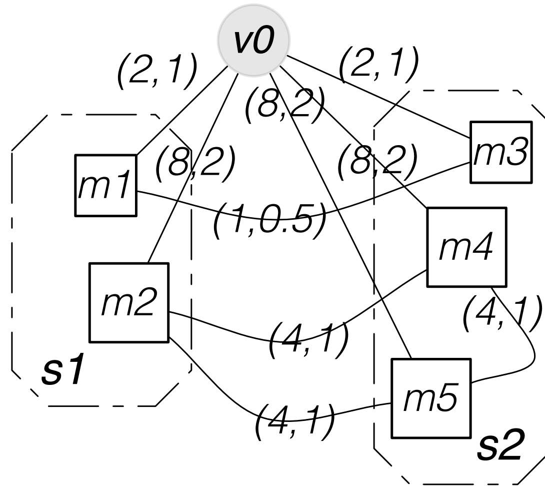


Figure 4.5: A Matrix Storage Graph Example

an edge $e = (\nu_0, m_i)$ *reflect the cost of materializing the matrix* m_i *and retrieving it directly. On the other hand, for an edge between two matrices, e.g.,* $e = (m_2, m_5)$*, the weights denote the storage cost of the corresponding delta and the recreation cost of applying that delta. In Figure 4.6 and 4.7, two matrix storage plans are shown.*

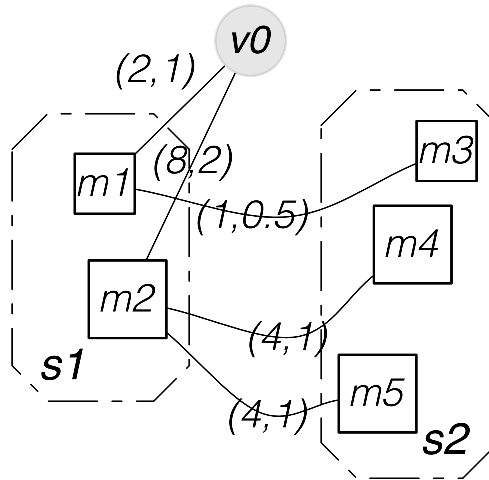


Figure 4.6: Optimal Matrix Storage Plan without Constraints

For a matrix storage plan $\mathcal{P}_V(\mathcal{V}_P, \mathcal{E}_P)$, PAS stores all its edges and is able to

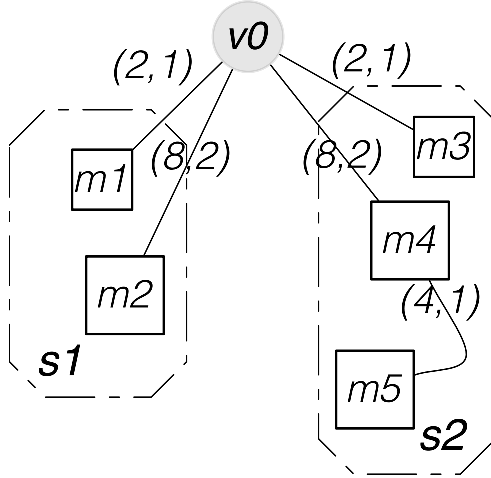


Figure 4.7: Optimal Matrix Storage Plan Constrained by ${\mathcal{C}_r}^{\psi_i}(s_1) \leq 3 \wedge {\mathcal{C}_r}^{\psi_i}(s_2) \leq 6$

recreate any matrix m_i following a path starting from ν_0. The *total storage cost* of $\mathcal{P}_V$, denoted as $\mathcal{C}_s(\mathcal{P}_V)$, is simply the sum of edge storage costs, i.e.

$$\mathcal{C}_s(\mathcal{P}_V) = \sum_{e \in \mathcal{E}_P} c_s(e)$$

Computation of the *average snapshot recreation cost* is more involved and depends on the retreival scheme used:

- *Independent* scheme recreates each matrix m_i one by one by following the shortest path (Υ_{ν_0, m_i}) to m_i from ν_0. In that case, the recreation cost is simply computed by summing the recreation costs for all the edges along the shortest path.

- *Parallel* scheme accesses all matrices of a snapshot in parallel (using multiple threads); the longest shortest path from ν_0 defines the recreation cost for the snapshot.

- *Reusable* scheme considers caching deltas on the way, i.e., if paths from ν_0

to two different matrices overlap, then the shared computation is only done once. In that case, we need to construct the lowest-cost *Steiner tree* ($\mathcal{T}_{\mathcal{P}_V, s_i}$) involving ν_0 and the matrices in the snapshot. However, because multiple large matrices need to be kept in memory simultaneously, the memory consumption of this scheme can be large.

Retrieval Scheme	*Recreation* $\mathcal{C}_r{}^{\psi}(\mathcal{P}_V, s_i)$	*Solution of Prob.1*
Independent (ψ_i)	$\sum_{m_j \in s_i} \sum_{e_k \in \Upsilon_{\nu_0, m_j}} c_r(e_k)$	Spanning tree
Parallel (ψ_p)	$\max_{m_j \in s_i} \{\sum_{e_k \in \Upsilon_{\nu_0, m_j}} c_r(e_k)\}$	Spanning tree
Reusable (ψ_r)	$\sum_{e_k \in \mathcal{T}_{\mathcal{P}_V, s_i}} c_r(e_k)$	Subgraph

Table 4.4: Recreation Cost of a Snapshot s_i $\mathcal{C}_r(\mathcal{P}_V, s_i)$ in a plan $\mathcal{P}_V$

PAS can be configured to use any of these options during the actual query execution. However, solving the storage optimization problem with *Reusable* scheme is nearly impossible; since the Steiner tree problem is NP-Hard, just computing the cost of a solution becomes intractable making it hard to even compare two different storage solutions. Hence, during the storage optimization process, PAS can only support *Independent* or *Parallel* schemes.

In the example above, the edges are shown as being undirected indicating that the deltas are symmetric. In general, we allow for directed deltas to handle asymmetric delta functions, and also for multiple directed edges between the same two matrices. The latter can be used to capture different options for storing the delta; e.g., we may have one edge corresponding to a remote storage option, where

the storage cost is lower and the recreation cost is higher; whereas another edge (between the same two matrices) may correspond to a local SSD storage option, where the storage cost is the highest and the recreation cost is the lowest. Our algorithms can thus automatically choose the appropriate storage option for different deltas.

Similarly, PAS is able to make decisions at the level of byte segments of float matrices, by treating them as separate matrices that need to be retrieved together in some cases, and not in other cases. This, combined with the ability to incorporate different storage options, is a powerful generalization that allows PAS to make decisions at a very fine granularity.

Given this notation, we can now state the problem formally. Since there are multiple optimization metrics, we assume that constraints on the retrieval costs are provided and ask to minimize the storage.

Problem 1 (Optimal Parameter Archival Storage) *Given a matrix storage graph $\mathcal{G}_V(\mathcal{V}, \mathcal{E}, c_s, c_r)$, let θ_i be the* snapshot recreation cost budget *for each $s_i \in S$. Under a retrieval scheme ψ, find a matrix storage plan ${\mathcal{P}_V}^*$ that minimizes the* total storage cost*, while satisfying recreation constraints, i.e.:*

$$\underset{\mathcal{P}_V}{\text{minimize}} \quad \mathcal{C}_s(\mathcal{P}_V); \quad s.t. \quad \forall s_i \in S, {\mathcal{C}_r}^{\psi}(\mathcal{P}_V, s_i) \leq \theta_i$$

Example 5 *In Figure 4.6, without any recreation constraints, we show the best storage plan, which is the minimum spanning tree based on c_s of the matrix storage graph, $\mathcal{C}_s(\mathcal{P}_V) = 19$. Under independent scheme ψ_i, ${\mathcal{C}_r}^{\psi_i}(\mathcal{P}_V, s_1) = 3$ and ${\mathcal{C}_r}^{\psi_i}(\mathcal{P}_V, s_2) = 7.5$. In Figure 4.7, after adding two constraints $\theta_1 = 3$ and $\theta_2 = 6$,*

we shows an optimal storage plan ${\mathcal{P}_V}^*$ *satisfying all constraints. The storage cost increases,* $\mathcal{C}_s({\mathcal{P}_V}^*) = 24$*, while* ${\mathcal{C}_r}^{\psi_i}({\mathcal{P}_V}^*, s_1) = 3$ *and* ${\mathcal{C}_r}^{\psi_i}({\mathcal{P}_V}^*, s_2) = 6$*.*

Although this problem variation might look similar to the ones considered in recent work [1], none of the variations studied there can handle the co-usage constraints (i.e., the constraints on simultaneously retrieving a group of versioned data artifacts). One way to enforce such constraints is to treat the entire snapshot as a single data artifact that is stored together; however, that may force us to use an overall suboptimal solution because we would not be able to choose the most appropriate delta at the level of individual matrices. Another option would be to sub-divide the retrieval budget for a snapshot into constraints on individual matrices in the snapshot. As our experiments show, that can lead to significantly higher storage utilization. Thus the formulation above is a strict generalization of the formulations considered in that prior work.

Theorem 1 *Optimal Parameter Archival Storage Problem is NP-hard for all retrieval schemes in Table 4.4.*

Proof 1 *We reduce Prob.5 in [1] to the independent scheme* ψ_i*, and Prob.6 to the parallel scheme* ψ_p *in [1], by mapping each datasets as vertices in storage graph, and introducing a snapshot holding all matrices with recreation bound* Θ_g*. For reuse scheme* ψ_r*, it is at least as hard as weighted set cover problem if reducing a set to an edge* e *with storage cost* $c_s(e)$ *as weight, an item to an vertex in* $\mathcal{G}_V(\mathcal{V}, \mathcal{E})$*, and set recreation budget* $\Theta_g = \infty$*.*

Lemma 1 *The optimal solution for Problem 1 is a spanning tree when retrieval scheme ψ is* independent *or* parallel.

Proof 2 *Suppose we have a non-tree solution $\mathcal{P}_V$ satisfying the constraints, and also minimize the objective. Note that parallel and independent schemes are based on shortest path $\Upsilon_{\nu_0,m}$ in $\mathcal{P}_V$ from ν_0 to each matrix m, so the union of each shortest path forms a shortest path tree. If we remove edges which are not in the shortest path tree from the plan to ${\mathcal{P}_V}'$, it results in a lower objective $\mathcal{C}_s({\mathcal{P}_V}')$, but still satisfying all recreation constraints, which leads to a contradiction.*

Note the above lemma is not true for the *reusable* scheme (ψ_r); snapshot Steiner trees satisfying different recreation constraints may share intermediate nodes resulting in a subgraph solution.

Lemma 1 shows ${\mathcal{P}_V}^*$ is a spanning tree and connects our problem to a class of constrained minimum spanning tree problems.

4.4.3.1 Constrained Spanning Tree Problem

In Problem 1, storage cost minimization while ignoring the recreation constraints leads to a minimum spanning tree (MST) of the storage matrix; whereas the snapshot recreation constraints are best satisfied by using a shortest path tree (SPT). These problems are often referred to as constrained spanning tree problems [103] or shallow-light tree constructions [104], which have been studied in areas other than dataset versioning, such as VLSI designs. Khuller et al. [65] propose an algorithm called LAST to construct such a "balanced" spanning tree in an undi-

rected graph G. LAST starts with a minimum spanning tree of the provided graph, traverses it in a DFS manner, and adjusts the tree by changing parents to ensure the path length in constructed solution is within $(1+\epsilon)$ times of shortest path in G, i.e. $\mathcal{C}_r(T, v_i) \leq (1+\epsilon)\mathcal{C}_r(\Upsilon_{\nu_0, v_i}, v_i)$, while total storage cost is within $(1+\frac{2}{\epsilon})$ times of MST. In our problem, the co-usage constraints of matrices in each snapshot form hyperedges over the matrix storage graph making the problem more difficult.

In the rest of the discussion, we adapt meta-heuristics for constrained MST problems to develop two algorithms: the first one (PAS-MT) is based on an iterative refinement scheme, where we start from an MST and then adjust it to satisfy constraints; the second one is a priority-based tree construction algorithm (PAS-PT), which adds nodes one by one and encodes heuristic in the priority function. Both algorithms aim to solve the parallel and independent schemes, and can also find feasible solution for reusable scheme. Due to large memory footprints of intermediate matrices, we do not discuss algorithm for improving reusable scheme solutions.

4.4.3.2 PAS-MT

The algorithm starts with T as the MST of $\mathcal{G}_V(\mathcal{V}, \mathcal{E})$, and iteratively adjusts T to satisfy the broken snapshot recreation constraints, $U = \{s_i | \mathcal{C}_r(T, s_i) > \theta_i\}$, by swapping one edge at a time. We denote p_i as the parent of v_i, $(p_i, v_i) \in T$ and $p_0 = \phi$, and successors of v_i in T as $\mathcal{D}_i$. A <u>*swap operation*</u> on (p_i, v_i) to edge $(v_s, v_i) \in \mathcal{E} - T$ changes parent of v_i to v_s in T.

Lemma 2 *A swap operation on v_i changes storage cost of $\mathcal{C}_s(T)$ by $c_s(p_i, v_i) -$*

$c_s(v_s, v_i)$, *and changes recreation costs of* v_i *and its successors* $\mathcal{D}_i$ *by:* $\mathcal{C}_r(T, v_i) - \mathcal{C}_r(T, v_s) - c_r(v_s, v_i)$.

The proof can be derived from definition of C_s and C_r by inspection. When selecting edges in $\mathcal{E} - T$, we choose the one which has the largest marginal gain for unsatisfied constraints:

Eq. 4.1 sums the gain of recreation cost changes among all matrices in the same snapshot s_i (for the independent scheme), while Eq. 4.2 uses the max change instead (for the *parallel* scheme). The actual formula used is somewhat more complex, and

Input: $\mathcal{G}_V(\mathcal{V}, \mathcal{E}, c_s, c_r)$, snapshots S, recreation cost $\{\theta_i \geq 0 \mid s_i \in S\}$.

Output: A spanning tree T satisfying constraints $\{\mathcal{C}_r(T, s_i) \leq \theta_i\}$

1: let T = MST of $\mathcal{G}_V(\mathcal{V}, \mathcal{E})$;
2: **while** unsatisfied constraints $U = \{s_i \mid \mathcal{C}_r(T, s_i) > \theta_i\} \neq \emptyset$ **do**
3:     **for** each edge $e_{si} = (v_s, v_i) \in \mathcal{E} - T$ **do**
4:         calculate $gain(e_{si})$ with Eq. 4.1 (Eq. 4.2 for scheme ψ_p)
5:     **end for**
6:     find $e'_{si} = \max\{e_{si} \mid gain(e_{si})\}$
7:     **break** if $gain(e'_{si}) \leq 0$
8:     *swap* (p_i, v_i) with e'_{si}: $T = (T - \{(p_i, v_i)\}) \cup \{e'_{si}\}$
9: **end while**
10: **return** T unless $U \neq \emptyset$

Algorithm 1: PAS-MT

handles negative denominators.

$$\psi_i : \max_{(v_s,v_i)\in\mathcal{E}-T}\{\frac{\sum_{s_k\in U}\sum_{v_j\in s_k\cap\mathcal{D}_i}(\mathcal{C}_r(T,v_i)-\mathcal{C}_r(T,v_s)-c_r(v_s,v_i))}{c_s(v_s,v_i)-c_s(p_i,v_i)}\} \tag{4.1}$$

$$\psi_p : \max_{(v_s,v_i)\in\mathcal{E}-T}\{\frac{\sum_{s_k\in U}(\mathcal{C}_r(T,v_i)-\mathcal{C}_r(T,v_s)-c_r(v_s,v_i))}{c_s(v_s,v_i)-c_s(p_i,v_i)}\} \tag{4.2}$$

The algorithm iteratively swaps edges and stops if all recreation constraints are satisfied or no edge returns a positive gain. A single step examines $|\mathcal{E} - T|$ edges and $|U|$ unsatisfied constraints, and there are at most $|\mathcal{E}|$ steps. Thus the complexity is bounded by $O(|\mathcal{E}|^2|S|)$. The pseudo-code is shown in Algorithm 1.

4.4.3.3 PAS-PT

This algorithm constructs a solution by "growing" a tree starting with an empty tree. In the matrix storage graph $\mathcal{G}_V(M, D, c_s, c_r)$, due to the co-usage constraint, previous tree growth algorithms [1,65] do not work any more, as adding one node at a time cannot determine whether a group constraint is satisfied. Instead, PAS-PT examines the edges in $\mathcal{G}_V(\mathcal{V}, \mathcal{E})$ in the increasing order by the storage cost c_s; a priority queue is used to maintain all the candidate edges and is populated with all the edges from v_0 in the beginning. At any point, the edges in Q are the ones that connect a vertex T, to a vertex outside T. Using an edge $e_{ij} = (v_i, v_j)$ (s.t., $v_i \in V_T \land v_j \in \mathcal{V} - V_T$) popped from Q, the algorithm tries to add v_j to T with minimum storage increment $c_s(e_{ij})$. Before adding v_j, it examines whether the constraints of affected groups s_a (s.t., $v_j \in s_a$) are satisfied using actual and

estimated recreation costs for vertices $\{v_k \in s_a\}$ in V_T and $\mathcal{V} - V_T$ respectively; if $v_k \in V_T$, actual recreation cost $\mathcal{C}_r(T, v_k)$ is used, otherwise the lower bound of it, i.e. $c_r(\nu_0, v_k)$ is used as an estimation. We refer the estimation for s_a as $\hat{\mathcal{C}}_r(T, s_a)$.

Once an edge e_{ij} is added to T, the inner edges $\mathcal{I}_T^j = \{(v_k, v_j) | v_k \in V_T\}$ of newly added v_j are dequeued from Q, while the outer edges $\mathcal{O}_T^j = \{(v_j, v_k) \mid v_k \in \mathcal{V} - V_T\}$ are enqueued. If the storage cost of existing vertices in T can be improved (i.e. $\mathcal{C}_s(T, v_k) > c_s(v_k, v_j)$), and recreation cost is not more (i.e. $\mathcal{C}_r(T, v_k) \geq \mathcal{C}_r(T, v_j) + c_r(v_k, v_j)$), then the parent p_k of v_k in T is replaced to v_j via the swap operation, decreasing the storage but not increasing affected group recreation cost.

The algorithm stops if Q is empty and T is a spanning tree. In the case when Q is empty but $V_T \subset \mathcal{V}$, an *adjustment operation* on T to increase storage cost and satisfy the group recreation constraints is performed. For each $v_u \in \mathcal{V} - V_T$, we append it to ν_0, then in each unsatisfied group s_i that v_u belongs to, optimally, we want to choose a set of $\{v_g\} \subseteq s_i \cap T$ to change their parents in T, such that the decrement of storage cost is minimized while recreation cost is satisfied. The optimal adjustment itself can be viewed as a knapsack problem with extra non-cyclic constraint of T, which is NP-hard. Instead, we use the same heuristic in Eq. 4.1 to adjust $v_g \in s_i \cap T$ one by one by swapping its parent p_g to v_s until the group constraints cannot improved. Similarly, the parallel scheme ψ_p uses Eq. 4.2 for the adjustment operation. The complexity of this algorithm is $O(|\mathcal{E}|^2|S|)$. The pseudo-code is shown in Algorithm 2.

Input: $\mathcal{G}_V(\mathcal{V}, \mathcal{E}, c_s, c_r)$, snapshots S, recreation cost $\{\theta_i \geq 0 \mid s_i \in S\}$.

Output: A spanning tree T satisfying constraints $\{\mathcal{C}_r(T, s_i) \leq \theta_i\}$

1: let $T = \emptyset$ and Q be a priority queue of edges based on c_s

2: **push** $\{(\nu_0, v_i) \mid v_i \in \mathcal{V}\}$ in Q

3: **while** $Q \neq \emptyset$ **do**

4: **pop** $e_{ij} = (v_i, v_j)$ from Q; let $T = T \cup \{e_{ij}\}$

5: let constraints satisfaction flag be $\Theta^{e_{ij}}_{satisfy} = true$

6: **for** each snapshot constraint $s_a \in \{s \mid s \in S \wedge v_j \in s\}$ **do**

7: estimate recreation cost $\hat{\mathcal{C}}_r(T, s_a)$

8: $\Theta^{e_{ij}}_{satisfy} = false$ and **break** if $\hat{\mathcal{C}}_r(T, s_a) > \theta_a$

9: **end for**

10: **if** $\Theta^{e_{ij}}_{satisfy}$ is $false$, **then** $T = T - \{e_{ij}\}$ and **goto** line 3

11: **pop** inner edges of v_j $\mathcal{I}^j_T = \{(v_k, v_j) \mid v_k \in T\}$ from Q

12: **push** outer edges $\mathcal{O}^j_{\mathcal{E}-T} = \{(v_j, v_k) \mid v_k \in \mathcal{E} - T\}$ to Q

13: **for** $(v_k, v_j) \in T$, change p_k improves $\mathcal{C}_s$, and no worse $\mathcal{C}_r$ **do**

14: *swap* $(p_k, v_k) \in T$ with (v_j, v_k)

15: **end for**

16: **end while**

17: **if** T is not a spanning tree **then**

18: **for each** $v_u \in \mathcal{V} - V_T$, **do** $T = T \cup \{e_{0u} = (\nu_0, v_u)\}$

19: *adjust* T using PAS-MT heuristic.

20: **end if**

21: **return** T if T is a *matrix storage plan*

Algorithm 2: PAS-PT

4.4.4 Model Evaluation Scheme in PAS

Model evaluation, i.e., applying a DNN forward on a data point to get the prediction result, is a common task to explore, debug and understand models. Given a PAS storage plan, an `dlv eval` query requires uncompressing and applying deltas along the path to the model. We develop a novel model evaluation scheme utilizing the segmented design, that progressively accesses the low-order segments only when necessary, and guarantees no errors for arbitrary data points.

The basic intuition is that: when retrieving segmented parameters, we know the minimum and maximum values of the parameters (since higher order bytes are retrieved first). If the prediction result is the same for the entire range of those values, then we do not need to access the lower order bytes. However, considering the high dimensions of parameters, non-linearity of the DNN model, unknown full precision value when issuing the query, it is not clear if this is feasible.

We define the problem formally, and illustrate the determinism condition that we use to develop our algorithm. Our technique is inspired from theoretical stability analysis in numerical analysis. We make the formulation general to be applicable to other prediction functions. The basic assumption is that the prediction function returns a vector showing relative strengths of the classification labels, then the dimension index with the maximum value is used as the predicted label.

Problem 2 (Parameter Perturbation Error Determination) *Given a prediction function* $\mathcal{F}(d, W) : \mathbb{R}^m \times \mathbb{R}^n \mapsto \mathbb{R}^c$*, where* d *is the data and* W *are the learned weights, the prediction result* c_d *is the dimension index with the highest value in the*

output $o \in \mathbb{R}^c$. When W value is uncertain, i.e., each $w_i \in W$ in known to be in the range $[w_{i,\min}, w_{i,\max}]$, determine whether c_d can be ascertained without error.

When W is uncertain, the output o is uncertain as well. However, if we can bound the individual entries in o, then the following condition is an applicable necessary condition for determining error:

Lemma 3 *Let $o_i \in o$ vary in range $[o_{i,\min}, o_{i,\max}]$. If $\exists k$ such that $\forall i,\ o_{k,\min} > o_{i,\max}$, then prediction result c_d is k.*

Next we illustrate a query procedure, that given data d, evaluates a DNN with weight perturbations and determines the output perturbation on the fly. Recall that DNN is a nested function (Section 4.2), we derive the output perturbations when evaluating a model while preserving perturbations step by step:

$$x_{0,k} = \sum_j W_{0,k,j} d_j + b_{0,k}$$

$$x_{0,k,\min} = \sum_j \min\{W_{0,k,j} d_j\} + \min\{b_{0,k}\}$$

$$x_{0,k,\max} = \sum_j \max\{W_{0,k,j} d_j\} + \max\{b_{0,k}\}$$

Next, activation function σ_0 is applied. Most of the common activation functions are monotonic functions: $\mathbb{R} \mapsto \mathbb{R}$, (e.g. sigmoid, ReLu), while pool layer functions are min, max, avg functions over several dimensions. It is easy to derive the perturbation of output of the activation function, $[f_{0,k,\min}, f_{0,k,\max}]$. During the evaluation query, instead of 1-D actual output, we carry 2-D perturbations, as the actual parameter value is not available. Nonlinearity decreases or increases the perturbation range.

Now the output perturbation at f_i can be calculated similarly, except now both W and f_{i-1} are uncertain:

$$x_{i,k} = \sum_j W_{i,k,j} f_{i-1,j} + b_{i,k}$$

$$x_{i,k,\min} = \sum_j \min\{W_{i,k,j} f_{i-1,j}\} + \min\{b_{i,k}\}$$

$$x_{i,k,\max} = \sum_j \max\{W_{i,k,j} f_{i-1,j}\} + \max\{b_{i,k}\}$$

Applying these steps iteratively until last layer, we can then apply Lemma 3, the condition of error determinism, to check if the result is correct. If not, then lower order segments of the float matrices are retrieved, and the evaluation is re-performed.

This progressive evaluation query techniques dramatically improve the utility of PAS, as we further illustrate in our experimental evaluation. Note that, other types of queries, e.g., matrix plots, activation plots, visualizations, etc., can often be executed without retrieving the lower-order bytes either.

4.5 Evaluation Study

ModelHub is designed to work with a variety of deep learning backends; our prototype interfaces with `caffe` [72] through a ProvDB ingestor extension that can extract caffe training logs, and read and write parameters for training. We have also built a custom layer in `caffe` to support progressive queries. Similar to ProvDB shell ingestor, the `dlv` command-line suite is implemented as a Ruby gem, utilizing `git` as internal VCS and `sqlite3` and PAS as backends to manage the set

of heterogeneous artifacts in the local client. PAS is built in C++ with gcc 5.4.0. All experiments are conducted on a Ubuntu Linux 16.04 machine with an 8-core 3.0GHz AMD FX-380 processor, 16GB memory, and NVIDIA GTX 970 GPU. We use zlib for compression; unless specifically mentioned, the compression level is set to 6. When wrapping and modifying caffe, the code base version is rc3.

In this section, we present a comprehensive evaluation with real-world and synthetic datasets aimed at examining our design decisions, differences of configurations in PAS, and performance of archiving and progressive query evaluation techniques proposed in earlier sections.

4.5.1 Dataset Description

4.5.1.1 Real World Dataset

To study the performance of PAS design decisions, we use a collection of shared caffe models listed in Table 4.5 published in caffe repository or Model Zoo. In brief, LeNet-5 [93] is a convolutional DNN with 431k parameters. The reference model has 0.88% error rate on MNIST. AlexNet [94] is a medium-sized model with 61 million parameters, while VGG-16 [95] has 1.9 billion parameters. Both AlexNet and VGG-16 are tested on ILSVRC-2012 dataset. The downloaded models have 43.1%, and 31.6% top-1 error rate respectively. Besides, to study the delta performance on model repositories under different workloads (i.e., retraining, fine-tuning): we use VGG-16/19 and CNN-S/M/F [105], a set of similar models developed by VGG authors to study model variations. They are similar to VGG-16, and retrained from

Network	$\|W\|$ **(flops)**	Size	Purpose
LeNet-5	4.31×10^5	5MB	Small model
AlexNet	6×10^7	200MB	Medium model
VGG-16 [95]	1.96×10^{10}	500MB	Large model
CNN-S/M/F [105]	1.13×10^{10}	199/300MB	Similar models
VGG-Salient	1.96×10^{10}	500MB	Fine-tuning

Table 4.5: Real World DNN Models used in the Experiment Study

scratch; for fine-tuning, we use VGG-Salient [106] a fine-tuning VGG model which only changes last full layer.

4.5.1.2 Synthetic Datasets

Lacking sufficiently fine-grained real-world repositories of models, to evaluate performance of parameter archiving algorithms, we developed an automatic modeler to enumerate models and hyperparameters to produce a `dlv` repository. We generated a synthetic dataset (SD): simulating a modeler who is enumerating models to solve a face recognition task, and fine-tuning a trained VGG. SD results in similar DNNs and relatively similar parameters across the models. As retraining is always used, SD1 has a set of different models w.r.t. network architecture and parameters, while finetuning practice used in SD2 results in similar network architecture, and more similar parameters. The datasets are shared online[4].

To elaborate, the automation is driven by a state machine that applies model-

[4]Dataset Details: `http://www.cs.umd.edu/~hui/code/modelhub`

ing practices from the real world. For SD1, the modeler mutates the network architecture intensively by inserting/deleting layers and changing layer shapes, as well as by updating optimization related hyperparameters. While in SD2, the modeler updates the VGG network architecture slightly and changes VGG object recognition goal to a face prediction task (prediction labels changed from 1000 to 100, so the last layer is changed); various fine-tuning hyperparameter alternations are applied by mimicking practice [107]. SD in total has 54 model versions, each of which has 10 snapshots. A snapshot has 16 parametric layers and a total of 1.96×10^{10} floats.

4.5.2 Evaluation Results

4.5.2.1 Float Representation & Accuracy

We show the effect of different float encoding schemes on compression and accuracy in Figure 4.8; this is a tradeoff that the user often needs to consider when configuring ModelHub to save a model. In Figure 4.8, for each scheme, we plot the average compression ratio versus the average accuracy drop when applying PAS float schemes on the three real-world models. Here, *random* and *uniform* denote two standard quantization schemes. As we can see, we can get very high compression ratios (a factor of 20 or so) without a significant loss in accuracy, which may be acceptable in many scenarios.

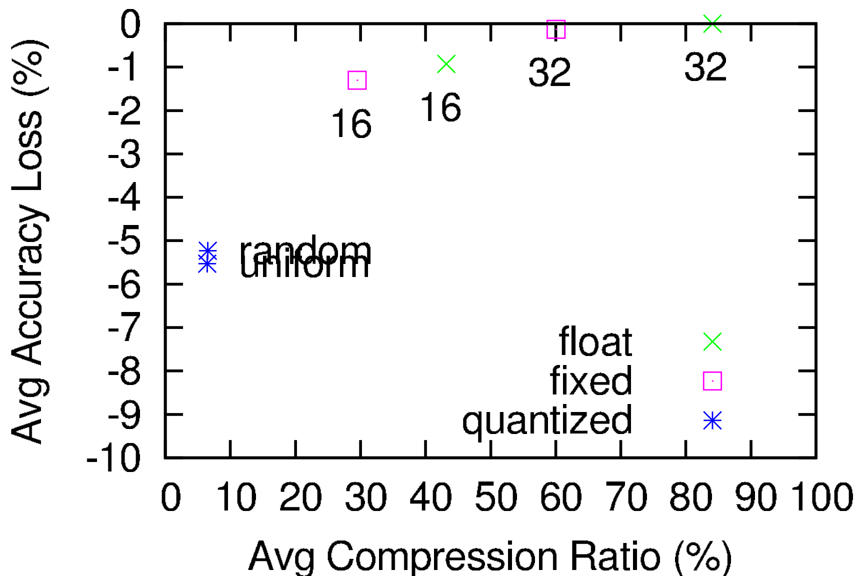


Figure 4.8: Compression-Accuracy Tradeoff for Float Representation Schemes

4.5.2.2 Delta Encoding & Compression Ratio Gain

Next we study the usefulness of delta encoding in real-world models in the following scenarios: **a)** *Similar*: latest snapshots across similar models (CNN-S/M/F, VGG-16); **b)** *Fine-tuning*: fine-tuning models (VGG-16, VGG-Salient); and **c)** *Snapshots*: snapshots for the same VGG models in SD between iterations. In Figure 4.9, for different delta schemes, namely, storing original matrices (*Materialize*), arithmetic subtraction (*Delta-SUB*), and bitwise XOR diff (*Delta-XOR*), the comparison is shown (i.e., we show the results of compressing the resulting matrices using **zlib**). The figure shows the numbers under lossless compression scheme (float 32), which has the largest storage footprint.

As we can see, delta scheme is not always good, due to the non-convexity and high entropy of parameters. For models under similar architectures, storing

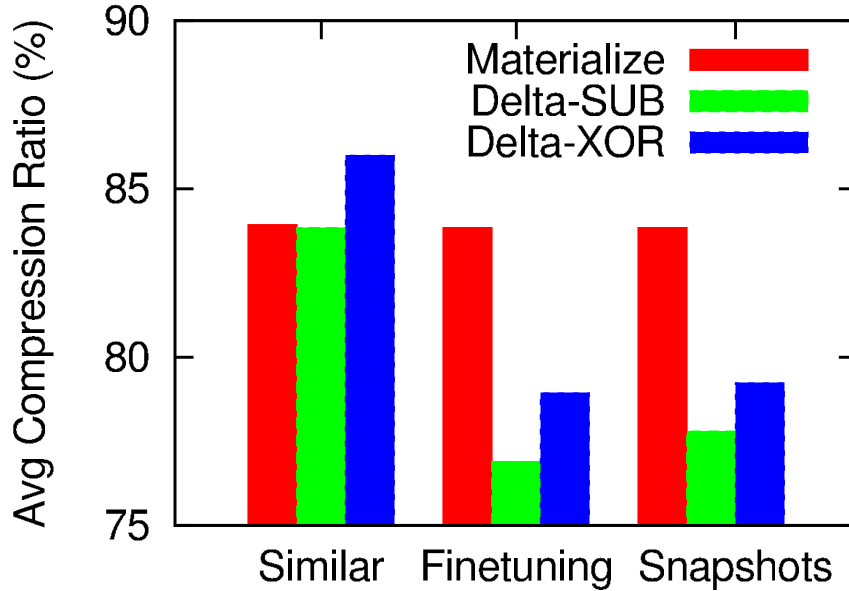


Figure 4.9: Compression Performance for Different Delta Schemes & Models

materialized original parameters is often better than applying delta encoding. With fine-tuning and nearby snapshots, the delta is always better, and arithmetic subtraction is consistently better than bitwise XOR. We saw similar results for many other models. These findings are useful for PAS implementation decisions, where we only perform delta between nearby snapshots in a single model, or for the fine-tuning setting among different models.

Table 4.6 shows the delta encoding results when using two lossy schemes, fixed point conversion and *normalization*, for fine-tuned VGG datasets, but without reducing the number of bits used (i.e., we still use 32 bits to store the numbers). Normalization refers to adding a sufficiently large number to all the floats so that the radixes and the signs are aligned, whereas fixed point conversion uses a single exponent for all the numbers in a matrix. As we can see, for both of these, delta encoding can result in sigificant gains. Introducing additional lossiness, e.g.,

Schemes	Configuration	Materialize	Delta-SUB
Float Number	Lossless	92.83%	86.39%
Representation	Lossless, bytewise	83.85%	76.89%
	Fix point	72.43%	57.15%
	Fix point, bytewise	58.68%	49.34%
After	Lossless	68.06%	47.69%
Normalization	Lossless, bytewise	56.15%	**36.60%**
	Fix point	69.11%	48.94%
	Fix point, bytewise	55.36%	36.88%

Table 4.6: Delta Performance For Lossless & Lossy Schemes, 32-bits

through using fewer bits, further improves the performance, but at the expense of significantly higher accuracy loss.

4.5.2.3 Optimal Parameter Archival Storage

Figure 4.10 shows the results of comparing PAS-PT, PAS-MT and the baseline LAST [65] for the optimal parameter archival problem. Using dataset SD, we derive nearby snapshot deltas as well as model-wise deltas among the latest snapshots. To compare with LAST clearly, we vary the recreation threshold using a scalar α to mimic a full precision archiving problem instance with different constraints, i.e., $\mathcal{C}_r(T, s_i) \leq \alpha \cdot \mathcal{C}_r(\text{SPT}, s_i)$. The SPT for SD is 22.77Gb and the MST is 15.44Gb. In Figure 4.10, the left y-axis denotes the storage cost ($\mathcal{C}_s$) while the

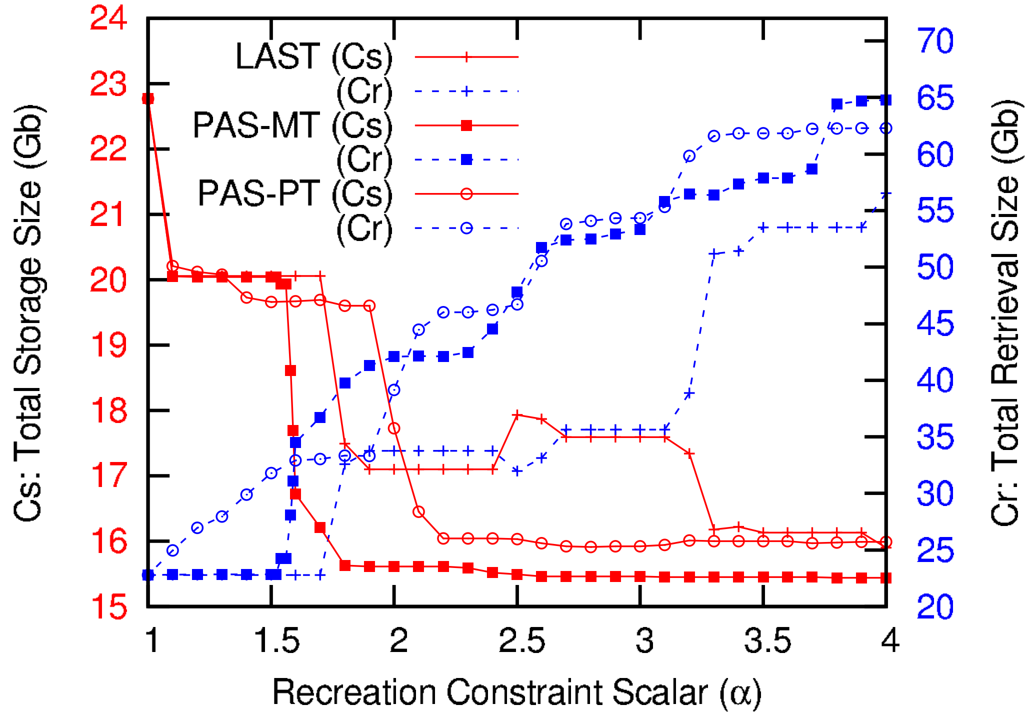


Figure 4.10: Comparing PAS Archival Storage Algorithms for SD

right y-axis is the recreation cost ($\mathcal{C}_r$).

As we can see, in most cases, PAS-MT and PT find much better storage solutions that are very close to the MST (the best possible) by exploiting the recreation thresholds. In contrast, LAST, which cannot handle group constraints, returns worse storage plans and cannot utilize the recreation constraints fully. Between MT and PT, since MT starts from the MST and adjusts it, when the constraints are tight (i.e., $\alpha < 1.5$), MT cannot alter it to very different trees and the recreation constraints are underutilized; however, PT can exploit the constraints when selecting edges to grow the tree. On the other hand, when the threshold is loose ($\alpha \in [1.5, 2]$), MT's edge swapping strategy is able to refine MST extensively, while PT prunes edges early and cannot find solutions close to MST. When the constraints

Storage Plan	Query	Independent (s)	Parallel (s)
Materialization	Full	3.49	2.16
Min Storage	Full	8.47	4.85
PAS ($\alpha = 1.6$)	Full	8.1	4.59
	2 bytes	3.19	0.38
	1 byte	1.60	0.18

Table 4.7: Recreation Performance Comparison of Storage Plans

continue to loosen, both PAS algorithms find good plans, while LAST can only do so at very late stages ($\alpha > 3$). In practice, the best option might be to execute both algorithms and pick the best solution for a given setting.

4.5.2.4 Retrieval Performance

Next we show the retrieval performance for PAS storage plans using the SD dataset. The main query type of interest is snapshot retrieval, which would retrieve all segments of a snapshot or, for a partial retrieval query, the high-order segments. In Table 4.7, the average recreation time of a snapshot for a moderate PAS storage plan ($\alpha = 1.6$) is compared with the two extreme cases, full materialization, and minimum storage without recreation constraints. As we can see, PAS is not only able to find good solutions which satisfy recreation constraints, but also supports flexible access schemes. Under partial access of high order bytes, the query times for segmented snapshots are better than uncompressing the fully materialized model.

4.5.2.5 Progressive Query Evaluation

We study the efficiency of the progressive evaluation technique using perturbation error determination scheme on real-world models (LeNet, AlexNet, VGG16) and their corresponding datasets. The original parameters are 4-byte floats, which are archived in segments in PAS. We modify `caffe` implementation of involved layers and pass two additional blobs (min/max errors) between layers. The perturbation error determination algorithm uses high order segments, and answers `eval` query on the test dataset. The algorithm determines whether top-k (1 or 5) result needs lower order bytes (i.e., matched index value range overlaps with $k+1$ index value range). The result is summarized in Figure 4.11. The y-axis shows the error rate. The x-axis shows the percentage of data that needs to be retrieved (i.e., 2 bytes or 1 byte per float). As one can see, the prediction errors requiring full pre-

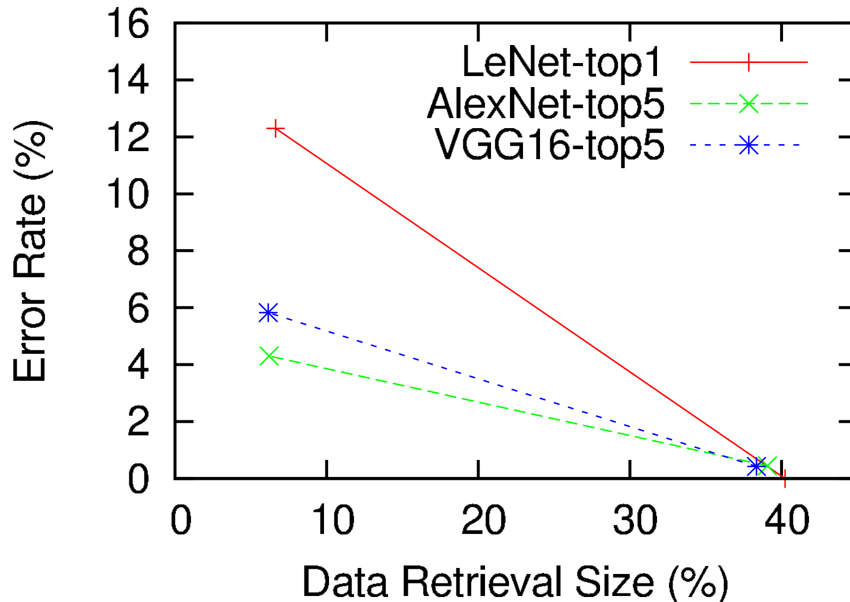


Figure 4.11: Progressive Evaluation Query Processing Using High-Order Bytes

cision lower-order bytes are very small. The less high-order bytes used, higher the chance of potential errors. The consistent result of progressive query evaluation on real models supports our design decision of segmented float storage.

4.6 Conclusion

In this chapter, we described how to build domain-specific ProvDB extensions to address some of the key data management challenges in learning, managing, and adjusting deep learning models. ModelHub attempts to address those challenges in a systematic fashion. The goals of ModelHub are multi-fold: (a) to make it easy for a user to explore the space of potential models by tweaking the network architecture and/or the hyperparameter values, (b) to minimize the burden in keeping track of the metadata including the accuracy scores and the fine-grained results, and (c) to compactly store a large number of models and constituent snapshots without compromising on query or retrieval performance. We presented several high-level abstractions, including a command-line version management tool and a domain-specific language, for addressing the first two goals. Anecdotal experience with our early users suggests that both of those are effective at simplifying the model exploration tasks. We also developed a read-optimized parameter archival storage for storing the learned weight parameters, and designed novel algorithms for storage optimization and for progressive query evaluation. Extensive experiments on real-world and synthetic models verify the design decisions we made and demonstrate the advantages of proposed techniques.

Chapter 5: Querying Collaborative Analytics Lifecycle Provenance

To support non-intrusive and extensible provenance ingestion mechanisms to collect rich information, ProvDB and other lifecycle management systems often use a graph data model (e.g., property graph) and query languages (e.g., Cypher) to represent and manipulate the stored provenance. However, due to the schema-later nature of the metadata, multiple versions of the same files, unfamiliar artifacts introduced by team members, and enormous provenance records collected continuously, querying the ingested provenance graph and utilizing it to a large extent is very challenging for the users. As the provenance graph is verbose and evolving, and the users only have partial knowledge of it, just using standard graph languages makes it very difficult to compose queries and utilize the valuable information. These observations echo the development of provenance systems in other domains, such as cybersecurity, where a clear workflow is not present and query issuers need to deal with verbose and evolving provenance graphs.

In this chapter, we propose general provenance graph query operators on standard PROV graph data model to address the verboseness and evolving nature of such provenance graphs. We mainly focus on querying workflow provenance for data science lifecycles which are collaboration pipelines consisting of versioned artifacts

and parametrized derivation steps. We discussed studies on data provenance for data science systems in Chapter 2. In Section. 5.1, we first overview the issues of querying provenance graphs on modern graph databases, and motivate the problem. Then in Section 5.2, we review the standard PROV data model, and point out the challenges and desiderata of querying collaborative analytics provenance graphs. In Section 5.3 and 5.4, we introduce graph query operators to induce and summarize provenance subgraphs of interest by allowing the users to only have partial knowledge of the lifecycle. We show the semantics of such queries and propose efficient evaluation techniques on top of a property graph data store. Next, the implementation of the operators and their position in ProvDB are illustrated in Section 5.5. In Section 5.6, we evaluate our query methods extensively on a variety of provenance graph datasets and show the effectiveness and efficiency of the proposed methods.

5.1 Introduction

As mentioned in early chapters, provenance management has been identified as an important problem for the prosperous data science activities [50, 85]. In general, capturing provenance allows the practitioners introspect the data analytics trajectories, monitor the ongoing modeling activities, and communicate the practice with others [85].

Compared with well-established data provenance systems for databases [5], and scientific workflow systems for e-science [3], building provenance systems for data science faces an *unstable* data science lifecycle that is often ad hoc, typically

featuring highly unstructured datasets, an amalgamation of different tools and techniques, significant back-and-forth among team members, and trial-and-error to identify the right analysis tools, models, and parameters.

Schema-later approaches and graph data model are often used to capture the lifecycle, versioned artifacts and associated rich information [50, 85], which also echoes the modern provenance data model standardization over a long period of time as a result of consolidation for scientific workflows [18] and the Web [19].

Though there is an enormous potential value of data science lifecycle provenance and high hype to reproduce the results or accelerate the modeling process, the evolving and verbose nature of the captured provenance graphs makes them difficult to store and manipulate. Depending on the granularity, storing the graphs could take dozens of GBs within several minutes [108]. More importantly, given the evolving lifecycle and verbose provenance graphs, it is difficult to write general queries to explore the graph and utilize it, because there are no predefined workflows, i.e., the pipelines change as the project evolves, and because of arbitrary steps (e.g., trial and error) in the modeling process. Though storing the provenance graph in a graph database seems like a natural choice, most of the provenance query types of interest involve paths [109], and require returning paths instead of answering questions about reachability [21], which are beyond the capability of the pattern matching query (BPM) and regular path query (RPQ) support in popular modern graph databases [110, 111]. For example, answering 'how is today's result file generated from today's data file' requires a segment of the provenance graph including not only the mentioned files but also other files that are not on the path and the

user may not know at all (e.g., ‘a configuration file’); answering ‘how do the team members typically generate the result file from the data file?’ requires summarizing several of the query results of the above query and visualize at different resolutions.

Lack of proper query facilities in modern graph databases not only limits the value of lifecycle provenance systems for data science, but also of other provenance systems. Provenance queries have specialized query types of interest [21, 109], and provenance systems often implement specialized storage systems [112] and query interfaces [22, 23] on their own [3]. Recent works on provenance graphs in the provenance community propose various graph transformations for different tasks, which are essentially different template queries from the graph querying perspective, such as grouping nodes together to handle publishing policies [24], summarizing verbose graphs by node types to understand commonalities and outliers [25], segmenting provenance graphs via declarative languages to support feature extractions for cybersecurity [26]. We attempt to draw connections between provenance graph query types of interest and modern graph databases capabilities while building a provenance system to aid the data analytics lifecycle.

In this chapter, we propose two graph operators for common provenance queries to let the user explore the evolving provenance graph without fully understanding the underlying provenance graph structure. The operators not only help our purpose in the context of data science but also the applications using standard provenance data models [18, 19] that have no clear workflow skeletons and are verbose in nature [25, 26, 108, 113].

First, we introduce a flexible graph segmentation operator, which queries the

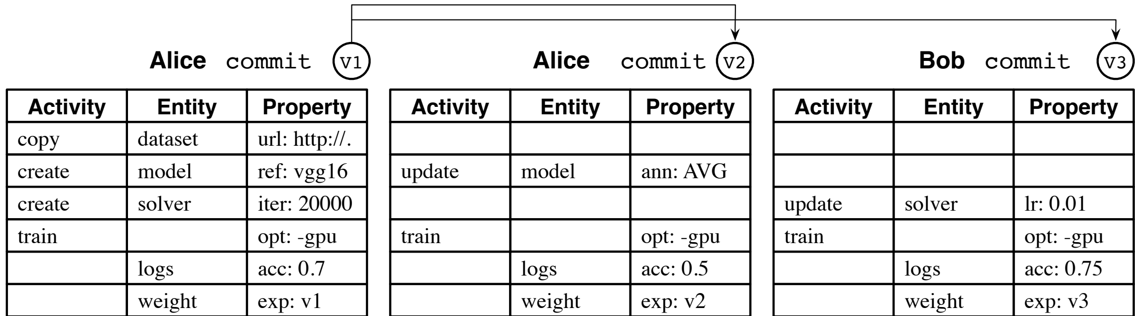
provenance of a collection of user-given nodes (e.g., versioned file snapshots, authors, an entered command) given boundary criteria (e.g., hops, timestamps, path patterns). We show the semantics of such a query in a context free language, and discuss the connection with RPQ and propose efficient evaluation techniques on top of a property graph store. Second, we propose a graph summarization operator for aggregating the segmentation results, which allows multi-resolution interaction with the query results to understand similar and abnormal behaviors in those segments.

5.2 Challenges & Desiderata

Next, we give motivating examples, and introduce standard provenance data model and adaptations in our context. Then we summarize the typical provenance query types of interest and analyze them from the perspective of graph queries.

5.2.1 Motivating Example

In the lifecycle of a data analytics project [**?**,77,78,83], given specific datasets (e.g., face images and labels) and a goal (e.g., prediction function from a face to a label with high accuracy), the data scientists in a team collaborate with each other and try different models repetitively. Using a lifecycle provenance management system [50,85], details of the project progress, versions of the artifacts and associated provenance are captured and managed. In Example 6, we use a classification task using neural networks to illustrate the system background, provenance model and query type.



Activity	Entity	Property
copy	dataset	url: http://.
create	model	ref: vgg16
create	solver	iter: 20000
train		opt: -gpu
	logs	acc: 0.7
	weight	exp: v1

Activity	Entity	Property
update	model	ann: AVG
train		opt: -gpu
	logs	acc: 0.5
	weight	exp: v2

Activity	Entity	Property
update	solver	lr: 0.01
train		opt: -gpu
	logs	acc: 0.75
	weight	exp: v3

Figure 5.1: A Data Analytics Project Lifecycle Example & Associated Provenance

Example 6 *In Figure 5.1, Alice and Bob work together on a classification task to predict face ids given an image. Alice starts the project and creates a neural network by modifying a popular model. She downloads the dataset and edits the model definitions and solver hyperparameters, then invokes the training program with specific command options. After training the first model, she examines the accuracy in the log file, annotates the weight files, then commits a version using* `git`*. As the accuracy of the first model is not ideal, she changes the neural network by editing the model definition, trains it again and derives new log files and weight parameters. However the accuracy drops, and she turns to Bob for help. Bob examines what she did, trains a new model following some best practices by editing the solver configuration in version* v_1*, and commits a better model.*

Behind the scene, a lifecycle management system tracks user activities, manages project artifacts (e.g., datasets, models, solvers) and ingests provenance. In the Figure 5.1 tables, we show ingested information in detail: a) *history of user activities (e.g., the first* train *command uses model* v_1 *and solver* v_1 *and generates logs* v_1 *and weights* v_1*),* b) *versions and changes of entities (e.g., weights* v_1*,* v_2 *and* v_3*)*

and derivations among those entities (e.g., model v_2 is derived from model v_1), and c) *provenance records as associated properties to activities and entities, ingested via provenance system ingestors (e.g., dataset is copied from some url, Alice changes a pool layer type to AVG in v_2, accuracy in logs v_3 is* 0.75*).*

5.2.2 Provenance Model

The ingested provenance of the project lifecycle naturally forms a provenance graph, which is a directed acyclic graph[1] and encodes information with multiple aspects, such as a version graph representing the artifact changes, a workflow graph reflecting the derivations of those artifact versions, and a conceptual model graph showing the involvement of problem solving methods in the project [50, 85]. To represent the provenance graph and keep our discussion general to other provenance systems, we choose the W3C PROV data model [114], which is a standard interchange model for different provenance systems. Different aspects of the provenance graph are supported via a rich set of query facilities (Section 5.5) on top of the PROV data model.

The full PROV data model is complex in order to satisfy application needs for different domains [114]. For simplicity, we use the core subset of it, which is shown in Figure 5.2. There are three types of vertices ($\mathbb{V}$) in the provenance graph:

- Entities ($\mathcal{E}$): are the project artifacts (e.g., files, datasets, scripts) which the users work on and talk about in a project, and the underlying lifecycle man-

[1] In our system, we use versioning to avoid cyclic self-derivations of the same entity and overwritten entity generations by some activity.

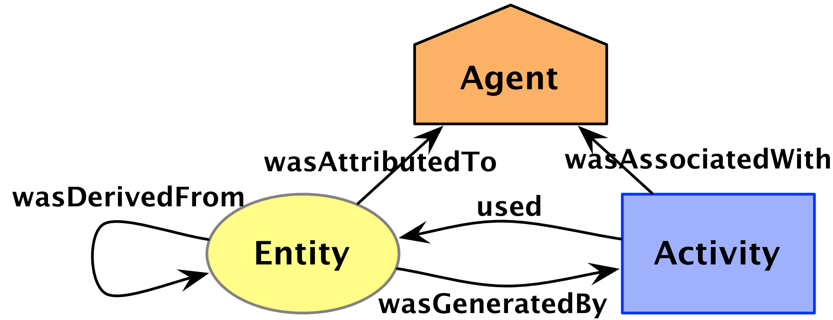


Figure 5.2: Illustration of the W3C PROV Data Model

agement system manages their provenance.

- Activities ($\mathcal{A}$): are the system or user actions (e.g., `train`, `git commit`, `cron` jobs) which act upon or with entities over a period of time, $[t_i, t_j)$.

- Agents ($\mathcal{U}$): are the parties who are responsible for some activity (e.g., a team member, a system component).

Among the vertices, we focus our discussion to five types of directed edges[2] ($\mathbb{E}$):

- 'used' ($U \subseteq \mathcal{A} \times \mathcal{E}$): An activity started at time t_i often uses some entities.

- 'wasGeneratedBy' ($G \subseteq \mathcal{E} \times \mathcal{A}$): Then some entities would be generated by the same activity at time t_j ($t_j \geq t_i$).

- 'wasAssociatedWith' ($S \subseteq \mathcal{A} \times \mathcal{U}$): An activity is always associated with some agent during its period of execution.

[2]There are 13 type of relationships among Entity, Activity and Agent. The proposed techniques in the chapter can be extended naturally to support more relation types in other provenance systems.

For instance, in Example 6, the activity `train` was associated with Alice, used a set of artifacts (model, solver, and dataset) and generated other artifacts (logs, weights).

- 'wasAttributedTo' ($A \subseteq \mathcal{E} \times \mathcal{U}$): Besides an entity's presence would be attributed to some agent, e.g., the dataset in Example 6 was added from external sources and attributed to Alice.

- 'wasDerivedFrom' ($D \subseteq \mathcal{E} \times \mathcal{E}$): An entity would be derived from another entity, such as different versions of the same artifact (e.g., different model versions in v_1 and v_2 in Figure 5.1).

In the provenance graph, both vertices and edges have a label to encode their vertex type in $\{\mathcal{E}, \mathcal{A}, \mathcal{U}\}$ or edge type in $\{U, G, S, A, D\}$. All other provenance records are modeled as properties, ingested by a set of configured project ingestors during the period of activity executions and represented as key-value pairs.

PROV standard defines various serializations of the concept model, such as RDF, XML, and JSON [19]. In our system, we use a physical property graph data model to store it, as it is more natural for the users to think of the artifacts as nodes when writing queries using Cypher or Gremlin. It is also more compact than RDF graph for the large amount of provenance records, which are treated as literal nodes. We discuss implementation details in Section 5.5. As a summary, we formally define the provenance graph used in the rest of the chapter.

Definition 3 (Provenance Graph) *Provenance in a data analytics project is represented as a directed acyclic graph,* $\mathcal{G}(\mathbb{V}, \mathbb{E}, \lambda_v, \lambda_e, \sigma, \omega)$*, where vertices have three types,* $\mathbb{V} = \mathcal{E} \cup \mathcal{A} \cup \mathcal{U}$*, and edges have five types,* $\mathbb{E} = U \cup G \cup S \cup A \cup D$*. Label*

functions, λ_v*:* $\mathbb{V} \mapsto \{\mathcal{E}, \mathcal{A}, \mathcal{U}\}$*, and* λ_e*:* $\mathbb{E} \mapsto \{U, G, S, A, D\}$ *are total functions associating each vertex and each edge to its type. Given a project, we refer to the set of property types as* $\mathcal{P}$ *and their values as* $\mathcal{V}$*, then vertex properties* σ*:* $\mathbb{V} \times \mathcal{P} \mapsto \mathcal{V}$ *and edge properties* ω*:* $\mathbb{E} \times \mathcal{P} \mapsto \mathcal{V}$ *are partial functions from vertex/edge and property type to some value.*

Example 7 *Using the* PROV *data model, in Figure 5.3, we show the corresponding provenance graph of the project lifecycle listed in Figure 5.1. Vertex shapes follow their type in Figure 5.2. Names of the vertices (e.g., 'model-v1', 'train-v3', 'Alice') are made by using their representative properties (i.e., project artifact names for entities, operation names for activities, and first names for agents) and suffixed using the version ids to distinguish different snapshots. Activity vertices are ordered from left to right w.r.t. the temporal order of their executions. We label the edges using their types and show a subset of the edges in Figure 5.1 to illustrate usages of five relationship types. Note there are many snapshots of the same artifact in different versions, and between the versions, we maintain derivation edges 'wasDerivedFrom' (D) for efficient versioning storage. The figure shows the provenance of those entities in all three versions. The property records are shown as white rectangles but not treated as vertices in the property graph model.*

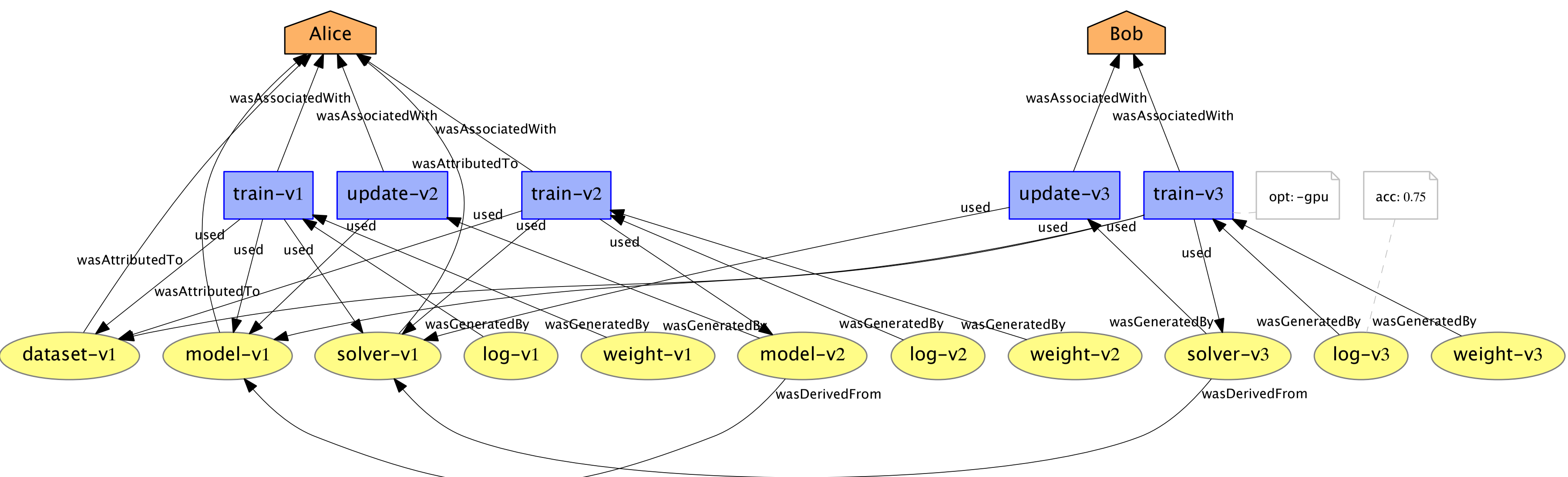


Figure 5.3: Provenance Graph for the Lifecycle Example (Some edges and properties are not shown)

Characteristics

The provenance graph has the following characteristics, which we need to consider when designing query facilities:

- **Versioned Artifact**: Each entity is a point-in-time snapshot of some artifact in the project. For instance, the query 'accuracy of this version of the model' discusses a particular **snapshot** of the model artifact, while 'what are the common updates for *solver* before `train`' refer to the **artifact** but not an individual snapshot. R1: The query facilities need to support both aspects in the graph.

- **Evolving Workflows**: Data analytics lifecycle is explorative and collaborative in nature, so *there is no static workflow skeleton, and no clear boundaries for individual runs* in contrast with workflow systems [3]. For instance, the modeling methods may change (e.g., from SVM to neural networks), the data processing steps may vary (e.g., split, transform or merge data files), and the user-committed versions may be mixed with code changes, error fixes, thus may not serve as boundaries of provenance queries for entities. R2: The query facility for snapshots should not assume workflow skeleton and should allow flexible boundary conditions.

- **Partial Knowledge in Collaboration**: Each team member may work on and be familiar with a subset of artifacts and activities, and may use different tools or approaches, e.g., in Example 6, Alice and Bob use different approaches to improve accuracy. When querying retrospective provenance of

the snapshots attributed to other members or understanding activity process over team behaviors, the user may only have partial knowledge at query time, thus may find it difficult to compose the right graph query. R3: The query facility should support provenance queries with partial information reflecting users' understanding and induce correct result.

- **Verboseness for Usage**: In practice, the provenance graph would be very verbose for humans to use and in large volume for the system to store. With similar goals to recent research efforts [21, 24–26], our system aims to let the users understand the essence of provenance at their preference level by transforming the provenance graph. R4: The query facility should be able to aggregate snapshots derivations, not only to reflect the commonalities but also anomalies among derivations.

5.2.3 Provenance Queries & Challenges

Despite the exchange data model, the provenance standards (e.g., PROV, OPM) do not describe query models, as different systems have their own application-level meanings of those nodes [18, 19]. Many provenance systems focus on ingestion methods and rely on standard query language (e.g., SQL, SPARQL, Cypher) provided by the backend DBMS to let the user manipulate the underlying information [3]. However, general queries to express provenance retrieval tend to be very complex. To improve usability, a few systems provide novel query facilities [10, 23], and some of them propose special query languages [21, 22]. Recent provenance

systems which adopt W3C PROV data model naturally use graph stores as backends [108, 113]; while the usability of standard graph query language often cannot satisfy the needs [21, 110], a set of graph manipulation techniques is often proposed to utilize the provenance [24–26].

Along the same lines, we aim to draw connections between the research to improve provenance graph usages with graph query techniques. By observing the characteristics of the provenance graph in analytics lifecycle and identifying the requirements for the query facilities (Section 5.2.2), we propose two graph operators (i.e., segmentation and summarization) for general provenance graphs in PROV data model. We first illustrate the queries in examples and emphasize the differences from prior work. We defer their formal discussion to Section 5.3 and 5.4.

5.2.3.1 Segmentation

A very important provenance query type of interest is querying ancestors and descendants of entities, which have different names (e.g., reachability, lineage) and subtle differences in formulations (only return true/false or construct paths; support regular paths). In our context, the users introspect the lifecycle and identify issues by analyzing dependencies among snapshots. Lack of a workflow skeleton and clear boundaries makes the queries over the provenance graph more difficult. Also note that the user may not specify all interested entities in a query due to partial knowledge. We propose a segmentation operator that allows the user to specify sets of source and destination entities, and the operator induces other important unknown

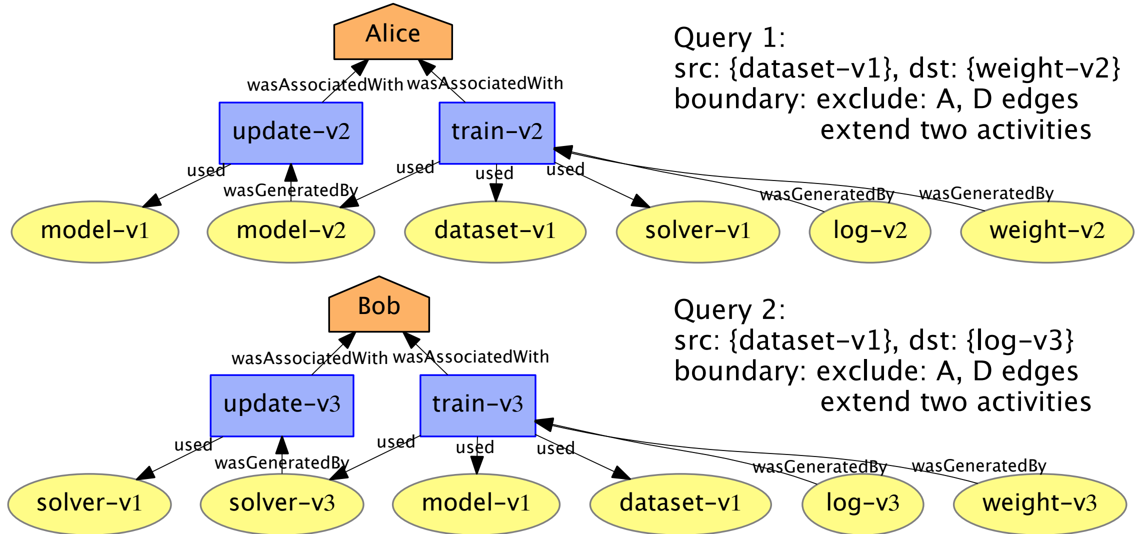


Figure 5.4: Segmentation Query Examples

entities in the query result. We also allow a set of boundary criteria to address the verboseness of the return graph.

Example 8 *In Figure 5.4, we show two examples of provenance graph segmentation query. In Query 1 (Q_1), Bob was interested in what Alice did in version v_2. He did not know the details of activities and the entities Alice touched, instead he set* {dataset, weight} *as querying entities to see how the* weight *in Alice's version v_2 was connected to the* dataset. *To exclude actions in earlier commits (e.g., v_1), he set the boundaries as two activities away from those querying entities. In the figure, the system found connections among the querying entities, and included the vertices within the boundaries. After interpreting the result, Bob knew Alice updated the model definitions in* model. *On the other hand, Alice would ask query to understand how Bob improved the accuracy and learn from him. In Query 2 (Q_2), instead of learned* weight, *accuracy property associated* log *entity is used as querying entity*

along with dataset. *The result showed Bob only updated solver configuration and did not use her new* model *committed in* v_2.

In contrast with similar queries in scientific workflow provenance systems [3, 17], their processes are predefined in template workflow skeletons, and multiple executions generate different instance-level provenance *run graphs*. By taking advantages of the workflow skeleton, there are lines of research for advanced ancestry query processing, such as defining user views over such skeleton to aid queries on verbose run graphs [23], executing reachability query on the run graphs efficiently [115], storing run graphs generated by the workflow skeletons compactly [27], and using workflow visualization as examples to ease query construction [10].

5.2.3.2 Summarization

In workflow systems, querying the workflow skeleton (a.k.a prospective provenance) is an important use case (e.g., business process [20]) and included in the provenance challenge [109]. In our context, even though a static workflow skeleton is not present, summarization of activity commonalities and identifying abnormal behaviors are very useful query capabilities. However, general graph summarization techniques [116–118] are not applicable to provenance graphs due to constraints of the data model definitions [24, 25, 119]. Inspired by the graph analytics work [116, 117, 120] and graph manipulations specific to provenance graphs [24, 25], we propose a summarization operator with multi-resolution capabilities for provenance graphs. To support different aspects of the provenance, we design summa-

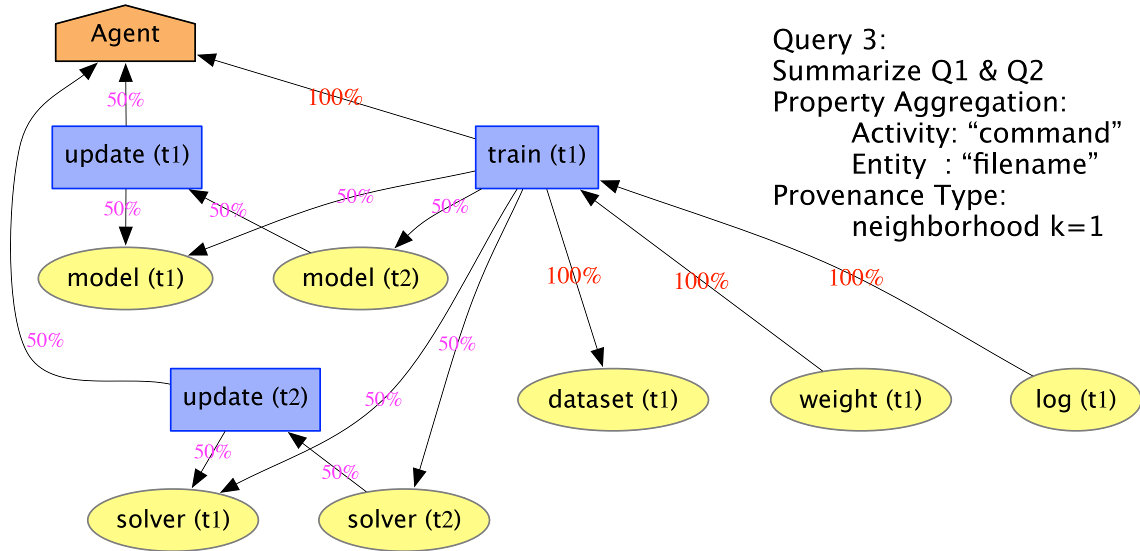


Figure 5.5: Summarization Query Examples

rization operator to be performed over query results of the segmentation operator; to ensure the rigidness of provenance entities, we use path constraints as vertex identifiers [25, 121].

Example 9 *We show a summarization query example in Figure 5.5. An outsider to the team (e.g., some auditor, new team member, or project manager) wanted to understand the activity overview in the project. Segmentation queries (e.g., Q_1, Q_2 in Figure 5.4) only show individual trails of the analytics process at the snapshot level. The outsider issued a summarization query, Query 3 (Q_3), by specifying the aggregation over three types of vertices, and defining the provenance types. The query result merged Q_1 and Q_2 into a summary graph G_S according to the query. In the figure, the vertices in G_S suffixed name with provenance types to show alternative generation process, while edges are labeled with their frequency of appearance among vertices in G_S. The query issuer would change the query conditions to derive various G_S at different resolutions.*

In contrast with other summarization work [118], our operator is designed for provenance graphs which include multiple types of nodes rather than a single node type [120, 122]; it works on query results rather than entire graph structure [116]; the summarization requirements are specific to provenance graphs rather than general ones [117]; we also consider aggregating graph structure and property values together, which is not studied before to the best of our knowledge.

In the following sections, we describe the proposed query models in detail. In Section 5.3, we introduce the segmentation operator (PgSeg) for snapshot-level retrospective analysis of the project activities. It returns an induced subgraph of the evolving workflow by allowing the users to only have partial knowledge of the project. In Section 5.4, a summarization operator (PgSum) is described for queries at the artifact level. It merges a set of segmentation results and generates a prospective overview for identifying commonalities and abnormalities at certain resolution by user-defined aggregations.

5.3 Segmentation Operation

Among the snapshots, collected provenance graph describes important ancestry relationships which form 'the heart of provenance data' [21]. Often lineages w.r.t. a query or a run graph trace w.r.t. a workflow are used to formulate ancestry queries in relational databases or scientific workflows [22]. However, in our context, there are no clear boundaries of logical runs, or query scopes to cleanly define the input and the output. Though a provenance graph could be collected, the key ob-

stacle is lack of formalisms to analyze the verbose information. In similar situations for querying script provenance [17], Prolog was used to traverse graph imperatively, which may be an overkill and require additional skill-set for team members. In our system, we design PgSeg to let the users who may only have partial knowledge to query retrospective provenance. PgSeg semantics induce a connected subgraph to show the ancestry relationships (e.g., lineage) among the entities of interest and include other causal and participating vertices within a specified boundary that is adjustable by the users. Next we first define the elements of the operator and query semantics, followed by query evaluation techniques.

5.3.1 Semantics of Segmentation (PgSeg)

At a high level, we view the PgSeg operator as a 3-tuple query $(\mathcal{V}_{src}, \mathcal{V}_{dst}, \mathcal{B})$ on a provenance graph $\mathcal{G}$ asking how a set of *source entities* $\mathcal{V}_{src} \subseteq \mathcal{E}$ are involved in generating a set of *destination entities* $\mathcal{V}_{dst} \subseteq \mathcal{E}$. PgSeg induces *induced vertices* (entities, activities and agents) $\mathcal{V}_{ind}$ that show the detailed generation process and satisfy certain *boundary criteria* $\mathcal{B}$. It returns a connected subgraph $\mathcal{S}(\mathbb{V}_{\mathcal{S}}, \mathbb{E}_{\mathcal{S}}) \subseteq \mathcal{G}$, where $\mathbb{V}_{\mathcal{S}}{=}\mathcal{V}_{src}{\cup}\mathcal{V}_{dst}{\cup}\mathcal{V}_{ind}$, and $\mathbb{E}_{\mathcal{S}}{=}\mathbb{E} \cap \mathbb{V}_{\mathcal{S}}{\times}\mathbb{V}_{\mathcal{S}}$.

When discussing the elements of PgSeg below, we use the following notations for paths in $\mathcal{G}$. A ***path*** π_{v_0,v_n} connecting vertices v_0 and v_n is a vertex-edge alternating sequence $\langle v_0, e_1, v_1, \cdots, v_{n-1}, e_n, v_n \rangle$, where $n > 1$, $\forall i \in [0,n]\ v_i \in \mathbb{V}$, and $\forall j \in (0,n]\ e_j = (v_{j-1}, v_j) \in \mathbb{E}$.

Given a path π_{v_0,v_n}, we define its ***path segment*** $\hat{\pi}_{v_0,v_n}$ by simply ignoring v_0

and v_n from the beginning and end of its path sequence, i.e., $\langle e_1, v_1, \cdots, v_{n-1}, e_n \rangle$.

A ***path label function*** τ maps a path π or path segment $\hat{\pi}$ to a word by concatenating labels of the elements in its sequence order. Unless specifically mentioned, the label of each element (vertex or edge) is derived via $\lambda_v(v)$ and $\lambda_e(e)$. For example, from a to c, there is a path $\pi_{\text{a,c}}= \langle a, e_a, b, e_b, c \rangle$, where $a, c \in \mathcal{E}$, $b \in \mathcal{A}$, $e_a \in G$ and $e_b \in U$; its path label $\tau(\pi_{\text{a,c}}) = \mathcal{E} G \mathcal{A} U \mathcal{E}$, and its path segment label $\tau(\hat{\pi}_{\text{a,c}}) = G \mathcal{A} U$.

For ease of describing path patterns, for ancestry edges (used, wasGeneratedBy), i.e., $e_k = (v_i, v_j)$ with label $\lambda_e(e_k) = U$ or $\lambda_e(e_k) = G$, we introduce its virtual *inverse edge* ${e_k}^{-1} = (v_j, v_i)$ with the inverse label $\lambda_e({e_k}^{-1}) = U^{-1}$ or $\lambda_e({e_k}^{-1}) = G^{-1}$ respectively. A ***inverse path*** is defined by reversing the sequence, e.g., ${\pi_{\text{a,c}}}^{-1} = \langle c, {e_b}^{-1}, b, {e_a}^{-1}, a \rangle$, while $\tau({\pi_{\text{a,c}}}^{-1}) = \mathcal{E} U^{-1} \mathcal{A} G^{-1} \mathcal{E}$, $\tau({\hat{\pi}_{\text{a,c}}}^{-1}) = U^{-1} \mathcal{A} G^{-1}$.

Next we discuss PgSeg semantics and our rationale in detail.

5.3.1.1 Source ($\mathcal{V}_{src}$) and Destination Entities ($\mathcal{V}_{dst}$)

Provenance is about the entities. In a project, users know the committed snapshots (e.g., data files, scripts, and their metadata) better than the detailed processes generating them. When writing a PgSeg query, we assume the user believes $\mathcal{V}_{src}$ may be ancestry entities w.r.t. $\mathcal{V}_{dst}$. Then PgSeg reasons the connectivity among $\mathcal{V}_{src}$ and $\mathcal{V}_{dst}$, and shows other vertices and the generation process which the user may not know and be able to write query with. Note that the user may not know the existence order of entities either, so we allow $\mathcal{V}_{src}$ and $\mathcal{V}_{dst}$ to overlap, and even

be identical. In the latter case, the user could be a program [26] and not familiar with the generation process at all.

5.3.1.2 Induced Vertices $\mathcal{V}_{ind}$

Given $\mathcal{V}_{src}$ and $\mathcal{V}_{dst}$, we refer $\mathcal{V}_{ind}$ as the induced vertices (entities, activities and agents) contributing to the generation process. What vertices should be in $\mathcal{V}_{ind}$ is the core question to ask. It should reflect the generation process precisely and concisely in order to assist the user introspect the generation details to make decisions.

Prior work on inducing subgraphs from a set of vertices do not fit our needs. First, lineage query would generate all ancestors of $\mathcal{V}_{dst}$, which is not concise or even precise: siblings of $\mathcal{V}_{dst}$ and siblings of entities along the paths may be excluded as they do not have path from $\mathcal{V}_{dst}$ or to $\mathcal{V}_{src}$ in $\mathcal{G}$. Second, at another extreme, a provenance subgraph induced from some paths [22] or all paths [24] among vertices in $\mathcal{V}_{src} \cup \mathcal{V}_{dst}$ will only include vertices on the paths, thus exclude other contributing ancestors for $\mathcal{V}_{dst}$. Moreover, quantitative techniques used in other domains other than provenance cannot be applied directly either, such as keyword search over graph data techniques [123] which also do not assume that users have full knowledge of the graph, and let users use keywords to match vertices and then induce connected subgraph among keyword vertices. However, the techniques often use tree structures (e.g., least common ancestor, Steiner tree) connecting $\mathcal{V}_{src} \cup \mathcal{V}_{dst}$ and are not aware of provenance domain knowledge, thus cannot reflect the ancestry

relationships precisely.

Instead of defining $\mathcal{V}_{ind}$ quantitatively, we define PgSeg qualitatively by a set of domain rules: *(a)* to be precise, PgSeg includes other participating vertices not in the lineage and not in the paths among $\mathcal{V}_{src} \cup \mathcal{V}_{dst}$; *(b)* to be concise, PgSeg utilizes the path shapes between $\mathcal{V}_{src}$ and $\mathcal{V}_{dst}$ given by the users as a heuristic to filter the ancestry lineage subgraph. We define and categorize the rules that generate subsets of $\mathcal{V}_{ind}$ as follows:

1. *Vertices on Direct Path ($\mathcal{V}_{ind}^{C_1}$):* Activities and entities along any direct path π_{v_j,v_i} between an entity $v_i \in \mathcal{V}_{src}$ and an entity $v_j \in \mathcal{V}_{dst}$ are the most important ancestry information. It helps the users answer classic provenance questions, such as reachability, i.e., whether there exists a path; workflow steps, i.e., if there is a path, what activities occurred. We refer entities and activities on such direct path as $\mathcal{V}_{ind}^{C_1}$, which is defined as follows:

$$\mathcal{V}_{ind}^{C_1} = \bigcup_{v_i \in \mathcal{V}_{src}, v_j \in \mathcal{V}_{dst}} \{v_k | \ \exists_{\pi_{v_j,v_i}} \ v_k \in \hat{\pi}_{v_j,v_i}\}$$

   Note not only the shortest path are of interest, but all such path π_{v_j,v_i} in the DAG $\mathcal{G}$ should be derived.

2. *Vertices on Similar Path ($\mathcal{V}_{ind}^{C_2}$):* Though $\mathcal{V}_{ind}^{C_1}$ is important, due to the partial knowledge of the user, just considering the direct paths may miss important ancestry information including: *a)* the entities generated together with $\mathcal{V}_{dst}$, *b)* the entities used together with $\mathcal{V}_{src}$, and *c)* more importantly, other entities and activities which are not on the direct path, but contribute to the derivations. The contributing vertices are particularly relevant to the query in our

context, as data analytics project consists of many back-and-forth repetitive and similar steps, such as preparing data in alternative ways, adjusting model templates, and evaluating experiments.

To define the induction scope, on one hand, all ancestors w.r.t. $\mathcal{V}_{dst}$ in the lineage subgraph would be returned, however it is very verbose and not concise to interpret. On the other hand, it is also difficult to let the user specify all the details of what should/should not be returned. Here we use a heuristic: *induce ancestors which are not on the direct path but contribute to $\mathcal{V}_{dst}$ in a similar way, i.e., path labels from $\mathcal{V}_{ind}^{C_2}$ to $\mathcal{V}_{dst}$ are the same with some directed path from $\mathcal{V}_{src}$.* In other words, one can think it is similar to a radius concept [26] to slice the ancestry subgraph w.r.t. $\mathcal{V}_{dst}$, but the radius is not measured by how many hops away from $\mathcal{V}_{dst}$ but by path patterns between both $\mathcal{V}_{dst}$ and $\mathcal{V}_{src}$ entities that are specified by the user query. Next we first formulate the path pattern in a context free language [124], L(SimProv), then $\mathcal{V}_{ind}^{C_2}$ can be defined as a $L-$constrained reachability query from $\mathcal{V}_{src}$ via $\mathcal{V}_{dst}$ over $\mathcal{G}$, only accepting path labels in the language.

A ***context-free* grammar** (CFG) over a provenance graph $\mathcal{G}$ and a PgSeg query Q is a 6-tuple $(\Sigma, N, P, S, \mathcal{G}, Q)$, where

- $\Sigma = \{\mathcal{E},\mathcal{A},\mathcal{U}\} \cup \{U,G,S,A,D\} \cup \mathcal{V}_{dst}$ is the alphabet consisting of vertex labels, edge labels in $\mathcal{G}$ and $\mathcal{V}_{dst}$ vertex identifiers (e.g., `id` in Neo4j) in Q
- N is a set of non-terminals, and S is the start symbol.
- P is the set of production rules. Each production rule in the form of

$l \rightarrow (\Sigma \cup N)*$ defines an acceptable way of concatenations of the RHS words for the LHS non-terminal l.

Given a CFG and a non-terminal $l_i \in N$ as the start symbol, a ***context-free* language** (CFL), $L(l_i)$, is defined as the set of all finite words over Σ by applying its production rules.

The following CFG defines a language $L(\textsc{SimProv})$ that describes the heuristic path segment pattern for the induced vertex set. The production rules expand from some $v_j \in \mathcal{V}_{dst}$ both ways to reach v_i and v_k, such that the concatenated path π_{v_i,v_k} has the destination v_j in the middle.

$$\begin{aligned} \textsc{SimProv} \rightarrow \quad & G^{-1}\mathcal{E}\ \textsc{SimProv}\ \mathcal{E}G \\ | \quad & U^{-1}\mathcal{A}\ \textsc{SimProv}\ \mathcal{A}U \\ | \quad & G^{-1}v_j G \qquad\qquad \forall v_j \in \mathcal{V}_{dst} \end{aligned}$$

Now we can use $L(\textsc{SimProv})$ to define $\mathcal{V}_{ind}^{C_2}$ accordingly: for any vertex v_k in $\mathcal{V}_{ind}^{C_2}$, there should be at least a path from a $v_i \in \mathcal{V}_{src}$ going through a $v_j \in \mathcal{V}_{dst}$ and v_k then reaching some vertex v_t, such that the path segment label $\tau(\hat{\pi}_{v_i,v_t})$ is a word in $L(\textsc{SimProv})$:

$$\mathcal{V}_{ind}^{C_2} = \bigcup_{v_i \in \mathcal{V}_{src}} \{v_k |\ \exists_{\pi_{v_i,v_t}}\ \tau(\hat{\pi}_{v_i,v_t}) \in L(\textsc{SimProv}) \wedge v_k \in \pi_{v_i,v_t}\}$$

Using CFG allows us to express the heuristic properly. Note that $L(\textsc{SimProv})$ cannot be described by regular expressions over the path(segment) label, as it can be viewed as a palindrome language [124]. Moreover, it allows us to extend the query easily by using other label functions, for example, instead of

$\lambda_v(v)$ and $\lambda_e(e)$ whose domains are PROV types, using property value $\sigma(v, p_i)$ or $\omega(e, p_j)$ in $\mathcal{G}$ allows us to describe interesting constraints, e.g., the induced path should use the same commands as the path from $\mathcal{V}_{src}$ to $\mathcal{V}_{dst}$, or the matched entities on both sides of the path should be attributed to the same agent. For example, the former case can simply modify the second production rule in the CFG as follows:

$$U^{-1}\ \sigma(a_i, p_0)\ \textsc{SimProv}\ \sigma(a_j, p_0)\ U$$

$$\text{s.t. } a_i, a_j \in \mathcal{A} \wedge p_0 = \text{`command'} \wedge \sigma(a_i, p_0) = \sigma(a_j, p_0)$$

This is a powerful generalization that allows PgSeg to constrain induction scope by describing repetitiveness and similarily ancestry paths at a very fine granularity.

3. *Entities Generated By Activities on Path ($\mathcal{V}_{ind}^{C_3}$):* As mentioned earlier, the sibling entities generated together with $\mathcal{V}_{dst}$ may not be induced from directed paths. The same applies to the siblings of entities induced in $\mathcal{V}_{ind}^{C_1}$ and $\mathcal{V}_{ind}^{C_2}$, if the siblings do not have paths to $\mathcal{V}_{dst}$. We refer to those entities as $\mathcal{V}_{ind}^{C_3}$ and define it as:

$$\mathcal{V}_{ind}^{C_3} = \bigcup_{v_i \in V'} \{v_\epsilon |\ (v_\epsilon, v_i) \in G \wedge v_\epsilon \notin \mathcal{V}_{ind}^{C_1} \cup \mathcal{V}_{ind}^{C_2}\}$$

$$\text{where } V' = (\mathcal{V}_{ind}^{C_1} \cup \mathcal{V}_{ind}^{C_2}) \cap \mathcal{A}$$

4. *Involved Agents ($\mathcal{V}_{ind}^{C_4}$):* Finally, the agents in the provenance graph may be important in some situations, e.g., from the derivation, identify who makes a mistake, like `git blame` in version control settings. On a provenance graph

$\mathcal{G}$, agents can be derived easily:

$$\mathcal{V}_{ind}^{C_4} = \bigcup_{v_i \in V'} \{v_u | \ v_u \in \mathcal{U} \wedge (v_i, v_u) \in S \cup A\}$$

$$\text{where } V' = \mathcal{V}_{src} \cup \mathcal{V}_{dst} \cup \mathcal{V}_{ind}^{C_1} \cup \mathcal{V}_{ind}^{C_2} \cup \mathcal{V}_{ind}^{C_3}$$

Note we not only induce agents of $\mathcal{V}_{dst}$ and $\mathcal{V}_{src}$, but also other induced vertices.

5.3.1.3 Boundary Criteria $\mathcal{B}$

On the induced subgraph, besides path shapes, the segmentation operator should be able to express users' logical boundaries when asking the ancestry queries. It is particulary useful in an interactive setting once the user examines the returned induced subgraph and wants to make adjustments. We categorize the boundary criteria support as a) exclusion constraints and b) expansion specifications.

First, boundaries would be constraints to exclude some parts of the graph, such as limiting ownership (authorship) (**who**), time intervals (**when**), project steps (particular version, file path patterns) (**where**), understanding capabilities (neighborhood size) (**what**), etc. Most of the boundaries can be defined as boolean functions mapping from a vertex or edge to true or false, i.e., $b_v(v) : \mathbb{V} \mapsto \{0, 1\}$, $b_e(e) : \mathbb{E} \mapsto \{0, 1\}$, which can be incorporated easily to the CFG framework for subgraph induction. We define the exclusion boundary criteria as two sets of boolean functions ($\mathcal{B}_v$ for vertices and $\mathcal{B}_e$ for edges), which could be provided by the system or defined by the user. Then the labeling function used for defining $\mathcal{V}_{ind}$ would be

adjusted by applying the boundary criteria as follows:

$$\mathcal{F}_v = \begin{cases} \lambda_v(v) & \bigwedge_{b_i \in \mathcal{B}_v} b_i(v) = 1 \\ \varepsilon & \text{otherwise} \end{cases}, \quad \mathcal{F}_e = \begin{cases} \lambda_e(e) & \bigwedge_{b_i \in \mathcal{B}_e} b_i(e) = 1 \\ \varepsilon & \text{otherwise} \end{cases}$$

In other words, a vertex or an edge that satisfies all exclusion boundary conditions, is mapped to its original label. Otherwise the empty word is used as its label, so that the membership of paths having that vertex to $L(\textsc{SimProv})$ would become invalid.

Second, instead of exclusion constraints, the user may wish to expand the induced subgraph. We allow the users to specify expansion criteria, $\mathcal{B}_x = \{b_x(V_x, k)\}$, denoting including paths which are k activities away from entities in $V_x \subseteq \mathcal{V}_{ind}$.

5.3.1.4 Discussion

Validness of provenance graph is an important constraint [26, 119]. In our system, the PgSeg operator does not introduce new vertices or edge. As long as the original provenance graph is valid, the induced subgraph is valid. However, at query time, the boundaries criteria could possibly let the operator result exclude important vertices. As an interactive system, we leave it to the user to adjust the vertex set of interest and boundary criteria in their queries.

5.3.2 Query Evaluation

5.3.2.1 Overview: Two-Step Approach

Given a PgSeg($\mathcal{V}_{src}$,$\mathcal{V}_{dst}$,$\mathcal{B}$) query, we separate the query evaluation into two steps: 1) **induce**: induce $\mathcal{V}_{ind}$ and construct the induced graph $\mathcal{S}$ using $\mathcal{V}_{src}$and $\mathcal{V}_{dst}$, 2) **adjust**: apply $\mathcal{B}$ interactively to filter induced vertices or retrieve more vertices from the property graph store backend. The rationale of the two-step approach is that the operator is designed for the users with partial knowledge who are willing to understand a local neighborhood in the provenance graph. Any induction heuristic applied would be unlikely to match the user's implicit interests and would require back-and-forth explorations.

In the rest of the discussion, we assume *a)* the provenance graph is stored in a backend property graph store, with constant time complexity to access arbitrary vertex and arbitrary edge by corresponding primary identifier; *b)* given a vertex, both its incoming and outgoing edges can be accessed equally, with linear time complexity w.r.t. the in- or out-degree. In our implementation (Section 5.5), we use Neo4j as our storage backend, which satisfies the conditions – both nodes and relationships are accessed via their `id`.

5.3.2.2 Induce Step

Given $\mathcal{V}_{src}$ and $\mathcal{V}_{dst}$, PgSeg induces $\mathcal{V}_{ind}$ which consists of four categories. We mainly focus our discussion on inducing vertices on direct and similar paths,

as the other two types, i.e., sibling entities and related agents can be derived in a straightforward manner by scanning 1-hop neighborhoods of the first two sets of results.

Cypher:

The definition of vertices on similar path requires a context-free language, and cannot be expressed by a regular language. When developing the system, we realize it can be decomposed into two regular language path segments, and express the query using path variables [125, 126]. We handcraft a Cypher query shown in Query 5.1. The query uses $\mathcal{V}_{src}$ (`b`) and $\mathcal{V}_{dst}$ (`e1`) to return all directed paths $\mathcal{V}_{ind}^{C_1}$ via path variables (`p1`), and uses Cypher `with` clause to hold the results. The second `match` finds the other half side of the SimProv via path variable `p2` which then joins with `p1` to

```
match p1=(b:E)<-[:U|G*]-(e1:E)
with p1
where id(b) in [0,1] and id(e1) in [30,31]
match p2=(c:E)<-[:U|G*]-(e2:E)
where id(e2) in [30,31] and
      extract(x in nodes(p1) | labels(x)[0])
           = extract(x in nodes(p2) | labels(x)[0]) and
      extract(x in relationships(p1) | type(x))
           = extract(x in relationships(p2) | type(x))
return p2;
```

Query 5.1: Cypher Q_1 for L(SimProv), $\mathcal{V}_{src}$= $\{0, 1\}$, $\mathcal{V}_{dst}$= $\{30, 31\}$

compare the node-by-node and edge-by-edge conditions to induce $\mathcal{V}_{ind}^{C_2}$. If we do not need to check properties, then we can use `length(p1) = length(p2)` instead of the two `extract` clauses. However, as shown later in the evaluation (Section 5.6), Neo4j takes more than 12 hours to return results for even very small graphs with about a hundred vertices. Note that regular pattern queries (RPQ) with path variables are not supported well in modern graph query languages and graph database [111,125], we develop our own PgSeg algorithm for provenance graphs.

CFL-reachability:

Given a vertex v and a CFL L, the problem of finding all vertieces $\{u\}$ such that there is a path $\pi_{v,u}$ with label $\tau(\pi_{v,u}) \in L$ is often referred as *single source CFL-reachability* (CFLR) problem or *single source L-Transitive Closure* problem [127, 128]. The *all-pairs* version, which aims to find all such pairs of vertices connected by a L path of the problem, has the same complexity. As $\mathcal{V}_{src}$ would be all vertices, we do not distinguish between the two in the rest of the discussion. Though the problem has been first studied in our community [127], there is little follow up and support in the context of modern graph databases. CFLR finds its main application in programming analysis and is recognized as a general formulation method for many program analysis tasks [128]. On graph representations of programs, program analysis tasks such as program slicing and pointer analysis, can be described in a CFL to specify path patterns.

State of the art CFLR algorithm [129] solves the problem in $O(n^3/log(n))$ time and $O(n^2)$ space w.r.t. the number of vertices in the graphs. It is based on a classic

$$
\begin{array}{llllll}
r_0 : \textsc{Qd} \rightarrow & v_j & \forall v_j \in \mathcal{V}_{dst} & & & \\
r_1 : \textsc{Lg} \rightarrow & G^{-1}\ \textsc{Qd} & r_3 : \textsc{La} \rightarrow & \mathcal{A}\ \textsc{Rg} & r_6 : \textsc{Ru} \rightarrow & \textsc{Lu}\ U \\
\mid & G^{-1}\ \textsc{Re} & r_4 : \textsc{Ra} \rightarrow & \textsc{La}\ \mathcal{A} & r_7 : \textsc{Le} \rightarrow & \mathcal{E}\ \textsc{Ru} \\
r_2 : \textsc{Rg} \rightarrow & \textsc{Lg}\ G & r_5 : \textsc{Lu} \rightarrow & U^{-1}\ \textsc{Ra} & r_8 : \textsc{Re} \rightarrow & \textsc{Le}\ \mathcal{E}
\end{array}
$$

Figure 5.6: SimProv Normal Form, SimProv $\rightarrow$ Re. Lg $\subseteq \mathcal{A} \times \mathcal{E}$; Rg, La, Ra $\subseteq \mathcal{A} \times \mathcal{A}$; Lu $\subseteq \mathcal{E} \times \mathcal{A}$; Ru, Le, Re, Qd $\subseteq \mathcal{E} \times \mathcal{E}$.

cubic time dynamic programming scheme [128, 130] which derives production facts non-repetitively via graph traversal, and uses the method of four Russians [131] during the traversal. In the rest of the discussion, we refer it as CflrB.

We first describe the algorithm briefly and then present improvement techniques for L(SimProv) on provenance graphs. Given a CFG, it works on its normal form [124], where each production has at most two RHS symbols, i.e., $A \rightarrow BC$ or $A \rightarrow B$. We show the SimProv normal form in Figure 5.6. At a high level, the algorithm traverses the graph and uses grammar as a guide to find new production facts $N(i, j)$, where N is a LHS nonterminal, i, j are graph vertices, and the found fact $N(i, j)$ denotes that there is a path from i to j whose path label satisfies N. To elaborate, similar to BFS, it uses a worklist W (queue) to track newly found fact $N(i, j)$ and a *fast set* data structure H with time complexity $O(n/log(n))$ for set diff/union and $O(1)$ for insert to memorize found facts. In the beginning, all facts $F(i, j)$ from all single RHS symbol rules $F \rightarrow A$ are enqueued. In SimProv case (r_0 in Figure 5.6), each $v_j \in \mathcal{V}_{dst}$ is added to W as Qd(v_j, v_j). From W, the algorithm processes one fact $F(i, j)$ at a time until W is empty. When processing a dequeued

fact $F(i,j)$, if F appears in any rule in the following cases:

$$N(i,j) \rightarrow F(i,j)$$

$$N(i,v) \rightarrow F(i,j)A(j,v)$$

$$N(u,j) \rightarrow A(u,i)F(i,j)$$

the new LHS fact $N(i,v)$ is derived by set diff $\{v \in A(j,v)\} \setminus \{v \in N(i,v)\}$ or $N(u,j)$ by $\{u \in A(u,i)\} \setminus \{u \in N(u,j)\}$ in H. Then the new facts of N are added to H to avoid repetition and W to explore it later. Once W is empty, the start symbol L facts $L(i,j)$ in H include all vertices pairs (i,j) which have a path with label that satisfies L. If path is needed, a parent table would be used similar to BFS. In SimProv(Figure. 5.6), the start symbol is Re, $\forall v_i \in \mathcal{V}_{src}$, $\text{Re}(v_i, v_t)$ facts include all v_t, s.t. between them there is $\tau(\hat{\pi}_{\text{i,t}}) \in L(\text{SimProv})$.

L(SimProv)**-reachability on** PROV:

Next we study the performance of CflrB for SimProv on a PROV graph, and show the *fast set* method is not suitable for PROV graph. Then we further explore grammar and PROV graph properties, instead of normal form, we rewrite the grammar to allow several pruning strategies and propose a linear-time algorithm if $|\mathcal{V}_{dst}|$ can be viewed as a constant.

Lemma 4 CflrB *solves* L(SimProv)*-reachability on a* PROV *graph in* $O(\frac{|\mathcal{A}||\mathcal{E}|^2}{\log|\mathcal{A}|} + \frac{|\mathcal{E}||\mathcal{A}|^2}{\log|\mathcal{E}|})$ *time if using* fast set. *Otherwise, it solves it in* $O(|G||\mathcal{E}| + |U||\mathcal{A}|)$ *time.*

Proof 3 *On* SimProv *normal form (Figure 5.6), for* $i \in [1,8]$, CflrB *derives* r_i *LHS facts by a* r_{i-1} *LHS fact dequeued from* W *(Note it also derives* r_1 *from* r_8*). For* $i \in \{1,2\}$, $r_i(u,v)$ *uses* G *edges in the graph during the derivation, e.g., from*

r_8 *LHS* $\textsc{Re}$ *to* $r_1 : \textsc{Lg}(u,v) \rightarrow G^{-1}(u,k)\ \textsc{Re}(k,v)$. *As* $\textsc{Re}(k,v)$ *can only be in the worklist* W *once, we can see that each 3-tuple* (u,k,v) *is formed only once on the RHS and there are at most* $|G||\mathcal{E}|$ *of such 3-tuples. To make sure* $\textsc{Lg}(u,v)$ *is not found before,* H *is checked. If not using fast set but a* $O(1)$ *time procedure for each instance* (u,k,v)*, then it takes* $O(|G||\mathcal{E}|)$ *to produce the LHS; on the other hand, if using a* fast set *on* u'*s domain* $\mathcal{A}$ *for each* u*, for each* $\textsc{Re}(k,v)$*,* $O(\frac{|\mathcal{A}|}{\log|\mathcal{A}|})$ *time is required, thus it takes* $O(\frac{|\mathcal{A}||\mathcal{E}|^2}{\log|\mathcal{A}|})$ *in total. Applying similar analysis on* r_5 *and* r_6 *using* U *to derive new facts, we can see it takes* $O(\frac{|\mathcal{E}||\mathcal{A}|^2}{\log|\mathcal{E}|})$ *with fast set and* $O(|U||\mathcal{A}|)$ *without fast set. Finally* r_3, r_4 *and* r_7, r_8 *can be viewed as following a vertex self-loop edge and does not affect the complexity result.*

In our context, the PROV graph is often sparse, and both the numbers of entities that an activity uses and generates can be viewed as a small constant, however the domain size of $\mathcal{A}$ and $\mathcal{E}$ are potentially large. The lemma shows a quadratic time scheme for $L(\textsc{SimProv})$-reachability if we can view the average in/out degree as a constant. Note that the quadratic time complexity is not surprising, as SimProv is a linear CFG, i.e., there is at most one nonterminal on RHS of each production rule. The CFLR time complexity for any linear grammar on general graphs $G(V,E)$ have been shown in theory as $O(|V||E|)$ by a transformation to general transitive closures [127].

Rewriting SimProv:

Most CFLR algorithms require the normal form mentioned earlier. However, under the normal form, it **a)** introduces more worklist entries, and **b)** misses important

$$
\begin{aligned}
r_1' : \textsc{Ee} \rightarrow \quad & v_j \qquad \forall v_j \in \mathcal{V}_{dst} & r_2' : \textsc{Aa} \rightarrow \quad & G^{-1}\ \textsc{Ee}\ G \\
| \quad & U^{-1}\ \textsc{Aa}\ U & | \quad & \mathcal{A}\ \textsc{Aa}\ \mathcal{A} \\
| \quad & \mathcal{E}\ \textsc{Ee}\ \mathcal{E} & &
\end{aligned}
$$

Figure 5.7: Proposed SimProv Rewriting, SimProv $\rightarrow$ Ee. Aa $\subseteq \mathcal{A} \times \mathcal{A}$; Ee $\subseteq \mathcal{E} \times \mathcal{E}$.

grammar properties. Instead, we rewrite SimProv as shown in Figure 5.7, and propose SimProvAlg and SimProvTst by adjusting CflrB. Comparing with standard normal forms, r_1' and r_2' have more than two RHS symbols. SimProvAlg utilizes the rewritten grammar and PROV-graph properties to improve CflrB. Moreover, SimProvTst solves L(SimProv)-reachability on a PROV graph in linear time and sublinear space if viewing $|\mathcal{V}_{dst}|$ as constant. The properties of the rewritten grammar and how SimProvAlg and SimProvTst utilize them are described below, which can be used in other CFLR problems:

1. Reduction for Worklist tuples: Note that r_2' in Figure 5.7 combines rules r_1, r_2 in the normal form, i.e., $\textsc{Rg}(a_1, a_2) \rightarrow \textsc{Lg}(a_1, e_2) G(e_2, a_2)$ and $\textsc{Lg}(a_1, e_2) \rightarrow G^{-1}(a_1, e_1)\textsc{Re}(e_1, e_2)$, is derived by

$$\textsc{Aa}(a_1, a_2) \rightarrow G^{-1}(a_1, e_1)\ \textsc{Ee}(e_1, e_2)\ G(e_2, a_2)$$

Instead of enqueue $\textsc{Lg}(a_1, e_2)$ and then $\textsc{Rg}(a_1, a_2)$, SimProvAlg adds $\textsc{Aa}(a_1, a_2)$ to W directly. In the previous normal form, there may be other cases that can also derive $\textsc{Aa}(a_1, a_2)$, i.e., in presence of $\textsc{Lg}(a_1, e_k)$ and $G(e_k, a_2)$. In the worst case, CflrB enqueued $|\mathcal{E}|$ number of Lg in W which later find the

same fact $\textsc{Rg}(a_1, a_2)$. It's worth mentioning that in SimProvAlg because $\textsc{Aa}(a_1, a_2)$ now would be derived by many $Ee(e_i, e_j)$ in r'_2, before adding it to W, we need to check if it is already in W. In SimProvAlg, we use two pairs of bitmaps for Ee and Aa for W and H respectively, the space cost is $O(\frac{|\mathcal{E}|^2}{\log|\mathcal{E}|} + \frac{|\mathcal{A}|^2}{\log|\mathcal{A}|})$. Compressed bitmaps would be used to improve space usage at the cost of non-constant time random read/write.

2. Symmetric property: In the rewritten grammar, both nonterminals Ee and Aa are symmetric, i.e., $\textsc{Ee}(e_i, e_j)$ implies $\textsc{Ee}(e_j, e_i)$, $\textsc{Aa}(a_i, a_j)$ implies $\textsc{Aa}(a_j, a_i)$, which is not held in normal forms. Intuitively $\textsc{Ee}(e_1, e_2)$ means some path label from e_1 to $v_j \in \mathcal{V}_{dst}$ is the same with some path label from e_2 to v_j. Using symmetric property, in SimProvAlg, we can use a straightforward **pruning strategy**: only process (e_i, e_j) in both H and W if $\texttt{id}(e_i) \leq \texttt{id}(e_j)$, and (a_i, a_j) if $\texttt{id}(a_i) \leq \texttt{id}(a_j)$; and an **early stopping rule**: for any $\textsc{Aa}(a_i, a_j)$ that both a_i's and a_j's order of being is before all PgSeg query $\mathcal{V}_{src}$ entities, we do not need to proceed further. Note the early stopping rule is SimProv and PROV-graph specific, while solving general CFLR, even in the *single-source* version, cannot take source information and we need to evaluate until the end. Though both strategies used by SimProvAlg do not improve the worst-case time complexity, as shown later in empirical studies (Section 5.6), they are very useful in realistic PROV graphs.

3. Transitive property: By definition SimProv does not have transitivity, i.e., given $\textsc{Ee}(e_1, e2)$ and $\textsc{Ee}(e_2, e_3)$, it does not imply $\textsc{Ee}(e_1, e_3)$. This is because

a PgSeg query allows multiple $\mathcal{V}_{dst}$, $\textsc{Ee}(e_1, e_2)$ and $\textsc{Ee}(e_2, e_3)$ may be found due to different $v_j \in \mathcal{V}_{dst}$. However, if we evaluate $v_j \in \mathcal{V}_{dst}$ separately, then Ee and Aa have transitivity, which leads to a linear algorithm SimProvTst for each v_j: instead of maintaining $\textsc{Ee}(e_i, e_j)$ or $\textsc{Aa}(a_i, a_j)$ tuples in H and W, we can use a set $[e]_m$ or $[a]_n$ to represent an equivalence class at iteration m or n where any pair in the set is a fact of Ee or Aa respectively. If at iteration m, the current W holds a set $[e]_m$, then $\textsc{Aa}(a, a) \rightarrow G^{-1}(a, e)\textsc{Ee}(e, e)$ is used to infer the next W (a set $[a]_{m+1}$); otherwise, W must hold a set $[a]_m$, then $\textsc{Ee}(e, e) \rightarrow U^{-1}(e, a)\textsc{Aa}(a, a)$ is used to infer next equivalence class $[e]_{m+1}$ as the next W. In the first case, as there are at most $|G|$ possible (a, e) tuples, the step takes $O(|G|)$ time; in the later case, similarly the step takes $O(|U|)$ time. The algorithm returns vertices in any equivalence classes $[v_i]_m$, s.t. $v_i \in \mathcal{V}_{src}$. Overall, because there are multiple $\mathcal{V}_{dst}$ vertices, the algorithm runs in $O(|\mathcal{V}_{dst}||G|+|\mathcal{V}_{dst}||U|)$ time and $O(\frac{|\mathcal{E}|}{log|\mathcal{E}|}+\frac{|\mathcal{A}|}{log|\mathcal{A}|})$ space. The early stop rule can applied as well, instead of a pair of activities in SimProvAlg, in SimProvTst all activities in an equivalent class $[a]_m$ are compared with entities in $\mathcal{V}_{src}$ in terms of their order of being; while the pruning strategy is not necessary, as all pairs are represented compactly in an equivalent class.

Theorem 2 SimProvTst *solves* $L(\textsc{SimProv})$*-reachability on a* PROV *graph in* $O(|G| + |U|)$ *time, if* $|\mathcal{V}_{dst}|$ *can be viewed as a constant.*

5.3.2.3 Adjust Step

Once the induced graph $\mathcal{S}(\mathbb{V}_\mathcal{S}, \mathbb{E}_\mathcal{S})$ is derived, the adjustment step applies boundary criteria to filter existing vertices and retrieve more vertices. Comparing with induction step, applying boundary criteria is rather straightforward. For exclusion constraints $\mathcal{B}_v$ and $\mathcal{B}_e$, we apply them on vertices and edges in $\mathcal{S}(\mathbb{V}_\mathcal{S}, \mathbb{E}_\mathcal{S})$ linearly if present. For $\mathcal{B}_x$, we traverse the backend store with specified entities for $2k$ hops through $G^{-1}$and U^{-1}edges to their ancestry activities and entities. To support back and forth interaction, we cache the induced graph instead of inducing multiple times. We expect k is small constant in our context as the generated graph is for human users to interpret, otherwise, a reachability index is needed.

For other purposes where the two-step approaches are not ideal, the exclusion constraints $\mathcal{B}_v$ and $\mathcal{B}_e$ can be evaluated together using CflrB, SimProvAlg and SimProvTst with small modifications. In CflrB the label function $\mathcal{F}_v$ of $\mathcal{B}_v$ can be applied at r_0, r_3, r_4, r_7, r_8 on $\mathcal{A}$ or $\mathcal{E}$, while $\mathcal{F}_e$ of $\mathcal{B}_e$ can be applied at rest of the rules involving U and G. For SimProvAlg and SimProvTst, $\mathcal{F}_v$ and $\mathcal{F}_e$ can be applied together at r'_1, r'_2.

5.3.2.4 Discussion

We focus on the ad-hoc query evaluation schemes. As of now, the granularity of provenance is at the level of command executions, the number of activities are constrained by project members' work rate. In case when the PROV graph becomes extremely large, indexing techniques and incremental algorithms are more practical.

5.4 Summarization Operation

In a collaborative analytics project, collected provenance graph of the repetitive modeling trails records verbose steps of various pipelines and reflects different roles and work routines of the team members. Using PgSeg, the users can navigate to their segments of interest, which may be about similar pipeline steps. For example, the query result of a single PgSeg($\mathcal{V}_{src}$,$\mathcal{V}_{dst}$,$\mathcal{B}$), e.g., '*yesterday's input data and prediction result*', shows a pipeline subgraph $\mathcal{S}_1$ about how $\mathcal{V}_{dst}$ (*prediction result*) was derived from $\mathcal{V}_{src}$ (*input data*) together with other induced $\mathcal{V}_{ind}$ (e.g., modeling steps). As there is no skeleton for the pipeline, given a different tuple ($\mathcal{V}^2_{src}$, $\mathcal{V}^2_{dst}$, $\mathcal{B}^2$) to get another segment $\mathcal{S}_2$, e.g., '*today's input data and model result*', the pipeline subgraph $\mathcal{S}_2$ may or may not be the same as $\mathcal{S}_1$. Given a set of segments, our design goal of PgSum is to produce a precise and concise provenance summary graph, Psg, which will not only allow the users to see commonality among those segments of interest (e.g., *yesterday's and today's pipelines are almost the same*), but also let them understand alternative routines (e.g., *an old step excluded in today's pipeline*). Though no workflow skeleton is defined, with that ability, Psg would enable the users to reason about prospective provenance in evolving workflows of analytics projects.

One way to combine PgSeg segment graphs is to use context-free graph grammars (CFGG) [20] which are able to capture recursive substructures. However without a predefined workflow skeleton CFGG, and due to the workflow noise resulting from the nature of analytics workload, inferring a minimum CFGG from a set of

subgraphs is not only an intractable problem, but also possibly leads to complex graph grammars that are more difficult to be understood by the users [132]. Instead, we view it as a graph summarization task by grouping vertices and edges in the set of segments to a PSG.

5.4.1 Semantics of Summarization (PGSUM)

Although there are many graph summarization schemes proposed over the years [118], they are neither aware of provenance domain desiderata [25] nor the meaning of PGSEG segments. Most of the work focuses on finding smaller representations for a very large graph by methods such as compression [133], attribute-aggregation [116] and bisimulation [134]; while there are a few works aiming at combining a set of query-returned trees [135] or graphs [117] to form a compact representation. Our PGSUM operator falls into the latter category, and is tailored for PGSEG segments which consist of similar or alternative steps among a set of entities of interest. A PGSUM query is designed to take a 2-tuple $(\mathcal{K}, \mathcal{R}_k)$ as input which describes the level of details of vertices and constrains the rigidness of the provenance; then it outputs a minimum provenance summary graph (PSG).

5.4.1.1 Property Aggregations & Provenance Types

Given a set of segments $\mathbb{S}$, each $\mathcal{S}_i$ of which is a PGSEG result, to combine vertices and edges across the segments, we first introduce two concepts: a) *property aggregation* $(\mathcal{K})$ and b) *provenance type* $(\mathcal{R}_k)$, which PGSUM takes as input and

allow the user to obfuscate vertex details and retain structural constraints.

Property Aggregation ($\mathcal{K}$):

Similar to an attribute-aggregation summarization query on a general graph [116], depending on the granularity level of interest, not all the details of a vertex are interesting to the user and some properties should be omitted, so that they can be combined together. For example, the user may neither care who performs an activity, nor an activity's detailed configuration; in the former case, all agent type vertices regardless of their property values (e.g., name) should be indistinguishable and in the latter case, the same activity type vertices even with different configuration settings (e.g., training parameters) in various PgSeg segments should be viewed as if the same thing (e.g., *a training activity*) has happened.

Formally, *property aggregation* $\mathcal{K}$ is a 3-tuple ($\mathcal{K}_{\mathcal{E}}$, $\mathcal{K}_{\mathcal{A}}$, $\mathcal{K}_{\mathcal{U}}$), where each of the tuple elements is a subset of the PROV graph property types, i.e., $\mathcal{K}_{\mathcal{E}}$,$\mathcal{K}_{\mathcal{A}}$,$\mathcal{K}_{\mathcal{U}}$$\subseteq$$\mathcal{P}$ (Definition 3). When used in a PgSum query, it discards other irrelevant properties for each vertex type, e.g., properties of entity $\mathcal{E}$ type in $\mathcal{P}\backslash\mathcal{K}_{\mathcal{E}}$ are ignored. For example, in Figure 5.5, $\mathcal{K}_{\mathcal{E}}$= {*'filename'*}, $\mathcal{K}_{\mathcal{A}}$= {*'command'*}, $\mathcal{K}_{\mathcal{U}}$= $\emptyset$, so properties such as version of the entity, details of an activity, names of the agents are ignored.

Provenance Type ($\mathcal{R}_k$):

In contrast with general-purpose graphs, in a provenance graph, the vertices with identical labels and property values may be very different [25]. For example, two identical activities that use different numbers of inputs or generate different entities should be considered different. In [25], Moreau proposes using a recursive definition

over a vertex's k-hop neighborhood to assign a vertex type for later aggregation. However, the definition ignores input/output degrees of vertices, and the recursive definition is exponential w.r.t. to k. It is worth mentioning that the former issue occurs in bisimulation-based method as well [134].

We extend the idea of preserving provenance meaning of a vertex and use the k-hop local neighborhood of a vertex to capture its *provenance type*: given a PgSeg segment $\mathcal{S}(\mathbb{V}_{\mathcal{S}}, \mathbb{E}_{\mathcal{S}})$, and a constant k, $k \geq 0$, provenance type $\mathcal{R}_k(v)$ is a function that maps a vertex $v \in \mathbb{V}_{\mathcal{S}}$ to its k-hop neighborhood subgraph in its segment $\mathcal{S}$, $\mathcal{R}_k \subseteq \mathcal{S}$. For example, in Figure 5.5, $k = 1$, thus provenance type of vertices is the 1-hop neighborhood, the vertices with label *'update'*, *'model'* and *'solver'* all have two different provenance types (marked as 't1', 't2').

Note one can generalize the definition of $\mathcal{R}_k(v)$ as a subgraph within $k-$hop neighborhood of v satisfying a pattern matching query, which has been proposed in [121] with application to entity matching where similar to provenance graphs, just using the vertex properties are not enough to represent the conceptual identity of a vertex.

Vertex Equivalence Relation ($\equiv_{\kappa}^{k}$):

Given a set of segments $\mathbb{S} = \{\mathcal{S}_i(\mathbb{V}_{\mathcal{S}i}, \mathbb{E}_{\mathcal{S}i})\}$, denoting the union of vertices as $\mathbb{V}_{\mathbb{S}} = \bigcup_i \mathbb{V}_{\mathcal{S}i}$, with the user specified property aggregation $\mathcal{K}$ and provenance type $\mathcal{R}_k$, we define a binary relation $\equiv_{\kappa}^{k}$ over $\mathbb{V}_{\mathbb{S}} \times \mathbb{V}_{\mathbb{S}}$, such that for each vertex pair $(v_i, v_j) \in \equiv_{\kappa}^{k}$:

a) vertex labels are the same, i.e., $\lambda_v(v_i) = \lambda_v(v_j)$,

b) all property values in $\mathcal{K}$ are equal, i.e., $\forall_{p \in \mathcal{K}}\ \sigma(v_i, p) = \sigma(v_j, p)$,

c) $\mathcal{R}_k(v_i)$ and $\mathcal{R}_k(v_j)$ are graph isomorphic w.r.t. the vertex label and properties in $\mathcal{K}$, i.e., there is a bijection f between $V_i \in \mathcal{R}_k(v_i)$ and $V_j \in \mathcal{R}_k(v_j)$, s.t., $f(v_m) = v_n$ if

1. $\lambda_v(v_m) = \lambda_v(v_n)$,
2. $\forall_{p \in \mathcal{K}}\ \sigma(v_m, p) = \sigma(v_n, p)$,
3. $\forall (v_m, v_t) \in E_i, (v_n, f(v_t)) \in E_j$.

It is easy to see that $\equiv_{\kappa}^{k}$ is an equivalence relation on $\mathbb{V}_{\mathbb{S}}$ by inspection. Using $\equiv_{\kappa}^{k}$, we can derive a partition $P^{\equiv_{\kappa}^{k}}$ of $\mathbb{V}_{\mathbb{S}}$, s.t., each set in the partition is an equivalence class by $\equiv_{\kappa}^{k}$, denoted by $[v]$, s.t., $[v_i] \cap [v_j] = \emptyset$ and $\bigcup_i [v_i] = \mathbb{V}_{\mathbb{S}}$. For each $[v]$, we can define its canonical label, e.g., the smallest vertex `id`, for comparing vertices.

In other words, vertices in each equivalence class $[v]$ by $\equiv_{\kappa}^{k}$ describe the homogeneous candidates which can be merged by PgSum. Its definition not only allows the users to specify *property aggregations* $\mathcal{K}$ to obfuscate unnecessary details in different resolutions, but also allows the users to set *provenance types* $\mathcal{R}_k$ in order to preserve local structures and ensure the meaning of provenance of a merged vertex.

5.4.1.2 Provenance Summary Graph (Psg)

Next, we introduce the output summary graph of PgSum operator, Psg.

Desiderata:

Due to the nature of provenance, the produced Psg should be *precise*, i.e., we should preserve paths that exist in one or more segments, at the same time, we should not introduce any path that does not exist in any segment. On the other hand, Psg

should be concise; the more vertices we can merge, the better summarization result it is considered to be. In addition, as a summary, to show the commonality and the rareness of a path among the segments, we annotate each edge with its appearance frequency in the segments.

Minimum PSG:

PGSUM combines segment vertices in their equivalence classes and ensures the paths in the output summary graph satisfy above conditions. Next we define a valid summary graph.

Given a set of segments $\mathbb{S} = \{\mathcal{S}_i(\mathbb{V}_{\mathcal{S}i}, \mathbb{E}_{\mathcal{S}i})\}$, and a PGSUM$(\mathcal{K}, \mathcal{R}_k)$ query, a *provenance summary graph*, PSG$(\mathcal{M}, E, \rho, \gamma)$, is a directed acyclic graph, where

- Each $\mu \in \mathcal{M}$ represents a subset of an equivalence class $\mu \subseteq [v]$ w.r.t. $\equiv_\kappa^k$ over $\mathbb{V}_\mathbb{S}$, and one segment vertex v can only be in one PSG vertex μ, i.e., $\forall \mu_m, \mu_n \in \mathcal{M}$, $\mu_m \cap \mu_n = \emptyset$; the vertex label function $\rho : \mathcal{M} \mapsto P^{\equiv_\kappa^k}$ maps a PSG vertex to its equivalence class;

- An edge $e_{m,n} = (\mu_m, \mu_n) \in E$ exists if there is a corresponding edge in some segment, i.e., $\exists_{\mathcal{S}_i}\ \mu_m \times \mu_n \cap \mathbb{E}_{\mathcal{S}i} \neq \emptyset$; the edge label function $\gamma : E \mapsto [0, 1]$ annotates the edge's frequencies over segments, i.e., $\gamma(e_{m,n}) = |\{\mathcal{S}_i | \mu_m \times \mu_n \cap \mathbb{E}_{\mathcal{S}i} \neq \emptyset\}| / |\mathbb{S}|$;

- There is a path $\pi_{m,n}$ from μ_m to μ_n ***iff*** $\exists_{\mathcal{S}_i} v_s \in \mu_m \cap \mathbb{V}_{\mathcal{S}i} \wedge v_t \in \mu_n \cap \mathbb{V}_{\mathcal{S}i}$, there is a path $\pi_{s,t}$ from v_s to v_t in $\mathcal{S}_i$, and their path labels are the same $\tau(\pi_{m,n}) = \tau(\pi_{s,t})$. Note that for PSG, we use equivalence classes' canonical label (e.g., smallest vertex `id`) as the vertex label in τ.

It is easy to see $\bigcup_i \mathcal{S}_i$ the union of all segments in $\mathbb{S}$ is a valid PSG. However, we are interested in a concise summary with fewer vertices. The best PSG one can get is the optimal solution of the following problem.

Problem 3 (Minimum PSG) *Given a set of segments* $\mathbb{S}$ *and a* PGSUM($\mathcal{K}$,$\mathcal{R}_k$) *query, find the* PSG($\mathcal{M}, E, \rho, \gamma$) *with minimum* $|\mathcal{M}|$.

5.4.1.3 Discussion

Though requiring all paths in PSG must exist in some segment may look strict and affect the compactness of the result, PGSUM operator allows using the property aggregation ($\mathcal{K}$) and provenance types ($\mathcal{R}_k$) to tune the compactness of PSG. Due to the rigidness and the utility of provenance, allowing paths that do not exist in any segment in the summary would cause misinterpretation of the provenance, thus would not be suitable for our context. In situations where extra paths in the summary graph is not an issue, problems with objectives such as minimizing the number of introduced extra paths, and minimizing the description length are interesting ones to be explored further.

5.4.2 Query Evaluation

Given $\mathbb{S}$= $\{\mathcal{S}_i(\mathbb{V}_{\mathcal{S}i}, \mathbb{E}_{\mathcal{S}i})\}$, after applying $\mathcal{K}$ and $\mathcal{R}_k$, PSG $g_0 = \bigcup_i \mathcal{S}_i$ is a labeled graph and contains all paths in segments; to find a smaller PSG, we have to merge vertices in $\mathbb{V}_{\mathbb{S}}$= $\bigcup_i \mathbb{V}_{\mathcal{S}i}$ in g_0 while keeping the PSG invariant, i.e., not introducing new paths.

In order to describe merging conditions, we describe *trace equivalence* relations in a PSG.

- in-trace equivalence ($\simeq^t_{in}$): If for every path $\pi_{\mathrm{a,u}}$ ending at $u \in \mathbb{V}_{\mathbb{S}}$, there is a path $\pi_{\mathrm{b,v}}$ ending at $v \in \mathbb{V}_{\mathbb{S}}$ with the same label, i.e., $\tau(\pi_{\mathrm{a,u}}) = \tau(\pi_{\mathrm{b,v}})$, we say u is in-trace dominated by v, denoted as $u \leq^t_{in} v$. The u and v are in-trace equivalent, written $u \simeq^t_{in} v$, iff $u \leq^t_{in} v \wedge v \leq^t_{in} u$.

- out-trace equivalence ($\simeq^t_{out}$): Similarly, if for every path starting at u, there is a path starting at v with the same label, then we say u is out-trace dominated by v, written $u \leq^t_{out} v$. u and v are out-trace equivalent, i.e., $u \simeq^t_{out} v$, iff $u \leq^t_{out} v \wedge v \leq^t_{out} u$.

Lemma 5 *Merging u to v does not introduce new paths, if and only if* 1) $u \simeq^t_{in} v$, *or* 2) $u \simeq^t_{out} v$, *or* 3) $u \leq^t_{in} v \wedge u \leq^t_{out} v$.

The lemma defines a partial order over the vertices in $\mathbb{V}_{\mathbb{S}}$. By applying the above lemma, we can merge vertices in a PSG greedily until no such pair exist, then we derive a minimal PSG. However, the problem of checking in-/out-trace equivalence is PSPACE-complete [136], which implies that the decision and optimization versions of the minimum PSG problem are PSPACE-complete.

Theorem 3 *Minimum* PSG *is PSPACE-complete.*

Instead of checking trace equivalence, we use *simulation* relations as its approximation [137, 138], which is less strict than bisimulation and can be computed efficiently in $O(|\mathbb{V}_{\mathbb{S}}||\mathbb{E}_{\mathbb{S}}|)$ in PSG g_0 [137]. A vertex u is in-simulate dominated by a

vertex v, written $u \leq_{in}^{s} v$, if **i)** their label in PSG is the same, i.e., $\rho(u) = \rho(v)$ and **ii)** for each parent p_u of u, there is a parent of p_v of v, s.t., $p_u \leq_{in}^{s} p_v$. We say u and v in-simulate each other, $u \simeq_{in}^{s} v$, iff $u \leq_{in}^{s} v \wedge v \leq_{in}^{s} u$. Similarly, u is out-simulate dominated ($\leq_{out}^{s}$) by v, if $\rho(u) = \rho(v)$ and for each child c_u of u, there is a child of c_v of v, s.t., $c_u \leq_{out}^{s} c_v$; and u and v out-simulate each other $u \simeq_{out}^{s} v$ iff they out-simulate dominate each other.

Note that a binary relation r_a *approximates* r_b, if $(e_i, e_j) \in r_a$ implies $(e_i, e_j) \in r_b$ [138]. In other words, if (u, v) in-/out-simulates each other, then (u, v) is in-/out-trace equivalence. By using simulation instead of trace equivalence in Lemma 5 as the merge condition, we can ensure the invariant.

Lemma 6 *If* 1) $u \simeq_{in}^{s} v$, *or* 2) $u \simeq_{out}^{s} v$, *or* 3) $u \leq_{in}^{s} v \wedge u \leq_{out}^{s} v$, *merging* u *to* v *does not introduce new paths.*

We develop the PGSUM algorithm by using the partial order derived from Lemma 6 merge condition in a PSG (initialized as g_0) to merge the vertices. To compute $\leq_{in}^{s}$ and $\leq_{out}^{s}$, we apply the similarity checking algorithm in [137] twice in $O(|\mathbb{V}_{\mathbb{S}}||\mathbb{E}_{\mathbb{S}}|)$ time.

From Lemma 6, we can ensure there is no new path introduced, and the merging operation does not remove paths, so PGSUM algorithm finds a valid PSG. Note that unlike Lemma 5, as the reverse of Lemma 6 does not hold, so we may not able to find the minimum PSG, as there may be (u, v) is in trace equivalence but not in simulation.

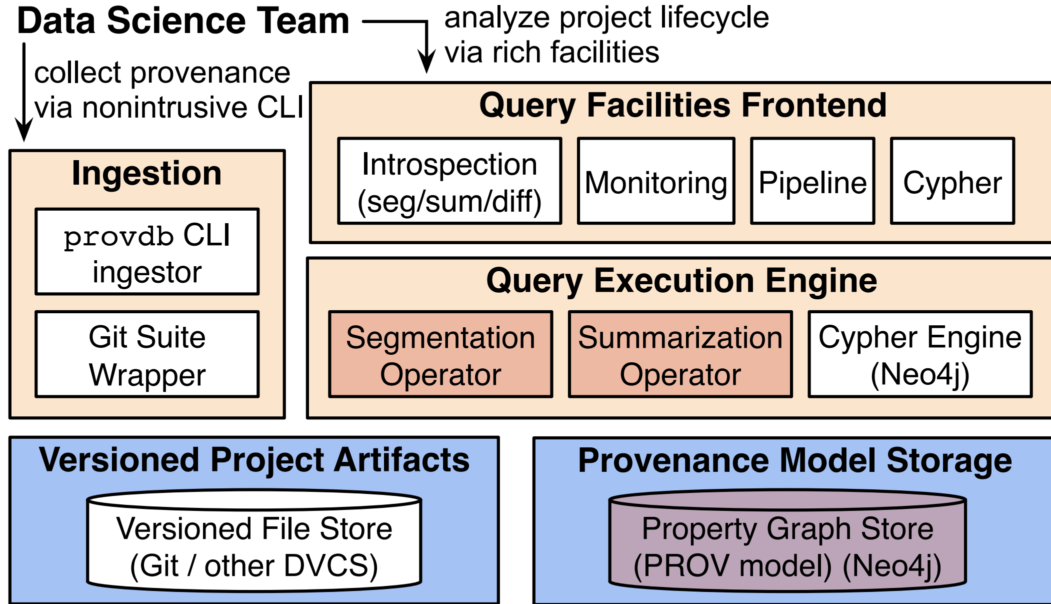


Figure 5.8: Position of Proposed Graph Query Operators in ProvDB

5.5 System Implementation

The proposed operators and techniques are implemented in ProvDB (Chapter 3) for data science project lifecycles. We highlight the positions of the proposed operators in ProvDB in Figure 5.8. As mentioned in Chapter 3, ProvDB collects provenance via a `git`-like command line interface (CLI) called `provdb`, which is supposed to be prefixed over command line executions of day-to-day commands (e.g., `provdb ingest 'cp model.config new_model.config'`) by the users. Therefore the context of the executed command (Activity, User) and changes of project artifacts (Entity) occurred before/during/after the execution can be captured by `provdb` and stored in a property graph store; currently we use Neo4j. On the other hand, ProvDB can replace `git` and acts as a data version control system (DVCS) to manage the versioned project artifacts. It is tailored for analytics project artifacts,

such as large model files mainly consisting of float numbers (Chapter 4).

Once versions of the project artifacts and provenance of the team activities are managed by ProvDB, the data science team members and stakeholders are able to analyze the lifecycle provenance using the rich set of query facilities in ProvDB (Chapter 3). The segmentation and summarization operators proposed in this chapter are implemented with templated query interface in the front end:

- For the segmentation operator, the users are allowed to identify snapshots of interest via search interface and specify boundary criteria within HTML GUI;
- For the summarization operator, previous segmentation operator results are selectable in the GUI and the property aggregation and the provenance type can be specified accordingly.

Returned results of both operators can be visualized and interacted with via `d3.js`.

The query execution engine for the proposed operators and their algorithms described in Section 5.3 and 5.4 are implemented in Java using Neo4j in embedded mode for better performance.

Discussion

Note the design decision of using a general purpose native graph backend (Neo4j) for high-performance provenance ingestion may not be ideal, as the volume of ingested provenance records would be very large in some applications, e.g., whole-system provenance recording at kernel level [108, 112] would generate GBs of data in minutes. The support of flexible and high performance graph ingestion on modern graph databases and efficient query evaluation remain an open question [139]. We

leave the issue to support similar operators for general PROV graph for our future steps. The proposed techniques in the chapter focus on enabling better utilization of the ingested provenance information via novel query facilities and are orthogonal to storage layers in provenance systems.

5.6 Evaluation Study

In this section, we study the proposed operators and techniques comprehensively. We first evaluate the efficiency of proposed techniques for the segmentation operator by varying provenance graph properties and comparing with state-of-the-art CFLR algorithm [129, 130] and the crafted Cypher query. We then study the effectiveness of the summarization operator on segmentation results with different stableness and compare with a summarization technique on graph query results [117]. All experiments are conducted on a Ubuntu Linux 16.04 machine with an 8-core 3.0GHz AMD FX-380 processor and 16GB memory. For the backend property graph store, we use Neo4j 3.2.5 community edition in embedded mode and access it via its Java APIs. ProvDB query operators are implemented in Java in order to work with Neo4j APIs. To limit the performance impact from the Neo4j, we always use the node id to seek the nodes, which can be done in constant time in Neo4j's physical storage. Unless specifically mentioned, the page cache for Neo4j is set to 2GB and the JVM version is 1.8.0_25 and -Xmx is set to 4GB.

5.6.1 Dataset Description

Unless lifecycle provenance management systems (e.g., Ground [50], ProvDB) are used by the practitioners for a long period of time, it is difficult to get real-world provenance graph from data science teams. Though using VCS (e.g., `git`) is common practice, VCS repositories only consist of versions of artifacts, but not the activities that occurred between commits. Publicly available real-world PROV provenance graph datasets in various application domains [25] are very small (KBs). We instead develop several synthetic PROV graph generators to examine different aspects of the proposed operators. The datasets and the generators are available online[3].

5.6.1.1 Provenance Graphs & PgSeg Queries

To study the efficiency of PgSeg, we generate a provenance graphs dataset (PG) for collaborative analytics projects by mimicking a group of project members performing a sequence of activities in a lifecycle management system. Each project artifact has many versions and each version is an entity in the graph. An activity uses one or more input entities and produces one or more output entities.

To elaborate, given N, the number of vertices in the output graph, we introduce $|\mathcal{U}| = \lfloor \log(N) \rfloor$ agents. To determine who performs the next activity, we use a Zipf distribution with skew s_w to model their work rate. Each activity is associated with an agent and uses $1 + m$ input entities and generates $1 + n$ output entities. m and n are generated from two Poisson distributions with mean λ_i and λ_o to model

[3] Datasets: `http://www.cs.umd.edu/~hui/code/provdbquery`

Name	PG_{50}	PG_{100}	PG_{500}	PG_{1k}	PG_{5k}	PG_{10k}	PG_{50k}
$\|\mathcal{E}\|$	37	88	343	796	3676	7580	37749
$\|\mathcal{A}\|$	13	26	126	251	1251	2501	12501
$\|U\|$	37	74	381	726	3784	7473	37532
$\|G\|$	33	84	342	794	3670	7579	37743

Table 5.1: Summary of Provenance Graph Datasets (PG)

different input and output size. In total, the generator produces $|\mathcal{A}| = \left\lfloor \frac{N}{2+\lambda_o} \right\rfloor$ activities, so that at the end of generation, the sum of entities $|\mathcal{E}|$, activities $|\mathcal{A}|$ and agents $|\mathcal{U}|$ is close to N. The m input entities are picked from existing entities; the probability of an entity being selected is modeled as the pmf of a Zipf distribution with skew s_e at its rank in the reverse order of being. If s_e is large, then the activity tends to pick the latest generated entity, while s_e is small, an earlier entity has better chance to be selected. The n output entities are always new entities, which would be the first version of an artifact, or a new version of an existing artifact. For the latter, we add a derivation edge to an ancestor entity uniformly.

We use the following values as default for the parameters: $s_w = 1.2$, $\lambda_i = 2$, $\lambda_o = 2$, and $s_e = 1.5$. In Table 5.1, we show the information about generated provenance graphs for varying $N \in [50, 100, 500, 1000, 5000, 10000, 50000]$.

On the provenance graphs in PG, we pick pairs ($\mathcal{V}_{src}$, $\mathcal{V}_{dst}$) as PgSeg queries to evaluate. Unless specifically mentioned, given a PG dataset, $\mathcal{V}_{src}$ are the first two entities, and $\mathcal{V}_{dst}$ are the last two entities, as they are always connected by some

path and the query is the most challenging PgSeg instance. In one evaluation, we vary $\mathcal{V}_{src}$ to show the effectiveness of the proposed pruning strategy.

5.6.1.2 Similar Segments & PgSum Queries

In order to study the effectiveness of PgSum, we design a synthetic generator (Sd) with the ability to vary shapes of conceptually similar provenance graph segments. In brief, the intuition is that as at different stages of the project, the stability of the underlying pipelines tends to differ, the effectiveness of summary operator could be affected; e.g., at the beginning of a project, many activities (e.g., clean, plot, train data) would happen after another one in no particular order, while at later stages, there are more likely to be stable pipelines, i.e., an activity type (e.g., preprocessing) is always followed by another activity type (e.g., train). For PgSum, the former case is more challenging than the latter one.

In detail, we model a segment as a Markov chain with k states and a transition matrix $M \in [0,1]^{k \times k}$ among states. Each row of the transition matrix is generated from a Dirichlet prior with the concentration parameter $\vec{\alpha}$, i.e., the ith row is a categorical distribution for state i; each M_{ij} represents the probability of moving to state j, i.e., pick an activity of type j. We set a single α for the vector $\vec{\alpha}$; for higher α, the transition tends to be a uniform distribution, while for lower α, the probability is more concentrated, i.e., fewer types of activities would be picked from.

Given a transition matrix, we can generate a set of segments $\mathbb{S}$, each of which consists of n activities labeled with k types, derived step by step using the transition

matrix. For the input/output entities and edges of each activity, we use λ_i, λ_o, and s_e the same way in PG, and all introduced entities have the same equivalent class.

We vary α, $|\mathbb{S}|$, k and n to study the PgSum effectiveness on different sets of segments. A PgSum query is applied on each $\mathbb{S}$, and produces a Psg. The effects of property aggregation and provenance types are reflected in the above label assignment process.

5.6.2 Evaluation Results

5.6.2.1 Segmentation Operator

Next, we show the evaluation results for PgSeg algorithms. We compare our algorithm SimProvAlg and SimProvTst with the state-of-the-art general context-free language reachability algorithm, CflrB [129]. It uses bit-based set operations to improve the Reps' [130] dynamic programming algorithm. We implement the fast set using Java BitSet in order to have constant random access time, which is not true for compressed BitMap alternatives. We also compare with the Cypher query mentioned in Section 5.3.2 in Neo4j.

Varying Graph Size N

In Figure 5.9, we study the scalability of all algorithms. x axis denotes N of the PG graph, while y axis shows the runtime in seconds to finish the PgSeg query. Note the figure is log-scale for both axes. As we see, SimProvAlg and SimProvTst run at least one order of magnitude faster than CflrB on all PG datasets. The main reasons are the utilization of the properties of the grammar and efficient prun-

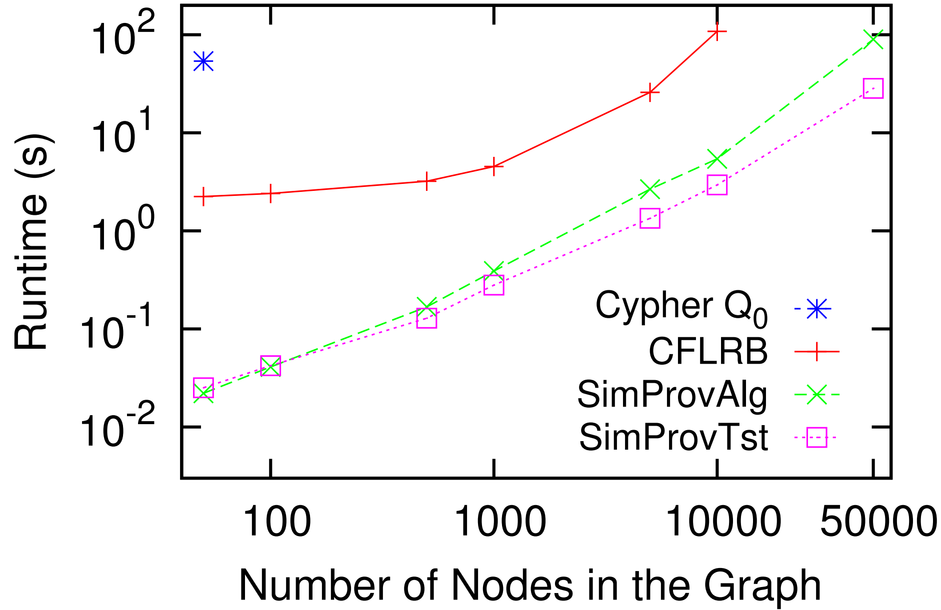


Figure 5.9: Comparing Cyper Query 5.1, CflrB, SimProvAlg and SimProvTst Efficiency by Varying Graph Size N

ing strategies. Note CflrB runs out of memory on PG_{50k} because of the much faster growth of the worklist than SimProvAlg, as CflrB uses normal forms and introduces an extra level. SimProvAlg runs slightly faster than SimProvTst for small instances while becomes much slower for large instances, e.g., PG_{50k}, it is 3x slower than SimProvTstfor the query. The reason that small instances SimProvAlg slightly faster is the because SimProvTst run $|\mathcal{V}_{dst}|$ times on the graph and each run's performance gain is not large enough. When the size of the graph instance increases, the superiority of the SimProvTst by using the transitivity property becomes significant.

On the other hand, the Cypher query can only return correct result for a very small graph PG_{50} and takes orders of magnitude longer. Surprisingly, even for

PG_{100}, it runs over 12 hours and we have to terminate it. From its query profiler, we know that Neo4j uses a path variable to hold all paths and joins them later which is exponential w.r.t. the path length and average out-degree. Due to the query language expressiveness, grammar properties cannot be used by the query planer.

Varying Input Selection Skew s_e

Next, in Figure 5.10, we study the effect of different selection behaviors on PG_{10k}. The x axis is s_e and the y axis is the runtime in seconds in log-scale. In practice, some analytics activities tend to try many model alternatives to get the best performance for an analytics task, e.g., through a grid search over hyperparameters, or changing a neural network architecture; while there are other analytics activities, where there are long chains of data transformation pipelines, e.g., feature engineering efforts. The former types of projects tend to always take an early entity as input (e.g., dataset, label), while the latter tend to take new entities (i.e., the output of the previous pipeline step) as inputs. Tuning s_e in opposite directions can mimic those project behaviors, as it tunes the probability of earlier entities been selected as inputs. In Figure 5.10, we vary s_e from 1.1 to 2.1, and the result is quite stable for SimProvAlg, SimProvTst and CflrB, which implies the query formulation and techniques can be applied to different project types with similar performance.

Varying Activity Input Mean λ_i

We study the effect of varying density of the graph in Figure 5.11 on PG_{10k}. The x axis varies the mean λ_i of the number of input entities. The y axis shows the runtime in seconds. Having a larger λ_i, the number of $|U|$ edges will increases linearly, thus

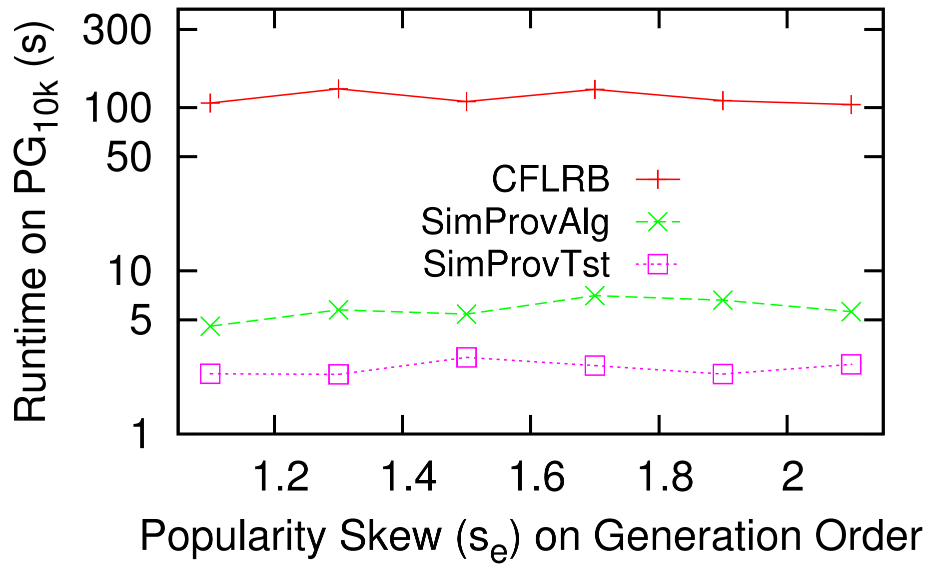


Figure 5.10: Evaluate Performance for Different Project Stereotypes by Varying λ_i

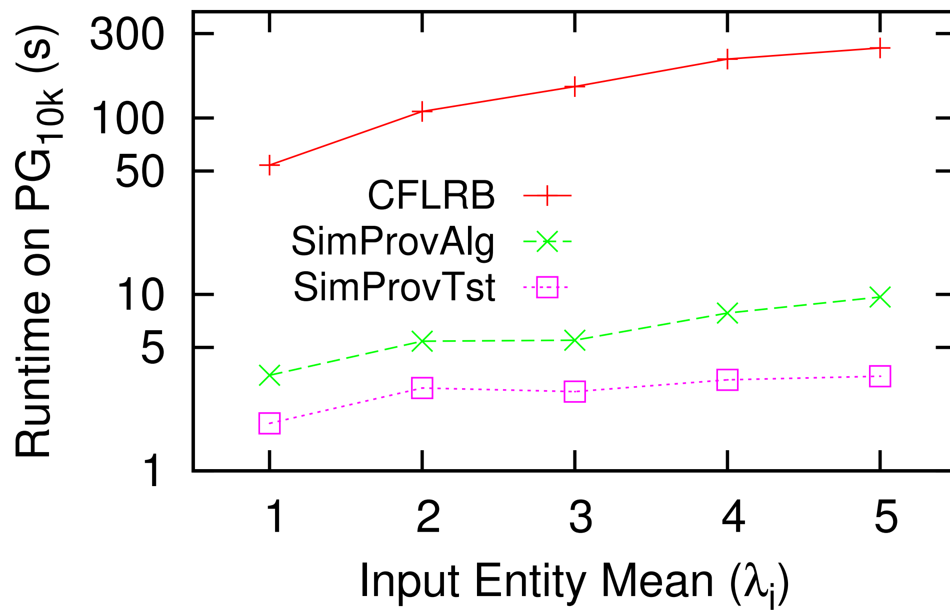


Figure 5.11: Impact of Input Size λ_i on CFLRB, SIMPROVALG and SIMPROVTST

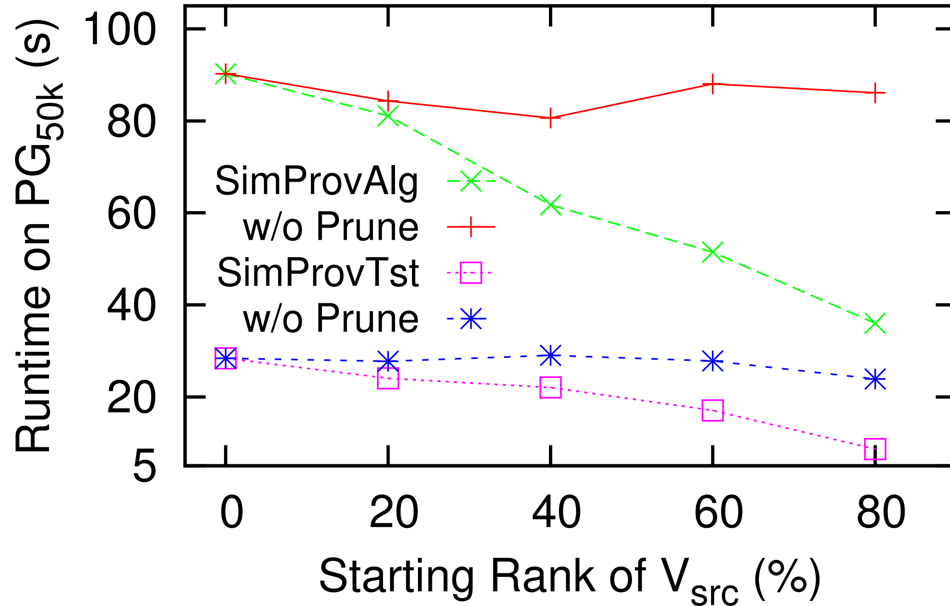


Figure 5.12: Effectiveness of SimProvAlg and SimProvTst with Early Stopping

the algorithms runtime increases as well. In Figure 5.11, we see SimProvAlg grows much more slowly than CflrB. Due to the pruning strategies, the growth in worklist utilization is avoided in SimProvAlg. SimProvTst further improves the SimProvAlg due to the utilization of the transitivity.

Effectiveness of Early Stopping

The above evaluations all use the most challenging PgSeg query on start and end entities. In practice, we expect the users will ask queries whose result they can understand by simple visualization. CflrB and general CFL don't have early stopping properties. SimProvAlg and SimProvTst use the temporal constraints of the provenance graph to support early stopping growing the result. In Figure 5.12, we vary the $\mathcal{V}_{src}$ of a PgSeg query and study the performance on PG_{50k}. The x axis is the starting position among all the entities, e.g., $x = 20$ means $\mathcal{V}_{src}$ is selected

at the end of 20% percentile w.r.t. the ranking of the order of being. The y axis is the runtime in seconds. As we can see, the shorter of the gap between $\mathcal{V}_{src}$ and $\mathcal{V}_{dst}$, the shorter the SimProvAlg and SimProvTst runtime. By utilizing the property of PROV graphs, we get better performance empirically even though the worst case complexity does not change.

5.6.2.2 Summarization Operator

Given a $\mathbb{S}=\{\mathcal{S}_i(\mathbb{V}_{\mathcal{S}i}, \mathbb{E}_{\mathcal{S}i})\}$, PgSum generates a precise summary graph $\text{Psg}(\mathcal{M}, E)$ by definition. Here we study its effectiveness in terms of conciseness. We use the compaction ratio defined as $c_r = |\mathcal{M}|/|\bigcup_i \mathbb{V}_{\mathcal{S}i}|$. As there are few graph query result summarization techniques available, in our study, we compare with `pSum` [117] which is designed for summarizing a set of graphs from keyword search graph queries. `pSum` works on undirected graphs and preserves path among keyword pairs and was shown to be more effective than summarization techniques on one large graph, e.g., SNAP [116]. To make `pSum` work on PgSeg segments, we introduce a conceptual (*start*, *end*) vertex pair as the keyword vertices, and let the *start* vertex connect to all vertices in $\mathbb{S}$ having 0 in-degree, and similarly let the *end* vertex connect to all vertices having 0 out-degree. In the rest of the experiments, by default, $\alpha = 0.1$, $k = 5$, $n = 20$ and $|\mathbb{S}| = 10$, and y-axis denotes the compaction ratio c_r in all figures.

Varying Transition Concentration α

In Figure 5.13, we change the concentration parameter to mimic segment sets at various stage of a project with different stableness. x axis denotes the value of α

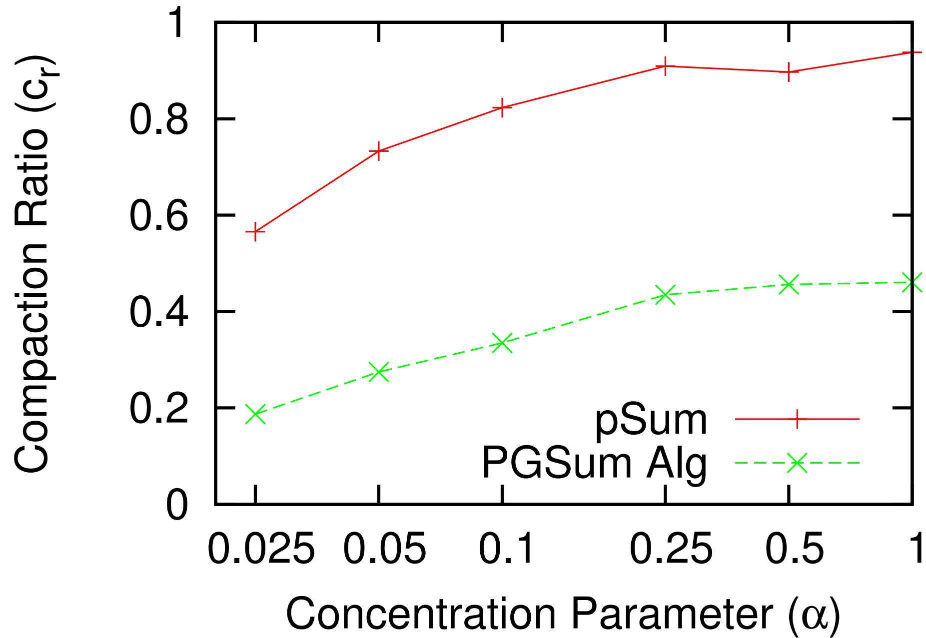


Figure 5.13: Studying PgSum on Segments at Different Stages of a Project by Varying Concentration α

in log-scale. Increasing α, the transition probability tends to be uniform, in other words, the pipeline is less stable, and paths are more likely be different, so the vertex pairs which would be merged become infrequent. As we can see, PgSum algorithm always performs better than `pSum`, and the generated Psg is about half the result produced by `pSum`, as `pSum` cannot combine some $\simeq^t_{in}$ and $\simeq^t_{out}$ pairs, which are important for workflow graphs. The finding is consistent in other experiments.

Varying Activity Types k

Next, in Figure 5.14, we vary the possible transition states, which reflects the complexity of the underlying pipeline. It can also be viewed as the effect of using property aggregations on activities (e.g., distinguish the commands with the same name but different options). Increasing k leads to more different path labels, as

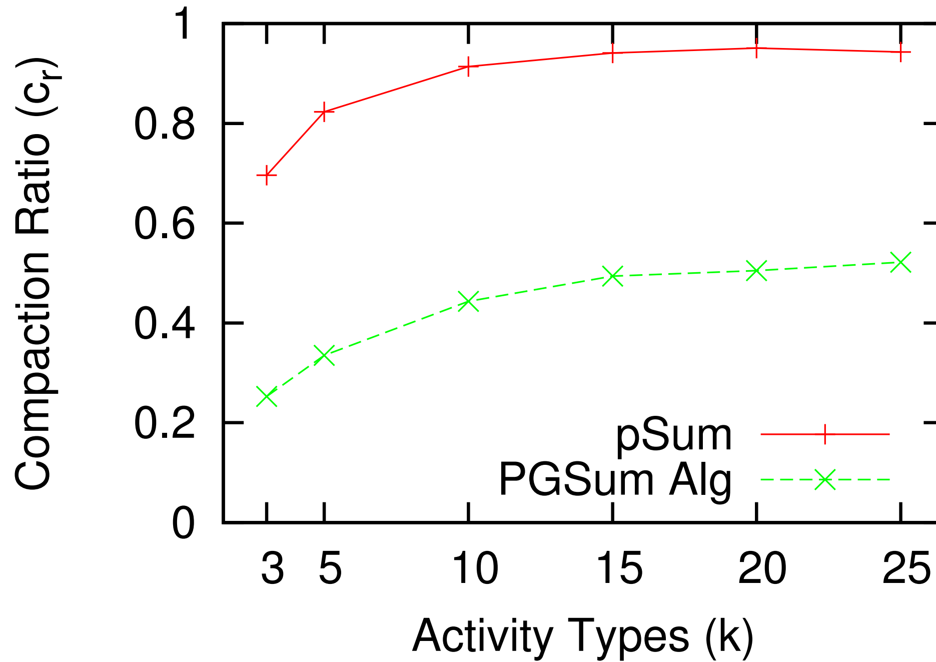


Figure 5.14: Studying PgSum on Segments with Different Complexity by Varying Activity Types k

shown in the Figure 5.14, and it makes the summarization less effective. Note that when varying k, the number of activities n in a segment is set to be 20, so the effect of k on compaction ratio tends to disappear when k increases.

Varying Segment Size n

We vary the size of each segment n when fixing α and k to study the performance of PgSum. Intuitively, the larger the segment is, the more intermediate vertices there are. The intermediate vertices are less likely to satisfy the merging conditions. As shown in Figure 5.15, the compaction ratio increases as the input instances are more difficult.

Varying Number of Segments $|\mathbb{S}|$

With all the shape parameters set ($\alpha = 0.25$), we increase the number of similar

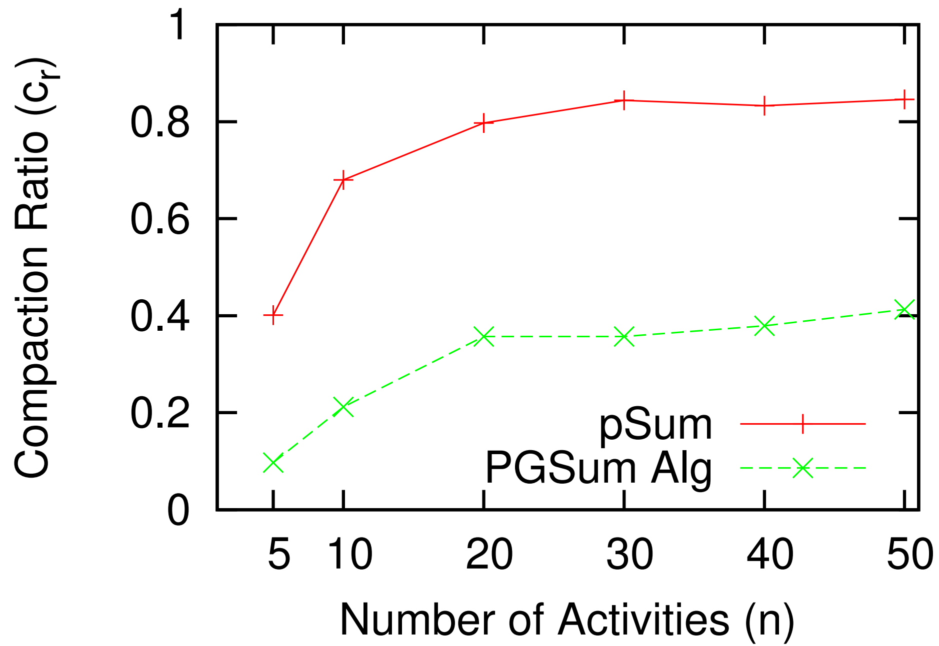


Figure 5.15: Studying PgSum on Segments with Different Size by Varying Number of Activities n

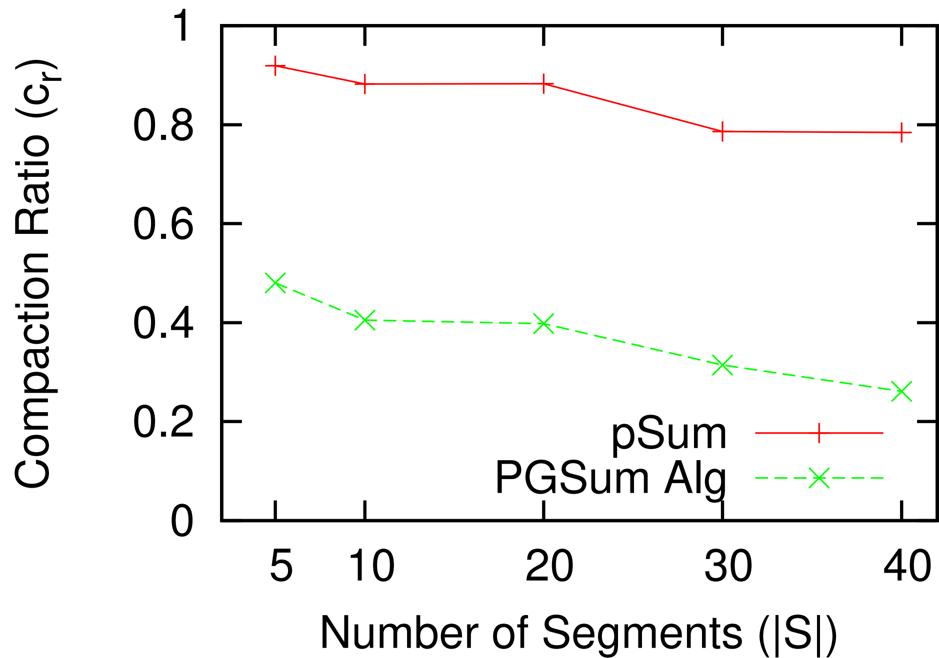


Figure 5.16: Studying PgSum Effectiveness by Varying $|\mathbb{S}|$

segments. As the segments are generated from the same transition matrix, they tend to have similar paths. As shown in Figure 5.16, the compaction ratio becomes better when more segments are given as input.

5.7 Conclusion

In this chapter, we described the key challenges of querying provenance graphs generated in evolving workflows without predefined skeletons, such as the ones collected by lifecyle management systems in collaborative analytics projects. At query time, the users only have partial knowledge about the ingested provenance, due to the schema-later nature of the properties, multiple versions of the same files, unfamiliar artifacts introduced by team members, and enormous provenance records collected continuously. Just using standard graph query model, it is very difficult to compose queries and utilize the valuable information. We presented two high-level graph query operators to address the verboseness and evolving nature of such provenance graphs. First, we introduced a graph segmentation operator that allows the users to only provide the vertices they are familiar with and then induces a subgraph representing the retrospective provenance of the vertices of interest. We formulated the semantics of such a query in a context free language, and developed efficient algorithms on top of a property graph backend. Second, we described a graph summarization operator that combines the results of multiple segmentation queries to assist the users to understand similar and abnormal behaviors in those conceptually similar segments with multi-resolution capabilities. Extensive experi-

ments on synthetic provenance graphs with different project characteristics show the operators and evaluation techniques are effective and efficient. The operators are also applicable for querying provenance graphs generated in other scenarios where there are no workflow skeletons, such as cybersecurity and system diagnosis.

Chapter 6: Discovering Hosted Analytics Projects

Alongside developing systems for scalable machine learning and collaborative data science activities, there is an increasing trend toward publicly shared data science projects, hosted in general or dedicated hosting services, such as GitHub, DataHub, Model Zoo, etc. The artifacts of the hosted projects are rich and include not only text files, but also versioned datasets, trained models, project documents, etc. With lifecycle management systems, like ProvDB, the shared project will even consist of provenance of the lifecycle, and rich set of metadata about those artifacts. Under the fast pace and expectation of data science activities, we argue model discovery, i.e., finding relevant data science projects to reuse is an important task.

In this chapter, we study the discovery task and explore system designs and research problems. Proper investigation of this issue requires a large number of hosted projects, which is not true for our ProvDB system. On the other hand, in general collaborative systems, such as GitHub, there is no clean structured data model for data science projects. Instead of structured queries, we propose a system vision via an information retrieval approach, and decompose the discovery task into three major steps: *1)* project query and matching, *2)* model comparison and ranking,

and *3)* processing and building ensembles with returned models. We first describe the motivation and desiderata of the model discovery problem, then we overview ongoing effort of enriching project repositories in Section 6.1. Then we describe our system vision, and present opportunities, challenges and techniques for each step in Section 6.2. In Section 6.3, we illustrate preliminary evaluation results of proposed techniques.

6.1 Motivation

In the "Big Data" era, datasets are collected blindly in different domains and industries by logging user and system behavior or labeling data via crowd-sourcing, and there is a large demand to conduct data science projects to find value from it. With more experienced practitioners and better system support to utilize the collected data and help the data science activities, more and more data science projects are built and shared. For example, there are tens of thousands of hosted Github projects using IPython/Jupyter notebooks (Figure 6.1) and the numbers have grown rapidly in the past few years; in 2016, Github made 'Jupyter Notebook' as a supported language type in the search engine for its repositories. For predictive models, due to the widely-used finetuning techniques and long training times, deep learning models have been shared by the community starting with publishing models on authors' websites; now training systems tend to have models hosted at a central portal for practitioners, e.g., Caffe Model Zoo. Along the same line, ProvDB and ModelHub mentioned in previous chapters in the dissertation also include a

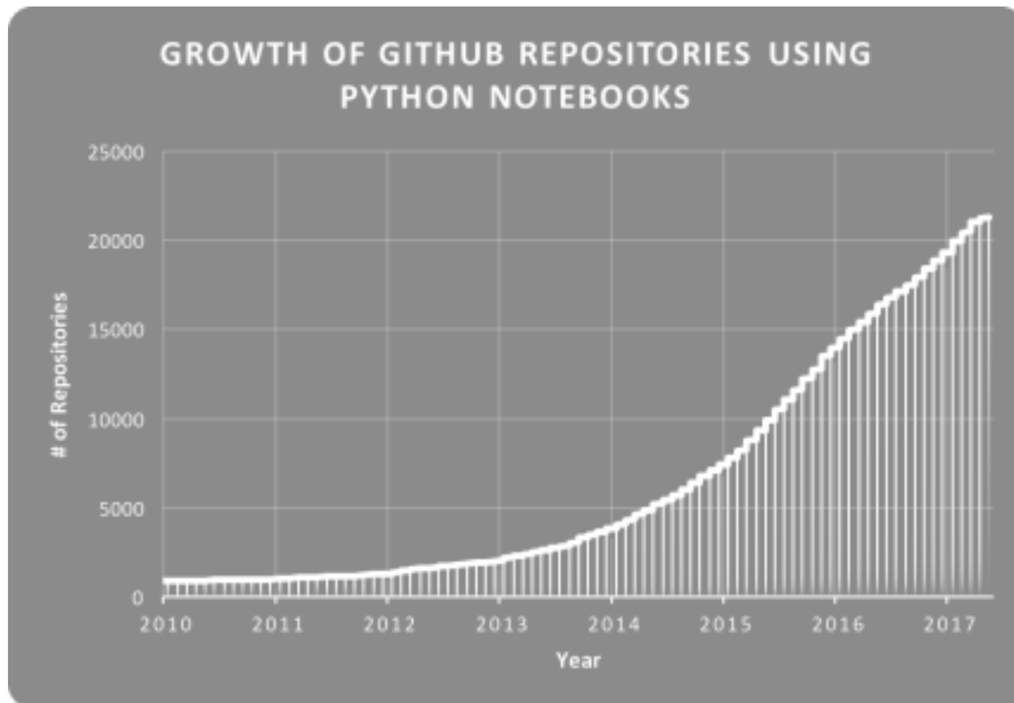


Figure 6.1: Growth of Github Repositories using iPyhont Notebooks (based on Github Weekly Dump Dataset on Google Big Query)

hosting service for data science projects with comprehensive model revision histories and provenance information generated during the process. In feature engineering-based modeling practice, users tend to try many combination of features; recent commercial platforms save the produced models and the context to help future modeling activities.

6.1.1 Model Discovery

Given the presence of large collections of data science projects uploaded by different groups of data scientists and hosted centrally by a system, the problem of model discovery is underserved by existing hosting services and is not well-studied

by machine learning lifecycle management tools. Model discovery is the problem of identifying relevant projects for a data science practitioner who is working on her own project and willing to find references, by asking queries such as

- 'What projects used a similar dataset like mine on a classification problem? (e.g. US census data, 256x256 images)',
- 'Show me a set of diverse projects which explore this specific dataset or use this particular modeling method (e.g., random forest)',
- 'Find or ensemble a model from projects having high recall but reasonable accuracy on a given validation dataset'.

Current hosting systems, such as Github, cannot answer such queries, as they do not distinguish data science projects from open source software repositories and data scientists from software developers, and they do not understand modeling artifacts and associated metadata and provenance in the lifecycle.

Example 10 *For example, a modeler wishes to identify others' projects that do mortgage default rate prediction and use a linear regression method in Python in GitHub. Using today's systems, the user could use 'site:github.com predict mortgage default rate linear regression python' on a modern search engine. However, general web search treats each URL separately and does not index repo as a whole; the returned results could not rank the most relevant repository properly. Instead, individual files, such as dataset, code snippet with some search keywords in the comments were returned. On the other hand, if the user uses GitHub directly, currently, it still uses repositories' names to match the queries, and no repository may be returned.*

	Category	Examples
Artifact	Physical file	Data file properties, schema, size, author, content; script AST, runtime profiling, lib dependencies, etc.
	Logical artifact	Model type, metric value, hyperparameters of a model, user annotations on a model, etc.
Workflow	Steps in a physical file	System IO calls, library methods (e.g. logistic regression, pandas dataframe), DNN network architecture, and their dependencies, etc.
	Steps in an logical artifact	Analysis step type (data loading, cleaning, feature engineering), step input, output, transitions, etc.
	Relations among physical files	File versions, file dependencies in the execution history, system library dependency, etc.
	Relations among logical artifacts	Logical artifact versions, e.g. earlier version for a model, lineages from modeling branches, etc.
User	Human collaborations	Users' contributions, communications, notes, comments, cognitive annotations, etc.

Figure 6.2: Enriched Information for Data Science Projects

We envision that by having properly designed model discovery system, a hosted data science project site should be able to help practitioners answer those queries and accelerate the modeling lifecycle and deliver results faster.

6.1.2 Enriched Project Repository

Data science projects are ad hoc, exploratory and have many iterations. Without proper data and lifecycle management systems for the modeling process, they are no more than a collection of files and possibly file histories if a version control system is used.

Similar to ProvDB described in earlier chapters, recently, there is a line of work, proposing what we call *lifecyle management systems* (LMS), that capture provenance metadata and manage modeling artifacts and lifecycle beyond the files,

so that practitioners can ask detailed queries about the artifact and the workflow [15, 17, 40, 50, 82, 83, 85] and potentially accelerate or even automate some steps in the lifecycle. Though the focus and the provenance data model vary in these systems, we argue a rich representation and storage of a data science project can be derived and stored. Such information, for modeling artifacts, includes the modeling method types, parameters, hyperparameters, input data and output result files, etc.; for project workflows, it includes file-level dependencies, function dependency graphs in scripts, temporal dependencies among different versions of files. We summarize the enriched information in different works in Figure 6.2.

The enriched information by the ongoing efforts leads to opportunities for novel approaches to the model discovery problem. In the rest of the discussion, we assume a data science project is enriched and managed by such an LMS, so that the projects include files as well as captured rich information from such systems.

6.2 Model Discovery System Vision

In this section, we present a system design to approach the model discovery problem. We refer the system as ModelHub Discovery. To simplify discussion, we focus on predictive models, i.e., given an observation dataset with labels, the goal is to find a prediction function that fits the observations. In the current prototype development, we also assume the models are a call to existing library routines with users' parameters, without special customization or self-implementation, for example, writing ones' own logistic regression implementation in Python due to

project-specific requirements. We argue that the impact of this assumption is acceptable as it covers many data science activities and could be addressed during the development of LMS.

6.2.1 System Overview

Data Model

ModelHub Discovery is a server-side service, designed to be used with client-side LMS, in particular ModelHub and ProvDB discussed in earlier chapters. The detailed data model for a project is described in Figure 3.1, and uses a schema-later approach with a property graph data model for ingested information and a minimal schema for version control. It has logical views (i.e., models and artifacts to lift file snapshots) and heterogeneous backends for different artifacts (e.g., binary store for weights, graph store for properties, and version index, etc.). When discussing techniques, we keep them general to use with other LMS.

Query Model

An important design decision we made for the query facility is supporting NLP keyword queries as a first-class citizen as in information retrieval (IR) systems. Using NLP queries, practitioners can easily describe project background, goals, datasets and methods using keywords and documents. We chose this option over category-based project organization or structured queries with ontological conditions because:

- There are many text files (e.g., READMEs, comments) and symbols (e.g., library methods) in a project, however, different teams have their own dialects

and preferences, and there is lack of a unified ontology or schema over data science projects;

- In structured queries, relevant ranking of matched projects needs to be provided by users;

- Many structural properties (e.g., accuracy, loss) of a model are project-specific, so a query condition over them for all projects does not make sense, but those are more meaningful as adjunctive filters to match similar projects.

Besides NLP queries, we envision a query facility by datasets. In Datahub, datasets are hosted with identifiers, versions and possibly schemas. When a user needs to explore a dataset, which might be public, proprietary, or hosted in Datahub, allowing her to use dataset information, such as structured metadata (i.e., name, columns), or samples of the dataset (i.e., a set of rows) to query hosted projects is a very useful navigation facility. However, it requires the LMS to have detailed file dependencies for good user experience.

System Architecture

We present the preliminary design of the ModelHub Discovery system in Figure 6.3. We show the query processing pipelines and data serving components. As an information retrieval approach, for each project, we use a bag-of-words model to represent it as a document, which includes words from its text artifacts, function symbols, code comments, etc. Then the vector-space model [140] is used and the project is indexed using the Apache Lucene full-text indexing engine. The project catalog is a graph-store which combines enriched information from lifecycle management systems, and

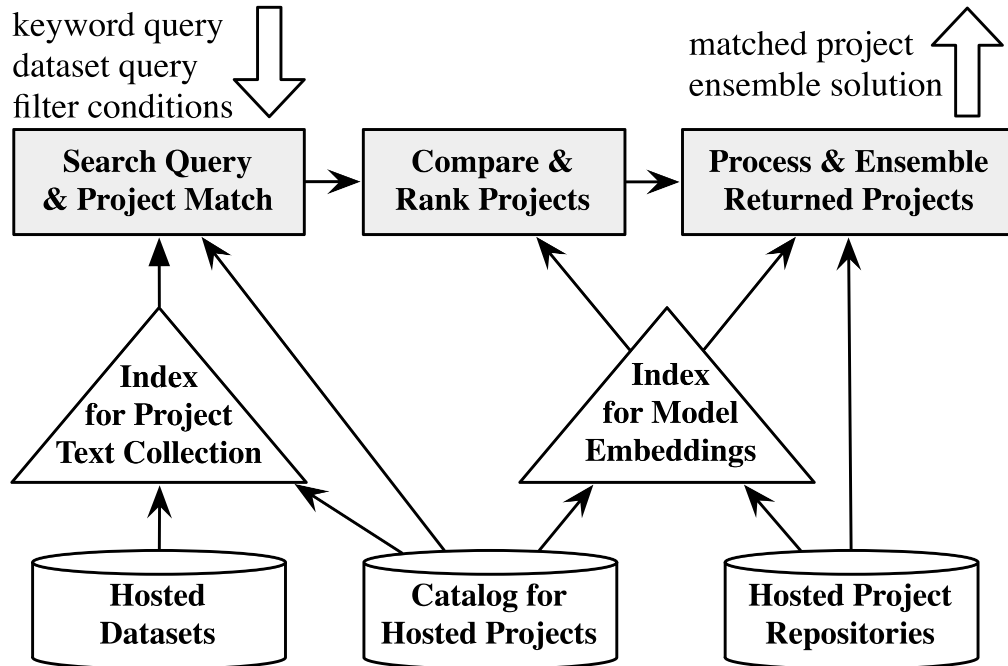


Figure 6.3: Overview of ModelHub Discovery Pipeline

is used in adjunctive conditions to filter the matched projects and constraints on the relevance to the query.

To deduplicate and rank returned projects and have diverse results, a comparison and ranking module is proposed. It uses pair-wise similarities of projects and prunes query results. In Section 6.2.2, we discuss the similarity problem among models and propose a novel strategy, which also allows embedding models as vectors, accelerates all-pairs similarity search and enables learn-to-rank strategies.

When the returned models solve the same problem on a dataset (e.g., deep learning models for ImageNet), we discuss the possibility of incorporating model ensemble techniques as a post processing step (Section 6.2.3), in order to support the case when the users have additional constraints in their projects, such as precision-recall, accuracy-fairness, or accuracy-resource trade-offs.

6.2.2 Compare & Rank Models

Model Similarity

A query potentially returns many projects with similar or even duplicated models. As one can observe from GitHub, a popular IPython/Jupyter notebook may be forked hundreds of times. In deep learning, finetuning is a common practice as a popular model trained from massive data may be transferred to other tasks; the weights are reused and finetuned, resulting in similar models.

Returning similar models affects usability dramatically. Deduplication is common practice in information retrieval systems. As the projects come from different teams, in order to compare projects we need a similarity model for models. However, it is challenging to determine when two models are similar. By just looking at the associated model properties, such as accuracy and confidence, it is not meaningful, as the measurements are specific to data points. To characterize a model, aside from the enriched information, the input datasets, I, the output result tables, O, and the trained parameters, P, can characterize model performance well. O is typically a structured table (data, label), while P is often float vectors, matrices or tensors, all of which are ordered multi-dimensional sequences. Due to the order, similar models are possibly obfuscated, e.g., the same dataset rows or columns may vary for different projects, while the parameter dimension may refer to different input dimensions. To address it, we propose an *alignment* operation to process the model pairs in the *best way* in order to develop a similarity method.

Alignment Operation

For ease of illustration, we focus on 2D matrices for P, as vectors and tensors alignment can be formulated similarly, while I and O can be operated on using similar ideas. The basic idea is given two matrices A and B, we permute B's rows and columns to determine the most similar B' w.r.t. a given distance function, e.g., euclidean distance, string edit distance, etc. As the matrices to be aligned do not necessarily have the same dimensions, we first define a *permutation matrix* capable of adapting to differences in dimensions.

Definition 4 (Permutation Matrix) *Let a permutation* $\psi = (\psi_1, \psi_2, \cdots, \psi_n) \in \Psi$*, where* Ψ *is the set of all possible* $n!$ *permutations. Given two positive integers* $s, t \in \mathbb{N}_+$*, a permutation matrix of* ψ *is a matrix* $\mathcal{P}(\psi, s, t) \in [0,1]^{s \times t}$*, where*

$$\mathcal{P}(\psi, s, t)_{i,j} = \begin{cases} 1 & \text{when } i \leq n, \psi_i = j \\ 0 & \text{otherwise} \end{cases}$$

Using the permutation matrix, a permutation can be used in matrix multiplication to reorder matrix by rows or columns.

Example 11 *A row permutation* $\psi = (2, 1)$ *of* $A \in \mathbb{R}^{2\times4}$ *to* $\mathbb{R}^{3\times4}$*:*

$$\mathcal{P}(\psi, 3, 2) \times A = \begin{bmatrix} 0 & 1 \\ 1 & 0 \\ 0 & 0 \end{bmatrix} \times \begin{bmatrix} 8 & 2 & 5 & 3 \\ 4 & 1 & 2 & 2 \end{bmatrix} = \begin{bmatrix} 4 & 1 & 2 & 2 \\ 8 & 2 & 5 & 3 \\ 0 & 0 & 0 & 0 \end{bmatrix}$$

A column permutation $\phi = (1, 3, 2, 4)$ *of* $A \in \mathbb{R}^{2\times4}$ *to* $\mathbb{R}^{2\times3}$*:*

$$A \times \mathcal{P}(\phi, 4, 3) = \begin{bmatrix} 8 & 2 & 5 & 3 \\ 4 & 1 & 2 & 2 \end{bmatrix} \times \begin{bmatrix} 1 & 0 & 0 \\ 0 & 0 & 1 \\ 0 & 1 & 0 \\ 0 & 0 & 0 \end{bmatrix} = \begin{bmatrix} 8 & 5 & 2 \\ 4 & 2 & 1 \end{bmatrix}$$

With the permutation matrix, given two matrices and a distance function, we formulate the alignment problem as follows:

Problem 4 (Matrix Alignment) *Given two real matrices, $A \in \mathbb{R}^{m \times n}$, $B \in \mathbb{R}^{s \times t}$, we want to find two permutations to reorder B to A, $\psi = (\psi_1, \psi_2, \cdots, \psi_s) \in \Psi_{row}$ and $\phi = (\phi_1, \phi_2, \cdots, \phi_t) \in \Phi_{col}$, s.t.:*

$$\psi^*, \phi^* = \underset{\psi \in \Psi_{row}, \phi \in \Phi_{col}}{\arg\min} \ \mathcal{D}(A - \mathcal{P}(\psi, m, s) \times B \times \mathcal{P}(\phi, t, n))$$

where $\mathcal{D} : \mathbb{R}^{m \times n} \mapsto \mathbb{R}$ is a distance function. We denote the best aligned matrix as $\Delta^*_{A,B} = A - \mathcal{P}(\psi^*, m, s) \times B \times \mathcal{P}(\phi^*, t, n)$.

Example 12 *Let $\mathcal{D}$ be the $\|.\|_2$ norm, the two matrices:*

$$A = \begin{bmatrix} 8 & 2 & 5 & 3 \\ 4 & 1 & 2 & 2 \end{bmatrix}, B = \begin{bmatrix} 1 & 1 & 1 \\ 8 & 5 & 2 \\ 4 & 2 & 1 \\ 2 & 2 & 2 \end{bmatrix},$$

the permutations $\psi^* = (2, 3, 1, 4), \phi^* = (1, 3, 2)$ minimize the distance function: $\mathcal{D}(\Delta^*_{A,B}) = \|A - \mathcal{P}(\psi^*, 2, 4) \times B \times \mathcal{P}(\phi^*, 3, 4)\|_2 =$

$$\left\| \begin{bmatrix} 8 & 2 & 5 & 3 \\ 4 & 1 & 2 & 2 \end{bmatrix} - \begin{bmatrix} 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \end{bmatrix} \begin{bmatrix} 1 & 1 & 1 \\ 8 & 5 & 2 \\ 4 & 2 & 1 \\ 2 & 2 & 2 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 \end{bmatrix} \right\|_2 = 13$$

Complexity Analysis

As the solutions of this matrix alignment problem lie in two permutation spaces, it is not easy to solve. A reduction from the graph edit distance problem (GED) [141] can be used to show it is NP-hard. Similar to state-of-the-art heuristic for GED [141], bipartite matching-based approximations can be used to find good solutions.

Alignment Property & Its Usages

Interestingly, the alignment operation can also be used to construct distance metric to calculate diverse results efficiently. We define the *alignment distance* of two matrices after alignment: $\mathcal{A}(A, B) = \mathcal{D}(\Delta^*_{A,B}) + \mathcal{D}(\Delta^*_{B,A}) + \epsilon(A, B)$, where ϵ is $\mathcal{D}$ on natively aligned matrices by indices and 0 padding.

Lemma 7 *If the distance $\mathcal{D}$ is a metric, then $\mathcal{A}$ is also a metric.*

Since the alignment distance $\mathcal{A}$ is also a metric, an embedding space of models can be computed using approaches such as multidimensional scaling (MDS) [142]. The central idea of MDS is to learn an inner product space so that the distances between points can be preserved or approximated. The objective function is defined by $\min_x \sum_{i,j}(||x_i - x_j|| - d(i,j))^2$ where $d(i,j)$ is the distance between model i and model j after alignment, and x_i, x_j are their embedding vectors respectively.

The solution of MDS produces a vector for each given project model, which allows us to find similar and diverse models efficiently in online settings [143], and even enables the hosting service to use a model vector representation to improve query performances by learning to rank from click logs.

Discussion & Challenges

The unstructured nature of project artifacts and workflows makes it challenging to identify similar projects and return diverse results. In addition to using alignment and scaling to reason about parameters and input-output result tables, we also need to develop techniques to analyze the workflows, the program scripts, and the datasets to both compare different projects and to rank them. There is also a need to develop *interactive* techniques that allow a user to navigate through the returned models to quickly identify the most suitable models for their needs.

6.2.3 Process & Ensemble Returned Models

Constrained Query Results

In many scenarios, users are interested in the most accurate predictive models for their specific tasks. Instead of returning the best model from the hosted projects, it is likely that a combination of multiple models performs better than any single model. Ensemble modeling is a widely used practice, and has been studied extensively [144]. The model ensemble could greatly benefit from a hosted service having many projects with a diverse set of ideas.

At the same time, users may have multiple search criteria about a model,

such as constraints on precision-recall, accuracy-fairness, accuracy-resource, etc. For example:

- Instead of the most accurate model, users may also query for a model which performs similarly to a specified precision-recall curve [145].
- Due to the possibility of disparate impact, a model returned should deal with the fairness requirements [146].
- The performance of a predictive model is also measured by the resources it requires, e.g., time consumption and computer memory cost. It is often the case that under limited resources, e.g., mobile systems, users may compromise by asking for any models that have accuracy above a given lower bound.

All of these constrained query scenarios require processing the returned models to meet the user's demand. We explore supporting model ensemble, an important case with well-established methodologies.

Ensemble models

The question for model assembly is how we generate one model from K selected models from repositories which are applicable to the same task in order to satisfy the user-provided measurements. This could be achieved by two meta-approaches: 1) creating a new model by parallel-connecting all of the models, whose output is a weighted average or voting of the output of all the models, and 2) creating a (tree-structured) cascade or decision tree of models such that each node contains one of the models and a decision function to decide whether to use the output of the current model or the rest of the models. Both of these approaches can be

realized by learning the averaging weights, voting weights or decision functions with the given evaluation measurements of single models to approximate the user-desired measurements.

The system problem of model assembly is to incrementally find models that are to be used in the final ensemble model. Caruana et al. [147] study the problem of ensemble selection from a model library, and propose a general approach: 1) initialize ensemble as empty; 2) pick the model that maximizes the performance and add it to the ensemble; 3) repeat 2 until constraints are reached or all models are selected; 4) output the generated ensemble.

Discussion & Challenges

However, to build such system, a discovery system should be able to validate models, and better train the models in different projects. In practice, there may be only a subset of projects that have such a property. ModelHub Discovery prototype is now built on top of ModelHub, which now only contains deep learning projects. The model training and testing procedures in each repository are well understood, so that it allows us to explore this direction. On the other hand, model ensemble is also an important task for automatic model selections services [89, 147]; compared with discovering hosted projects, they deal with finding the best model via predefined prediction methods by giving a dataset and a goal. Identifying scalable system methods in this task is useful to improve the productivity for practitioners.

6.3 Evaluation Study

In this section, we study the usefulness of model alignment technique to identify similar models. Note that it is a challenge to find ground truth in the real-world repositories, as the prediction results need to be reproduced from the selected repositories. Instead, we finetune a VGG deep learning model to derive a synthetic model dataset.

6.3.1 Dataset Description

Fine-tuning is a popular modeling practice in deep learning, when the user has an existing model properly trained in a large dataset (such as ImageNet) and wants to apply such model to a new dataset which can have a different set of prediction labels. The motivation of fine-tuning is that a well trained model should have good generalization to various data so it becomes unnecessary to do end-to-end training of every model which may consume a large amount of time.

We start from the VGG-16 network which was originally trained using ImageNet dataset and fine-tune the model on CASIA face recognition dataset. CASIA dataset has 10575 face categories while we sample 1000 of them as the last layer prediction output. The way to fine-tune a model is to replace the last fully connected layer with a new one and set small or even zero learning rates for existing layers. The newly mutated model trained on the new dataset will converge much faster than an end-to-end training. It is often unnecessary to fine-tune early layers but it is necessary to set a small non-zero learning rates to some later existing layers. By

enumerating different pivot layers for fine-tuning, our fine-tuning model generator replicates such behavior.

To elaborate, each time, we first choose a pivot parametric layer in VGG16, before which the parameters are fixed while after which the parametric layers are retained. The layer-wise learning rate and weight decay scalars are set to be $(1, 2)$ and $(1, 0)$ for (weight, bias) respectively. For the last full layer (fc8 in VGG16, fc8_casia1k in new models), the scalars are set to be $(10, 10)$ for (weight, bias). Once the pivot layer is chosen, then we enumerate global optimization hyperparameters, learning rate in $[10^{-3}, 10^{-4}, 5 \times 10^{-5}]$, and weight decay in $[2 \times 10^{-4}, 5 \times 10^{-4}]$. In total, there are 54 different model configurations. Each of them are retrained over 10000 iterations, and checkpointed per 1000 iteration.

The models with their fine-tuning configurations and validation accuracies are shown in Figure 6.4. Each oval node is a model which is labeled with its generation order and the accuracy. The path from root to a model shows the detailed retraining configurations. Red circled models have the top 3 validation accuracies.

6.3.2 Evaluation Result

On the generated models, we first compute pair-wise weights distances (L_2) of the model pairs. Then we derive their prediction performance difference. Note each model produces a vector in $[0, 1]^{1000}$ from the last layer, which we refer to as result table. On the testing dataset, we compute pairwise *cosine* distances between the model results.

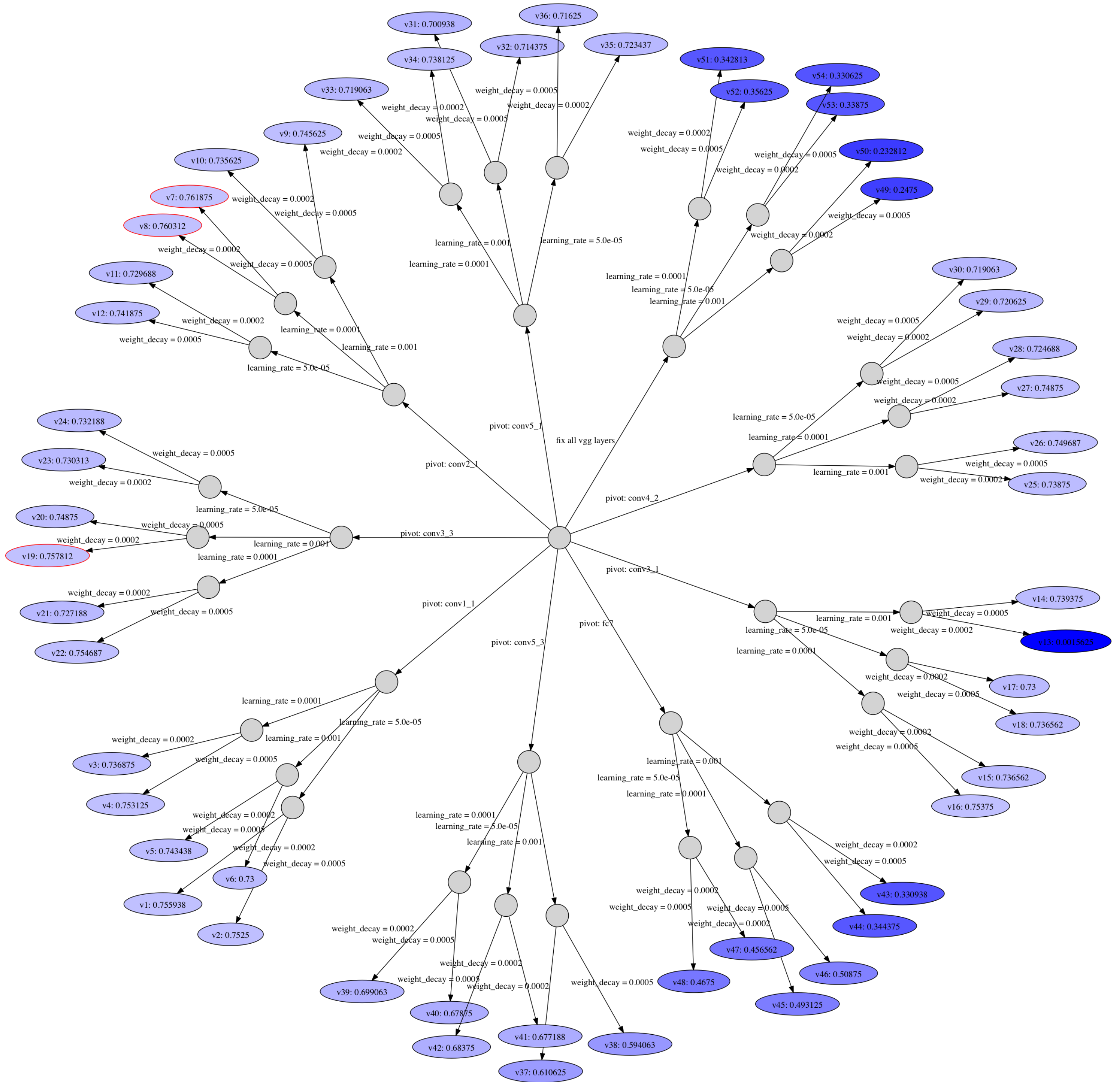


Figure 6.4: Finetuned VGG Models

In Figure 6.5, we show the relationship between model weight alignment distances and prediction performance distances; x axis shows the alignment distance,

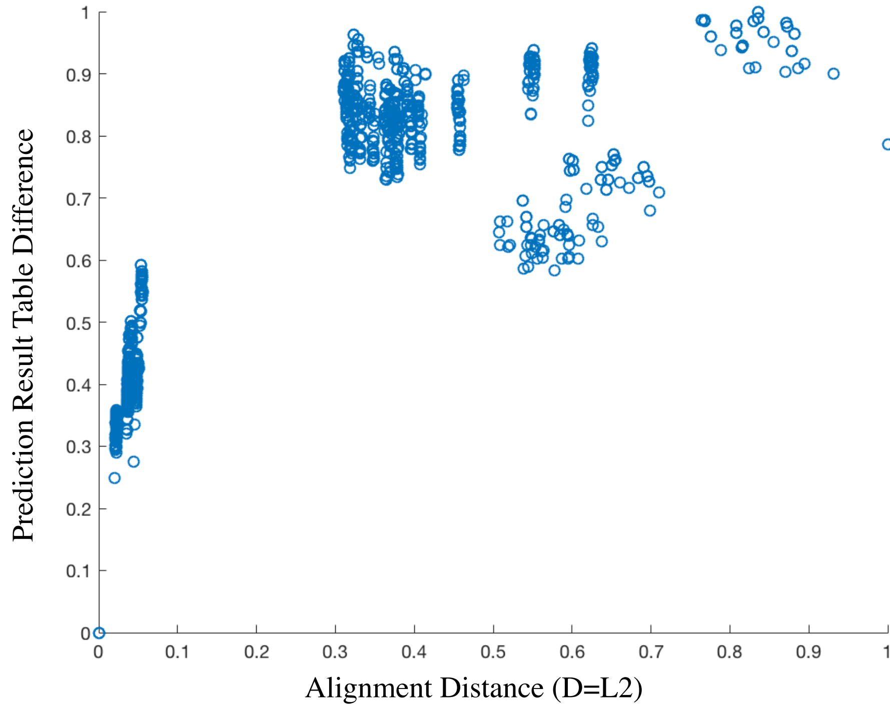


Figure 6.5: Correlation Between Aligned Distance and Prediction Results

y axis is the prediction performance distance. We normalized both of them to $[0, 1]$, so that the closer to 0, the more similar they are. As we can see from the figure, the smaller x (alignment distance) tend to have smaller y (prediction result table difference) as well. The Pearson correlation between x and y is 0.8224. From the preliminary evaluation, it shows the model similarly technique we propose would work for some modeling practices.

In a model discovery service, often the performance result is not known, due to the possible correlation between model weight alignment distance and the performance result difference, the service implementation would use the aligned distance as a signal to improve query results.

6.4 Conclusion

In this chapter, we explored systems research opportunity to enable data scientists to query and reuse data science projects hosted in a central service. In a hosted service storing enriched repositories from lifecycle management tools, we presented our vision of querying managed data science projects. Instead of querying on structured stores, we chose an information retrieval approach in order to better serve the needs from practitioners and described a search service allowing the user to query using project requirement languages, such as goals, datasets, model methods. We outlined the system architecture and proposed several challenging problems in building it, including developing model similarity methods and model assembly for constrained queries. Because in the real world, there were not that many repositories with well-extracted information yet, we conducted preliminary evaluation using synthetic models generated by following best practice, and showed the potential value and performance of proposed techniques.

Chapter 7: Conclusions

In this dissertation, we studied the practice of data science and explored the issues in the end-to-end lifecycle management of collaborative analytics workflows. We took a systems approach to unify the provenance and project artifacts management for a collaborative analytics team, and studied how to track provenance and derivation history of models, query modeling artifacts and processing pipelines, analyze unexpected behaviors, and search hosted repositories.

We first described ProvDB in Chapter 3, which is a general provenance ingestion and graph-based storage system that introduces a command line toolkit to wrap user commands and captures static and runtime information when users execute scripts. The capturing is done via a set of ingestor plugins, such as UNIX POSIX ingestors for basic commands (e.g., `mv` source and target), deep learning training tool ingestors (e.g., training accuracy and loss), core data science library ingestors (e.g., scikit-learn usages in a script), etc. It also features with a rich set of general query facilities tailored for data science lifecycles, such as introspecting the project artifacts and pipelines and monitoring the ongoing modeling activities.

By extending ProvDB ingestion mechanism, query facilities and storage backend, we described the ModelHub system in Chapter 4. It includes a specialized

version control system to capture parameters, hyperparameters, and relationships between revised deep learning models when the user adds and commits versions. Using this information, it extends ProvDB query facility and features a domain-specific query language to explore existing models and enumerate new models. By exploiting the model metadata and derivation history, it uses novel modeling archiving technique, which compacts the model storage while ensuring that higher quality models can be accessed within desired time constraints.

In Chapter 5, we studied how to fully exploit the ingested provenance and help analytics project teams. As collaborative analytics projects often have *unstable* lifecycles resulting in evolving and verbose provenance graphs, it is common that team members only have partial knowledge of the provenance graph. Without a predefined workflow skeleton and full understanding of contributing artifacts and steps, it is difficult to write graph queries and explore the provenance graph using modern graph databases and query languages. We formulated two graph query operators, segmentation and summarization, to query retrospective and prospective provenance of analytics workflows. The segmentation operator is able to induce a certain scope in the lineages of user-specified vertices to get insight about the derivation relationships among the vertices of interest. The summarization operator can combine similar segments together to show the common and alternative pipelines among those segments.

In the real world, there are more and more collaborative analysis projects being shared online. In Chapter 6, we presented our vision of a model discovery service, a system that enables data scientists to query and reuse data science projects hosted in

a central service, such as GitHub or ModelHub. Assuming many projects' repositories are enrinched by lifecycle management tools, we outlined a system architecture and proposed several challenging problems in building it, including developing model similarity methods and model assembly for constrained queries.

Bibliography

[1] Souvik Bhattacherjee, Amit Chavan, Silu Huang, Amol Deshpande, and Aditya G. Parameswaran. Principles of dataset versioning: Exploring the recreation/storage tradeoff. *Proceedings of the VLDB Endowment*, 8(12):1346–1357, 2015.

[2] Anant P. Bhardwaj, Souvik Bhattacherjee, Amit Chavan, Amol Deshpande, Aaron J. Elmore, Samuel Madden, and Aditya G. Parameswaran. Datahub: Collaborative data science & dataset version management at scale. In *CIDR 2015, Seventh Biennial Conference on Innovative Data Systems Research, Asilomar, CA, USA, January 4-7, 2015, Online Proceedings*, 2015.

[3] Juliana Freire, David Koop, Emanuele Santos, and Cláudio T. Silva. Provenance for computational tasks: A survey. *Computing in Science and Engineering*, 10(3):11–21, 2008.

[4] Susan B. Davidson and Juliana Freire. Provenance and scientific workflows: challenges and opportunities. In *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2008, Vancouver, BC, Canada, June 10-12, 2008*, pages 1345–1350, 2008.

[5] James Cheney, Laura Chiticariu, and Wang Chiew Tan. Provenance in databases: Why, how, and where. *Foundations and Trends in Databases*, 1(4):379–474, 2009.

[6] Bertram Ludäscher, Ilkay Altintas, Chad Berkley, Dan Higgins, Efrat Jaeger, Matthew Jones, Edward A Lee, Jing Tao, and Yang Zhao. Scientific workflow management and the Kepler system. *Concurrency and Computation: Practice and Experience*, 18(10):1039–1065, 2006.

[7] Tom Oinn, Mark Greenwood, Matthew Addis, M. Nedim Alpdemir, Justin Ferris, Kevin Glover, Carole Goble, Antoon Goderis, Duncan Hull, Darren Marvin, Peter Li, Phillip Lord, Matthew R. Pocock, Martin Senger, Robert Stevens, Anil Wipat, and Chris Wroe. Taverna: lessons in creating a workflow environment for the life sciences. *Concurrency and Computation: Practice and Experience*, 18(10):1067–1100, 2006.

[8] Belinda Giardine, Cathy Riemer, Ross C Hardison, Richard Burhans, Laura Elnitski, Prachi Shah, Yi Zhang, Daniel Blankenberg, Istvan Albert, James Taylor, Webb Miller, W. James Kent, and Anton Nekrutenko. Galaxy: a platform for interactive large-scale genome analysis. *Genome research*, 15(10):1451–1455, 2005.

[9] Stephen A Goff, Matthew Vaughn, Sheldon McKay, Eric Lyons, Ann E Stapleton, Damian Gessler, Naim Matasci, Liya Wang, Matthew Hanlon, Andrew Lenards, Andy Muir, Nirav Merchant, Sonya Lowry, Stephen Mock, Matthew Helmke, Adam Kubach, Martha Narro, Nicole Hopkins, David Micklos, Uwe Hilgert, Michael Gonzales, Chris Jordan, Edwin Skidmore, Rion Dooley, John Cazes, Robert McLay, Zhenyuan Lu, Shiran Pasternak, Lars Koesterke, William H Piel, Ruth Grene, Christos Noutsos, Karla Gendler, Xin Feng, Chunlao Tang, Monica Lent, Seung-Jin Kim, Kristian Kvilekval, B S Manjunath, Val Tannen, Alexandros Stamatakis, Michael Sanderson, Stephen M Welch, Karen A Cranston, Pamela Soltis, Doug Soltis, Brian O'Meara, Cecile Ane, Tom Brutnell, Daniel J Kleibenstein, Jeffery W White, James Leebens-Mack, Michael J Donoghue, Edgar P Spalding, Todd J Vision, Christopher R Myers, David Lowenthal, Brian J Enquist, Brad Boyle, Ali Akoglu, Greg Andrews, Sudha Ram, Doreen Ware, Lincoln Stein, and Dan Stanzione. The iPlant collaborative: cyberinfrastructure for plant biology. *Frontiers in plant science*, 2, 2011.

[10] Louis Bavoil, Steven P. Callahan, Carlos E. Scheidegger, Huy T. Vo, Patricia Crossno, Cláudio T. Silva, and Juliana Freire. Vistrails: Enabling interactive multiple-view visualizations. In *16th IEEE Visualization Conference, VIS 2005, Minneapolis, MN, USA, October 23-28, 2005*, pages 135–142, 2005.

[11] Ian T. Foster, Jens-S. Vöckler, Michael Wilde, and Yong Zhao. Chimera: Avirtual data system for representing, querying, and automating data derivation. In *Proceedings of the 14th International Conference on Scientific and Statistical Database Management, July 24-26, 2002, Edinburgh, Scotland, UK*, pages 37–46, 2002.

[12] Ewa Deelman, Gurmeet Singh, Mei-Hui Su, James Blythe, Yolanda Gil, Carl Kesselman, Gaurang Mehta, Karan Vahi, G. Bruce Berriman, John Good, Anastasia C. Laity, Joseph C. Jacob, and Daniel S. Katz. Pegasus: A framework for mapping complex scientific workflows onto distributed systems. *Scientific Programming*, 13(3):219–237, 2005.

[13] Yong Zhao, Michael Wilde, and Ian T. Foster. Applying the virtual data provenance model. In *Provenance and Annotation of Data, International Provenance and Annotation Workshop, IPAW 2006, Chicago, IL, USA, May 3-5, 2006, Revised Selected Papers*, pages 148–161.

[14] Philip J. Guo and Margo Seltzer. BURRITO: wrapping your lab notebook in computational infrastructure. In *4th Workshop on the Theory and Practice of Provenance, TaPP'12, Boston, MA, USA, June 14-15, 2012*, 2012.

[15] Fernando Seabra Chirigati, Dennis E. Shasha, and Juliana Freire. Reprozip: Using provenance to support computational reproducibility. In *5th Workshop on the Theory and Practice of Provenance, TaPP'13, Lombard, IL, USA, April 2-3, 2013*, 2013.

[16] Timothy M. McPhillips, Tianhong Song, Tyler Kolisnik, Steve Aulenbach, Khalid Belhajjame, Kyle Bocinsky, Yang Cao, Fernando Chirigati, Saumen C. Dey, Juliana Freire, Deborah N. Huntzinger, Christopher Jones, David Koop, Paolo Missier, Mark Schildhauer, Christopher R. Schwalm, Yaxing Wei, James Cheney, Mark Bieda, and Bertram Ludäscher. YesWorkflow: A user-oriented, language-independent tool for recovering workflow information from scripts. *International Journal of Digital Curation*, 10(1):298–313, 2015.

[17] Leonardo Murta, Vanessa Braganholo, Fernando Chirigati, David Koop, and Juliana Freire. noWorkflow: Capturing and analyzing provenance of scripts. In *Provenance and Annotation of Data and Processes - 5th International Provenance and Annotation Workshop, IPAW 2014, Cologne, Germany, June 9-13, 2014.*, pages 71–83.

[18] Luc Moreau, Ben Clifford, Juliana Freire, Joe Futrelle, Yolanda Gil, Paul Groth, Natalia Kwasnikowska, Simon Miles, Paolo Missier, Jim Myers, Beth Plale, Yogesh Simmhan, Eric Stephan, and Jan Van den Bussche. The open provenance model core specification (v1.1). *Future Generation Computer Systems*, 27(6):743 – 756, 2011.

[19] Luc Moreau and Paul Groth. PROV-overview. W3C note, W3C, 2013. http://www.w3.org/TR/2013/NOTE-prov-overview-20130430/.

[20] Catriel Beeri, Anat Eyal, Simon Kamenkovich, and Tova Milo. Querying business processes. In *Proceedings of the 32nd International Conference on Very Large Data Bases, Seoul, Korea, September 12-15, 2006*, pages 343–354, 2006.

[21] David A. Holland, Uri Jacob Braun, Diana Maclean, Kiran-Kumar Muniswamy-Reddy, and Margo I. Seltzer. Choosing a data model and query language for provenance. In *The 2nd International Provenance and Annotation Workshop*. Springer, 2008.

[22] Manish Kumar Anand, Shawn Bowers, and Bertram Ludäscher. Techniques for efficiently querying scientific workflow provenance graphs. In *EDBT 2010, 13th International Conference on Extending Database Technology, Lausanne, Switzerland, March 22-26, 2010, Proceedings*, pages 287–298, 2010.

[23] Olivier Biton, Sarah C. Boulakia, Susan B. Davidson, and Carmem S. Hara. Querying and managing provenance through user views in scientific workflows. In *Proceedings of the 24th International Conference on Data Engineering, ICDE 2008, April 7-12, 2008, Cancún, México*, pages 1072–1081, 2008.

[24] Paolo Missier, Jeremy Bryans, Carl Gamble, Vasa Curcin, and Roxana Dánger. Provabs: Model, policy, and tooling for abstracting PROV graphs. In *Provenance and Annotation of Data and Processes - 5th International Provenance and Annotation Workshop, IPAW 2014, Cologne, Germany, June 9-13, 2014*, pages 3–15, 2014.

[25] Luc Moreau. Aggregation by provenance types: A technique for summarising provenance graphs. In *Proceedings Graphs as Models, GaM@ETAPS 2015, London, UK, 11-12 April 2015.*, pages 129–144, 2015.

[26] Rui Abreu, Dave Archer, Erin Chapman, James Cheney, Hoda Eldardiry, and Adria Gascón. Provenance segmentation. In *8th Workshop on the Theory and Practice of Provenance, TaPP'16, Washington, D.C., USA, June 8-9, 2016*, 2016.

[27] Manish Kumar Anand, Shawn Bowers, Timothy M. McPhillips, and Bertram Ludäscher. Efficient provenance storage over nested data collections. In *EDBT 2009, 12th International Conference on Extending Database Technology, Saint Petersburg, Russia, March 24-26, 2009, Proceedings*, pages 958–969, 2009.

[28] Yulai Xie, Kiran-Kumar Muniswamy-Reddy, Dan Feng, Yan Li, and Darrell D. E. Long. Evaluation of a hybrid approach for efficient provenance storage. *TOS*, 9(4):14:1–14:29, 2013.

[29] Saumen C. Dey, Daniel Zinn, and Bertram Ludäscher. ProPub: Towards a declarative approach for publishing customized, policy-aware provenance. In *Proceedings of the 23rd International Conference, SSDBM 2011, Portland, OR, USA, July 20-22, 2011*, pages 225–243, 2011.

[30] James Cheney and Roly Perera. An analytical survey of provenance sanitization. In *Provenance and Annotation of Data and Processes - 5th International Provenance and Annotation Workshop, IPAW 2014, Cologne, Germany, June 9-13, 2014. Revised Selected Papers*, pages 113–126, 2014.

[31] Peter Buneman, Sanjeev Khanna, and Wang Chiew Tan. Why and where: A characterization of data provenance. In *Proceedings of the 8th International Conference on Database Theory, ICDT 2001, London, UK, January 4-6, 2001*, pages 316–330, 2001.

[32] Jennifer Widom. Trio: A system for integrated management of data, accuracy, and lineage. In *CIDR 2005, Second Biennial Conference on Innovative Data Systems Research, Asilomar, CA, USA, January 4-7, 2005, Online Proceedings*, pages 262–276, 2005.

[33] Peter Buneman, Adriane Chapman, and James Cheney. Provenance management in curated databases. In *Proceedings of the ACM SIGMOD International Conference on Management of Data, Chicago, Illinois, USA, June 27-29, 2006*, pages 539–550, 2006.

[34] Todd J. Green, Gregory Karvounarakis, and Val Tannen. Provenance semirings. In *Proceedings of the Twenty-Sixth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems, June 11-13, 2007, Beijing, China*, pages 31–40, 2007.

[35] Matteo Interlandi, Kshitij Shah, Sai Deep Tetali, Muhammad Ali Gulzar, Seunghyun Yoo, Miryung Kim, Todd Millstein, and Tyson Condie. Titian: Data provenance support in Spark. *Proceedings of the VLDB Endowment*, 9(3), 2015.

[36] Jayant Madhavan, Sreeram Balakrishnan, Kathryn Brisbin, Hector Gonzalez, Nitin Gupta, Alon Y. Halevy, Karen Jacqmin-Adams, Heidi Lam, Anno Langen, Hongrae Lee, Rod McChesney, Rebecca Shapley, and Warren Shen. Big Data Storytelling Through Interactive Maps. *IEEE Data Eng. Bull.*, 35(2):46–54, 2012.

[37] Zachary G. Ives, Nitin Khandelwal, Aneesh Kapur, and Murat Cakir. ORCHESTRA: rapid, collaborative sharing of dynamic data. In *CIDR 2005, Second Biennial Conference on Innovative Data Systems Research, Asilomar, CA, USA, January 4-7, 2005, Online Proceedings*, pages 107–118, 2005.

[38] Nodira Khoussainova, Magdalena Balazinska, Wolfgang Gatterbauer, YongChul Kwon, and Dan Suciu. A case for A collaborative query management system. In *CIDR 2009, Fourth Biennial Conference on Innovative Data Systems Research, Asilomar, CA, USA, January 4-7, 2009, Online Proceedings*, 2009.

[39] Bill Howe, Garrett Cole, Emad Souroush, Paraschos Koutris, Alicia Key, Nodira Khoussainova, and Leilani Battle. Database-as-a-service for long-tail science. In *Proceedings of the 23rd International Conference, SSDBM 2011, Portland, OR, USA, July 20-22, 2011*, pages 480–489, 2011.

[40] Eser Kandogan, Mary Roth, Peter M. Schwarz, Joshua Hui, Ignacio Terrizzano, Christina Christodoulakis, and Renée J. Miller. Labbook: Metadata-driven social collaborative data analysis. In *2015 IEEE International Conference on Big Data, Big Data 2015, Santa Clara, CA, USA, October 29 - November 1, 2015*, pages 431–440.

[41] `http://ckan.org`. CKAN: an Open-Source Data Portal (retrieved June 1, 2014).

[42] `http://www.quandl.com`. Quandl (retrieved June 1, 2014).

[43] http://www.factual.com. Factual Inc. (retrieved June 1, 2014).

[44] Joaquin Vanschoren, Jan N. van Rijn, Bernd Bischl, and Luis Torgo. Openml: networked science in machine learning. *SIGKDD Explorations*, 15(2):49–60, 2013.

[45] http://www.dominoup.com. Domino Data Lab (retrieved June 1, 2014).

[46] https://aws.amazon.com/sagemaker. Amazon SageMaker (retrieved May 1, 2018).

[47] https://cloud.google.com/datalab/. Google Datalab (retrieved May 1, 2018).

[48] http://www.datamarket.com. Data Market Inc. (retrieved June 1, 2014).

[49] Alon Y. Halevy, Flip Korn, Natalya Fridman Noy, Christopher Olston, Neoklis Polyzotis, Sudip Roy, and Steven Euijong Whang. Goods: Organizing google's datasets. In *Proceedings of the 2016 International Conference on Management of Data, SIGMOD Conference 2016, San Francisco, CA, USA, June 26 - July 01, 2016*, pages 795–806, 2016.

[50] Joseph M. Hellerstein, Vikram Sreekanti, Joseph E. Gonzalez, James Dalton, Akon Dey, Sreyashi Nag, Krishna Ramachandran, Sudhanshu Arora, Arka Bhattacharyya, Shirshanka Das, Mark Donsky, Gabriel Fierro, Chang She, Carl Steinbach, Venkat Subramanian, and Eric Sun. Ground: A data context service. In *CIDR 2017, 8th Biennial Conference on Innovative Data Systems Research, Chaminade, CA, USA, January 8-11, 2017, Online Proceedings*, 2017.

[51] https://www.collibra.com. Collibra (retrieved May 1, 2018).

[52] http://atlas.apache.org. Apache Atlas (retrieved May 1, 2018).

[53] https://alation.com. Alation (retrieved May 1, 2018).

[54] http://www.sas.com. SAS Business Analytics Software (retrieved Oct. 14, 2013).

[55] http://office.microsoft.com/en-us/excel/. Microsoft Excel (retrieved Oct. 14, 2013).

[56] http://www.r-project.org. The R Project (retrieved Oct. 14, 2013).

[57] http://www.mathworks.com. Mathworks Matlab (retrieved Oct. 14, 2013).

[58] http://mahout.apache.org. Apache Mahout Machine Learning Library (retrieved Oct. 14, 2013).

[59] http://ipython.org. IPython (retrieved June 1, 2014).

[60] http://scikit-learn.org. Scikit-Learn: Machine Learning in Python (retrieved June 1, 2014).

[61] http://pandas.pydata.org. Python Data Analysis Library (retrieved June 1, 2014).

[62] Amit Chavan, Silu Huang, Amol Deshpande, Aaron J. Elmore, Samuel Madden, and Aditya G. Parameswaran. Towards a unified query language for provenance and versioning. 2015.

[63] Amit Chavan and Amol Deshpande. DEX: query execution in a delta-based storage system. pages 171–186, 2017.

[64] Silu Huang, Liqi Xu, Jialin Liu, Aaron J. Elmore, and Aditya G. Parameswaran. OrpheusDB: Bolt-on versioning for relational databases. *PVLDB*, 10(10):1130–1141, 2017.

[65] Samir Khuller, Balaji Raghavachari, and Neal E. Young. Balancing minimum spanning trees and shortest-path trees. *Algorithmica*, 14(4):305–321, 1995.

[66] Evan R. Sparks, Ameet Talwalkar, Virginia Smith, Jey Kottalam, Xinghao Pan, Joseph E. Gonzalez, Michael J. Franklin, Michael I. Jordan, and Tim Kraska. MLI: an API for distributed machine learning. In *2013 IEEE 13th International Conference on Data Mining, Dallas, TX, USA, December 7-10, 2013*, pages 1187–1192, 2013.

[67] Yucheng Low, Joseph Gonzalez, Aapo Kyrola, Danny Bickson, Carlos Guestrin, and Joseph M. Hellerstein. Distributed GraphLab: A framework for machine learning in the cloud. *Proceedings of the VLDB Endowment*, 5(8):716–727, 2012.

[68] Mu Li, David G. Andersen, Jun Woo Park, Alexander J. Smola, Amr Ahmed, Vanja Josifovski, James Long, Eugene J. Shekita, and Bor-Yiing Su. Scaling distributed machine learning with the parameter server. In *11th USENIX Symposium on Operating Systems Design and Implementation, OSDI '14, Broomfield, CO, USA, October 6-8, 2014.*, pages 583–598, 2014.

[69] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, Manjunath Kudlur, Josh Levenberg, Rajat Monga, Sherry Moore, Derek Gordon Murray, Benoit Steiner, Paul A. Tucker, Vijay Vasudevan, Pete Warden, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. Tensorflow: A system for large-scale machine learning. In *12th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2016, Savannah, GA, USA, November 2-4, 2016.*, pages 265–283, 2016.

[70] http://www.numpy.org. NumPy (retrieved June 1, 2014).

[71] `http://www.scipy.org`. SciPy (retrieved June 1, 2014).

[72] Yangqing Jia, Evan Shelhamer, Jeff Donahue, Sergey Karayev, Jonathan Long, Ross B. Girshick, Sergio Guadarrama, and Trevor Darrell. Caffe: Convolutional architecture for fast feature embedding. In *Proceedings of the ACM International Conference on Multimedia, MM '14, Orlando, FL, USA, November 03 - 07, 2014*.

[73] Arun Kumar, Matthias Boehm, and Jun Yang. Data management in machine learning: Challenges, techniques, and systems. In *Proceedings of the 2017 ACM International Conference on Management of Data, SIGMOD Conference 2017, Chicago, IL, USA, May 14-19, 2017*, pages 1717–1722, 2017.

[74] Mert Akdere, Ugur Çetintemel, Matteo Riondato, Eli Upfal, and Stanley B. Zdonik. The case for predictive database systems: Opportunities and challenges. In *CIDR 2011, Fifth Biennial Conference on Innovative Data Systems Research, Asilomar, CA, USA, January 9-12, 2011, Online Proceedings*, pages 167–174, 2011.

[75] Xixuan Feng, Arun Kumar, Benjamin Recht, and Christopher Ré. Towards a unified architecture for in-RDBMS analytics. In *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2012, Scottsdale, AZ, USA, May 20-24, 2012*, pages 325–336, 2012.

[76] Arun Kumar, Jeffrey F. Naughton, and Jignesh M. Patel. Learning generalized linear models over normalized data. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data, Melbourne, Victoria, Australia, May 31 - June 4, 2015*, pages 1969–1984.

[77] Vineet Chaoji, Rajeev Rastogi, and Gourav Roy. Machine learning in the real world. *Proceedings of the VLDB Endowment*, 9(13):1597–1600, 2016.

[78] Neoklis Polyzotis, Sudip Roy, Steven Euijong Whang, and Martin Zinkevich. Data management challenges in production machine learning. In *Proceedings of the 2017 ACM International Conference on Management of Data, SIGMOD Conference 2017, Chicago, IL, USA, May 14-19, 2017*, pages 1723–1726, 2017.

[79] Ce Zhang, Arun Kumar, and Christopher Ré. Materialization optimizations for feature selection workloads. *ACM Transactions on Database Systems (TODS)*, 41(1):2, 2016.

[80] Daniel Crankshaw, Peter Bailis, Joseph E. Gonzalez, Haoyuan Li, Zhao Zhang, Michael J. Franklin, Ali Ghodsi, and Michael I. Jordan. The missing piece in complex analytics: Low latency, scalable model management and serving with Velox. In *CIDR 2015, Seventh Biennial Conference on Innovative Data Systems Research, Asilomar, CA, USA, January 4-7, 2015, Online Proceedings*, 2015.

[81] Christopher Olston, Noah Fiedel, Kiril Gorovoy, Jeremiah Harmsen, Li Lao, Fangwei Li, Vinu Rajashekhar, Sukriti Ramesh, and Jordan Soyke. Tensorflow-serving: Flexible, high-performance ml serving. In *NIPS Workshop on ML Systems (LearningSys)*, 2017.

[82] Manasi Vartak, Harihar Subramanyam, Wei-En Lee, Srinidhi Viswanathan, Saadiyah Husnoo, Samuel Madden, and Matei Zaharia. ModelDB: a system for machine learning model management. In *Proceedings of the Workshop on Human-In-the-Loop Data Analytics*, page 14. ACM, 2016.

[83] Hui Miao, Ang Li, Larry S. Davis, and Amol Deshpande. Towards unified data and lifecycle management for deep learning. In *33rd IEEE International Conference on Data Engineering, ICDE 2017, San Diego, CA, USA, April 19-22, 2017*, pages 571–582, 2017.

[84] Sebastian Schelter, Joos-Hendrik Boese, Johannes Kirschnick, Thoralf Klein, and Stephan Seufert. Automatically tracking metadata and provenance of machine learning experiments. In *NIPS Workshop on ML Systems (LearningSys)*, 2017.

[85] Hui Miao, Amit Chavan, and Amol Deshpande. ProvDB: Lifecycle management of collaborative analysis workflows. In *Proceedings of the 2nd Workshop on Human-In-the-Loop Data Analytics, HILDA@SIGMOD 2017, Chicago, IL, USA, May 14, 2017*, pages 7:1–7:6, 2017.

[86] Hui Miao, Ang Li, Larry S. Davis, and Amol Deshpande. On model discovery for hosted data science projects. In *Proceedings of the 1st Workshop on Data Management for End-To-End Machine Learning, DEEM@SIGMOD 2017, Chicago, IL, USA, May 14 - 19, 2017*, 2017.

[87] Denis Baylor, Eric Breck, Heng-Tze Cheng, Noah Fiedel, Chuan Yu Foo, Zakaria Haque, Salem Haykal, Mustafa Ispir, Vihan Jain, Levent Koc, Chiu Yuen Koo, Lukasz Lew, Clemens Mewald, Akshay Naresh Modi, Neoklis Polyzotis, Sukriti Ramesh, Sudip Roy, Steven Euijong Whang, Martin Wicke, Jarek Wilkiewicz, Xin Zhang, and Martin Zinkevich. TFX: A tensorflow-based production-scale machine learning platform. In *Proceedings of the 23rd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, Halifax, NS, Canada, August 13 - 17, 2017*, pages 1387–1395, 2017.

[88] Evan R. Sparks, Shivaram Venkataraman, Tomer Kaftan, Michael J. Franklin, and Benjamin Recht. KeystoneML: Optimizing pipelines for large-scale advanced analytics. In *33rd IEEE International Conference on Data Engineering, ICDE 2017, San Diego, CA, USA, April 19-22, 2017*, pages 535–546, 2017.

[89] `https://cloud.google.com/prediction`. Google Prediction API (retrieved Aug.20, 2016).

[90] Arun Kumar, Robert McCann, Jeffrey F. Naughton, and Jignesh M. Patel. Model selection management systems: The next frontier of advanced analytics. *SIGMOD Record*, 44(4):17–22, 2015.

[91] Randy H Katz. Toward a unified framework for version modeling in engineering databases. *ACM Computing Surveys (CSUR)*, 22(4):375–409, 1990.

[92] Yann LeCun, Yoshua Bengio, and Geoffrey E. Hinton. Deep learning. *Nature*, 521(7553):436–444, 2015.

[93] Yann LeCun, Bernhard E. Boser, John S. Denker, Donnie Henderson, Richard E. Howard, Wayne E. Hubbard, and Lawrence D. Jackel. Handwritten digit recognition with a back-propagation network. In *Advances in Neural Information Processing Systems 2, NIPS Conference, Denver, Colorado, USA, November 27-30, 1989*, pages 396–404, 1989.

[94] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E. Hinton. Imagenet classification with deep convolutional neural networks. In *Advances in Neural Information Processing Systems 25, NIPS Conference, Lake Tahoe, Nevada, USA, December 3-6, 2012*, pages 1106–1114, 2012.

[95] Karen Simonyan and Andrew Zisserman. Very deep convolutional networks for large-scale image recognition. In *Proceedings of the Third International Conference on Learning Representations (ICLR)*, 2015.

[96] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *2016 IEEE Conference on Computer Vision and Pattern Recognition, CVPR 2016, Las Vegas, NV, USA, June 27-30, 2016*, pages 770–778, 2016.

[97] Léon Bottou. Stochastic gradient descent tricks. In *Neural Networks: Tricks of the Trade*, pages 421–436. Springer, 2012.

[98] Vincent Vanhoucke, Andrew Senior, and Mark Z Mao. Improving the speed of neural networks on CPUs. In *Proc. Deep Learning and Unsupervised Feature Learning NIPS Workshop*, 2011.

[99] Song Han, Huizi Mao, and William J. Dally. Deep compression: Compressing deep neural networks with pruning, trained quantization and huffman coding. In *Proceedings of the Fourth International Conference on Learning Representations (ICLR)*, 2016.

[100] Matthieu Courbariaux, Yoshua Bengio, and Jean-Pierre David. Training deep neural networks with low precision multiplications. *arXiv preprint arXiv:1412.7024*, 2014.

[101] Eric R. Schendel, Ye Jin, Neil Shah, Jackie Chen, Choong-Seock Chang, Seung-Hoe Ku, Stéphane Ethier, Scott Klasky, Robert Latham, Robert B.

Ross, and Nagiza F. Samatova. ISOBAR preconditioner for effective and high-throughput lossless data compression. In *IEEE 28th International Conference on Data Engineering (ICDE 2012), Washington, DC, USA (Arlington, Virginia), 1-5 April, 2012*, pages 138–149.

[102] Souvik Bhattacherjee, Amol Deshpande, and Alan Sussman. Pstore: an efficient storage framework for managing scientific data. In *Conference on Scientific and Statistical Database Management, SSDBM '14, Aalborg, Denmark, June 30 - July 02, 2014*, pages 25:1–25:12, 2014.

[103] Narsingh Deo and Nishit Kumar. Computation of constrained spanning trees: A unified approach. In *Network Optimization*. 1997.

[104] Andrew B Kahng and Gabriel Robins. *On optimal interconnections for VLSI*, volume 301. Springer Science & Business Media, 1994.

[105] Ken Chatfield, Karen Simonyan, Andrea Vedaldi, and Andrew Zisserman. Return of the devil in the details: Delving deep into convolutional nets. In *British Machine Vision Conference, BMVC 2014, Nottingham, UK, September 1-5, 2014*, 2014.

[106] Jianming Zhang, Shugao Ma, Mehrnoosh Sameki, Stan Sclaroff, Margrit Betke, Zhe L. Lin, Xiaohui Shen, Brian L. Price, and Radomír Mech. Salient object subitizing. In *IEEE Conference on Computer Vision and Pattern Recognition, CVPR 2015, Boston, MA, USA, June 7-12, 2015*, pages 4045–4054, 2015.

[107] Ross B. Girshick. Fast R-CNN. In *2015 IEEE International Conference on Computer Vision, ICCV 2015, Santiago, Chile, December 7-13, 2015*, pages 1440–1448, 2015.

[108] Adam M. Bates, Dave Tian, Kevin R. B. Butler, and Thomas Moyer. Trustworthy whole-system provenance for the linux kernel. In *24th USENIX Security Symposium, USENIX Security 15, Washington, D.C., USA, August 12-14, 2015.*, pages 319–334, 2015.

[109] Provenance Challenge. `http://twiki.ipaw.info`. Accessed: 2017-07.

[110] Pablo Barceló Baeza. Querying graph databases. In *Proceedings of the 32nd ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems, PODS 2013, New York, NY, USA - June 22 - 27, 2013*, pages 175–188, 2013.

[111] Oskar van Rest, Sungpack Hong, Jinha Kim, Xuming Meng, and Hassan Chafi. PGQL: a property graph query language. In *Proceedings of the Fourth International Workshop on Graph Data Management Experiences and Systems, Redwood Shores, CA, USA, June 24 - 24, 2016*, page 7, 2016.

[112] Kiran-Kumar Muniswamy-Reddy, David A. Holland, Uri Braun, and Margo I. Seltzer. Provenance-Aware Storage Systems. In *USENIX Annual Technical Conference, General Track*, pages 43–56, 2006.

[113] Adam M. Bates, Wajih Ul Hassan, Kevin R. B. Butler, Alin Dobra, Bradley Reaves, Patrick T. Cable II, Thomas Moyer, and Nabil Schear. Transparent web service auditing via network provenance functions. In *Proceedings of the 26th International Conference on World Wide Web, WWW 2017, Perth, Australia, April 3-7, 2017*, pages 887–895, 2017.

[114] Paolo Missier and Luc Moreau. PROV-dm: The PROV data model. W3C recommendation, W3C, 2013. http://www.w3.org/TR/2013/REC-prov-dm-20130430/.

[115] Zhuowei Bao, Susan B. Davidson, Sanjeev Khanna, and Sudeepa Roy. An optimal labeling scheme for workflow provenance using skeleton labels. In *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2010, Indianapolis, Indiana, USA, June 6-10, 2010*, pages 711–722, 2010.

[116] Yuanyuan Tian, Richard A. Hankins, and Jignesh M. Patel. Efficient aggregation for graph summarization. In *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2008, Vancouver, BC, Canada, June 10-12, 2008*, pages 567–580, 2008.

[117] Yinghui Wu, Shengqi Yang, Mudhakar Srivatsa, Arun Iyengar, and Xifeng Yan. Summarizing answer graphs induced by keyword queries. *Proceedings of the VLDB Endowment*, 6(14):1774–1785, 2013.

[118] Arijit Khan, Sourav S. Bhowmick, and Francesco Bonchi. Summarizing static and dynamic big graphs. *Proceedings of the VLDB Endowment*, 10(12):1981–1984, 2017.

[119] Luc Moreau, James Cheney, and Paolo Missier. Constraints of the PROV data model. W3C recommendation, W3C, 2013. http://www.w3.org/TR/2013/REC-prov-constraints-20130430/.

[120] Peixiang Zhao, Xiaolei Li, Dong Xin, and Jiawei Han. Graph cube: on warehousing and OLAP multidimensional networks. In *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2011, Athens, Greece, June 12-16, 2011*, pages 853–864, 2011.

[121] Wenfei Fan, Zhe Fan, Chao Tian, and Xin Luna Dong. Keys for graphs. *Proceedings of the VLDB Endowment*, 8(12):1590–1601, 2015.

[122] Zhengkui Wang, Qi Fan, Huiju Wang, Kian-Lee Tan, Divyakant Agrawal, and Amr El Abbadi. Pagrol: Parallel graph olap over large-scale attributed graphs. In *IEEE 30th International Conference on Data Engineering, Chicago, ICDE 2014, IL, USA, March 31 - April 4, 2014*, pages 496–507, 2014.

[123] Haixun Wang and Charu C. Aggarwal. A survey of algorithms for keyword search on graph data. In *Managing and Mining Graph Data*, pages 249–273. 2010.

[124] John E. Hopcroft, Rajeev Motwani, and Jeffrey D. Ullman. *Introduction to Automata Theory, Languages, and Computation.* Pearson Addison Wesley, 3rd edition, 2007.

[125] Peter T. Wood. Query languages for graph databases. *SIGMOD Rec.*, 41(1):50–60, April 2012.

[126] Pablo Barceló, Leonid Libkin, Anthony Widjaja Lin, and Peter T. Wood. Expressive languages for path queries over graph-structured data. *ACM Trans. Database Syst.*, 37(4):31:1–31:46, 2012.

[127] Mihalis Yannakakis. Graph-theoretic methods in database theory. In *Proceedings of the Ninth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems, April 2-4, 1990, Nashville, Tennessee, USA*, pages 230–242, 1990.

[128] Thomas W. Reps. Program analysis via graph reachability. In *ILPS '97, Proceedings of the 1997 International Symposium on Logic Programming, Port Jefferson, Long Island, NY, USA, October 13-16, 1997*, pages 5–19, 1997.

[129] Swarat Chaudhuri. Subcubic algorithms for recursive state machines. In *Proceedings of the 35th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2008, San Francisco, California, USA, January 7-12, 2008*, pages 159–169, 2008.

[130] David Melski and Thomas W. Reps. Interconvertibility of a class of set constraints and context-free-language reachability. *Theor. Comput. Sci.*, 248(1-2):29–98, 2000.

[131] V. L. Arlazarov, E. A. Dinic, M. A. Kronrod, and I. A. Faradzev. On economical construction of the transitive closure of a directed graph. *Doklady Akademii Nauk SSSR*, 194(3):487, 1970.

[132] Istvan Jonyer, Lawrence B. Holder, and Diane J. Cook. Mdl-based context-free graph grammar induction and applications. *International Journal on Artificial Intelligence Tools*, 13(1):65–79, 2004.

[133] Saket Navlakha, Rajeev Rastogi, and Nisheeth Shrivastava. Graph summarization with bounded error. In *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2008, Vancouver, BC, Canada, June 10-12, 2008*, pages 419–432, 2008.

[134] Raghav Kaushik, Pradeep Shenoy, Philip Bohannon, and Ehud Gudes. Exploiting local similarity for indexing paths in graph-structured data. In *Proceedings of the 18th International Conference on Data Engineering, San Jose, CA, USA, February 26 - March 1, 2002*.

[135] Yu Huang, Ziyang Liu, and Yi Chen. Query biased snippet generation in XML search. In *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2008, Vancouver, BC, Canada, June 10-12, 2008*, pages 315–326, 2008.

[136] Larry J. Stockmeyer and Albert R. Meyer. Word problems requiring exponential time: Preliminary report. In *Proceedings of the 5th Annual ACM Symposium on Theory of Computing, April 30 - May 2, 1973, Austin, Texas, USA*, pages 1–9, 1973.

[137] Monika Rauch Henzinger, Thomas A. Henzinger, and Peter W. Kopke. Computing simulations on finite and infinite graphs. In *36th Annual Symposium on Foundations of Computer Science, Milwaukee, Wisconsin, 23-25 October 1995*, pages 453–462, 1995.

[138] Tova Milo and Dan Suciu. Index structures for path expressions. In *Database Theory - ICDT '99, 7th International Conference, Jerusalem, Israel, January 10-12, 1999, Proceedings.*, pages 277–295, 1999.

[139] Ruslan Mavlyutov, Carlo Curino, Boris Asipov, and Philippe Cudré-Mauroux. Dependency-driven analytics: A compass for uncharted data oceans. In *CIDR 2017, 8th Biennial Conference on Innovative Data Systems Research, Chaminade, CA, USA, January 8-11, 2017, Online Proceedings*, 2017.

[140] Christopher D. Manning, Prabhakar Raghavan, and Hinrich Schütze. *Introduction to information retrieval*. Cambridge University Press, 2008.

[141] Kaspar Riesen. *Structural Pattern Recognition with Graph Edit Distance: Approximation Algorithms and Applications*. Springer, 2016.

[142] Ingwer Borg and Patrick JF Groenen. *Modern multidimensional scaling: Theory and applications*. Springer Science & Business Media, 2005.

[143] Roberto J. Bayardo, Yiming Ma, and Ramakrishnan Srikant. Scaling up all pairs similarity search. In *Proceedings of the 16th International Conference on World Wide Web, WWW 2007, Banff, Alberta, Canada, May 8-12, 2007*, pages 131–140, 2007.

[144] Zhi-Hua Zhou. *Ensemble methods: foundations and algorithms*. CRC press, 2012.

[145] Brandyn A. White, Andrew E. Miller, and Larry S. Davis. Classifier-as-a-service: Online query of cascades and operating points. In *Workshop on Big*

Data Meets Computer Vision 2012, co-located with NIPS 2012, pages 1–5, 2012.

[146] Michael Feldman, Sorelle A. Friedler, John Moeller, Carlos Scheidegger, and Suresh Venkatasubramanian. Certifying and removing disparate impact. In *Proceedings of the 21th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, Sydney, NSW, Australia, August 10-13, 2015*, pages 259–268, 2015.

[147] Rich Caruana, Alexandru Niculescu-Mizil, Geoff Crew, and Alex Ksikes. Ensemble selection from libraries of models. In *Machine Learning, Proceedings of the Twenty-first International Conference (ICML 2004), Banff, Alberta, Canada, July 4-8, 2004*, 2004.