EFFECTIVE DATA VERSIONING
FOR COLLABORATIVE DATA ANALYTICS

BY

SILU HUANG

DISSERTATION

Submitted in partial fulfillment of the requirements
for the degree of Doctor of Philosophy in Computer Science
in the Graduate College of the
University of Illinois at Urbana-Champaign, 2019

Urbana, Illinois

ABSTRACT

With the massive proliferation of datasets in a variety of sectors, data science teams in these sectors spend vast amounts of time collaboratively constructing, curating, and analyzing these datasets. Versions of datasets are routinely generated during this data science process, via various data processing operations like data transformation and cleaning, feature engineering and normalization, among others. However, no existing systems enable us to effectively store, track, and query these versioned datasets, leading to massive redundancy in versioned data storage and making true collaboration and sharing impossible. In this thesis, we develop solutions for versioned data management for collaborative data analytics.

In the first part of this thesis, we extend a relational database to support versioning of structured data. Specifically, we build a system, OrpheusDB, on top of a relational database with a carefully designed data representation and an intelligent partitioning algorithm for fast version control operations. OrpheusDB inherits much of the same benefits of relational databases, while also compactly storing, keeping track of, and recreating versions on demand. However, OrpheusDB implicitly makes a few assumptions, namely that: *(a) the SQL assumption:* a SQL-like language is the best fit for querying data and versioning information; *(b) the structural assumption:* the data is in a relational format with a regular structure; *(c) the from-scratch assumption:* users adopt OrpheusDB from the very beginning of their project and register each data version along with full metadata in the system. In the second part of this thesis, we remove each of these assumptions, one at a time. First, we remove the SQL assumption and propose a generalized query language for querying data along with versioning and provenance information. Second, we remove the structural assumption and develop solutions for compact storage and fast retrieval of arbitrary data representations. Finally, we remove the "from-scratch" assumption, by developing techniques to infer lineage relationships among versions residing in an existing data repository.

To my parents and my husband, for their love and support.

ACKNOWLEDGMENTS

First of all, I would like to express my sincere appreciation to my awesome advisor Professor Aditya Parameswaran. It is my great fortune to be working with him during my whole Ph.D. study. I learned a lot from him, not only about how to do research but also about how to be a good researcher. He helped me build confidence in myself and makes me believe I can make it. I will always remember and cherish these valuable memories. Thanks a lot.

I have also been fortunate to work with some other great professors—Professors Aaron Elmore, Saurabh Sinha, and Amol Deshpande. Thanks, Aaron for his kind support and thought-provoking discussions during our four years' collaboration. I enjoyed working with him a lot. Thanks, Saurabh for providing me much flexibility and many useful suggestions on our project. Thanks, Amol for giving me encouragement and nice advice during my first-year study. I would also like to express my gratitude to my committee members: Professors Jiawei Han, Saurabh Sinha, and Aaron Elmore for spending their valuable time on my prelim, defense, and thesis. Thanks for their insightful questions, helpful suggestions, and comments.

In addition, I would like to thank my amazing mentors during my three internships, Bolin Ding, Chi Wang, Jialu Liu, and Flip Korn. Their kind help and insightful advice shape me to be a better researcher. Special Thanks to Bolin for giving me, a first-year Ph.D. student back then, the opportunity to work with him as a research intern in Microsoft Research.

This thesis cannot be done without the kind help and support from my great collaborators: Liqi Xu, Amit Chavan, Souvik Bhattacherjee, Suhail Rehman, and Jialin Liu. I appreciate and enjoy working with them a lot. Also, thanks, National Institutes of Health (NIH), National Science Foundation (NSF), 3M, and Microsoft for providing me funds during my Ph.D. study. Special thanks to Microsoft and Data Management, Exploration and Mining group (DMX) for giving me the fellowship award, which is tremendous recognition to me.

Life is great here thanks to my cool friends and officemates: Yunhui Long, Le Xu, Yanhua Sun, Yihan Gao, Qi Wang, Jia Wang, Liwei Wang, Liqi Xu, Sajjadur Rahman, Tarique Siddiqui, Mangesh Bendre. Thanks a lot for all their help, company, and the happiness they bring to me.

Last but not least, I would like to thank my awesome parents, Ronghui Huang and Yanmei Cheng, as well as my amazing husband Xiaobin Gao for their unconditional support, company, and love. I love you.

TABLE OF CONTENTS

CHAPTER 1: INTRODUCTION

1.1 BACKGROUND

Thanks to advances in data gathering and collection mechanisms, increasing volumes of data are available across a variety of sectors. Data science teams in these sectors spend vast amounts of time collaboratively constructing, curating, and analyzing these datasets. Typically, the collaboration is facilitated by means of a hosted shared file system that every team member has access to. When team members want to analyze a particular dataset, they typically make a private copy of some version of that dataset and then make changes to this copy in the process of analyzing it, including, but not limited to: adding derived columns, normalizing data, and removing erroneous values, resulting in a new version. Subsequently, this new dataset version may be shared with other team members, who may then proceed to add their own modifications. Overall, hundreds to thousands of versions of the same dataset may be constructed in this manner, all of which are stored in the same shared file system. This process of collaboration leads to massive redundancy in the stored datasets and makes the sharing of the newly constructed dataset versions near impossible. Furthermore, there is little understanding of the provenance of datasets and the dependencies between them, leading to additional confusion. Due to these issues, collaborative data science or curation projects almost always end up with very poor data management and sharing.

Consider a concrete example from a computational biology group in the Brain Institution of MIT. The group has around 20 to 30 students, postdocs, and researchers, with around 100 terabytes of data shared via a networked file system. Every TB of data costs around 800 dollars per year from a local storage provider for unlimited read and write access. This amounts to around 100K dollars per year, which is not a small amount for a university research group. When one of the group members wants to perform analysis, they will first make a private copy of some version, modify and analyze it, with an ad-hoc name assigned to each newly produced version like "dataset_v1.csv". These new versions are again stored back in a folder, which is shared across all of the team members. There are three major issues with ad-hoc management of data of this form:

- [Massive Redundancy] There is lots of duplication and redundancy, leading to a waste of storage. As a result, students are periodically asked to clean up the disk because of space and cost constraints. Furthermore, due to the lack of metadata, students find it hard to identify which versions are important and which versions can be deleted safely.
- [No True Collaboration] There is no easy way to "merge in" modification from collabo-

rators or share your newly produced results with others.

- [No Query Capabilities] There are no querying capabilities, allowing members to analyze version dependencies, metadata, or perform some analysis to compare across versions.

The first issue stems from the system perspective, while the last two issues stem from the user's perspective. We can see there is a pressing need for a system to help manage these versioned datasets. Accordingly, we identify properties for a version management system: *(a)* compact storage to reduce storage consumption; *(b)* efficient versioning capabilities to enable true collaboration; *(c)* data manipulation and analytics to support reasoning across versions.

OrpheusDB **for Structured Data.** Motivated by these problems, we have built a system, called OrpheusDB, to help foster collaborative data analysis. OrpheusDB is a full-fledged management system for effective *structured* data versioning. In particular, OrpheusDB is a dataset version control system that "*bolts on*" versioning capabilities to a traditional relational database system, thereby gaining the analytics capabilities of the database "for free", while the database itself is unaware of the presence of dataset versions. Users can interact with OrpheusDB using git-style version control commands and SQL-like commands, performing data analytics within a version or across versions as well as reasoning about provenance or versioning information. Furthermore, we have carefully designed the storage representation in OrpheusDB for maintaining full data and versioning information, and developed a partition optimizer for faster version control operations.

Towards General-purpose Data Versioning. However, there are some implicit assumptions and constraints in OrpheusDB, due to the fact that OrpheusDB is built as a wrapper on top of relational databases: *(a) SQL assumption:* OrpheusDB assumes that a SQL-like language is the best fit for reasoning about data and versioning information; *(b) structural assumption:* OrpheusDB assumes that the data is in a relational format with a regular structure; *(c) from-scratch assumption:* OrpheusDB assumes that data science teams use OrpheusDB from the very beginning of their project and register each dataset version with OrpheusDB along with complete metadata (such as derivation relationships and author information). In the second part of this thesis, we relax each of these assumptions and build general-purpose modules, including a generalized query language, a generalized storage representation, and a generalized provenance manager. Specifically, to target assumption (a), we propose a generalized query language for versioning, data, and provenance, making the syntax easier to understand and work with compared to SQL, while also providing additional power. Then, to target assumption (b), we develop a generalized storage representation that can work with data with varying degrees of structure, including structured, semi-structured, and unstructured data. Finally, our ongoing work infers lineage relationships between versions automatically, when users do not register their dataset version with, and do not provide appropriate derivation metadata to, a hosted platform like OrpheusDB.

1.2 ORGANIZATION

This thesis consists of two major parts: *(a)* our relational dataset versioning system, OrpheusDB; *(b)* our attempts towards general-purpose versioning by removing various assumptions made by OrpheusDB. Specifically,

- In the first part, we introduce our system called OrpheusDB, which is built on top of relational databases, providing "bolt-on" versioning capabilities to the database. The full paper was published in VLDB'17 [1], and a demo paper was published in SIGMOD'17 [2].
  - We begin with OrpheusDB's architecture and its query language (Chapter 3).
  - We then describe its data model and experimentally verify its effectiveness (Chapter 4).
  - We finally introduce its partition optimizer to make version retrieval faster (Chapter 5).

  This work was done with Ph.D. student Liqi Xu. I was responsible for the data model design, the partition optimizer, as well as experimental evaluation.

- In the second part, we remove some of the assumptions made in OrpheusDB, one at a time.
  - We remove the SQL assumption in OrpheusDB, by developing a generalized query language (Chapter 6). This paper was published in Tapp'15 [3]. This work was done with Ph.D. student Amit Chavan. I was jointly responsible for the proposed language design.
  - We remove the structural assumption in OrpheusDB, by developing a generalized storage representation (Chapter 7). This paper was published in VLDB'15 [4]. This work was done with Ph.D. students Souvik Bhattacherjee and Amit Chavan. I was responsible for the hardness proofs, the ILP formulation and implementation, as well as the design of the modified Prim's algorithm.

- We remove the "from-scratch" assumption in OrpheusDB, by developing a generalized provenance manager (Chapter 8). This is still an ongoing project. This work was done with Ph.D. student Suhail Rehman. I was responsible for inferring the derivation relationships for row-preserving operations and designing the end-to-end workflow.

We cover overall related work in Chapter 2, and work specific to each project in Chapter 5-8.

Remark 1.1. The notion and terminology for Chapter 3, 4 and 5 will be uniform and are meant to be read in sequence, while Chapter 6, 7, and 8 can be read each independently without requiring readers to read Chapter 3-5 first.

CHAPTER 2: RELATED WORK

We now survey work from multiple areas related to the management of versioned datasets.

Source Version Control. Perhaps the most closely related prior work is source code versioning systems like Git, Mercurial, and SVN, that are widely used for managing source code repositories. Despite their popularity, these systems use fairly simple algorithms underneath that are optimized to work with modest-sized source code files and their on-disk structures are optimized to work with line-by-line diffs. These systems are known to have significant limitations when handling large files and/or large numbers of versions [5]. As a result, a variety of extensions like git-annex [6] and git-bigfiles [7], have been developed to make them work reasonably well with large files. However, none of these tools support querying across data versions.

Restricted Dataset Versioning. There have been some open-source projects on versioning topics. LiquiBase [8] tracks schema evolution as the only applicable modifications giving rise to new versions: our goal is to capture both the data-level modifications and schema-level modifications. On the other hand, DBV [9] is focused on recording SQL operations that give rise to new versions such that these operations can be "replayed" on new datasets—thus the emphasis is on reuse of workflows rather than on efficient versioning. Like other recent projects, Dat [10] can be used to share and sync local copies of dataset across machines, while Mode [11] integrates various analytics tools into a collaborative data analysis platform. However, neither of the tools are focused on providing advanced querying and versioning capabilities.

Temporal Databases. There is a rich body of work on time travel (or temporal) databases, e.g., [12, 13, 14, 15, 16, 17], focusing on data management when the state of the data at a specific time is important. Temporal databases support a linear clock, or a linear chain of versions, whereas our work focuses on enabling non-linear histories with git-like branching and merging common in collaborative data analysis. There has been some work on developing temporal databases by "bolting-on" capabilities to a traditional database [18], with DB2 [19, 20] and Teradata [21] supporting time-travel in this way. Other systems adopt an "in-database" approach [22]. For example, the SAP HANA database [23] maintains a *Timeline Index* [22] to efficiently support temporal join, aggregation, and time travel. Kaufmann et al. [24] provide a good summary of the temporal features in databases, while Kulkarni et al. [25] describe the temporal features in SQL2011.

There has been limited work on branched temporal databases [26, 27], with multiple chains of linear evolution as opposed to arbitrary branching and merging. While there has been some work on developing indexing [28, 29] techniques in that context, these techniques are specifically tailored for queries that select a specific branch, and a time-window within that branch, which therefore have no correspondences in our context.

Dataset Version Control. A vision paper on Datahub [30] acknowledges the need for database systems to support collaborative data analytics—we execute on that vision by supporting collaborative analytics using a traditional relational database, thereby seamlessly leveraging the sophisticated analysis capabilities (OrpheusDB as introduced in Chapter 3). Decibel [31] describes a version-oriented storage engine designed "from the ground up" to support versioning. Unfortunately, the architecture involves several choices that make it impossible to support within a traditional relational database without substantial changes at all layers of the stack. For example, the eventual solution requires the system to log and query tuple membership on compressed bitmaps, reason about and operate on "delta files", and execute new and fairly complex algorithms for even simple operations such as branch (in our case checkout) or merge (in our case commit). It remains to be seen how this storage engine can be made to interact with other components, such as the parser, the transaction manager, and the query optimizer, and all the other benefits that come "for free" with a relational database.

Delta Encoding. Many prior efforts have looked at the problem of minimizing the total storage cost for storing a collection of related files. Specifically, Quinlan et al. [32] propose an archival "deduplication" storage system that identifies duplicate blocks across files and only stores them once for reducing storage requirements. Zhu et al. [33] present several optimizations on the basic theme. Douglis et al. [34] present several techniques to identify pairs of files that could be efficiently stored using delta compression even if there is no explicit derivation information known about the two files. Ouyang et al. [35] studied the problem of compressing a large collection of related files by performing a sequence of pairwise delta compressions. They proposed a suite of text clustering techniques to prune the graph of all pairwise delta encodings and find the optimal branching (i.e., MCA) that minimizes the total weight. Burns and Long [36] present a technique for in-place re-construction of delta-compressed files using a graph-theoretic approach. Similar dictionary-based reference encoding techniques have been used by [37] to efficiently represent a target web page in terms of additions/modifications to a small number of reference web pages. Kulkarni et al. [38] present a more general technique that combines several different techniques to identify similar blocks among a collection of files, and use delta compression to reduce the total storage cost (ignoring the recreation costs). We refer the reader to a recent survey [39] for a more comprehensive coverage of this line of work. Our goal in this thesis is to not just reduce storage but also time to retrieve or recreate specific versions.

Incremental View Maintenance. The problem of incremental view maintenance, e.g., [40], is also related since it implicitly considers the question of storage versus query efficiency, which is one of the primary concerns in data versioning. However, the considerations and challenges are very different, making the solutions not applicable to data versioning. Buneman et al. [41] introduce a range encoding approach to track the versioning of hierarchical data in scientific databases, but their method focuses on XML data and is not applicable to the relational datasets.

Provenance. There has been much prior work [42] on capturing and maintaining the derivation relationship among data artifacts. In general, these works can be classified into two categories: invasive vs. post-processing. The invasive approach refers to the adoption of some management systems to help maintain the provenance information upon each artifact's generation. Specifically, ProvDB [43] keeps track of the lineage information by analyzing user's shell commands via ingesters, while our proposed OrpheusDB captures the derivation information by git-style commands. On the other hand, the post-processing approaches do not explicitly influence end-user's behavior. The goal is to capture the provenance information without enforcing the adoption of some particular system. In particular, Goods [44] tries to organize and reconnect datasets within Google in a post-processing manner, with a focus on extracting metadata and provenance information from logs.

CHAPTER 3: ORPHEUSDB OVERVIEW

OrpheusDB is a dataset version management system that is built on top of standard relational databases. It inherits much of the same benefits of relational databases, while also compactly storing, tracking, and recreating versions on demand. OrpheusDB has been developed as open-source software (`orpheus-db.github.io`). We now describe fundamental version-control concepts, followed by the design of OrpheusDB.

3.1 DATASET VERSION CONTROL

The fundamental unit of storage within OrpheusDB is a *collaborative versioned dataset* (CVD) to which one or more users can contribute. Each CVD corresponds to a relation and implicitly contains many *versions* of that relation. A *version* is an instance of the relation, specified by the user and containing a set of records. Versions within a CVD are related to each other via a *version graph*—a directed acyclic graph—representing how the versions were derived from each other: a version in this graph with two or more parents is defined to be a *merged version*. Records in a CVD are *immutable*, i.e., any modifications to any record attributes result in a new record, and are stored and treated separately within the CVD. Overall, there is a many-to-many relationship between records and versions: each record can belong to many versions, and each version can contain many records. Each version has a unique version id, *vid*, and each record has its unique record id, *rid*. The record ids are used to identify immutable records within the CVD and are not visible to end-users of OrpheusDB. In addition, the relation corresponding to the CVD may have primary key attribute(s); this implies that for any version no two records can have the same values for the primary key attribute(s). However, across versions, this need not be the case. OrpheusDB can support multiple CVDs at a time. However, in order to better convey the core ideas of OrpheusDB, in the rest of the chapter, we focus our discussion on a single CVD.

3.2 SYSTEM ARCHITECTURE

We implement OrpheusDB as a middleware layer or wrapper between end-users (or application programs) and a traditional relational database system—in our case, PostgreSQL. PostgreSQL is completely unaware of the existence of versioning, as versioning is handled entirely within the middleware. Figure 3.1 depicts the overall architecture of OrpheusDB. OrpheusDB consists of six core modules: the *query translator* is responsible for parsing

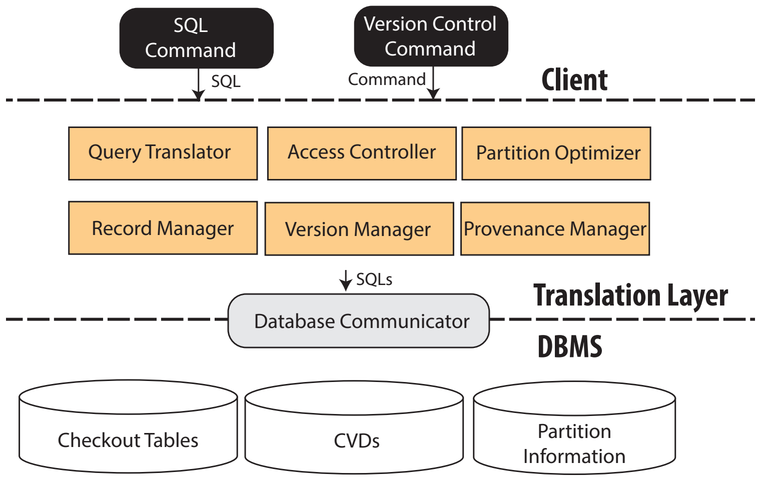


Figure 3.1: OrpheusDB Architecture

the input and translating it into SQL statements understandable by the underlying database system; the *access controller* monitors user permissions to various tables and files within OrpheusDB; the *partition optimizer* is responsible for periodically reorganizing and optimizing the partitions comprising different subsets of dataset versions along with a *migration engine* to migrate data from one partitioning scheme to another, and is the focus of Chapter 5; the *record manager* is in charge of recording and retrieving information about records in CVDs; the *version manager* is in charge of recording and retrieving versioning information, including the *rid*s each version contains as well as the metadata for each version; and the *provenance manager* is responsible for the metadata of uncommitted tables or files, such as their parent version(s) and the creation time. At the backend, a traditional DBMS, we maintain CVDs that consist of versions, along with the records they contain, as well as metadata about versions. In addition, the underlying DBMS contains a temporary staging area consisting of all of the materialized tables that users can directly manipulate via SQL without going through OrpheusDB. Understanding how to best represent and operate on these CVDs within the underlying DBMS is an important challenge—this is the focus of Chapter 4.

3.3 ORPHEUSDB QUERY LANGUAGE

Users interact with OrpheusDB via the command line, using both SQL queries, as well as git-style version control commands. We also describe an interactive user interface depicting the version graph, for users to easily explore and operate on dataset versions [45]. To make modifications to versions, users can either use SQL operations issued to the relational database that OrpheusDB is built on top of, or can alternatively operate on them using programming or scripting languages. We begin by describing the version control commands.

3.3.1 Version control commands

Users can operate on CVDs much like they would with source code version control. The first operation is *checkout*: this command materializes a specific version of a CVD as a newly created regular table within a relational database that OrpheusDB is connected to. The table name is specified within the checkout command, as follows:

```
checkout [cvd] -v [vid] -t [table name]
```

Here, the version with id *vid* is materialized as a new table [table name] within the database, to which standard SQL statements can be issued, and which can later be added to the CVD as a new version. The version from which this table was derived (i.e., *vid*) is referred to as the *parent version* for the table.

Instead of materializing one version at a time, users can materialize multiple versions, by listing multiple *vid*s in the command above, essentially *merging* multiple versions to give a single table. When merging, the records in the versions are added to the table in the precedence order listed after -v: for any record being added, if another record with the same primary key has already been added, it is omitted from the table. This ensures that the eventually materialized table also respects the primary key property. There are other conflict-resolution strategies, such as letting users resolve conflicted records manually; for simplicity, we use a precedence based approach. Internally, the checkout command records the versions that this table was derived from (i.e., those listed after -v), along with the table name. Note that only the user who performed the checkout operation is permitted access to the materialized table, so they can perform any analysis and modification on this table without interference from other users, only making these modifications visible when they use the *commit* operation, described next.

The *commit* operation adds a new version to the CVD, by making the local changes made by the user on their materialized table visible to others. The commit command has the following format:

```
commit -t [table name] -m [commit message]
```

The command does not need to specify the intended CVD since OrpheusDB internally maintains a mapping between the table name and the original CVD. In addition, since the versions that the table was derived from originally during checkout are internally known to OrpheusDB, the table is added to the CVD as a new version with those versions as parent versions. During the commit operation, OrpheusDB checks the primary key constraint if PK is specified, and compares the (possibly) modified materialized table to the parent versions. If any records were added or modified these records are treated as new records and added to the CVD. (Recall that records are immutable within a CVD.) An alternative is to compare the new records with all of the existing records in the CVD to check if any of the new records have existed in any version in the past, which would take longer to execute. At the same time, the latter approach would identify records that were deleted then re-added later. Since we believe that this is not a common case, we opt for the former approach, which would only lead to modest additional storage at the cost of much less computation during commit. We call this the *no cross-version diff* implementation rule. Lastly, if the schema of the table that is being committed is different from the CVD it derived from, we alter the CVD to incorporate the new schema; we discuss this in Section 4.3, but for most of the chapter we consider the static schema case.

In order to support data science workflows, we additionally support the use of *checkout* and *commit* into and from csv (comma separated value) files via slightly different flags: -f for csv instead of -t. The csv file can be processed in external tools and programming languages such as Python or R, not requiring that users perform the modifications and analysis using SQL. However, during commit, the user is expected to also provide a schema file via a -s flag so that OrpheusDB can make sure that the columns are mapped in the correct manner. An alternative would be to use schema inference tools, e.g., [46, 47], which could be seamlessly incorporated if need be. Internally, OrpheusDB also tracks the name of the csv file as being derived from one or more versions of the CVD, just like it does with the materialized tables.

In addition to checkout and commit, OrpheusDB also supports other commands, described very briefly here: *(a) diff:* a standard differencing operation that compares two versions and outputs the records in one but not the other. *(b) init:* initialize either an external csv file or a database table as a new CVD in OrpheusDB. *(c) create_user, config, whoami:* allows users to register, login, and view the current user name. *(d) ls, drop*: list all the CVDs or drop a particular CVD. *(e) optimize:* as we will see later, OrpheusDB can benefit from intelligent incremental partitioning schemes (enabling operations to process much less data). Users can set up the corresponding parameters (e.g., storage threshold, tolerance factor, described later) via the command line; the OrpheusDB backend will periodically invoke the partitioning optimizer to improve the versioning performance.

In brief, we now describe how the components in Figure 3.1 work with each other for the basic checkout and commit commands, once the command is parsed. For checkout, the query translator generates SQL queries to retrieve records from the relevant versions, which are then handled and materialized in the temporary staging area by the record manager; the provenance manager logs the related derivation information and other metadata; and finally the access controller to grant permissions to the relevant user. On commit, the record manager appends new records to the CVD, also performs cleanup by removing the table from the staging area; the version manager updates the metadata of the newly added version.

3.3.2 SQL commands

OrpheusDB supports the use of SQL commands on CVDs via the command line using the *run* command, which either takes a SQL script as input or the SQL statement as a string. Instead of materializing a version (or versions) as a table via the checkout command and explicitly applying SQL operations on that table, OrpheusDB also allows users to directly execute SQL queries on a specific version, using special keywords VERSION, OF, and CVD via syntax

```
SELECT ... FROM VERSION [vid] OF CVD [cvd], ...
```

without having to materialize it. For example, in Figure 3.2 scientists can quickly overview a small number of (e.g., 50) records within the first two versions of the *Interaction* CVD whose *coexpression* attribute is greater than 80 via the following SQL command:

```
SELECT * FROM VERSION 1, 2 OF CVD Interaction
WHERE coexpression > 80 LIMIT 50;
```

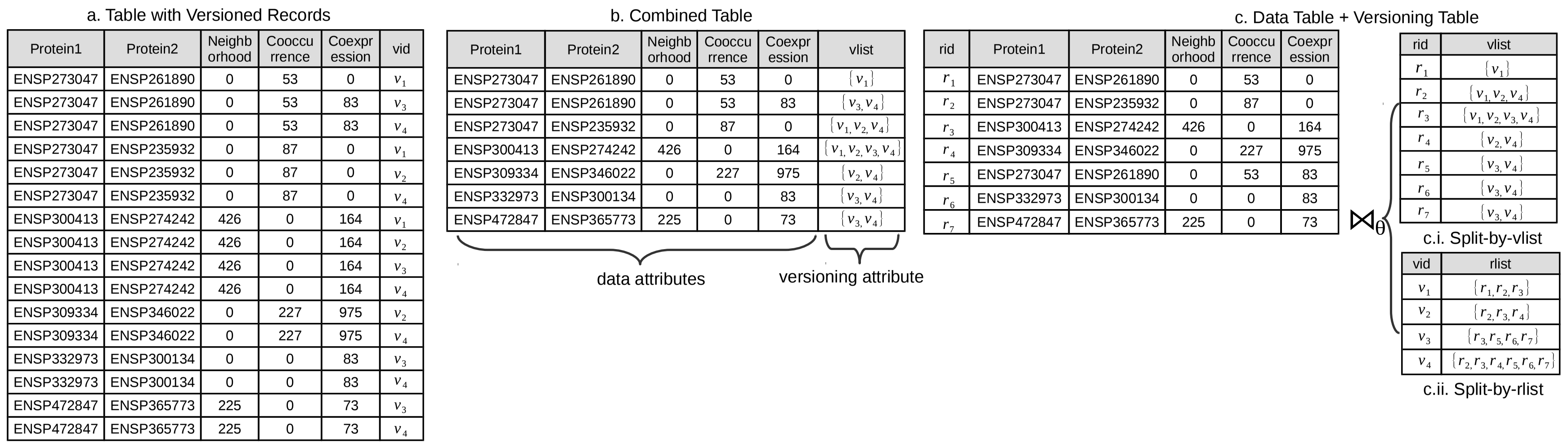
Further, by using renaming, users can operate directly on multiple versions (each as a relation) within a single SQL statement, enabling operations such as joins across multiple versions.

However, listing each version individually as described above may be cumbersome for some types of queries that users wish to run, e.g., applying an aggregate across a collection of versions, or identifying versions that satisfy some property. For this, OrpheusDB also supports constructs that enable users to issue aggregate queries across CVDs grouped by version ids, or select version ids that satisfy certain constraints. The corresponding syntax can be written as:

```
SELECT vid, ... FROM CVD [cvd], ... GROUP BY vid, ....
```

Internally, these constructs are translated into regular SQL queries that can be executed by the underlying database system. In addition, OrpheusDB provides shortcuts for several types of queries that operate on the version graph, e.g., listing the descendant or ancestors of a specific version, or querying the metadata, e.g., identify the last modification (in time) to the CVD. These operations are accessible via functional primitives that can be included as predicates within a query: *(a) ancestor(*vid*)/descendant(*vid*), parent(*vid*)*: The function takes a *vid* as the input and returns an array of all the ancestors/descendant, or its parent(s) of the *vid* in the version graph. *(b) v_diff(*vid/ARRAY*(*vid*),* vid/ARRAY*(*vid*)):* The function takes two arguments, each of which could be either a *vid* integer or an array of vids. It returns records in the data table that exist in the first argument but not in the second argument. *(c) v_intersect(* ARRAY*(*vid*)):* This is an aggregation function which takes an array of versions as the input and returns records in the data table that exist in all of these input versions.

Now we have introduced the architecture and query language used in OrpheusDB, next we will describe how to compactly store the data and versioning information in the underlying database.



a. Table with Versioned Records

Protein1	Protein2	Neighborhood	Cooccurrence	Coexpression	vid
ENSP273047	ENSP261890	0	53	0	v_1
ENSP273047	ENSP261890	0	53	83	v_3
ENSP273047	ENSP261890	0	53	83	v_4
ENSP273047	ENSP235932	0	87	0	v_1
ENSP273047	ENSP235932	0	87	0	v_2
ENSP273047	ENSP235932	0	87	0	v_4
ENSP300413	ENSP274242	426	0	164	v_1
ENSP300413	ENSP274242	426	0	164	v_2
ENSP300413	ENSP274242	426	0	164	v_3
ENSP300413	ENSP274242	426	0	164	v_4
ENSP309334	ENSP346022	0	227	975	v_2
ENSP309334	ENSP346022	0	227	975	v_4
ENSP332973	ENSP300134	0	0	83	v_3
ENSP332973	ENSP300134	0	0	83	v_4
ENSP472847	ENSP365773	225	0	73	v_3
ENSP472847	ENSP365773	225	0	73	v_4

b. Combined Table

Protein1	Protein2	Neighborhood	Cooccurrence	Coexpression	vlist
ENSP273047	ENSP261890	0	53	0	$\{v_1\}$
ENSP273047	ENSP261890	0	53	83	$\{v_3, v_4\}$
ENSP273047	ENSP235932	0	87	0	$\{v_1, v_2, v_4\}$
ENSP300413	ENSP274242	426	0	164	$\{v_1, v_2, v_3, v_4\}$
ENSP309334	ENSP346022	0	227	975	$\{v_2, v_4\}$
ENSP332973	ENSP300134	0	0	83	$\{v_3, v_4\}$
ENSP472847	ENSP365773	225	0	73	$\{v_3, v_4\}$

c. Data Table + Versioning Table

rid	Protein1	Protein2	Neighborhood	Cooccurrence	Coexpression
r_1	ENSP273047	ENSP261890	0	53	0
r_2	ENSP273047	ENSP235932	0	87	0
r_3	ENSP300413	ENSP274242	426	0	164
r_4	ENSP309334	ENSP346022	0	227	975
r_5	ENSP273047	ENSP261890	0	53	83
r_6	ENSP332973	ENSP300134	0	0	83
r_7	ENSP472847	ENSP365773	225	0	73

rid	vlist
r_1	$\{v_1\}$
r_2	$\{v_1, v_2, v_4\}$
r_3	$\{v_1, v_2, v_3, v_4\}$
r_4	$\{v_2, v_4\}$
r_5	$\{v_3, v_4\}$
r_6	$\{v_3, v_4\}$
r_7	$\{v_3, v_4\}$

c.i. Split-by-vlist

vid	rlist
v_1	$\{r_1, r_2, r_3\}$
v_2	$\{r_2, r_3, r_4\}$
v_3	$\{r_3, r_5, r_6, r_7\}$
v_4	$\{r_2, r_3, r_4, r_5, r_6, r_7\}$

c.ii. Split-by-rlist

Figure 3.2: Data models for protein interaction data [48]

CHAPTER 4: DATA MODELS FOR CVDS

In this chapter, we consider and compare methods to represent and operate on CVDs within a backend relational database, starting with the data within versions, and then the metadata about versions.

4.1 VERSIONS AND DATA: THE MODELS

A concrete example of data versioning occurs with biologists who operate on shared datasets, such as a gene annotation dataset [49] or a protein-protein interaction dataset [48], both of which are rapidly evolving, by periodically checking out versions, performing local analysis, editing, and cleaning operations, and committing these versions into a branched network of versions. This network of versions is also often repeatedly explored and queried for global statistics and differences (e.g., the aggregate count of protein-protein tuples with confidence in interaction greater than 0.9, for each version) and for versions with specific properties (e.g., versions with a specific gene annotation record, or versions with "a bulk delete", ones with more than 100 tuples deleted from their parents).

To explore alternative storage models, we consider the array-based data models, shown in Figure 3.2, and compare them to a delta-based data model, which we describe later. The table(s) in Figure 3.2 displays simplified protein-protein interaction data [48], and has a composite primary key <*protein1, protein2*>, along with numerical attributes indicating sources and strength of interactions: *neighborhood* represents how frequently the two proteins occur close to each other in runs of genes, *cooccurrence* reflects how often the two proteins co-occur in the species, and *coexpression* refers to the level to which genes are co-expressed in the species.

One approach to capture versioning information is to augment the CVD's relational schema with an additional versioning attribute. For example, in Figure 3.2(a) <ENSP273047, ENSP261890, 0, 53, 83> exists in two versions: v_3 and v_4. (Note that even though <protein1, protein2> is the primary key, it is only the primary key for any single version and not across all versions.) There are two records with <ENSP273047, ENSP261890> that have different values for the other attributes: one with (0, 53, 83) that is present in v_3 and v_4, and another with (0, 53, 0) that is present in v_1. However, this approach implies that we would need to duplicate each record as many times as the number of versions it is in, leading to severe storage overhead due to redundancy, as well as inefficiency for several operations, including checkout and commit. We focus on alternative approaches that are more space

Command	SQL Translation		
	with combined-table	with Split-by-vlist	with Split-by-rlist
CHECKOUT	`SELECT * into T' FROM T` `WHERE ARRAY[`v_i`] <@ vlist`	`SELECT * into T'` `FROM dataTable,` `(SELECT rid AS rid_tmp` `FROM versioningTable` `WHERE ARRAY[`v_i`] <@ vlist)` `AS tmp` `WHERE rid = rid_tmp`	`SELECT * into T'` `FROM dataTable,` `(SELECT unnest(rlist) AS rid_tmp` `FROM versioningTable` `WHERE vid = `v_i`)` `AS tmp` `WHERE rid = rid_tmp`
COMMIT	`UPDATE T SET vlist=vlist+`v_j `WHERE rid in` `(SELECT rid FROM T')`	`UPDATE versioningTable` `SET vlist=vlist+`v_j `WHERE rid in` `(SELECT rid FROM T')`	`INSERT INTO versioningTable` `VALUES (`v_j`,` `ARRAY[SELECT rid FROM T'])`

Table 4.1: SQL Queries for Checkout and Commit Commands with Different Data Models

efficient and discuss how they can support the two most fundamental operations—commit and checkout—on a single version at a time. Considerations for multiple version checkout is similar to that for a single version; our findings generalize to that case as well.

Approach 4.1: The Combined Table Approach. Our first approach of representing the data and versioning information for a CVD is the *combined table approach.* As before, we augment the schema with an additional versioning attribute, but now, the versioning attribute is of type array and is named *vlist* (short for version list) as shown in Figure 3.2(b). For each record the *vlist* is the ordered list of version ids that the record is present in, which serves as an inverted index for each record. Returning to our example, there are two versions of records corresponding to <ENSP273047, ENSP261890>, with coexpression 0 and 83 respectively—these two versions are depicted as the first two records, with an array corresponding to v_1 for the first record, and v_3 and v_4 for the second.

Even though array is a non-atomic data type, it is commonly supported in many database systems [50, 51, 52]; thus OrpheusDB can be built with any of these systems as the back-end database. As our implementation uses PostgreSQL, we focus on this system for the rest of the discussion, even though similar considerations apply to the rest of the databases listed. PostgreSQL provides a number of useful functions and operators for manipulating arrays, including append operations, set operations, value containment operations, and sorting and counting functions.

For the combined table approach, committing a new version to the CVD is time-consuming due to the expensive append operation for every record present in the new version. Consider the scenario where the user checks out version v_i into a materialized table T' and then immediately commits it back as a new version v_j. The query translator parses the user commands and generates the corresponding SQL queries for checkout and commit as shown in Table 4.1. In the checkout statement, the containment operator 'int[] <@ int[]' returns true if the array on the left is contained within the array on the right. When checking out v_i into a materialized table T', the array containment operator 'ARRAY[v_i] <@ vlist' first examines whether v_i is contained in *vlist* for each record in CVD, then all records that satisfy that condition are added to the materialized table T'. Next, when T' is committed back to the CVD as a new version v_j, for each record in the CVD, if it is also present in T' (i.e., the WHERE clause), we append v_j to the attribute *vlist* (i.e., vlist=vlist+v_j). In this case, since there are no new records that are added to the CVD, no new records are added to the combined table. However, even this process of appending v_j to *vlist* can be expensive especially when the number of records in v_j is large, as we will demonstrate.

Approach 4.2: The Split-by-vlist Approach. Our second approach addresses the limitations of the expensive commit operation for the combined table approach. We store two tables, keeping the versioning information separate from the data information, as depicted in Figure 3.2(c)—the *data table* and the *versioning table*. The data table contains all of the original data attributes along with an extra primary key *rid*, while the versioning table maintains the mapping between versions and *rid*s. The *rid* attribute was not needed in the previous approach since it was not necessary to associate identifiers with the immutable records. Specifically, the relation primary key— <protein1, protein2> —is not sufficient to distinguish between multiple copies of the same record. For example, $r1$ and $r5$ are two versions of the same record (i.e., the record with a given <protein1, protein2>). There are two ways we can store the versioning data. The first approach is to store the *rid* along with the *vlist*, as depicted in Figure 3.2(c.i). We call this approach *split-by-vlist*. Split-by-vlist has a similar SQL translation as combined-table for commit, while it incurs the overhead of joining the data table with the versioning table for checkout. Specifically, we select the *rid*s that are in the version to be checked out and store it in the table tmp, followed by a join with the data table. For example, when checking out version v_1, tmp will comprise the relevant *rid*s r_1, r_2, r_3, which are identified by looking at the *vlist* for each record in the versioning table and checking if v_1 is present, which is then joined with the data table to extract the appropriate results into the materialized table T'.

Approach 4.3: The Split-by-rlist Approach. Alternatively, we can organize the versioning table with a primary key as *vid* (version id), and another attribute *rlist*, containing the array of the records present in that particular version, as in Figure 3.2(c.ii). We call this approach the *split-by-rlist* approach. When committing a new version v_j from the materialized table T', we only need to add a single tuple in the versioning table with *vid* equal to v_j, and *rlist* equal to the list of record ids in T'. This eliminates the expensive array appending operations that are part of the previous two approaches, making the commit command much more efficient. For the checkout command for version v_i, we first extract the record ids associated with v_i from the versioning table, by applying the unnesting operation: unnest(*rlist*), following which we join the *rid*s with the data table to identify all of the relevant records. For example, for checking out v_1, instead of examining the entire versioning table, we simply need to examine the tuple corresponding to v_1, unnest those *rid*s—r_1, r_2, r_3, followed by a join.

So far, all our models support convenient rewriting of arbitrary and complex versioning queries into SQL queries understood by the backend database; see details in our demo paper [45]. However, our delta-based model, discussed next, does not support convenient rewritings for some of the more advanced queries, e.g., "find versions where the total count of tuples with *protein1* as ENSP273047 is greater than 50": in such cases, delta-based model essentially needs to recreate all of the versions, and/or perform extensive and expensive computation outside of the database. Thus, even though this model does not support advanced analytics capabilities "for free", we include it in our comparison to contrast its performance to the array-based models.

Approach 4.4: Delta-based Approach. Here, each version records the modifications (or deltas) from its precedent version(s). Specifically, each version is stored as a separate table, with an added tombstone boolean attribute indicating the deletion of a record. In addition, we maintain a precedent metadata table with a primary key *vid* and an attribute *base* indicating from which version *vid* stores the delta. When committing a new version v_j, a new table stores the delta from its previous version v_i. If v_j has multiple parents, we will store v_j as the modification from the parent that shares the largest common number of records with v_j. (Storing deltas from multiple parents would make reconstruction of a version complicated, since we would need to trace back multiple paths in the version graph, or alternatively materialize each version in the version graph in a top-down manner, merging versions based on conflict resolution mechanisms. Here, we opt for the simpler solution.) A new record is then inserted into the metadata table, with *vid* as v_j and *base* as v_i. For the *checkout* command for version v_i, we trace the version lineage (via the *base* attribute) all the way back to the root. If an incoming record has occurred before, it is discarded; otherwise, if it is marked as "insert", we insert it into the checkout table T'.

Approach 4.5: The A-Table-Per-Version Approach. Our final array-based data model is impractical due to excessive storage, but is useful from a comparison standpoint. In this approach, we store each version as a separate table. We include a-table-per-version in our comparison; we do not include the approach in Figure 3.2a, containing a table with duplicated records, since it would do similarly in terms of storage and commit times to a-table-per-version, but worse in terms of checkout times.

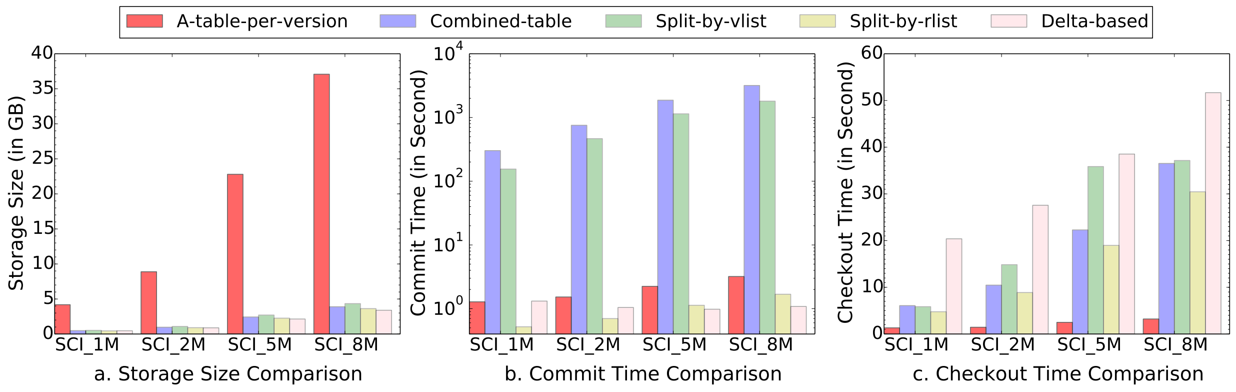


Figure 4.1: Comparison Between Different Data Models

4.2 VERSIONS AND DATA: THE COMPARISON

We perform an experimental evaluation between the approaches described in the previous section on storage size, and commit and checkout time. We focus on the commit and checkout times since they are the primitive versioning operations on which the other more complex operations and queries are built on. It is important that these operations are efficient, because data scientists checkout a version to start working on it immediately, and often commit a version to have their changes visible to other data scientists who may be waiting for them.

In our evaluation, we use four versioning benchmark datasets SCI_1M, SCI_2M, SCI_5M and SCI_8M, each with $1M$, $2M$, $5M$ and $8M$ records respectively, that will be described in detail in Section 5.5.1. For split-by-vlist, a physical primary key index is built on *rid* in both the data table and the versioning table; for split-by-rlist, a physical primary key index is built on *rid* in the data table and on *vid* in the versioning table. When calculating the total storage size, we count the index size as well. Our experiment involves first checking out the latest version v_i into a materialized table T' and then committing T' back into the CVD as a new version v_j. We depict the experimental results in Figure 4.1.

Storage. From Figure 4.1(a), we can see that a-table-per-version takes $10\times$ more storage than the other data models. This is because each record exists on average in 10 versions. Compared to a-table-per-version and combined-table, split-by-vlist and split-by-rlist deduplicate the common records across versions and therefore have roughly similar storage. In particular, split-by-vlist and split-by-rlist share the same data table, and thus the difference can be attributed to the difference in the size of the versioning table. For the delta-based approach, the storage size is similar to or even slightly smaller than split-by-vlist and split-by-rlist. This is because our versioning benchmark contains only a few deleted tuples (opting instead for updates or inserts); in other cases, where deleted tuples are more prevalent, the storage in the delta-based approach is worse than split-by-vlist/rlist, since the deleted records will be repeated. We also remark that the storage size for array-based approaches can be further reduced by applying compression techniques like range-encoding [41].

Commit. From Figure 4.1(b), we can see that the combined-table and split-by-vlist take multiple orders of magnitude more time than split-by-rlist for commit. We also notice that the commit time when using combined-table is almost $10^4 s$ as the dataset size increases: when using combined-table, we need to add v_j to the attribute *vlist* for each record in the CVD that is also present in T'. Similarly, for split-by-vlist, we need to perform an append operation for several tuples in the versioning table. On the contrary, when using split-by-rlist, we only need to add one tuple to the versioning table, thus getting rid of the expensive array appending operations. A-table-per-version also has higher latency for commit than split-by-rlist since it needs to insert all the records in T' into the CVD. For the delta-based approach, the commit time is small since the new version v_j is exactly the same as its precedent version v_i. It only needs to update the precedent metadata table, and create a new empty table. The commit time of the delta-based approach is not small in general when there are extensive modifications to T', as illustrated by other experiments (not displayed); For instance, for a committed version with 250K records of which 30% of the records are modified, delta-based takes 8.16s, while split-by-rlist takes 4.12s.

Checkout. From Figure 4.1 (c), we can see that split-by-rlist is a bit faster than combined-table and split-by-vlist for checkout. Not surprisingly, a-table-per-version is the best for this operation since it simply requires retrieving all the records in a specific table (corresponding to the desired version). We dive into the query plan for the other data models. Combined-table requires one full scan over the combined table to check whether each record is in version v_i. On the other hand, split-by-vlist needs to first scan the versioning table to retrieve the *rid*s in version v_i, and then join the *rid*s with the data table. Lastly, split-by-rlist retrieves the *rid*s in version v_i using the primary key index on *vid* in the versioning table, and then joins the *rid*s with the data table. For both split-by-vlist and split-by-rlist, we used a hash-join, which was the most efficient[1], where a hash table on *rid*s is first built, followed by a sequential scan on the data table by probing each record in the hash table. Overall, combined-table, split-by-vlist, and split-by-rlist all require a full scan on the combined table or the data table, and even though split-by-rlist introduces the overhead of building a hash table, it reduces the expensive array operation for containment checking as in combined-

[1] We also tried alternative join methods—the findings were unchanged; we will discuss this further in Section 5.1. We also tried using an additional secondary index for *vlist* for split-by-vlist which reduced the time for checkout but increased the time for commit even further.

table and split-by-vlist. For the delta-based approach, the checkout time is large since it needs to probe into a number of tables, tracing all the way back to the root, remembering which records were seen.

Takeaways. Overall, considering the space consumption, the commit and checkout time, plus the fact that delta-based models are inefficient in supporting advanced queries as discussed in Section 4.1, we claim that split-by-rlist is preferable to the other data models in supporting versioning within a relational database. Thus, we pick split-by-rlist as our data model for representing CVDs. That said, from Figure 4.1(c), we notice that the checkout time for split-by-rlist grows with dataset size. For instance, for dataset SCI_8M with $8M$ records in the data table, the checkout time is as high as 30 seconds. On the other hand, a-table-per-version has very low checkout times on all datasets; it only needs to access the relevant records instead of all records as in split-by-rlist. This motivates the need for the partition optimizer module in OrpheusDB, which tries to attain the best of both worlds by adopting a hybrid representation of split-by-rlist and a-table-per-version, described in Chapter 5.

Remark 4.1. The canonical approach to recording time in temporal databases (see Chapter 2) is via attributes indicating the start and end time, which differs a bit depending on whether the time is the "transaction time" or the "valid time". In either case, if one extends temporal databases to support arrays capturing versions instead of the start and end time, we will end up as a solution like the one in Figure 3.2b, which as shown severely limits performance. Thus, the techniques we describe in the chapter on evaluating efficient data models and partitioning in the next chapter are still relevant and complement this prior work.

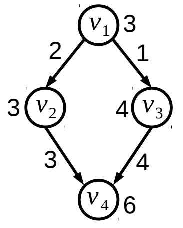
Most work in this area focuses on supporting constructs that do not directly apply to OrpheusDB, due to the lack of time-oriented notions such as: (a) queries that probe interval related-properties, such as which tuples were valid in a specific time interval, via range indexes [53], or queries that roll back to specific points [54]; (b) temporal aggregation [22] to aggregate some attributes for every time interval granularity, and temporal join [55] to join tuples if they overlap in time; (c) queries that involve time-related constructs such as AS OF, OVERLAPS, PRECEDES.

4.3 VERSION DERIVATION METADATA

Version Provenance. As discussed in Section 3.2, the version manager in OrpheusDB keeps track of the derivation relationships among versions and maintains metadata for each version. We store version-level provenance information in a separate table called the *metadata table*; Figure 4.2(a) depicts the metadata table for the example in Figure 3.2. It contains attributes including version id, parent/child versions, creation time, commit time, a commit message, and an array of attributes present in the version. Using the data contained in this table, users can easily query for the provenance of versions and for other metadata. In addition, using the attribute *parents* we can obtain each version's derivation information and visualize it as a directed acyclic graph that we call a *version graph*. Each node in the version graph is a version and each directed edge points from a version to one of its children version(s). An example is depicted in Figure 4.2(b), where version v_2 and v_3 are both derived from version v_1, and version v_2 and v_3 are merged into version v_4. We will return to this concept in Section 5.2.

vid	parents	checkoutT	commitT	msg	attributes
v_1	NULL	NULL	t_1	···	$\{a_1, a_2, a_3, a_4, a_6\}$
v_2	$\{v_1\}$	t_2	t_3	···	$\{a_1, a_2, a_3, a_4, a_6\}$
v_3	$\{v_1\}$	t_2	t_4	···	$\{a_1, a_2, a_3, a_4, a_6\}$
v_4	$\{v_2, v_3\}$	t_5	t_6	···	$\{a_1, a_2, a_3, a_4, a_6\}$

a. Metadata Table



b. Version Graph

Figure 4.2: Metadata Table and Version Graph (Fixed Schema)

vid	parents	checkoutT	commitT	msg	attributes
v_1	NULL	NULL	t_1	···	$\{a_1, a_2, a_3, a_4\}$
v_2	$\{v_1\}$	t_2	t_3	···	$\{a_1, a_2, a_3, a_5\}$
v_3	$\{v_1\}$	t_2	t_4	···	$\{a_1, a_2, a_3, a_4, a_6\}$
v_4	$\{v_2, v_3\}$	t_5	t_6	···	$\{a_1, a_2, a_3, a_5, a_6\}$

a. Metadata Table With Schema Changes

attr_id	attr_name	data_type
a_1	protein1	string
a_2	protein2	string
a_3	neighborhood	integer

b. Attribute Table

attr_id	attr_name	data_type
a_4	cooccurrence	integer
a_5	cooccurrence	decimal
a_6	coexpression	integer

c. Attribute Table (cont.)

Figure 4.3: Metadata Table and Attribute Table (Schema Changes)

Schema Changes. During a commit, if the schema of the table being committed is different from the schema of the CVD it was derived from, we update the schema of CVD to incorporate the changes. More precisely, in OrpheusDB, we maintain an attribute table (as in Figure 4.3) where each tuple represents an attribute with a unique identifier, along with the corresponding attribute name and data type; any change of a property of an attribute results in a new attribute entry in the table. If the data type of any attribute changes, we transform the attribute type to a more general data type (e.g., from integer to string as in Jain et al. [56]), and insert a new tuple into the attribute table with the updated datatype. All of our array-based models can adapt to changes in the set of attributes: a simple solution for new attributes is so use the ALTER command to add any new attributes to the model, assigning NULLs to the records from the previous versions that do not possess these new attributes. Attribute deletions only require an update in the version metadata table. To illustrate, we modify the previous example in Figure 4.2 (which showed a static schema) to a dynamic one. For example, as shown in Figure 4.3, initially version v_1 has four attributes: protein1, protein2, neighborhood and cooccurrence. When a user commits version v_2, with the data type of the cooccurrence attribute (a_4) changed from integer to decimal, within OrpheusDB, we create another attribute (a_5) in the attribute table with data type decimal, log a_5 in the metadata table for v_2 and alter the cooccurrence attribute to decimal within the CVD. Moreover, when a new coexpression attribute is added in v_3, we generate a corresponding attribute (a_6) in the attribute table, add a_6 in the metadata table for v_3, and add the coexpression attribute to the CVD. During the merge, the resulting version includes all attributes from its parents and contains the more general data type for conflicting attributes (e.g., attributes in v_4). This simple mechanism is similar to the single pool method proposed in a temporal schema versioning context by De Castro et al. [57]. Compared to the multi pool method where any schema change results in the new version being stored separately, the single pool method has fewer records with duplicated attributes and therefore has less storage consumption overall. Even though ALTER TABLE is indeed a costly operation, due to the partitioning schemes we describe later, we only need to ALTER a smaller partition of the CVD rather than a giant CVD, and consequently the cost of an ALTER operation is substantially mitigated. In Section 5.3.3, we describe how our partitioning schemes (described next in Chapter 5) can adapt to the single pool mechanism with comparable guarantees; for ease of exposition, for the rest of this chapter, we focus on the static schema case, which is still important and challenging. There has been some work on developing schema versioning schemes [58, 59, 60] and we plan to explore these and other schema evolution mechanisms (including hybrid single/multi-pool methods) as future work.

Summary. In this chapter, we have illustrated that Split-by-rlist can achieve the best performance in terms of storage size and commit time. Recall that in addition to the storage size and commit time, we also consider checkout time in guiding the development of OrpheusDB. However, even with the optimized data model Split-by-rlist, we still observe high latency during checkout. In the next chapter, we will propose a partitioning scheme to further reduce the checkout latency.

CHAPTER 5: PARTITION OPTIMIZER

Recall that Figure 4.1(c) in Chapter 4 indicated that as the number of records within a CVD increases, the checkout latency of our data model (split-by-rlist) increases—this is because the number of "irrelevant" records, i.e., the records that are not present in the version being checked out, but nevertheless require processing increases. Even with an index on *rid*, the checkout latency is still high since records are scattered across the whole data table, and hundreds of thousands of random accesses are eventually reduced to a full table scan as we will demonstrate. In this chapter, we introduce the concept of partitioning a CVD by breaking up the data and versioning tables, in order to reduce the number of irrelevant records during checkout. We formally define our partitioning problem, demonstrate that this problem is NP-HARD, and identify a light-weight approximation algorithm. We provide a convenient table of notation in Table 5.1.

5.1 PROBLEM OVERVIEW

The Partitioning Notion. Let $V = \{v_1, v_2, ..., v_n\}$ be the n versions and $R = \{r_1, r_2, ..., r_m\}$ be the m records in a CVD. We can represent the presence of records in versions using a version-record bipartite graph $G = (V, R, E)$, where E is the set of edges—an edge between v_i and r_j exists if the version v_i contains the record r_j. The bipartite graph in Figure 5.1(a) captures the relationships between records and versions in Figure 3.2.

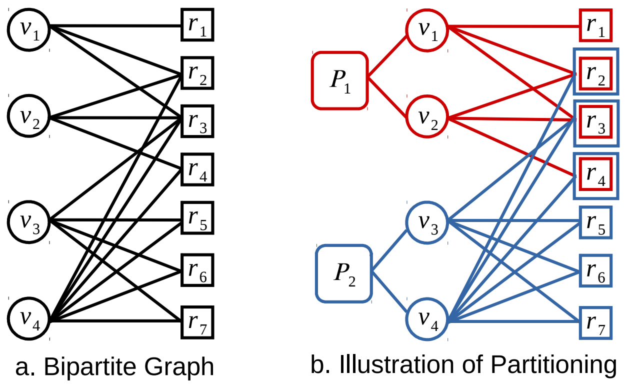


Figure 5.1: Version-Record Bipartite Graph & Partitioning

The goal of our partitioning problem is to partition G into smaller subgraphs, denoted as $\mathcal{P}_k$. We let $\mathcal{P}_k = (\mathcal{V}_k, \mathcal{R}_k, \mathcal{E}_k)$, where $\mathcal{V}_k$, $\mathcal{R}_k$ and $\mathcal{E}_k$ represent the set of versions, records and bipartite graph edges in partition $\mathcal{P}_k$ respectively. Note that $\cup_k \mathcal{E}_k = E$, where E is the set of edges in the original version-record bipartite graph G. We further constrain each

Symb.	Description	Symb.	Description
G	bipartite graph	E	bipartite edge set in G
V	version set in G	n	total number of versions
R	record set in G	m	total number of records
v_i	version i in V	r_j	record j in R
$\mathcal{P}_k$	k^{th} partition	$\mathcal{V}_k$	version set in $\mathcal{P}_k$
$\mathcal{R}_k$	record set in $\mathcal{P}_k$	$\mathcal{E}_k$	bipartite edge set set in $\mathcal{P}_k$
$\mathcal{S}$	total storage cost	γ	storage threshold
$\mathcal{C}_i$	checout cost for v_i	$\mathcal{C}_{avg}$	average checkout cost
$\mathbb{G}$	version graph	$\mathbb{V}$	version set in $\mathbb{G}$
$\mathbb{E}$	edge set in $\mathbb{G}$	e	$e = (v_i, v_j)$: v_i derives v_j
$\mathbb{T}$	version tree	$e.w$	# of common records on e
$l(v_i)$	level # of v_i in $\mathbb{G}$	$p(v_i)$	parent version(s) of v_i in $\mathbb{G}$
$R(v_i)$	record set in v_i	ℓ	# of recursive levels in Alg 1

Table 5.1: Notations

version in the CVD to exist in only one partition, while each record can be duplicated across multiple partitions. In this manner, we only need to access one partition when checking out a version, consequently simplifying the checkout process by reducing the overhead from accessing multiple partitions. (While we do not consider it in this thesis, in a distributed setting, it is even more important to ensure that as few partitions are consulted during a checkout operation.) Thus, our partition problem is equivalent to partitioning V, such that each partition ($\mathcal{P}_k$) stores all of the records corresponding to all of the versions assigned to that partition. Figure 5.1(b) illustrates a possible partitioning strategy for Figure 5.1(a). Partition $\mathcal{P}_1$ contains version v_1 and v_2, while partition $\mathcal{P}_2$ contains version v_3 and v_4. Note that records r_2, r_3 and r_4 are duplicated in $\mathcal{P}_1$ and $\mathcal{P}_2$.

Metrics. We consider two criteria while partitioning: the storage cost and the time for checkout. Recall that the time for commit is fixed and small—see Figure 4.1(b), so we only focus on checkout.

The overall storage costs involve the cost of storing all of the partitions of the data and the versioning table. However, we observe that the versioning table simply encodes the bipartite graph, and as a result, its cost is fixed. Furthermore, since all of the records in the data table have the same (fixed) number of attributes, so instead of optimizing the actual storage we will optimize for the number of records in the data table across all the partitions. Thus, we define the *storage cost*, $\mathcal{S}$, to be the following:

$$\mathcal{S} = \sum_{k=1}^{K} |\mathcal{R}_k| \tag{5.1}$$

Next, we note that the time taken for checking out version v_i is proportional to the size of the data table in the partition $\mathcal{P}_k$ that contains version v_i, which in turn is proportional to the number of records present in that data table partition. We theoretically and empirically justify this in Section 5.5.5. So we define the *checkout cost of a version* v_i, $\mathcal{C}_i$, to be $\mathcal{C}_i = |\mathcal{R}_k|$, where $v_i \in \mathcal{V}_k$. The *checkout cost*, denoted as $\mathcal{C}_{avg}$, is defined to be the average of $\mathcal{C}_i$, i.e., $\mathcal{C}_{avg} = \frac{\sum_i \mathcal{C}_i}{n}$. While we focus on the average case, which assumes that each version is checked out with equal frequency—a reasonable assumption when we have no other information about the workload, our algorithms generalize to the *weighted* case as described in Section 5.3.2. (The weighted case can help represent the workload in real world settings, where recent versions may be checked out more frequently.) On rewriting the expression for $\mathcal{C}_{avg}$ above, we get:

$$\mathcal{C}_{avg} = \frac{\sum_{k=1}^{K} |\mathcal{V}_k||\mathcal{R}_k|}{n} \tag{5.2}$$

The numerator is simply sum of the number of records in each partition, multiplied by the number of versions in that partition, across all partitions—this is the cost of checking out all of the versions, equivalent to $\sum_{i=1}^{n} \mathcal{C}_i$—this is the cost of checking out all of the versions.

Formal Problem. Our two metrics $\mathcal{S}$ and $\mathcal{C}_{avg}$ interfere with each other: if we want a small $\mathcal{C}_{avg}$, then we need more storage, and if we want the storage to be small, then $\mathcal{C}_{avg}$ will be large. Typically, storage is under our control; thus, our problem can be stated as:

Problem 5.1 (Minimize Checkout Cost)**.** *Given a storage threshold γ and a version-record bipartite graph $G = (V, R, E)$, find a partitioning of G that minimizes $\mathcal{C}_{avg}$ such that $\mathcal{S} \leq \gamma$.*

We can show that Problem 5.1 is NP-Hard using a reduction from the 3-Partition problem, whose goal is to decide whether a given set of n integers can be partitioned into $\frac{n}{3}$ sets with equal sum. 3-Partition is known to be strongly NP-Hard, i.e., it is NP-Hard even when its numerical parameters are bounded by a polynomial in the length of the input.

Theorem 5.1. *Problem 5.1 is* NP-hard.

Proof. We reduce the well known NP-hard 3-partition problem to our Problem 5.1. The 3-partition problem is defined as follows: Given an integer set $\mathcal{A} = \{a_1, \cdots, a_n\}$ where n is divisible by 3, partition $\mathcal{A}$ into $\frac{n}{3}$ sets $\{A_1, A_2, A_j \cdots A_{\frac{n}{3}}\}$ such that for any A_j, $\sum_{a_i \in A_j} a_i = \frac{B}{n/3}$ where $B = \sum_{a_i \in \mathcal{A}} a_i$.

To reduce 3-partition to our Problem 5.1, we first construct a version-record bipartite graph $G = (V, R, E)$ (Figure 5.2) that consists of B versions and $(B + D)$ records, where D is the number of dummy records and can be any positive integer. Specifically:

- For each integer $a_i \in \mathcal{A}$:
  - Create a_i versions $\{v_i^1, v_i^2, \cdots, v_i^{a_i}\}$ in V;
  - Create a_i records $\{r_i^1, r_i^2, \cdots, r_i^{a_i}\}$ in R;
  - Connect each v_i^j with r_i^τ in E, where $1 \leq j \leq a_i$ and $1 \leq \tau \leq a_i$. This forms a biclique between $\{v_i^1, \cdots, v_i^{a_i}\}$ and $\{r_i^1, \cdots, r_i^{a_i}\}$.
- We also create dummy records R_D and edges E_D:
  - R_D: create D dummy records $R_D = \{r_0^1, r_0^2, \cdots, r_0^D\}$ in R, where $D \geq 1$;
  - E_D: connect each dummy record with every version $v \in V$.

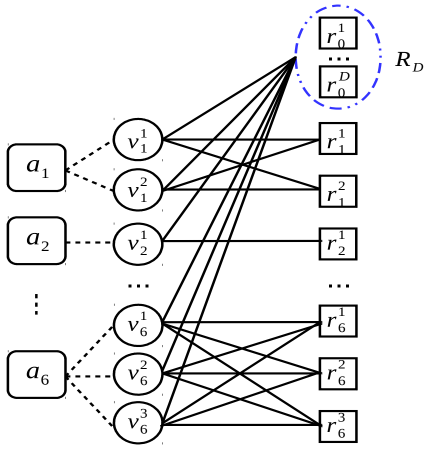


Figure 5.2: An Example of a Constructed Graph G

As inputs to Problem 5.1, we take the constructed graph G and set storage threshold $\gamma = \frac{n}{3} \cdot D + B$. We have the following two claims for the optimal solution to Problem 5.1:

Claim 5.1. *For each a_i, its corresponding versions $\{v_i^1, v_i^2, \cdots, v_i^{a_i}\}$ must be in the same partition.*

Claim 5.2. *The optimal solution must have $\frac{n}{3}$ partitions, i.e, $K = \frac{n}{3}$.*

We prove our first claim by contradiction. For a fixed a_i, if $\{v_i^1, v_i^2, \cdots, v_i^{a_i}\}$ are in different partitions, denoted as $P' = \{\mathcal{P}_{\tau_1}, \mathcal{P}_{\tau_2}, \cdots\}$, we can reduce the average checkout cost while maintaining the same storage cost by moving all these versions into the same partition $\mathcal{P}_{k^*} \in P'$ with the smallest $|\mathcal{R}_{k^*}|$. Furthermore, the only common records between v_i^x and v_j^y, where $i \neq j$, are the dummy records in R_D, thus only these dummy records will be duplicated across different partitions. Consequently, the total storage cost from records except the dummy record, i.e., $R \setminus R_D$, in all partitions is a constant B, regardless of the partitioning scheme.

Based on the first claim, we have $|\mathcal{R}_k| = |\mathcal{V}_k| + D, \forall k$ and our optimization objective function can be represented as follows:

$$\mathcal{C}_{avg} = \frac{1}{B}\sum_{k=1}^{K} |\mathcal{V}_k| \times (|\mathcal{V}_k| + D) = \frac{1}{B}(\sum_{k=1}^{K} |\mathcal{V}_k|^2 + B \cdot D) \tag{5.3}$$

Next, we prove the correctness of our second claim. First, we show that keeping the total storage cost $\sum_{k=1}^{K} |\mathcal{R}_k| \leq \frac{n}{3} \times D + B$ is equivalent to keeping the number of partitions $K \leq \frac{n}{3}$. From our first claim, we know that no record in $R \setminus R_D$ will be duplicated and the total number of records that corresponds to $R \setminus R_D$ in all of the partitions is B. On the other hand, each partition $\mathcal{P}_k$ must include all dummy records R_D, which is of size D. Thus, the number of partitions K must be no larger than $\frac{n}{3}$. Furthermore, we claim that the optimal solution must have $\frac{n}{3}$ partitions, i.e., $K = \frac{n}{3}$; otherwise, we can easily reduce the checkout cost by splitting any partition into multiple partitions.

Lastly, we prove that the optimal $\mathcal{C}_{avg}$ equals $B/K + D$ if and only if the decision problem to 3-PARTITION is correct. First, since $\sum_{k=1}^{K} |\mathcal{V}_k| = B$, $\mathcal{C}_{avg}$ in Equation 5.3 is minimized when all $|\mathcal{V}_k| = B/K, \forall k$. Returning to the 3-PARTITION problem, if our decision to 3-PARTITION is true, then we can partition the versions in the constructed graph G accordingly and $\mathcal{C}_{avg} = B/K + D$ with each $|\mathcal{V}_k| = \frac{B}{K} = \frac{B}{n/3}$. Second, if the decision problem is false, then $\mathcal{C}_{avg}$ must be larger than $B/K + D$. Otherwise, all $|\mathcal{V}_k|$ must be the same and equal to B/K. Subsequently, we can easily partition $\mathcal{A}$ into $\frac{n}{3}$ sets with equal sum for 3-PARTITION, which contradicts the assumption that the decision problem is false.

We now clarify one complication between our formalization so far and our implementation. ORPHEUSDB uses the *no cross-version diff rule*: that is, while performing a commit operation, to minimize computation, ORPHEUSDB does not compare the committed version against all of the ancestor versions, instead only comparing it to its parents. Therefore, if some records are deleted and then re-added later, these records would be assigned different *rid*s, and are treated as different. As it turns out, Problem 5.1 is still NP-HARD when the space of instances are those that can be generated when this rule is applied. For the rest of this chapter, we will use the formalization with the no cross-version diff rule in place, since that relates more closely to practice.

5.2 PARTITIONING ALGORITHM

Given a version-record bipartite graph $G = (V, R, E)$, there are two extreme cases for partitioning. At one extreme, we can minimize the checkout cost by storing each version in the CVD as one partition; there are in total $K = |V| = n$ partitions, and the storage cost is $\mathcal{S} = \sum_{k=1}^{n} |\mathcal{R}_k| = |E|$ and the checkout cost is $\mathcal{C}_{avg} = \frac{1}{n}\sum_{k=1}^{n} (|\mathcal{V}_k||\mathcal{R}_k|) = \frac{|E|}{|V|}$. At another extreme, we can minimize the storage by storing all versions in one single partition; the storage cost is $\mathcal{S} = |R|$ and $\mathcal{C}_{avg} = |R|$. We illustrate these schemes in Figure 5.3. We also list them as formal observations below:

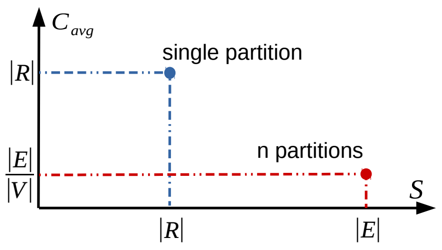


Figure 5.3: Extreme Schemes

Observation 5.1. *Given a bipartite graph $G = (V, R, E)$, the checkout cost $\mathcal{C}_{avg}$ is minimized by storing each version as one separate partition: $\mathcal{C}_{avg} = \frac{|E|}{|V|}$.*

Observation 5.2. *Given a bipartite graph $G = (V, R, E)$, the storage cost $\mathcal{S}$ is minimized by storing all versions in a single partition: $\mathcal{S} = |R|$.*

Version Graph Concept. Our goal in designing our partitioning algorithm, LyreSplit[1], is to trade-off between these two extremes. Instead of operating on the version-record bipartite graph, which may be very large, LyreSplit operates on the much smaller version graph instead, which makes it a lot more lightweight. We recall the concept of a *version graph* from Section 4.3, and depicted in Figure 4.2. We denote a version graph as $\mathbb{G} = (\mathbb{V}, \mathbb{E})$, where each vertex $v \in \mathbb{V}$ is a version and each edge $e \in \mathbb{E}$ is a derivation relationship. Note that $\mathbb{V}$ is essentially the same as V in the version-record bipartite graph. An edge from vertex v_i to a vertex v_j indicates that v_i is a parent of v_j; this edge has a weight $w(v_i, v_j)$ equals the number of records in common between v_i and v_j. We use $p(v_i)$ to denote the parent versions of v_i. For the special case when there are no merge operations, $|p(v_i)| \leq 1, \forall i$, and the version graph is a tree, denoted as $\mathbb{T} = (\mathbb{V}, \mathbb{E})$. Lastly, we use $R(v_i)$ to be the set of all records in version v_i, and $l(v_i)$ to be the depth of v_i in the version graph $\mathbb{G}$ in a topological sort[2] of the graph—the root has depth 1. For example, in Figure 4.2, version v_2 has $|R(v_2)| = 3$ since it has three records, and is at level $l(v_2) = 2$. Further, v_2 has a single parent $p(v_2) = v_1$, and shares two records with its parent, i.e., $w(v_1, v_2) = 2$. Next, we describe the algorithm for LyreSplit when the version graph is a tree (i.e., no merge operations). We then naturally extend our algorithm to other settings, as we will describe next.

[1]A lyre was the musical instrument of choice for Orpheus.

[2]In each iteration r, topological sorting algorithm finds vertices V' with in-degree equals 0, removes V', and updates in-degree of other vertices. $l(v_i) = r, \forall v_i \in V'$.

The Version Tree Case. Our algorithm is based on the following lemma, which intuitively states that if every version v_i shares a large number of records with its parent version, then the checkout cost is small, and bounded by some factor of $\frac{|E|}{|V|}$, where $\frac{|E|}{|V|}$ is the lower bound on the optimal checkout cost (from Observation 5.1).

Lemma 5.1. *Given a bipartite graph $G = (V, R, E)$, a version tree $\mathbb{T} = (\mathbb{V}, \mathbb{E})$, and a parameter $\delta \leq 1$, if the weight of every edge in $\mathbb{E}$ is larger than $\delta|R|$, then the checkout cost $\mathcal{C}_{avg}$ when all of the versions are in one single partition is less than $\frac{1}{\delta} \cdot \frac{|E|}{|V|}$.*

Proof. Consider the nodes of the version tree $\mathbb{T}$ level-by-level, starting from the root. That is, all of a version's ancestors are considered before it is evaluated. Now, given a version v_i, the number of new records added by v_i is $R(v_i) - w(v_i, p(v_i))$. Thus, we have:

$$\begin{aligned}|R| &= |\cup_{i=1}^{|V|} R(v_i)| \\ &= R(v_1) + \sum_{l(v_i)=2} (R(v_i) - w(v_i, p(v_i))) \\ &\quad + \sum_{l(v_i)=3} (R(v_i) - w(v_i, p(v_i))) + \cdots \qquad (5.4) \\ \implies |R| &= \sum_{i=1}^{|V|} R(v_i) - \sum_{i=2}^{|V|} (w(v_i, p(v_i)))\end{aligned}$$

Since each edge weight is larger than $\delta|R|$, i.e., $w(v_i, p(v_i)) > \delta|R|, \forall 2 \leq i \leq |V|$, we have:

$$|R| < |E| - \delta(|V| - 1)|R| \leq |E| - \delta|V||R| + |R| \qquad (5.5)$$

where the last inequality is because $\delta \leq 1$. Thus, we have $|R| < \frac{1}{\delta} \cdot \frac{|E|}{|V|}$. Since $\mathcal{C}_{avg} = |R|$ when we have only one partition, the result follows. ∎

Lemma 5.1 indicates that when $\mathcal{C}_{avg} \geq \frac{1}{\delta} \cdot \frac{|E|}{|V|}$, there must exist some version v_j that only shares a small number of common records with its parent version v_i, i.e., $w(v_i, v_j) \leq \delta|R|$; otherwise $\mathcal{C}_{avg} < \frac{1}{\delta} \cdot \frac{|E|}{|V|}$. Intuitively, such an edge (v_i, v_j) with $w(v_i, v_j) \leq \delta|R|$ is a potential edge for splitting since the overlap between v_i and v_j is small.

LyreSplit Illustration. We describe a version of LyreSplit that accepts as input a parameter δ, and then recursively applies partitioning until the overall $\mathcal{C}_{avg} < \frac{1}{\delta} \cdot \frac{|E|}{|V|}$; we will adapt this to Problem 5.1 later. The pseudocode is provided in Algorithm 5.1, and we illustrate its execution on an example in Figure 5.4.

As before, we are given a version tree $\mathbb{T} = (\mathbb{V}, \mathbb{E})$. We start with all of the versions in one partition. We first check whether $|R||V| < \frac{|E|}{\delta}$ (line 1). If yes, then we terminate; otherwise, we pick one edge e^* with weight $e^*.w \leq \delta|R|$ (lines 5–6) to cut in order to split the partition

Algorithm 5.1: LyreSplit ($\mathbb{G}, |R|, |V|, |E|, \delta$)

```
Input  : Version tree $\mathbb{G} = (\mathbb{V}, \mathbb{E})$ and parameter $\delta$
Output : Partitions $\{\mathcal{P}_1, \mathcal{P}_2, \cdots, \mathcal{P}_K\}$
1  if $|R| \times |V| < \frac{|E|}{\delta}$ then
2      return $V$
3  end
4  else
5      $\Omega \leftarrow \{e | e.w \leq \delta \times |R|, e \in \mathbb{E}\}$
6      $e^* \leftarrow$ PickOneEdgeCut($\Omega$)
7      Remove $e^*$ and split $\mathbb{G}$ into two parts $\{\mathbb{G}_1, \mathbb{G}_2\}$
8      Update the number of records, versions and bipartite edges in $\mathbb{G}_1$, denoted as $|R_1|$,
         $|V_1|$ and $|E_1|$
9      Update the number of records, versions and bipartite edges in $\mathbb{G}_2$, denoted as $|R_2|$,
         $|V_2|$ and $|E_2|$
10     $\mathcal{P}_1$=LyreSplit ($\mathbb{G}_1, |R_1|, |V_1|, |E_1|, \delta$)
11     $\mathcal{P}_2$=LyreSplit ($\mathbb{G}_2, |R_2|, |V_2|, |E_2|, \delta$)
12     return $\{\mathcal{P}_1, \mathcal{P}_2\}$
13 end
```

into two. According to Lemma 5.1, if $|R||V| \geq \frac{|E|}{\delta}$, there must exist some edge whose weight is no larger than $\delta|R|$. The algorithm does not prescribe a method for picking this edge if there are multiple; the guarantees hold independent of this method. For instance, we can pick the edge with the smallest weight; or the one such that after splitting, the difference in the number of versions in the two partitions is minimized. In our experiments, we use the latter, and break a tie by selecting the edge that balances the records between two partitions in addition to the number of versions. In our example in Figure 5.4(a), we first find that having the entire version tree as a single partition violates the property, and we pick the red edge to split the version tree $\mathbb{T}$ into two partitions—as shown in Figure 5.4(b), we get one partition $\mathcal{P}_1$ with the blue nodes (versions) and another $\mathcal{P}_2$ with the red nodes (versions).

After each edge split, we update the number of records, versions and bipartite edges (lines 8–9), and then we recursively call the algorithm on each partition (lines 10–11). In the example, we terminate for $\mathcal{P}_2$ but we split the edge (v_2, v_4) for $\mathcal{P}_1$, and then terminate with three partitions—Figure 5.4(c). We define ℓ be the recursion level number. In Figure 5.4 (a) (b) and (c), $\ell = 0$, $\ell = 1$ and $\ell = 2$ respectively. We will use this notation in the performance analysis next.

Analysis of δ. Now that we have an algorithm for the δ case, we can simply apply binary search on δ and obtain the best δ for Problem 5.1. Given a storage budget γ in Problem 5.1, we can simply perform a binary search on δ and get the best δ as the input for Algorithm 5.1. This claim is evidenced by the fact that the same sequence of edges are snipped for different

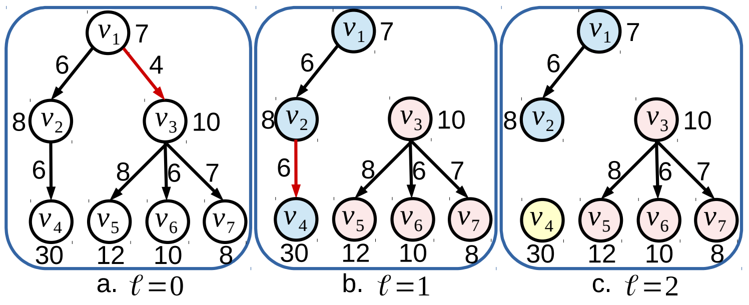


Figure 5.4: Illustration of LyreSplit ($\delta = 0.5$)

δ. In general, as δ increases, there are more partitions, consequently less checkout cost and larger storage cost.

- *Superset property of δ.* Consider two different δ: δ_1 and δ_2, without loss of generality we assume $\delta_1 < \delta_2$, and to simplify the analysis we pick the smallest weight as the splitting edge in each iteration. First we claim that Algorithm 5.1 takes more iterations when $\delta = \delta_2$ than $\delta = \delta_1$. This is because $\delta_1 < \delta_2$ and the termination constraint is $|R||V| < \frac{|E|}{\delta}$. Next, we assert that the edges cut when $\delta = \delta_1$ is a subset of the same sequence of $\delta = \delta_2$. This is because in each iteration, the edge with the smallest weight is cut for both δ_1 and δ_2, and when δ_1 terminates ($|R||V| < \frac{|E|}{\delta_1}$), δ_2 may still goes on since $|R||V| \geq \frac{|E|}{\delta_2}$. Thus, compared to δ_1, δ_2 has more splits, larger storage cost, and less checkout cost.

- *Binary search on δ.* Initially, the search space for δ is $[\frac{|E|}{|R||V|}, 1]$, where each version is stored in a separate partiton(i.e., $\delta = 1$) and all versions are in the same partition(i.e., $\delta = \frac{|E|}{|R||V|}$). We first try $\delta = \frac{1}{2}(\frac{|E|}{|R||V|} + 1)$ in Algorithm 5.1 and get the resulting storage cost $\mathcal{S}$ after partitioning. If $\mathcal{S} < \gamma$, then the search space for δ is reduced to $[\frac{1}{2}(\frac{|E|}{|R||V|} + 1), 1]$; otherwise, $[\frac{|E|}{|R||V|}, \frac{1}{2}(\frac{|E|}{|R||V|} + 1)]$. Repeat this process until $0.99\gamma \leq \mathcal{S} \leq \gamma$.

Performance Analysis. Overall, the lowest storage cost is $|R|$ and the lowest checkout cost is $\frac{|E|}{|V|}$ respectively (as formalized in Observation 5.1 and 5.2). We now analyze the performance in terms of these quantities: an algorithm has an approximation ratio of (X, Y) if its storage cost $\mathcal{S}$ is no larger than $X \cdot R$ while its checkout cost $\mathcal{C}_{avg}$ is no larger than $Y \cdot \frac{|E|}{|V|}$. We first study the impact of a single split edge.

Lemma 5.2. *Given a bipartite graph $G = (V, R, E)$, a version tree $\mathbb{T} = (\mathbb{V}, \mathbb{E})$ and a parameter δ, let $e^* \in \mathbb{E}$ be the edge that is split in* LyreSplit*, then after splitting the storage cost $\mathcal{S}$ must be within $(1 + \delta)|R|$.*

Proof. First according to Lemma 5.1, if $|R||V| \geq \frac{|E|}{\delta}$, there must exist some edge $e^* = (v_i, v_j)$ whose weight is less than $\delta|R|$, i.e., $e^*.w \leq \delta|R|$. Then, we remove one such e^* and split $\mathbb{G}$ into two parts $\{\mathbb{G}_1, \mathbb{G}_2\}$ as depicted in line 7-9 in Algorithm 5.1. The current storage cost $\mathcal{S} = |R_1| + |R_2|$. The common records between $\mathbb{G}_1$ and $\mathbb{G}_2$ is exactly the common records shared by version v_i and v_j, i.e., $e^*.w$. Thus, we have:

$$\begin{aligned} |R| = |R_1 \cup R_2| = |R_1| + |R_2| - e^*.w \geq |R_1| + |R_2| - \delta|R| \\ \implies \mathcal{S} = |R_1| + |R_2| \leq (1+\delta)|R| \end{aligned} \tag{5.6}$$

Hence proved.

Now, overall, we have:

Theorem 5.2. *Given a parameter δ,* LyreSplit *results in a $((1+\delta)^\ell, \frac{1}{\delta})$-approximation for partitioning.*

Proof. Let us consider all partitions when Algorithm 5.1 terminates at level ℓ. Each partition (e.g., Figure 5.4(c)) corresponds to a subgraph of the version tree (e.g., Figure 5.4(a)). According to Lemma 5.1, the total checkout cost $\mathcal{C}_k$ in each partition $\mathcal{P}_k = (\mathcal{V}_k, \mathcal{R}_k, \mathcal{E}_k)$ must be smaller than $\frac{|\mathcal{E}_k|}{\delta}$, where $|\mathcal{E}_k|$ is the number of bipartite edges in partition $\mathcal{P}_k$. Since $\sum_{k=1}^{K} |\mathcal{E}_k| = |E|$, we prove that the overall average checkout cost $\mathcal{C}_{avg}$ is $\frac{\sum \mathcal{C}_k}{|V|} < \frac{1}{\delta} \cdot \frac{|E|}{|V|}$. Next, we consider the storage cost. The analysis is similar to the complexity analysis for quick sort. Our proof uses a reduction on the recursive level number ℓ. First, when $\ell = 0$, all versions are stored in a single partition (e.g. Figure 5.4(a)). Thus, the storage cost is $|R|$. Next, as the recursive algorithm proceeds, there can be multiple partitions at each recursive level ℓ. For instance, there are two partitions at level $\ell = 1$ and three partitions at level $\ell = 2$ as shown in Figure 5.4(b) and (c). Assume that there are τ partitions $\{\mathcal{P}_1, \mathcal{P}_2, \cdots, \mathcal{P}_\tau\}$ at level $\ell = \alpha$, and the storage cost for these partitions is no bigger than $(1+\delta)^\alpha \cdot |R|$. Then according to Lemma 5.2, for each partition $\mathcal{P}_k$ at level $\ell = \alpha$, after splitting the storage cost at level $(\alpha+1)$ will be no bigger than $(1+\delta)$ times that at level α. Thus, we have the total storage cost at level $(\alpha+1)$ must be no bigger than $(1+\delta)^{\alpha+1} \cdot |R|$.

Complexity. At each recursive level of Algorithm 5.1, it takes $O(n)$ time for checking the weight of each edge in the version tree (line 5). The update in line 8–9 can also be done in $O(n)$ using one pass of tree traversal for each partition. The total time complexity is $O(n\ell)$, where ℓ is the recursion level number when the algorithm terminates.

5.3 GENERALIZATIONS

We can naturally extend our algorithms for the case where the version graph is a DAG: in short, we first construct a version tree $\hat{\mathbb{T}}$ based on the original version graph $\mathbb{G}$, then apply LyreSplit on the constructed version tree $\hat{\mathbb{T}}$. We will also discuss the *weighted* case, where the weight can help represent the workload in real world settings, e.g., recent versions may be checked out more frequently. At last, we will talk about how our algorithm can be adapted to the scenario with schema change.

5.3.1 Version graph is a DAG

When there are merges between versions, the version graph $\mathbb{G} = (\mathbb{V}, \mathbb{E})$ is a DAG. We can simply transform the $\mathbb{G}$ to a version tree $\hat{\mathbb{T}}$ and then apply LyreSplit as before. Specifically, for each vertex $v_i \in \mathbb{V}$, if there are multiple incoming edges, we retain the edge with the highest weight and remove all other incoming edges. In other words, for each merge operation in the version graph $\mathbb{G}$, e.g., where v_i is merged with v_j to obtain v_k, the corresponding operation in $\hat{\mathbb{T}}$ with the removed edge (v_j, v_k) is to inherit records only from one parent v_i and (conceptually) create new records in the CVD for all other records in v_k even though some records have exactly the same value as that in v_j.

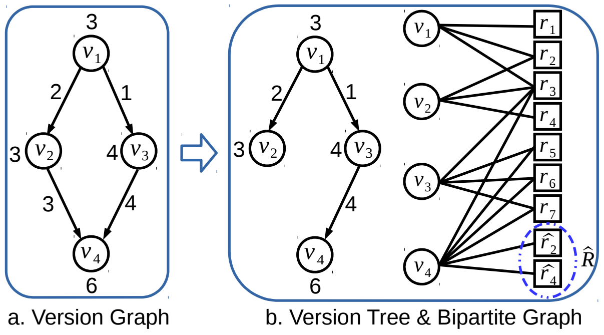


Figure 5.5: $\hat{\mathbb{T}}$ and $\hat{G}$ for $\mathbb{G}$ in Figure 4.2

For example, for the version graph $\mathbb{G}$ shown in Figure 7.1(a), its version v_4 has two parent versions v_2 and v_3. Since $3 = w(v_2, v_4) < w(v_3, v_4) = 4$, we remove edge (v_2, v_4) from $\mathbb{G}$ and obtain the version tree $\hat{\mathbb{T}}$ in Figure 7.1(b). Moreover, conceptually, we can draw a bipartite graph $\hat{G}$ corresponding to $\hat{\mathbb{T}}$ as shown in Figure 7.1(b) with two duplicated records, i.e., $\{\hat{r_2}, \hat{r_4}\}$. That is, v_4 in $\hat{\mathbb{T}}$ inherits 4 records from v_3 and creates two new records $\hat{R} = \{\hat{r_2}, \hat{r_4}\}$ even though $\hat{r_2}$ ($\hat{r_4}$) is exactly the same as r_2 (r_4). Thus, we have 9 records with $|\hat{R}| = 2$ and 16 bipartite edges in Figure 7.1(b).

Performance analysis. The number of bipartite edges in the bipartite graph $\hat{G}$ (corresponding to $\hat{\mathbb{T}}$) is the same as that in G (corresponding to $\mathbb{G}$), i.e., $|E|$. However, compared to G, the number of records in $\hat{G}$ is larger, i.e., $|R| + |\hat{R}|$, where R is the set of records in the original version-record bipartite graph G and $\hat{R}$ is the set of duplicated records. According to Theorem 5.2, given δ, LyreSplit provides a partitioning scheme with the checkout cost within $\frac{1}{\delta} \cdot \frac{|E|}{|V|}$ and the storage cost within $(1+\delta)^{\ell}(|R| + |\hat{R}|)$. We formally state the performance guarantee in Theorem 5.3. Moreover, this analysis is obtained by treating $\hat{R}$ as different from R when calculating the storage cost and checkout cost. In post-processing, we can combine $\hat{R}$ with R when calculating the real storage cost and checkout cost, making the real $\mathcal{S}$ and $\mathcal{C}_{avg}$ even smaller.

Theorem 5.3. *Given a version graph $\mathbb{G}$ with merges and a parameter δ,* LyreSplit *results in a $(\frac{|R|+|\hat{R}|}{|R|}(1+\delta)^{\ell}, \frac{1}{\delta})$-approximation for partitioning.*

5.3.2 Weighted Checkout Cost

In this section, we focus on the weighted checkout cost case, where versions are checked out with different frequencies.

Problem formulation. Let $\mathcal{C}_w$ denote the weighted checkout cost; say version v_i is checked out with probability or frequency f_i Then the weighted checkout cost $\mathcal{C}_w$ can be represented as $\mathcal{C}_w = \frac{\sum_{i=1}^{n}(f_i \times \mathcal{C}_i)}{\sum_{i=1}^{n} f_i}$. With this weighted checkout cost, we can modify the problem formulation for Problem 5.1 by simply replacing $\mathcal{C}_{avg}$ with $\mathcal{C}_w$.

Proposed Algorithm. Without the loss of generality, we assume that f_i for any version v_i is an integer. Given a version tree[3] $\mathbb{T} = (\mathbb{V}, \mathbb{E})$ and the frequency f_i for each version v_i, we construct a version tree $\mathbb{T}' = (\mathbb{V}', \mathbb{E}')$ in the following way:

- For each version $v_i \in \mathbb{V}$:
  - $\mathbb{V}'$: Create f_i versions $\{v_i^1, v_i^2, \cdots, v_i^{f_i}\}$ in $\mathbb{V}'$;
  - $\mathbb{E}'$: Connect v_i^j with v_i^{j+1} to form a chain in $\mathbb{E}'$, where $1 \leq j < f_i$
- For each edge $(v_i, v_j) \in \mathbb{E}$:
  - $\mathbb{E}'$: Connect $v_i^{f_i}$ with v_j^1 in $\mathbb{E}'$

The basic idea of constructing $\mathbb{T}'$ is to duplicate each version $v_i \in \mathbb{V}$ f_i times. Afterwards, we apply LyreSplit directly on $\mathbb{T}'$ to obtain the partitioning scheme. However, after partitioning, $v_i^j \in \mathbb{V}'$ with the same i may be assigned to different partitions, denoted as P'.

[3]if the version graph is a DAG instead, we first transform it into a version tree as discussed in Section 5.3.1.

Thus, as a post process, we move all v_i^j $(1 \leq j \leq f_i)$ into the same partition $\mathcal{P} \in P'$ that has the smallest number of records. Correspondingly, we get a partitioning scheme for $\mathbb{V}$, i.e., for each $v_i \in \mathbb{V}$, assign it to the partition where $v_i^j \in \mathbb{V}'$ $(1 \leq j \leq f_i)$ is in.

Performance analysis. At one extreme, when each version is stored in a separate table, the checkout cost $\mathcal{C}_w$ for $\mathbb{T}$ is the lowest with each $\mathcal{C}_i = |R(v_i)|$, the number of records in version v_i; thus, $\mathcal{C}_w = \frac{\sum_{i=1}^{n}(f_i \times |R(v_i)|)}{\sum_{i=1}^{n} f_i}$, denoted as ζ. At the other extreme, when all versions are stored in a single partition, the total storage cost is the smallest, i.e., $|R|$. In the following, we study the performance of the extended algorithm in the weighted case, and compare the storage cost and weighted checkout cost with $|R|$ and ζ respectively.

First, consider the bipartite graph $G' = (V', R', E')$ corresponding to the constructed version tree $\mathbb{T}'$. The number of versions $|V'|$ equals $\sum_{i=1}^{n} f_i$, since there are f_i replications for each version v_i; the number of records $|R'|$ is the same as $|R|$, since there are no new records added; the number of bipartite edges $|E'|$ is $\sum_{i=1}^{n}(\sum_{j=1}^{f_i} |R(v_i^j)|) = \sum_{i=1}^{n}(f_i \times |R(v_i)|)$, since the number of records in each version v_i^j with the same i is in fact $|R(v_i)|$. Next, based on Theorem 5.2, the average checkout cost after appyling Algorithm 5.1 is within $\frac{1}{\delta} \cdot \frac{|E'|}{|V'|} = \frac{1}{\delta} \cdot \frac{\sum_{i=1}^{n}(f_i \times |R(v_i)|)}{\sum_{i=1}^{n} f_i} = \frac{1}{\delta} \cdot \zeta$, while the storage cost is within $(1+\delta)^{\ell} \cdot |R'| = (1+\delta)^{\ell} \cdot |R|$, where ℓ is the termination level in Algorithm 5.1. After post-processing, the total storage cost as well the average checkout cost decreases since we pick the partition with the smallest number of records for all v_i^j with a fixed i. At last, note that after mapping the partitioning scheme from $\mathbb{T}'$ to $\mathbb{T}$, the total storage cost and the average (unweighted) checkout cost for $\mathbb{T}'$ are in fact the total storage cost and the weighted checkout cost for $\mathbb{T}$ respectively. Thus, with the extended algorithm, we achieve the same approximation bound as in Theorem 5.2 with respect to the lowest storage cost and weighted checkout cost, i.e., in the weighted checkout case, our algorithm also results in $((1 + \delta)^{\ell}, \frac{1}{\delta})$-approximation for partitioning.

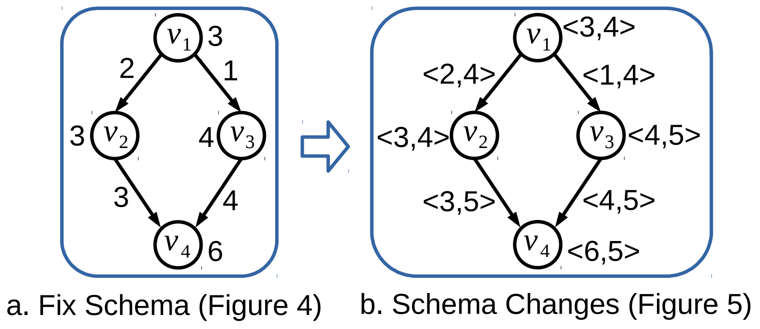


Figure 5.6: Version Graph $\mathbb{G}$ with/without Schema Changes

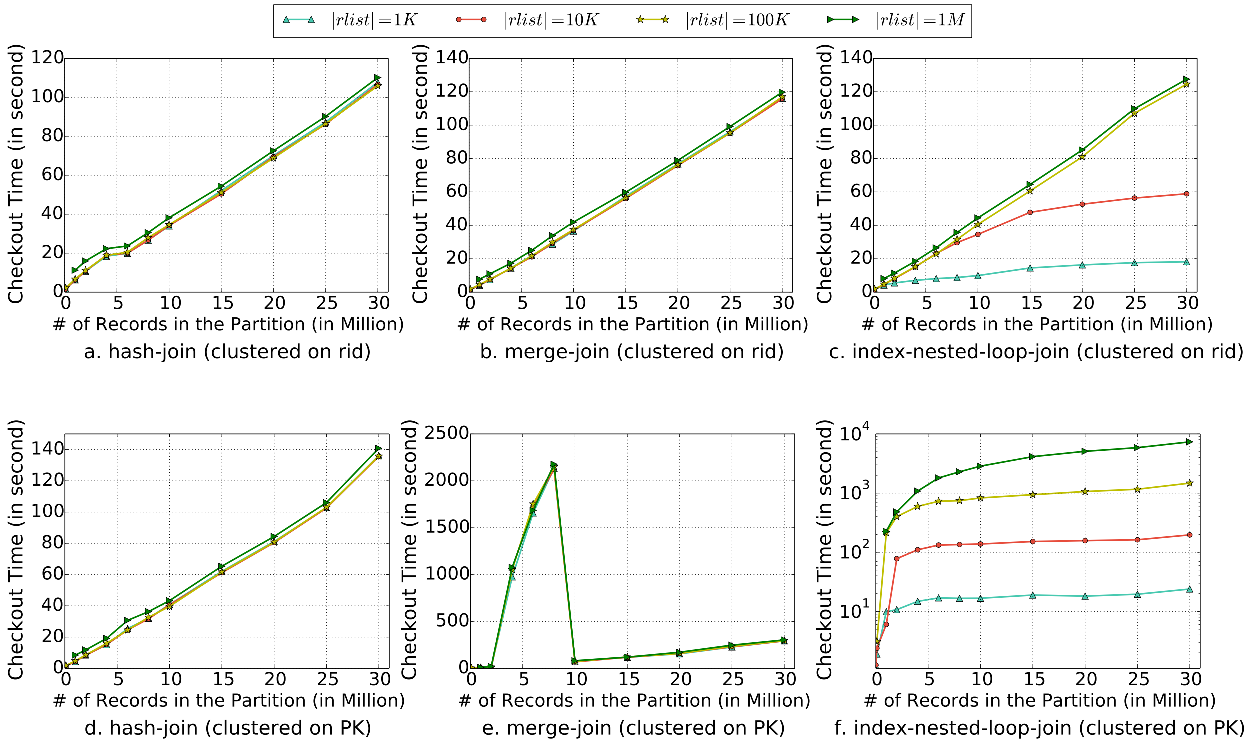


Figure 5.7: Checkout Cost Model Validation

5.3.3 Schema Changes

Our algorithm can be adapted to the single-pool setting described in Section 4.3 in Chapter 4 with schema changes. Recall the examples in Figure 4.2 and 4.3, corresponding to the fixed and dynamic schema settings. Figure 5.6a only maintains the number of records in each node (version), and the number of common records between two versions for each edge. In addition, Figure 5.6b also records the number of attributes and common attributes for each node and edge respectively. For instance, v_3 has five attributes and shares four common attributes with v_1.

Given a version graph, let A be the total number of attributes in all versions. For instance, Figure 5.6b, corresponding to Figure 4.3, has five attributes in total. Without partitioning, the storage cost and the checkout cost can be represented as $\mathcal{S} = |A||R|$ and $\mathcal{C}_{avg} = |A||R|$ respectively, where $|R|$ is the number of records. Next, let $a(v_i)$ and $a(v_i, v_j)$ denote the number of attributes in version v_i and the number of common attributes between version v_i and v_j, respectively. Recall that $w(v_i, v_j)$ denotes the number of common records between version v_i and v_j, disregarding the schema. For instance, if version v_j is obtained by deleting an attribute from v_i, then $a(v_i, v_j) = a(v_i) - 1$ and $w(v_i, v_j) = |R(v_i)|$.

The high-level idea is similar to LyreSplit: split an edge if its "weight" is smaller than some threshold. However, the weight here not only depends on the number of common records $w(v_i, v_j)$, but also the number of common atrributes $a(v_i, v_j)$. Specifically, if $a(v_i, v_j) \times w(v_i, v_j) \leq \delta \times |A||R|$, edge (v_i, v_j) is considered as a candidate splitting edge[4]. Note that when there is no schema change, $a(v_i, v_j) = |A|$, and the constraint is reduced to $w(v_i, v_j) \leq \delta|R|$ (line 5 in Algorithm 5.1). The remaining algorithm is the same as Algorithm 5.1.

5.4 INCREMENTAL PARTITIONING

LyreSplit can be explicitly invoked by users or by OrpheusDB when there is a need to improve performance or a lull in activity. We now describe how the partitioning identified by LyreSplit is incrementally maintained during the course of normal operation, and how we reduce the migration time when LyreSplit identifies a new partitioning.

Online Maintenance. When a new version v_i is committed, OrpheusDB applies the same intuition as LyreSplit to determine whether to add v_i to an existing partition, or to create a new partition. This is again a trade-off between the storage cost and the checkout cost. Compared to creating a new table, adding v_i to an existing partition has smaller storage cost but larger checkout cost. Sharing the same intuition with LyreSplit: if v_i has a large number of common records with one of its parent version v_j, we opt to add v_i into the partition $\mathcal{P}_k$ where v_j is in. This is because the added storage cost is minimized and the added checkout cost is guaranteed to be small as stated in Lemma 5.1. Essentially, the online maintenance is performing incremental partitioning in the version graph as new versions are coming in. Specifically, if $w(v_i, v_j) \leq \delta^*|R|$ and $\mathcal{S} < \gamma$, where δ^* was the splitting parameter used during the last invocation of LyreSplit, then we create a new version; otherwise, v_i is added to partition $\mathcal{P}_k$. Recall that γ is the storage threshold and $|R|$ is the number of records currently.

Even with the proposed online maintenance scheme, the checkout cost tends to diverge from the best checkout cost that LyreSplit can identify under the current constraints. This is because LyreSplit performs global partitioning using the full version graph as input, while online maintenance makes small changes to the existing partitioning. To maintain the checkout performance, OrpheusDB allows for a tolerance factor μ on the current checkout cost (users can also set μ explicitly). We let $\mathcal{C}_{avg}$ and $\mathcal{C}^*_{avg}$ be the current checkout cost and the best checkout cost identified by LyreSplit respectively. If $\mathcal{C}_{avg} > \mu\mathcal{C}^*_{avg}$, the *migration engine* is triggered, and we reorganize the partitions by migrating data from the old partitions

[4]If each attribute is of different size, we can simply replace ”the number of attributes” with ”the number of bytes” in the whole algorithm.

to the new ones; until then, we perform online maintenance. In general, when μ is small, the migration engine is invoked more frequently. Next, we discuss how migration is performed.

Migration Approach. Given the existing partitioning $P = \{\mathcal{P}_1, \mathcal{P}_2, \ldots, \mathcal{P}_\alpha\}$ and the new partitioning $P' = \{\mathcal{P}'_1, \mathcal{P}'_2, ..., \mathcal{P}'_\beta\}$ identified by LyreSplit, we need an algorithm to efficiently migrate the data from P to P' without dropping all existing tables and recreating the partitions from scratch, which could be very costly. The question asked here is whether we can make use of the existing tables and only perform some small modifications accordingly. To do so, OrpheusDB needs to identify, for every $\mathcal{P}'_i \in P'$, the closest partition $\mathcal{P}_j \in P$, in terms of modification cost, defined as $|\mathcal{R}'_i \setminus \mathcal{R}_j| + |\mathcal{R}_j \setminus \mathcal{R}'_i|$, where $\mathcal{R}'_i \setminus \mathcal{R}_j$ and $\mathcal{R}_j \setminus \mathcal{R}'_i$ are the records needed to be inserted and deleted respectively to transform $\mathcal{P}_j$ to $\mathcal{P}'_i$. This task consists of two main steps: 1) calculate the number of modifications needed for each partition pair $(\mathcal{P}'_i, \mathcal{P}_j)$; 2) find the closest partition $\mathcal{P}_j$ for each $\mathcal{P}'_i \in P'$. For step one, if we calculate the modification cost directly based on $\mathcal{R}'_i$ and $\mathcal{R}_j$, it may be very expensive especially when the number of records is large. Instead, we first find the common versions in $\mathcal{P}'_i$ and $\mathcal{P}_j$, and then calculate the number of common records based on the version graph $\mathbb{G}$ without probing into $\mathcal{R}'_i$ or $\mathcal{R}_j$. Next, for step two, we greedily pick the partition pair $(\mathcal{P}'_i, \mathcal{P}_j)$ with the smallest modification cost and assign $\mathcal{P}_j$ to $\mathcal{P}'_i$. Finally, we perform insertions and deletions on $\mathcal{P}_j$ accordingly to obtain $\mathcal{P}'_i$. Note that if the modification cost is larger than $|\mathcal{R}'_i|$, we would prefer to build partition $\mathcal{P}'_i$ from scratch rather than modifying the existing partition $\mathcal{P}_k$.

5.5 PARTITIONING EVALUATION

While Section 4.2 in Chapter 4 explores the performance of data models, this section evaluates the impact of partitioning. In Section 5.5.2, we evaluate if LyreSplit can be more efficient than existing partitioning techniques; in Section 5.5.3, we ask whether versioned databases strongly benefit from partitioning; and lastly, in Section 5.5.4 we evaluate how LyreSplit performs for online scenarios.

5.5.1 Experimental Setup

Datasets. We evaluated the performance of LyreSplit using the versioning benchmark datasets from Maddox et al. [31]. The versioning model used in the benchmark is similar to `git`, where a branch is a working copy of a dataset. For simplicity, we can think of branches as different users' working copies. We selected the Science (SCI) and Curation (CUR) workloads since they are most representative of real-world use cases. The SCI workload simulates the working patterns of data scientists, who often take copies of an evolving dataset for isolated data analysis. The version graph here can be visualized as a mainline (i.e., a single linear version chain) with various branches at different points—both from different points on the mainline as well as from other already existing branches. Thus, the version graph is analogous to a tree with branches. The CUR workload simulates the evolution of a canonical dataset that many individuals are contributing to—these individuals not just branch from the canonical dataset but also periodically merge their changes back in, resulting in a DAG of versions. Branches can be created from existing branches, and then merged back into the parent branch. We varied the following parameters when we generated the benchmark datasets: the number of branches $\mathbb{B}$, the total number of records $|R|$, as well as the number of inserts (or updates) from parent version(s) $\mathbb{I}$. We list our configurations in Table 5.2. For instance, dataset SCI_1M represents a SCI workload dataset where the input parameter corresponding to $|R|$ in the dataset generator is set to $1M$ records. Note that due to the inherent randomness in the dataset generator, the actual number of records generated does not perfectly match the value of $|R|$ we input to the generator. Furthermore, since the version graphs for all CUR_* datasets are DAGs (i.e., have multiple merges between versions), we also list their $|\hat{R}|$, the number of duplicated records described in Section 5.3.1. Compared with $|R|$, $|\hat{R}|$ is about 7 to 10 percent of $|R|$. In all of our datasets, each record contains 100 attributes, each of which is a 4-byte integer.

Dataset	$\lvert V\rvert$	$\lvert R\rvert$	$\lvert E\rvert$	$\lvert\mathbb{B}\rvert$	$\lvert\mathbb{I}\rvert$	$\lvert\hat{R}\rvert$
SCI_1M	1K	944K	11M	100	1000	-
SCI_2M	1K	1.9M	23M	100	2000	-
SCI_5M	1K	4.7M	57M	100	5000	-
SCI_8M	1K	7.6M	91M	100	8000	-
SCI_10M	10K	9.8M	556M	1000	1000	-
CUR_1M	1.1K	966K	31M	100	1000	90K
CUR_5M	1.1K	4.8M	157M	100	5000	0.35M
CUR_10M	11K	9.7M	2.34G	1000	1000	0.9M

Table 5.2: Dataset Description

Setup. We conducted our evaluation on a HP-Z230-SFF workstation with an Intel Xeon E3-1240 CPU and 16 GB memory running Linux OS (LinuxMint). We built OrpheusDB as a wrapper written in C++ over PostgreSQL 9.5[5], where we set the memory for sorting and hash operations as $1GB$ (i.e., `work_mem=1GB`) to reduce external memory sorts and joins. In addition, we set the buffer cache size to be minimal (i.e., `shared_buffers =128KB`) to

[5]PostgreSQL's version 9.5 added the feature of dynamically adjusting the number of buckets for hash-join.

eliminate the effects of caching on performance. In our evaluation, for each dataset, we randomly sampled 100 versions and used them to get an estimate of the checkout time. Each experiment was repeated 5 times, with the OS page cache being cleaned before each run. Due to experimental variance, we discarded the largest and smallest number among the five trials, and then took the average of the remaining three trials.

Algorithms. We compare LyreSplit against two partitioning algorithms in the NScale graph partitioning project [61]: the Agglomerative Clustering-based one (Algorithm 4 in [61]) and the KMeans Clustering-based one (Algorithm 5 in [61]), denoted as Agglo and Kmeans respectively: Kmeans had the best performance, while Agglo is an intuitive method for clustering versions. After mapping their setting into ours, like LyreSplit, NScale [61]'s algorithms group versions into different partitions while allowing the duplication of records. However, the NScale algorithms are tailored for arbitrary graphs, not for bipartite graphs (as in our case).

We implement Agglo and Kmeans as described. Agglo starts with each version as one partition and then sorts these partitions based on a shingle-based[6] ordering. Then, in each iteration, each partition is merged with a candidate partition that it shares the largest number of common shingles with. The candidate partitions have to satisfy two conditions (1) the number of the common shingles is larger than a threshold τ, which is set via a uniform sampling-based method, and (2) the number of records in the new partition after merging is smaller than a constraint BC, a pre-defined maximum number of records per partition. Furthermore, based on the shingle ordering, NScale proposes that each partition only considers its following l partitions as its merging candidates and l is adjusted dynamically. In our experiments, initially l is set to 100. To address Problem 5.1 with storage threshold γ, we conduct a binary search on BC and find the best partitioning scheme under the storage constraint.

For Kmeans, there are two input parameters: partition capacity BC as in Agglo, and the number of partitions K. Initially, K random versions are assigned to partitions, the centroid of which is initialized as the set of records in each partition. Next, we assign the remaining versions to their nearest centroid based on the number of common records, after which each centroid is updated to the union of all records in the partition. In subsequent iterations, each version is moved to a partition, such that after the movement, the total number of records across partitions is minimized, while respecting the constraint that the number of records in each partition is no larger than BC. The number of Kmeans iterations is set to 10. In our experiment, we vary K and set BC to be infinity. We tried other values

[6]Shingles are calculated as signatures of each partition based on a min-hashing based technique.

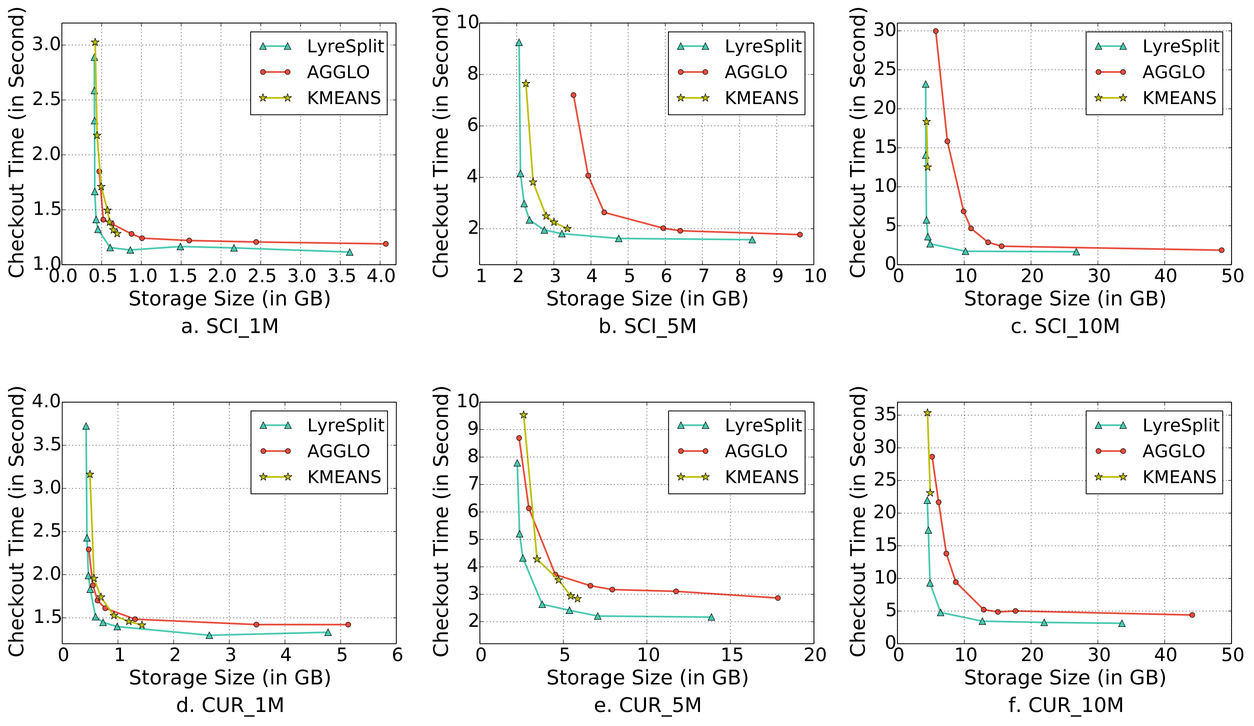


Figure 5.8: Storage Size vs. Checkout Time

for BC; the results are similar to that when BC is infinity. Overall, with an increase of K, the total storage cost increases and the checkout cost decreases. Again, we use binary search to find the best K for KMEANS and minimize the checkout cost under the storage constraint γ for Problem 5.1.

5.5.2 Comparison of Partitioning Algorithms

In these experiments, we consider both datasets where the version graph is a tree, i.e., there are no merges (SCI_1M, SCI_5M and SCI_10M), and datasets where the version graph is a DAG (CUR_1M, CUR_5M and CUR_10M). We first compare the effectiveness of different partitioning algorithms: LYRESPLIT, AGGLO and KMEANS, in balancing the storage size and the checkout time. Then, we compare the efficiency of these algorithms by measuring their running time.

Effectiveness Comparison.

LYRESPLIT dominates AGGLO and KMEANS with respect to the storage size and checkout time after partitioning, i.e., with the same storage size, LYRESPLIT's partitioning scheme provides a smaller checkout time.

Figure 5.9: Summary of Trade-off between Storage Size and Checkout Time.

In order to trade-off between $\mathcal{S}$ and $\mathcal{C}_{avg}$, we vary δ for LYRESPLIT, BC for AGGLO and

K for KMEANS to obtain the overall trend between the storage size and the checkout time. The results are shown in Figure 5.8, where the x-axis depicts the total storage size for the data table in gigabytes (GB) and the y-axis depicts the average checkout time in seconds for the 100 randomly selected versions. Recall that for a CVD, its versioning table is of constant storage size for different partitioning schemes, so we do not include this in the storage size computation. Each point in Figure 5.8 represents a partitioning scheme obtained by one algorithm with a specific input parameter value. We terminated the execution of KMEANS when its running time exceeded 10 hours for each K, which is why there are only two points with star markers in Figure 5.8(c) and 5.8(f) respectively. The overall trend for AGGLO, KMEANS, and LYRESPLIT is that with the increase in storage size, the average checkout time first decreases and then tends to a constant value—the average checkout time when each version is stored as a separate table, which in fact corresponds to the smallest possible checkout time. For instance, in Figure 5.8(f) with LYRESPLIT, the checkout time decreases from 22s to 4.8s as the storage size increases from 4.5GB to 6.5GB, and then converges at around 2.9s.

Furthermore, LYRESPLIT has better performance than the other two algorithms in both the SCI and CUR datasets in terms of the storage size and the checkout time, as shown in Figure 5.8. For instance, in Figure 5.8(b), with 2.3GB storage budget, LYRESPLIT can provide a partitioning scheme taking 2.9s for checkout on average, while both KMEANS and AGGLO give schemes taking more than 7s. Thus, with equal or lesser storage size, the partitioning scheme selected by LYRESPLIT achieves much less checkout time than the ones proposed by AGGLO and KMEANS, especially when the storage budget is small. The reason is that LYRESPLIT takes a "global" perspective to partitioning, while AGGLO and KMEANS take a "local" perspective. Specifically, each split in LYRESPLIT is decided based on the derivation structure and similarity between various versions, as opposed to greedily merging partitions with partitions in AGGLO, and moving versions between partitions in KMEANS.

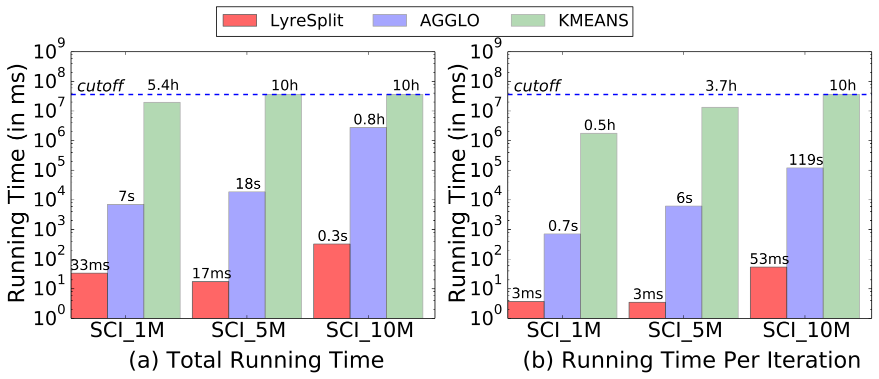


Figure 5.10: Algorithms' Running Time Comparison (SCI_*)

Efficiency Comparison.

When minimizing the checkout time under a storage constraint (Problem 5.1), LyreSplit is on average $10^3\times$ faster than Agglo, and more than $10^5\times$ faster than Kmeans for all SCI_* and CUR_* datasets.

Figure 5.11: Summary of Comparison of Running Time of Partitioning Algorithms.

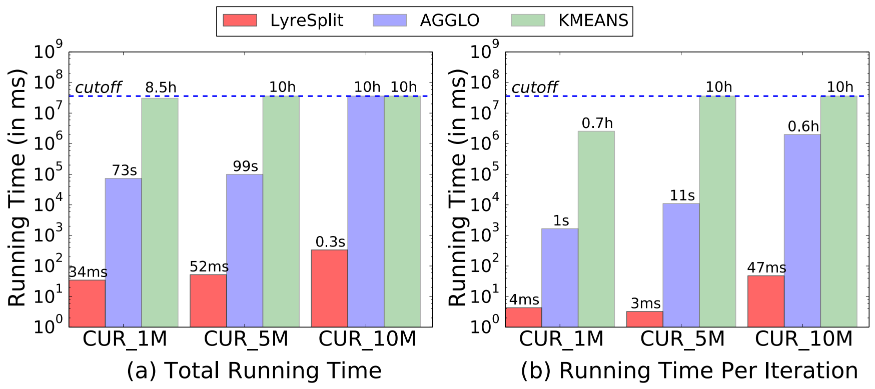


Figure 5.12: Algorithms' Running Time Comparison (CUR_*)

As discussed, given a storage constraint in Problem 5.1, we use binary search to find the best δ, BC, and K for LyreSplit, Agglo and Kmeans respectively. In this experiment, we set the storage threshold as $\gamma = 2|R|$, and terminate the binary search process when the resulting storage cost $\mathcal{S}$ meets the constraint: $0.99\gamma \leq \mathcal{S} \leq \gamma$. Figure 5.10a and 5.12a shows the total running time during the end-to-end binary search process, while Figure 5.10b and 5.12b shows the running time per binary search iteration. Again, we terminate Kmeans and Agglo when the running time exceeds 10 hours, thus we cap the running time in Figure 5.10 and 5.12 at 10 hours. We can see that LyreSplit takes much less time than Agglo and Kmeans. Consider the largest dataset SCI_10M in Figure 5.10 as an example: with LyreSplit the entire binary search procedure and each binary search iteration took 0.3s and 53ms respectively; Agglo takes 50 minutes in total; while Kmeans does not even finish a single iteration in 10 hours.

Overall, LyreSplit is $10^2\times$ faster than Agglo for SCI_1M, $10^3\times$ faster for SCI_5M, and $10^4\times$ faster for SCI_10M respectively (and is $10^3\times$ faster than Agglo for CUR_1M and CUR_5M and $10^5\times$ faster for CUR_10M respectively), and more than $10^5\times$ faster than Kmeans for all datasets. This is mainly because LyreSplit only needs to operate on the version graph while Agglo and Kmeans operate on the version-record bipartite graph, which is much larger than the version graph. Furthermore, Kmeans can only finish the binary search process within 10 hours for SCI_1M and CUR_1M. This algorithm is extremely slow due to the pairwise comparison between each version with each centroid in each iteration, especially when the number of centroids K is large. Referring back to Figure 5.8(f), the running times for the left-most point on the KMEANS line takes 3.6h with $K = 5$, while the right-most point takes 8.8h with $K = 10$. Thus our proposed LYRESPLIT is much more scalable than AGGLO and KMEANS. Even though KMEANS is closer to LYRESPLIT in performance (as seen in the previous experiments), it is impossible to use in practice.

5.5.3 Benefits of Partitioning

With only a 2× increase on the storage, we can achieve a substantial 3×, 10× and 21× reduction on checkout time for SCI_1M, SCI_5M, and SCI_10M, and 3×, 7× and 9× reduction for CUR_1M, CUR_5M, and CUR_10M respectively.

Figure 5.13: Summary of Checkout Time Comparison with and without Partitioning.

We now study the impact of partitioning and demonstrate that with a relatively small increase in storage, the checkout time can be substantially reduced. We conduct two sets of experiments with the storage threshold as $\gamma = 1.5 \times |R|$ and $\gamma = 2 \times |R|$ respectively, and compare the average checkout time with and without partitioning. Figure 5.14 and 5.15 illustrate the comparison on checkout time and storage size for SCI_* and CUR_* respectively. Each collection of bars in Figure 5.14 and Figure 5.15 corresponds to one dataset. Consider SCI_5M in Figure 5.14 as an example: the checkout time without partitioning is 16.6s while the storage size is 2.04GB; when the storage threshold is set to be $\gamma = 2 \times |R|$, the checkout time after partitioning is 1.71s and the storage size is 3.97GB. As illustrated in Figure 5.14, with only 2× increase in the storage size, we can achieve 3× reduction on SCI_1M, 10× reduction on SCI_5M, and 21× reduction on SCI_10M for the average checkout time compared to that without partitioning. Thus, with partitioning, we can eliminate the time for accessing irrelevant records. Consequently, the checkout time remains small even for large datasets.

The results shown in Figure 5.15 are similar to those in Figure 5.14: with 2× increase on the storage size, we can achieve 3× reduction on CUR_1M, 7× reduction on CUR_5M, and 9× reduction on CUR_10M for average checkout time compared to that without partitioning. However, the reduction in Figure 5.15a is smaller than that in Figure 5.14a. The reason is the following. We can see that the checkout time without partitioning is similar for SCI and CUR datasets, but the checkout time after partitioning for CUR dataset is greater than the corresponding SCI dataset. This is because the average number of records in each version, i.e., $\frac{|E|}{|V|}$, in CUR is around 3 to 4 times greater than that in the corresponding SCI, as depicted in Table 5.2. Recall that $\frac{|E|}{|V|}$ is the minimal checkout cost $\mathcal{C}_{avg}$ after partitioning as stated in Observation 5.1. Thus, the smallest possible *checkout* time for CUR, which is where the blue lines with triangle markers (corresponding to LYRESPLIT) in Figure 5.8(d)(e)(f) converges to, is typically larger than that for the corresponding SCI in Figure 5.8(a)(b)(c). Overall, as demonstrated in Figure 5.14 and 5.15, with a small increase in the storage size, we can reduce the average *checkout* time to within a few seconds even when the number of records in a CVD increases dramatically. Referring back to our motivating experiment in Figure 4.1(c), we claim that with partitioning the checkout time using split-by-rlist is comparable to that by a-table-per-version.

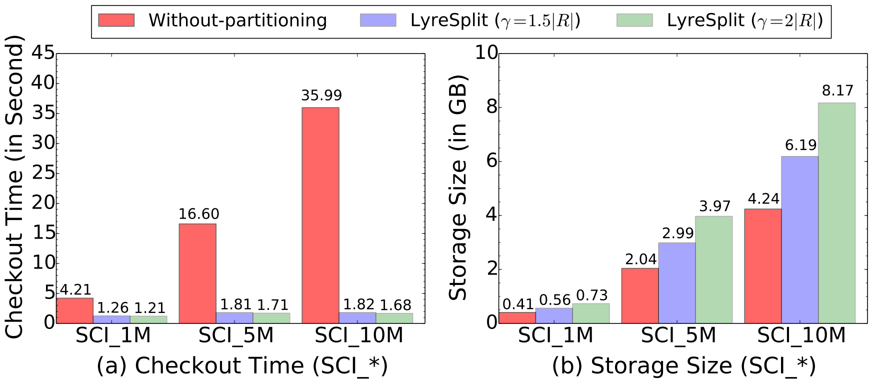


Figure 5.14: Comparison With and Without Partitioning

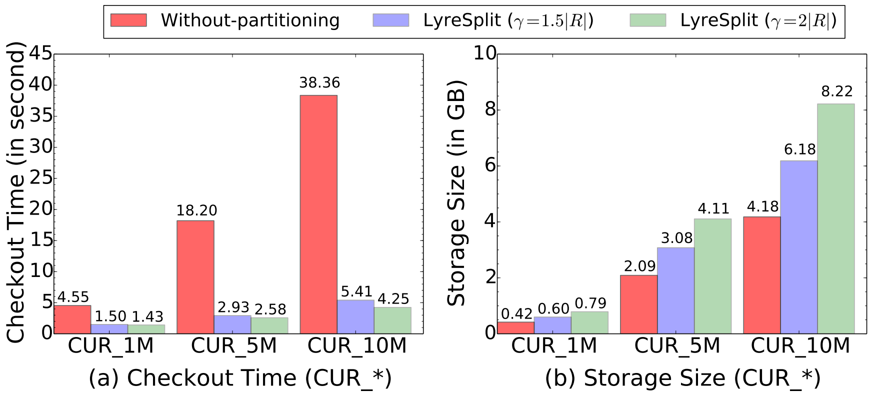


Figure 5.15: Comparison With and Without Partitioning

5.5.4 Maintenance and Migration

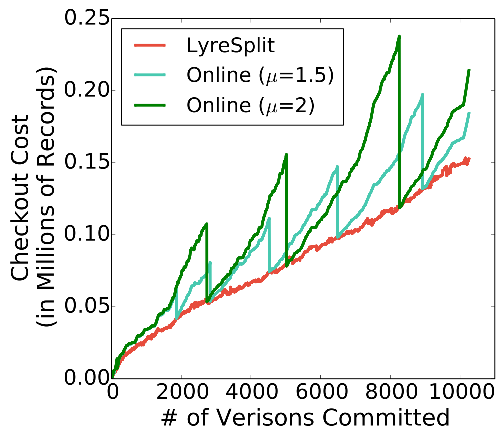
We now evaluate the performance of OrpheusDB's partitioning optimizer over the course of an extended period with many versions being committed to the system. We employ our SCI_10M dataset, which contains the largest number of versions (i.e. $10k$). Here, the versions are streaming in continuously; as each version commits, we perform online maintenance based on the mechanism described in Section 5.4. When $\frac{\mathcal{C}_{avg}}{\mathcal{C}^*_{avg}}$ reaches the tolerance factor μ, the migration engine is automatically invoked, and starts to perform the migration of data from the old partitions to the new ones identified by LyreSplit. We first examine how our online maintenance performs, and how frequently migration is invoked. Next, we test the latency of our proposed migration approach. The storage threshold is set to be $\gamma = 1.5|R|$ and $\gamma = 2|R|$ respectively.

Online Maintenance.

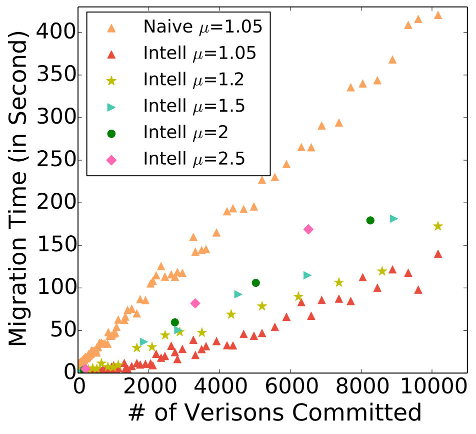
With our proposed online maintenance mechanism, the checkout cost $\mathcal{C}_{avg}$ diverges slowly from the best checkout cost $\mathcal{C}^*_{avg}$ identified by LyreSplit. When $\mu = 1.5$, our migration engine is triggered only 7 and 4 times across a total of 10,000 committed versions when $\gamma = 1.5|R|$ and $\gamma = 2|R|$ respectively.

Figure 5.16: Summary of Online Maintenance Compared to LyreSplit.

As shown in Figure 5.17(a) and 5.19(a), the red line depicts the best checkout cost $\mathcal{C}^*_{avg}$ identified by LyreSplit (note that LyreSplit is lightweight and can be run very quickly after every commit), while the blue and green lines illustrate the current checkout cost $\mathcal{C}_{avg}$ with tolerance factor $\mu = 1.5$ and $\mu = 2$, respectively. We can see that with online maintenance, the checkout cost $\mathcal{C}_{avg}$ (blue and green lines) starts to diverge from $\mathcal{C}^*_{avg}$ (red line). When $\frac{\mathcal{C}_{avg}}{\mathcal{C}^*_{avg}}$ exceeds the tolerance factor μ, the migration engine is invoked, and the blue and green lines jump back to the red line once migration is complete. With the increase of μ, the frequency of triggering migration decreases. As depicted in Figure 5.17(a), when $\mu = 1.5$, migration is triggered 7 times, while it is only triggered 3 times when $\mu = 2$, across a total of 10000 versions committed. Thus, our proposed online maintenance performs well, diverging slowly from LyreSplit. This can be explained by the same intuition shared by the online maintenance scheme and LyreSplit.



(a) Online Maintenance



(b) Migration Time

Figure 5.17: Online Partitioning and Migration ($\gamma = 1.5|R|$)

Migration Time.

Figure 5.17(b) and 5.19(b) depict the migration time when the *migration engine* is invoked. Figure 5.17(b) is in correspondence with Figure 5.17(a) sharing the same x-axis. For instance, When $\mu = 1.05$, the migration time with our proposed method is on average $\frac{1}{10}$ of that with naive approach of rebuilding the partitions from scratch when $\gamma = 1.5|R|$ and $\gamma = 2|R|$. As μ decreases, the migration time with our proposed method decreases.

Figure 5.18: Summary of Comparison of Running Time of Migration

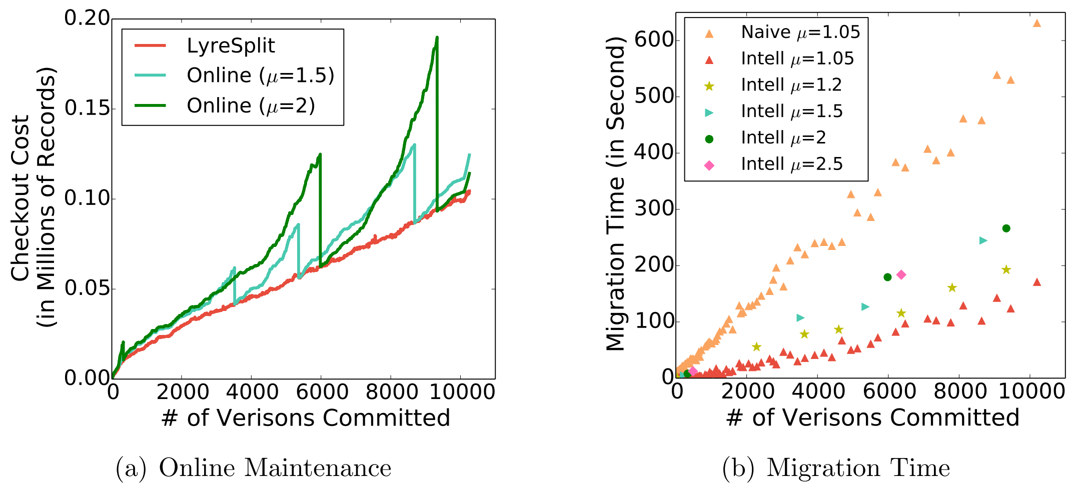


Figure 5.19: Online Partitioning and Migration ($\gamma = 2|R|$)

with $\mu = 2$, when the 5024th version commits, the *migration engine* is invoked as shown by the green line in Figure 5.17(a). Correspondingly, the migration takes place, and we record the migration time with the green circle ($\mu = 2$) in Figure 5.17(b). Hence, there are three green circles in Figure 5.17(b), corresponding to the three migrations in Figure 5.17(a). Same are Figure 5.19(a) and Figure 5.19(b).

We now compare our intelligent migration approach from Section 5.4, denoted *intell*, with the naive approach of rebuilding partitions from scratch, denoted *naive*. The points with upward triangles in Figure 5.17(b) all have $\mu = 1.05$, with the red points representing *intell*, and the brown representing *naive*: we see that *intell* takes at most $\frac{1}{3}$, and on average $\frac{1}{10}$ of the time of *naive*. For the sake of clarity, we omit the migration times for different μ using *naive*, since they roughly fall on the same line as that of $\mu = 1.05$. Next, consider the migration time with different μ using *intell*. Overall, as μ decreases, the migration time decreases. To see this, one can connect the points corresponding to each μ (denoted using different markers) to form lines in Figure 5.17(b). When μ is smaller, migration takes place more frequently, due to which the new partitioning scheme identified by LyreSplit is more similar to the current one, and hence fewer modifications need to be performed. Essentially, we are amortizing the migration cost across multiple migrations. Similar results can be found in Figure 5.19 when $\gamma = 2|R|$.

5.5.5 Additional Experiments

Next, we will show some additional experiments we have conducted.

Verification of Checkout Cost Model

In the following, we both analyze and experimentally evaluate the checkout cost model proposed in Section 5.1. We demonstrate that the checkout cost $\mathcal{C}_i$ of a version v_i grows linearly with the number of records in the partition $\mathcal{P}_k$ that contains v_i, i.e., $\mathcal{C}_i \propto |\mathcal{R}_k|$.

As depicted in the SQL query in Table 4.1 in Chapter 4, the checkout cost is impacted by the cost of two operations: (a) obtaining the list of records *rlist* associated with v_i; (b) joining data table with *rlist* to get all valid records. The cost from part (a) is a constant regardless of the partitioning scheme we use, and it is small since *rlist* can be obtained efficiently using a physical primary key index on *vid*. Thus, we focus our analysis on the cost from part (b).

We focus on three important types of join operations: hash-join, merge-join and nested-loop-join. In the following, we evaluate the checkout cost model for all these join algorithms and provide a detailed analysis. We vary the number of records in the checkout version ($|rlist|$) and the number of records in its corresponding partition ($|\mathcal{R}_k|$) in our experiments. The parameter $|\mathcal{R}_k|$ is varied from 1K to 30M and $|rlist|$ is varied from 1K to 1M, where *rlist* is a sorted list of randomly sampled *rid*s from $\mathcal{R}_k$. In addition, we have two different physical layouts for the data table, one clustered on *rid* and another clustered on its original relation primary key (PK)— <protein1, protein2> in Figure 3.2. For each of the three join types, we compare the checkout time (in seconds) vs. the estimated checkout cost (in millions of records). Note that we build an *index* on *rid* in the data table, otherwise, the nested-loop-join would be very time-consuming since each outer loop requires a full scan on the inner table. The results are presented in Figure 5.7, where each line is plotted with a fixed $|rlist|$ (1K, 10K, 100K, and 1M respectively) and varying $|\mathcal{R}_k|$. We now describe the performance of the individual join algorithms below.

Hash-join. No matter which physical layout is used, the query plan for a hash-join based approach is to first build a hash table for *rlist* and then sequentially scan the data table with each record probing the hash table. By benefiting from the optimized implementation of the hash-join in PostgreSQL, the cost of probing each *rid* in the hash table is almost a constant. With fixed $|rlist|$, the building phase in hash-join is the same, while the running time in the probing phase is proportional to $|\mathcal{R}_k|$. Hence, as depicted in Figure 5.7(a) and (d), with a fixed $|rlist|$, the running time increases linearly with the growth of $|\mathcal{R}_k|$.

Merge-join. When the data table is clustered on *rid*, the query plan for a merge-join based approach is to first sort *rlist* obtained from the versioning table, then conduct an index scan using *rid* index on the data table and merge with the *rlist* from the versioning table. First, since *rlist* from the versioning table has already been sorted, quicksort can immediately terminate after the first iteration. Second, since the data table is physically clustered on *rid*, an index scan on *rid* is equivalent to a sequential scan in the data table. Thus, with fixed $|rlist|$, the running time grows linearly with the increase of $|\mathcal{R}_k|$, which is experimentally verified in Figure 5.7(b).

On the other hand, when the data table is clustered on the relation primary key, PostgreSQL gives different query plans for different $|\mathcal{R}_k|$. When $|\mathcal{R}_k|$ is equal to 4M, 6M and 8M, the query plan is the same as the above—sort *rlist*, conduct an index scan on *rid* and merge with *rlist*. However, since the physical layout is no longer clustered on *rid*, having an index scan on *rid* is equivalent to performing random access $|\mathcal{R}_k|$ times into the data table, which is very time-consuming as illustrated in Figure 5.7(e). For other $|\mathcal{R}_k|$ except 4M, 6M and 8M, the query plan is to first sort *rlist* from the versioning table, conduct a sequantial scan on the data table, sort the *rid*s, and then finally merge *rid*s with *rlist*. Thus, with fixed $|rlist|$, the running time is proportional to $|\mathcal{R}_k|$, but greater than the hash-join based approach due to the overhead of sorting, as shown by the last five points in Figure 5.7(e).

Index-nested-loop-join. No matter which physical layout is used, the query plan for an index-nested-loop-join based approach is to perform a random I/O in the data table for each *rid* in *rlist* from the versioning table. Consider the scenario where $|rlist|$ is fixed and the data table is clustered on *rid*. When $|rlist|$ is much smaller than $|\mathcal{R}_k|$, the running time is almost the same since each random I/O is a constant and $|rlist|$ is fixed. This is also verified by the right portion of the blue line ($|rlist|$=1K) and red line ($|rlist|$=10K) in Figure 5.7(c). However, when $|rlist|$ is comparable to $|\mathcal{R}_k|$, the running time is proportional to $|\mathcal{R}_k|$ as illustrated in the green ($|rlist|$=1M) and yellow ($|rlist|$=100K) line in Figure 5.7(c). This is because hundreds of thousands of random I/Os are eventually reduced to a full sequential scan on the data table when $\mathcal{R}_k$ is clustered on *rid*. Returning to the checkout cost model, since partitioning algorithms tend to group similar versions together, after partitioning, $|rlist|$ is very likely to be comparable to $|\mathcal{R}_k|$ and thus the checkout time can be quantified by $|\mathcal{R}_k|$. Furthermore, the yellow line ($|rlist|$=100K) in Figure 5.7(c) indicates that even when $\frac{|rlist|}{|\mathcal{R}_k|} = \frac{1}{300}$, random I/Os will still be reduced to a sequential scan, consequently the running time grows linearly with $|\mathcal{R}_k|$.

However, note that when the data table is not clustered on *rid*, each random I/O takes almost constant time as shown in Figure 5.7(f). Since random I/O is more time-consuming than sequential I/O, the index-nested-loop-join performs much worse than hash-join as shown in Figure 5.7(d) and (f).

Overall Takeaways. When the data table is clustered on *rid*, the checkout cost can be quantified by $|\mathcal{R}_k|$ for hash-join and merge-join based approaches; while for index-nested-loop-join, the checkout cost can also be quantified by $|\mathcal{R}_k|$ when $\frac{|rlist|}{|\mathcal{R}_k|} \geq \frac{1}{300}$, which is typically the case in the partitions after partitioning especially for latest versions. On the other hand, when the data table is not clustered on *rid*, the checkout cost for the hash-join based approach can still be quantified by $|\mathcal{R}_k|$, while the merge-join and the index-nested-loop-join based approaches perform worse than that of hash-join for most cases. Overall, a hash-join based approach has the following advantages: *(a)* the checkout time using hash-join does not rely on any index on *rid*; *(b)* hash-join based approach has good and stable performance regardless of the physical layout; *(c)* the checkout cost using hash-join is easy to model, laying foundation for further optimization on *checkout* time. Thus, throughout this chapter we have focused on hash-join for the checkout command and model the checkout cost $\mathcal{C}_i$ as linear in the number of records $|\mathcal{R}_k|$ in the partition that contains v_i.

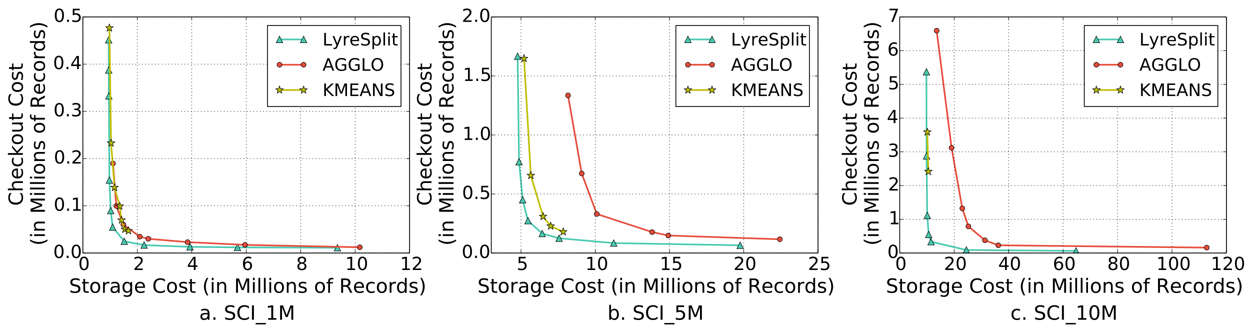


Figure 5.20: Estimated total Storage Cost vs. Estimated Checkout Cost (SCI_*)

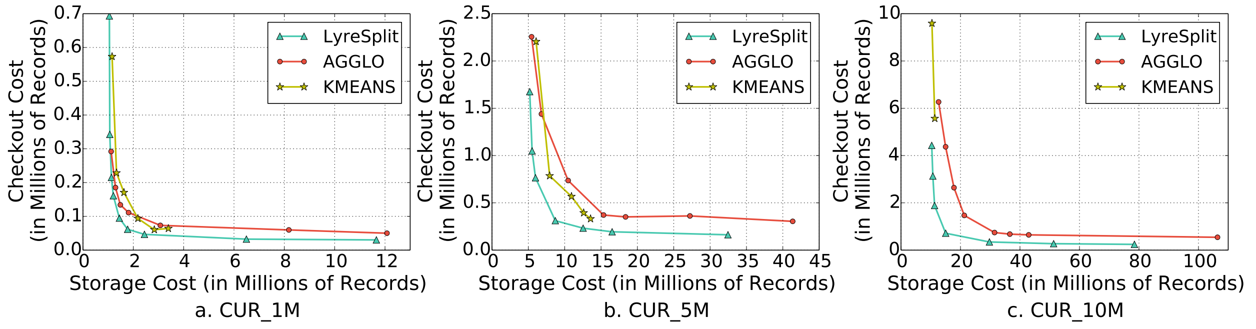


Figure 5.21: Estimated Storage Cost vs. Estimated Checkout Cost (CUR_*)

5.6 ADDITIONAL RELATED WORK

In addition to the related work in Chapter 2, there has been a lot of work on graph partitioning [62, 63, 64, 65], with applications ranging from distributed systems and parallel computing, to search engine indexing. The state-of-the-art in this space is NScale [61], which proposes algorithms to pack subgraphs into the minimum number of partitions while keeping the computation load balanced across partitions. In our setting, the versions are related to each other in very specific ways; and by exploiting these properties, our algorithms are able to beat the NScale ones in terms of performance, while also providing a $10^3\times$ speedup. Kumar et al. [66] study workload-aware graph partitioning by performing balanced k-way cuts on the tuple-query hypergraph for data placement and replication on the cloud; in their context, however, queries are allowed to touch multiple partitions.

Now we have introduced OrpheusDB, next we will relax the assumptions made in OrpheusDB, one at a time, as steps towards general-purpose data versioning. In particular, we will focus on a generalized query language in the next chapter.

CHAPTER 6: GENERALIZED QUERY LANGUAGE

In OrpheusDB, we implicitly made the assumption that a SQL-like language is the best fit for data querying and version reasoning, due to the fact that OrpheusDB is built on top of relational databases. However, SQL is overall ill-suited to traversing a (version) graph structure for analysis—one of our key requirements, and further, it has a cumbersome aggregation syntax that results in unwieldy queries when comparing across versions [67]. In this chapter, we present an initial design of our generalized query language, called VQuel, that aims to support such unified querying over both provenance and versioning information, as well as the intermediate and final results of analyses. VQuel is a *version-aware query language*, capable of querying dataset versions, dataset provenance (e.g., which datasets a given dataset was derived from), and record-level provenance (if available).

6.1 MOTIVATING EXAMPLE

To illustrate the features of our query language, we describe an example collaborative data analysis scenario, and then present examples of queries we would like to issue:

Example 6.1. Genome assembly *of a whole genome sequence dataset is a complex task — apart from huge computational demands, it is not always known a priori which tools and settings will work best on the available sequence data for an organism [68]. The process typically involves testing multiple tools, parameters and approaches to produce the best possible assembly for downstream analysis. The assemblies are evaluated on a host of metrics (e.g., the N50 statistic) and the choice of which assembly is the best one is also not always clear. One potential sequence of steps might be: Sequenced reads (FastQ files) $\rightarrow$ Error correction tools (Quake, Sickle, etc.) $\rightarrow$ Input analysis, k-mer calculation (KmerGenie) $\rightarrow$ Assembly tool (SOAPdenovo, ABySS) $\rightarrow$ Assembly analysis and selection (QUAST).*

A group of researchers may collaboratively try to analyze this data in various ways, building upon the work done by the others in the team, but also trying out different algorithms or tools. New data is also likely to be ingested at various points, either as updates/corrections to the existing data or as results of additional experiments. As one can imagine, the ad hoc nature of this process and the desire not to lose any intermediate synthesized result means that the researchers will be left with a large number of datasets and analyses, with large overlaps between them and complex derivational dependencies. Similar collaborative workflows can be seen in many other data science application domains.

Before moving forward, we describe our notion of the term "version". For us, a version consists of one or more datasets that are semantically grouped together (in some sense, it is equivalent to the notion of a "commit" in `git/svn`). A version, identified by an ID, is immutable and any update to a version conceptually results in a new version with a different version ID (note that the physical data structures are not necessarily immutable and we would typically not want to copy all the data over, but rather maintain differences [69]). New versions can also be created through the application of transformation programs to one or more existing versions. The version-level provenance that captures these processes is maintained as a "version graph", that we discuss in more detail later.

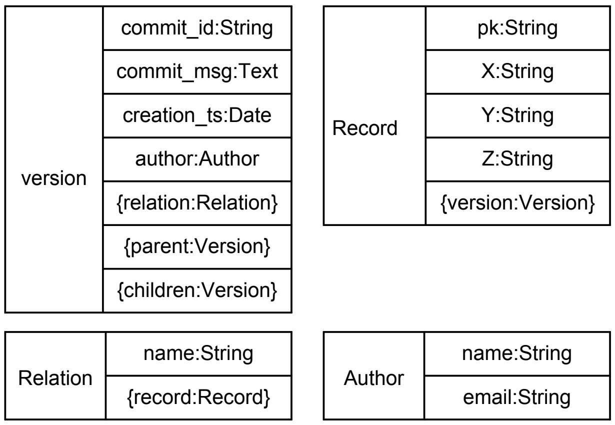
There is a wide range of queries that may be of interest in such a setting as above. Simple queries include: (a) identifying versions based on the metadata information (e.g., authors); (b) identifying versions that were derived (directly or through a chain of derivations) from a specific outdated version; and (c) finding versions that differ from their predecessor version by a large number of records. More complex queries include: (d) finding versions where the data within satisfies certain aggregation conditions; (e) finding the intersection of a set of versions (representing, e.g., the final synthesized results of different pipelines); and (f) finding versions that contain any records derived from a specific record in a version. We note here that a key challenge that we face is identifying a useful set of queries/tasks and abstracting language features from them, and we hope to engage with a wide variety of users to accomplish that.

These examples illustrate some of the key requirements for a query language, namely the ability to:

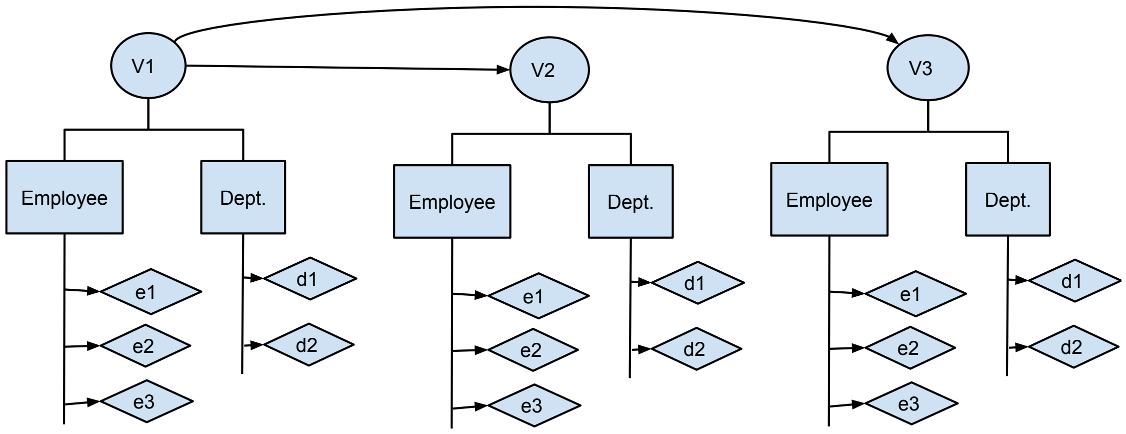
- Traverse the version graph (i.e., version-level provenance information) and query the metadata associated with the versions and the derivation/update edges.
- Compare several versions to each other in a flexible manner.
- Run declarative queries over data contained in a version, to the extent allowable by the structure in the data.
- Query the tuple-level provenance information, when available, in conjunction with the version-level provenance information.

6.2 PRELIMINARIES

As introduced in Chapter 3, OrpheusDB enables users to keep track of datasets and their versions, by means of a version graph that encodes derivation relationships among them. As we discussed earlier, a version refers to a collection of files or relations that are semantically grouped together. Figure 6.1(b) shows an example of a few versions along with the *version graph* connecting them.



(a)



(b)

Figure 6.1: (a) Conceptual Data model: the notation "{T}" denotes a set of values of T; fields in the Records entity can be conceptually thought of as a union of all fields across records; other fields and entities (for instance Authors) are not shown to keep the discussion brief; for each entity, entries in the left and right column denote the attribute name and type respectively. (b) An example version graph where circles denote versions; version V1 has two Relations, `Employee` and `Department`, each having a set of records, {E1, E2, E3} and {D1, D2} respectively; version V2 adds new records to both the `Employee` and `Department` relations and also adds a new File, `Forms.csv`. Edge annotations (not shown) are used to capture information about the derivation process itself, including references to transformation programs or scripts if needed.

Figure 6.1(a) shows a portion of the conceptual data model that we use to write queries against. The data model consists of four essential tables: `Version`, `Relation`, `File`, and `Record`. Additional tables like `Column` and `Author` are required but not essential for the purpose of this discussion. The difference between `Relation` and `File` is that a relation has a fixed schema for all its records (recorded in the `Column` table) while a file has no such requirement. To that effect, we denote the records in a relation as tuples.

The `Version` table maintains the information about the different versions in the database, including the "commit_id" (unique across the versions), and various attributes capturing metadata about the version, such as the creation time and author, as well as "commit_msg" and "creation_ts", representing the commit message and creation time respectively. There are four set-valued attributes called "relations", "files", "parents" and "children", recording the relations and files contained in the version, and the direct parents and children in version graph respectively. The last two refer back to the `Version` table, whereas the first two refer to the `Relation` and `File` tables respectively. A tuple in the `Relation` table, in turn, records the information for a relation including its schema; we view the tuples in the relation as a set-valued attribute of this table itself — this allows us to locate a relation and then query on the data inside it as we will see in the next section. The `Files` table is analogous, but records information appropriate for an unstructured file. Note that neither of these tables has a primary key but rather the attributes "name" and "full_path" serve as *discriminators*, and must be combined with the version "id" to construct primary keys. The "changed" attribute is a derived (redundant) attribute that indicates whether the relation/file changed from the parent version, and is very useful for version-oriented queries.

Finally, `Record` is a virtual table that can be conceptually thought of as a union of all tuples and records in all relations and files across the versions. The one exception is the "parents" and "children" attributes, which refer back to the `Record` table and can be used to refer to fine-grained provenance information within queries. This table is never directly referenced in the queries, but is depicted here for completeness. The provenance information must "obey" the version graph, e.g., in the example shown, records in version V2 can only have records in version V1 as parents.

We note here that this data model is a high-level conceptual one mainly intended for ease of querying and aims to maximize data independence. For instance, although the fine-grained provenance information is conceptually maintained in the `Record` table here and can be queried using the "parents" and "children" attributes, the implementation could maintain that information at schema-level wherever feasible to minimize the storage requirements.

6.3 OVERVIEW OF VQUEL

VQuel is largely a generalization of the Quel language [70] (tuple variables which enable iterating over objects at any level of the complex nested data model as described in

Section 6.3.1. while also introducing certain syntactic conveniences that Quel does not possess), and combines features from GEM [71] (tuple-reference attributes as described in Section 6.3.1) and path-based query languages. This means that VQuel is a *full-fledged relational query language*, and in addition, it enables the seamless querying of the nested data model described in the previous section, encoding versioning derivation relationships, as well as versioning metadata.

VQuel will be illustrated using example queries on the repository shown in Figure 6.1(b), with certain deviations introduced when necessary. We will introduce the constructs in VQuel incrementally, starting from those present in Quel to the new ones designed for a data versioning management setting. For ease of understanding, we first present a version that is clear and easy to understand, but results in longer queries. In Section 6.3.2 we describe additional constructs to make the queries concise.

6.3.1 Examples

We begin with some simple VQuel queries. Most of these queries are also straightforward to write in SQL; the queries that cannot be written in SQL easily begin in Section 6.3.3. Here, we gradually introduce the constructs of VQuel as a prelude to the more complex queries combining versioning and data.

Query 6.1. *Who is the author of version with id "v01"?*

```
range of V is Version
retrieve V.author.name
where V.id = ||v01||
```

A VQuel query has two elements: iterator setup (range above) and retrieval (retrieve above) of objects satisfying a predicate (where above). Iterators in VQuel are similar to tuple variables in Quel, but more powerful, in the sense that they can iterate over objects at any level of our hierarchical data model. They are declared with a statement of the form:

```
range of <iterator-variable> is <set>
```

The retrieve statement is used to select the object properties, and is of the form:

```
retrieve [into <iterator >][unique]<target-list>
[where <predicate>]
[sort by <attribute> [asc/desc] {, <attribute> [asc/desc]}]
```

The retrieve statement fetches all the object attributes specified in the `target-list` for those objects satisfying the where clause.

Query 6.2. *What commits did Alice make after January 01, 2015?*

```
range of V is Version
retrieve V.all
where V.author.name = ||Alice|| and V.creation_ts >= ||01/01/2015||
```

In Queries 6.1 and 6.2, note the use of GEM-style tuple-reference attributes, namely `V.author`, and the keyword `all` from Quel. The comparators =, !=, <, <=, > and >= are allowed in comparisons, and the logical connectives `and`, `or`, and `not` can be used to combine comparisons.

Multiple iterators can be set up before a retrieval statement, and their respective sets can be defined as a function of previously declared iterators. The next example illustrates this idea. The first range clause sets up an iterator V over all the versions. The second range clause defines an iterator over all relations inside a version.

Query 6.3. *List the commit timestamps of versions that contain the Employee relation.*

```
range of V is Version
range of R is V.Relations
retrieve V.commit_ts
where R.name = ||Employee||
```

Query 6.4. *Show the commit history of the Employee relation in reverse chronological order.*

```
range of V is Version
range of R is V.Relations
retrieve V.creation_ts , V.author.name, V.commit_message
where R.name = ||Employee|| and R.changed = true
sort by V.creation_ts desc
```

Similarly, we can set up a range clause over tuples inside a relation. Analogous to a relational database, the user needs to be familiar with the schema to be able to pose such a query.

Query 6.5. *Show the history of the tuple with employee id "e01" from Employee relation.*

```
range of V is Version
range of R is V.Relations
range of E is R.Tuples
retrieve E.all , V.commit_id , V.creation_ts
where E.employee_id = ||e01|| and R.name = ||Employee||
sort by V.creation_ts
```

6.3.2 Syntactic Sweetenings

In this section, we introduce some shorthand constructs to keep the size of the queries small. These constructs are meant only for brevity, and each of them can be mapped to an equivalent query without using shorthands.

The first one is analogous to a *filter* operation over a set declaration: we can use predicates in the set declaration block of the `range` statement. For instance, in the following example, both queries iterate over the same set of versions. Note that the `retrieve into` clause in (b1) sets up a new iterator V over all the versions satisfying constraints in `where` clause.

```
(a1) range of V is Version(id = ||v01||)

(b1) range of T is Version
     retrieve into V (T.all)
     where T.id = ||v01||
```

The next example shows the principle in action on a query that would otherwise become quite long. Again, (a2) and (b2) below show identical queries written using the short notation (a) and their equivalent form (b).

Query 6.6. *Find all Employee tuples in version "v01" that are different in version "v02".*

```
(a2) range of E1 is Version(id = ||v01||)
          .Relations(name = ||Employee||).Tuples
     range of E2 is Version(id = ||v02||)
                .Relations(name = ||Employee||).Tuples
     retrieve E1.all
     where E1.employee_id = E2.employee_id and E1.all != E2.all

(b2) range of V1 is Version
     range of R1 is V1.Relations
     range of E1 is R1.Tuples
     range of V2 is Version
     range of R2 is V2.Relations
     range of E2 is R2.Tuples
     retrieve E1.all
     where V1.id=||v01|| and R1.name=||Employee||
     and V2.id=||v02|| and R2.name=||Employee||
     and E1.employee_id = E2.employee_id and E1.all != E2.all
```

6.3.3 Aggregate operators

The aggregate functions sum, `avg`, `count`, any, min and max are also provided in VQuel. Any expression involving components of iterated entity attributes, constants and arithmetic symbols can be used as the argument of these functions. Due to the nested nature of iterators, we introduce the `_all` version of these operators, i.e. `count_all`, `sum_all`, etc. The general syntax of an aggregate expression is:

```
agg_op([<agg-attribute>/<iterator-variable>]
     [group by <grouping-attributes>] [where <predicate>])
```

This evaluates the `agg_op` on each group of `<agg-attribute>` of objects that satisfy the `<predicate>`. We see two examples next.

Query 6.7. *For each version, count the number of relations inside it.*

```
range of V is Version
range of R is V.Relations
retrieve V.id, count(R)
```

Query 6.8. *Find all versions containing precisely 100 Employees with last name "Smith".*

```
range of V is Version
range of E is V.Relations(name = ||Employee||).Tuples
retrieve V.commit_id
where count(E.employee_id where E.last_name = ||Smith||) = 100
```

In both queries above, the aggregation is performed only over objects at the *innermost* level of an iterator expression. In query 6.7, R is an iterator over relations inside a version V, and `count` iterates only over the innermost level of this iterator hierarchy, that is, R. Similarly, in query 6.8, the `count` expression only iterates over the tuples inside a relation inside a version.

Notice that the latter query is not very easy to express in vanilla SQL: there is no easy way to use SQL to retrieve version numbers, which in a traditional non-versioned context would either be considered as schema-level information, or involve multiple joins depending on the level of normalization of the schema. VQuel, on the other hand, allows us to set up the nested iterators that makes such queries very easy to express.

The next two examples show the usage of `count_all` operator. The difference from the `count` operator is that all the "parent" iterators are evaluated, instead of only the innermost iterator, to compute the value of the aggregate. Another way to reason about this behavior is that count has an implicit grouping list of attributes in its by clause: query 6.9 is identical to query 6.8.

Query 6.9. *Find all versions containing precisely 100 employees with last name "Smith".*

```
range of V is Version
range of R is V.Relations(name = ||Employee||)
range of E is R.Tuples
retrieve V.commit_id
where count_all(E.employee_id group by R, V
          where E.last_name = ||Smith||) = 100
```

Aggregates having a group by clause can also be used in the predicate to restrict the results of the query. In query 6.9, the result of count_all for each group is compared against 100. Query 6.10 gives another example.

Query 6.10. *Find all versions containing precisely 100 tuples in all relations put together inside a version.*

```
range of V is Version
range of R is V.Relations
range of T is R.Tuples
retrieve V.all
where count_all(T group by V) = 100
```

The next few examples show how we can use aggregate operators across a set of versions to answer a variety of questions about the data.

Query 6.11. *Among a group of versions, find the version containing most tuples that satisfy a predicate. For instance, which version contains the most number of employees above age 50?*

```
range of V is Version
range of E is V.Relations(name = ||Employee||).Tuples
retrieve into T (V.id as id, count(E.id where E.age > 50) as c)
retrieve T.id
where T.c = max(T.c)
```

Up until now, for an iterator, we have been exploring "down" the hierarchy. We also provide appropriate functions, depending on the type of iterator, to refer to values of entities "up" in the hierarchy. In the next query, Version(T) is used to refer to the version attributes of tuples in T.

Query 6.12. *Which versions are such that the natural join between relations S and T has more than 100 tuples?*

```
range of V is Version
range of S is V.Relations(name = ||S||).Tuples
range of T is V.Relations(name = ||T||).Tuples
retrieve into Q(V.id as id,
    count_all(S.id group by V
    where S.id = T.s_id and Version(S).id = Version(T).id) as c)
retrieve Q.id
where Q.c >= 100
```

6.3.4 Version graph traversal

VQuel has three constructs aimed at traversing the version graph. Each of these operates on a version at a time, specified over an iterator.

- P(<integer>): Return the set of ancestor version of this version, until integer number of hops in the version graph. If the number of hops is not specified, we go till the first version. Duplicates are removed.
- D(<integer>): Similar to P() except that it returns the descendant/derived versions.
- N(<integer>): Similar to P() except that it returns the versions that are <integer> number of hops away.

The next few queries illustrate these constructs. Notice once again that queries of this type are not very easy to express in SQL, which does not permit the easy traversal of graphs, or specification of path queries. The constructs we introduce are reminiscent of constructs in graph traversal languages [72]; these combined with the rest of the power of VQuel enable some fairly challenging queries to be expressed rather easily.

Query 6.13. *Find all versions within 2 commits of "v01" which have less than 100 employees.*

```
range of V is Version(id = ||v01||)
range of N is V.N(2)
range of E is N.Relations(name = ||Employee||).Tuples
```

```
retrieve N.all
where count(E) < 100
```

Query 6.14. *Find all versions where the delta from the previous version is greater than 100 tuples.*

```
range of V is Version
range of P is V.P(1)
retrieve unique V.all
where abs(count(V.Relations.Tuples) − count(P.Relations.Tuples)) > 100
```

Query 6.15. *For each tuple in Employee relation as of version "v01", find the parent version where it first appeared.*

```
range of V is Version(id = ||v01||)
range of E is V.Relations(name = ||Employee||).Tuples
range of P is V.P()
range of PE is P.Relations(name = ||Employee||).Tuples
retrieve E.id, P.id
where E.employee_id = PE.employee_id and P.commit_ts = min(P.commit_ts)
```

6.3.5 Extensions to fine-grained provenance

Finally, in some cases, we may have complete transparency into the operations performed by data scientists. In such cases, we can record, reason about, and access tuple-level provenance information. Here is an example of a query that can refer to tuple-level provenance:

Query 6.16. *For tuples in version "v01" in relation S that satisfy a predicate, say value of attribute* `attr = x`*, find all parent tuples that they depend on.*

```
range of E is Version(id = ||v01||).Relations(name = ||S||).Tuples
range of P is E.parents
retrieve E.id, P.id
where E.attr = x
```

Similar queries can be used to "walk up" the derivation path of given tuples, for example, to identify the origins of specific tuples.

6.4 ADDITIONAL RELATED WORK

While there has been some work on temporal query languages [73], these languages do not apply to our setting since they assume a linear chain of versions — in our case, we could have an arbitrary branching structure of versions as is common in collaborative data analysis. Extensions have been proposed to SQL [74] to work with nested relational model which allows for relation-valued attributes; but overall SQL is ill-suited to traversing a graph structure— one of our key requirements, and further, it has a cumbersome aggregation syntax that results in unwieldy queries when comparing across versions [67]. Similarly, while there has been substantial work on query languages for provenance, ranging from adapting SQL [75], Prolog [76, 77], SPARQL [78, 79] to specialized languages such as QLP [80, 81], PQL [82], ProQL [83] ([84], [85] have additional examples), much of this work centers on well-defined workflows and tuple-based provenance rather than collaborative settings where multiple users interact through a derivation graph of versions in an ad hoc manner. Furthermore, query languages are generally tied to a particular method of recording provenance information, e.g., semiring annotations [86], COMAD [87], etc., and adapting them to other provenance data and storage models is often clunky [88]. Finally, we note that although our proposed language is different from the aforementioned ones, we might be able to build upon some of their query execution strategies (e.g., [79]) and add user-defined operators to aid in specific analysis tasks (e.g., [77]). This is, however, ongoing work and is not the focus of this chapter.

To the best of our knowledge, ours is the first query language proposal tailored for an ad hoc derivation graph of versions of structured records. Our proposal draws from constructs introduced in the historical Quel [70] and GEM [71] languages, neither of which had a temporal component.

In this chapter, we removed the SQL assumption and introduced our proposed generalized query language VQuel. In the next chapter, we will relax the structural assumption and focus on the generalized storage representation for data at varying degrees of structure.

CHAPTER 7: COMPACT STORAGE ENGINE FOR DATA VERSIONING

As discussed in Chapter 1, the relative ease of collaborative data science and analysis has led to a proliferation of many thousands or millions of *versions* of the same datasets in many scientific and commercial domains, acquired or constructed at various stages of data analysis across many users, and often over long periods of time. Managing, storing, and recreating these dataset versions is a non-trivial task. In this chapter, we study how to compactly store the versioned datasets irrespective of degree of structure, and at the same time achieve fast retrieval of versions. The fundamental challenge here is the *storage-recreation trade-off*: the more storage we use, the faster it is to recreate or retrieve versions, while the less storage we use, the slower it is to recreate or retrieve versions. In particular, we study this trade-off in a principled manner: we formulate six problems under various settings, trading off these quantities in various ways, demonstrate that most of the problems are intractable, and propose a suite of inexpensive heuristics drawing from techniques in delay-constrained scheduling, and spanning tree literature, to solve these problems. We demonstrate, via extensive experiments, that our proposed heuristics provide efficient solutions in practical dataset versioning scenarios.

The main contributions of this chapter are given as follows:

- We formally define and analyze the dataset versioning problem and consider several variations of the problem that trade off storage cost and recreation cost in different manners, under different assumptions about the differencing mechanisms and recreation costs (Section 7.2). Table 7.1 summarizes the problems and our results. We show that most of the variations of this problem are NP-Hard (Section 7.3).
- We provide two light-weight heuristics: one, when there is a constraint on average recreation cost, and one when there is a constraint on maximum recreation cost; we also show how we can adapt a prior solution for balancing minimum-spanning trees and shortest path trees for undirected graphs (Section 7.4).
- We have built a prototype system where we implement the proposed algorithms. We present an extensive experimental evaluation of these algorithms over several synthetic and real-world workloads demonstrating the effectiveness of our algorithms at handling large problem sizes (Section 7.5).

Remark 7.1. In this chapter, we do not assume any particular format of the data. Our proposed algorithm is based on delta-encoding, which is generic and can work with any data format, including structured, semi-structured, and non-structured data.

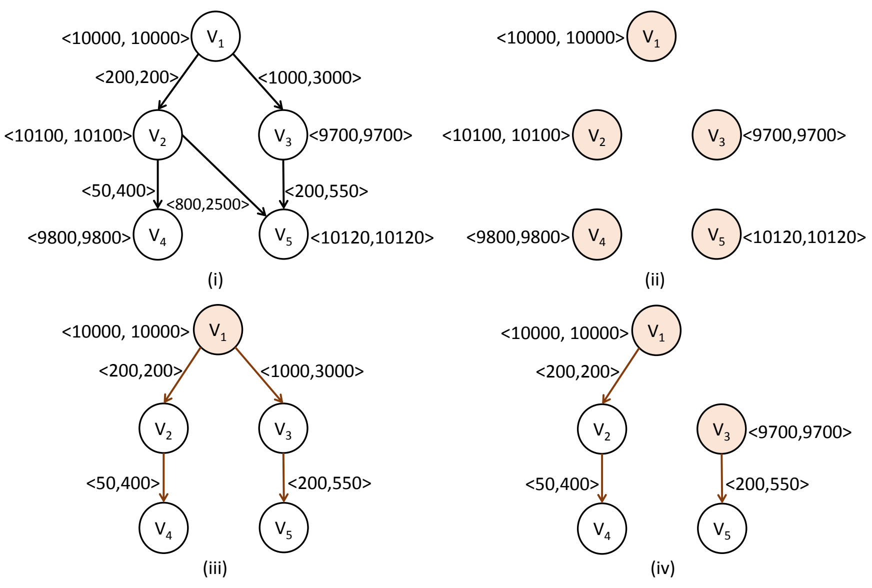


Figure 7.1: (i) A version graph over 5 datasets – annotation $\langle a, b\rangle$ indicates a storage cost of a and a recreation cost of b; (ii, iii, iv) three possible storage graphs

7.1 MOTIVATING EXAMPLE

In this chapter, we focus on the problem of trading off storage costs and recreation costs in a principled fashion. Specifically, the problem we address in this chapter is: given a collection of datasets as well as (possibly) a directed version graph connecting them, minimize the overall storage for storing the datasets and the recreation costs for retrieving them. The two goals conflict with each other — minimizing storage cost typically leads to increased recreation costs and vice versa. We illustrate this trade-off via an example.

Example 7.1. *Figure 7.1(i) displays a version graph, indicating the derivation relationships among 5 versions. Let V_1 be the original dataset. Say there are two teams collaborating on this dataset: team 1 modifies V_1 to derive V_2, while team 2 modifies V_1 to derive V_3. Then, V_2 and V_3 are merged and give V_5. As presented in Figure 7.1, V_1 is associated with $\langle 10000, 10000\rangle$, indicating that V_1's storage cost and recreation cost are both* 10000 *when stored in its entirety (we note that these two are typically measured in different units – see the second challenge below); the edge $(V_1 \rightarrow V_3)$ is annotated with $\langle 1000, 3000\rangle$, where* 1000 *is the storage cost for V_3 when stored as the modification from V_1 (we call this the* delta *of V_3 from V_1) and* 3000 *is the recreation cost for V_3 given V_1, i.e, the time taken to recreate V_3 given that V_1 has already been recreated.*

One naive solution to store these datasets would be to store all of them in their entirety (Figure 7.1 (ii)). In this case, each version can be retrieved directly but the total storage cost *is rather large, i.e.,* $10000+10100+9700+9800+10120 = 49720$*. At the other extreme, only one version is stored in its entirety while other versions are stored as modifications or deltas to that version, as shown in Figure 7.1 (iii). The total storage cost here is much smaller (*$10000 + 200 + 1000 + 50 + 200 = 11450$*), but the recreation cost is large for* V_2, V_3, V_4 *and* V_5*. For instance, the path* $\{(V_1 \to V_3 \to V_5)\}$ *needs to be accessed in order to retrieve* V_5 *and the recreation cost is* $10000 + 3000 + 550 = 13550 > 10120$*.*

Figure 7.1 (iv) shows an intermediate solution that trades off increased storage for reduced recreation costs for some version. Here we store versions V_1 *and* V_3 *in their entirety and store modifications to other versions. This solution also exhibits higher storage cost than solution (ii) but lower than (iii), and still results in significantly reduced retrieval costs for versions* V_3 *and* V_5 *over (ii).*

In this chapter, we initiate a formal study of the problem of deciding how to jointly store a collection of dataset versions with arbitrary structure, provided along with a version or derivation graph. Aside from being able to handle the scale, both in terms of dataset sizes and the number of versions, there are several other considerations that make this problem challenging.

- Different application scenarios and constraints lead to many variations on the basic theme of balancing storage and recreation cost (see Table 7.1). The variations arise both out of different ways to reconcile the conflicting optimization goals, as well as because of the variations in how the differences between versions are stored and how versions are reconstructed. For example, some mechanisms for constructing differences between versions lead to symmetric differences (either version can be recreated from the other version) — we call this the *undirected* case. The scenario with asymmetric, one-way differences is referred to as *directed* case.
- Similarly, the relationship between storage and recreation costs leads to significant variations across different settings. In some cases the recreation cost is proportional to the storage cost (e.g., if the system bottleneck lies in the I/O cost or network communication), but that may not be true when the system bottleneck is CPU computation. This is especially true for sophisticated differencing mechanisms where a compact derivation procedure might be known to generate one dataset from another.
- Another critical issue is that computing deltas for all pairs of versions is typically not feasible. Relying purely on the version graph may not be sufficient and significant redundancies across datasets may be missed.
- Further, in many cases, we may have information about relative *access frequencies* indicating the relative likelihood of retrieving different datasets. Several baseline algorithms

	Storage Cost	Recreation Cost	Undirected Case, $\Delta = \Phi$	Directed Case, $\Delta = \Phi$	Directed Case, $\Delta \neq \Phi$
Problem 7.1	minimize $\{\mathcal{C}\}$	$\mathcal{R}_i < \infty, \forall i$	PTime, Minimum Spanning Tree		
Problem 7.2	$\mathcal{C} < \infty$	minimize $\{\max\{\mathcal{R}_i \mid 1 \leq i \leq n\}\}$	PTime, Shortest Path Tree		
Problem 7.3	$\mathcal{C} \leq \beta$	minimize $\{\sum_{i=1}^{n} \mathcal{R}_i\}$	NP-hard, LAST Algorithm†	NP-hard, LMG Algorithm	
Problem 7.4	$\mathcal{C} \leq \beta$	minimize $\{\max\{\mathcal{R}_i \mid 1 \leq i \leq n\}\}$		NP-hard, MP Algorithm	
Problem 7.5	minimize $\{\mathcal{C}\}$	$\sum_{i=1}^{n} \mathcal{R}_i \leq \theta$	NP-hard, LAST Algorithm†	NP-hard, LMG Algorithm	
Problem 7.6	minimize $\{\mathcal{C}\}$	$\max\{\mathcal{R}_i \mid 1 \leq i \leq n\} \leq \theta$		NP-hard, MP Algorithm	

Table 7.1: Problem Variations With Different Constraints, Objectives and Scenarios.

for solving this problem cannot be easily adapted to incorporate such access frequencies. We note that the problem described thus far is inherently "online" in that new datasets and versions are typically being created continuously and are being added to the system. In this chapter, we focus on the static, off-line version of this problem and focus on formally and completely understanding that version. We plan to address the online version of the problem in the future.

7.2 PROBLEM OVERVIEW

In this section, we first introduce essential notations and then present the various problem formulations. We then present a mapping of the basic problem to a graph-theoretic problem, and also describe an integer linear program to solve the problem optimally.

7.2.1 Essential Notations and Preliminaries

Version Graph. We let $\mathcal{V} = \{V_i\}, i = 1, \ldots, n$ be a collection of versions. The derivation relationships between versions are represented or captured in the form of a *version graph*: $\mathcal{G}(\mathcal{V}, \mathcal{E})$. A directed edge from V_i to V_j in $\mathcal{G}(\mathcal{V}, \mathcal{E})$ represents that V_j was derived from V_i (either through an update operation, or through an explicit transformation). Since branching and merging are permitted in collaborative data analytics, $\mathcal{G}$ is a DAG (directed acyclic graph) instead of a linear chain. For example, Figure 7.1 represents a version graph $\mathcal{G}$, where V_2 and V_3 are derived from V_1 separately, and then merged to form V_5.

Storage and Recreation. Given a collection of versions $\mathcal{V}$, we need to reason about the *storage cost*, i.e., the space required to store the versions, and the *recreation cost*, i.e., the time taken to recreate or retrieve the versions. For a version V_i, we can either:

- Store V_i in its entirety: in this case, we denote the storage required to record version V_i fully by $\Delta_{i,i}$. The recreation cost in this case is the time needed to retrieve this recorded version; we denote that by $\Phi_{i,i}$. A version that is stored in its entirety is said to be *materialized*.
- Store a "delta" from V_j: in this case, we do not store V_i fully; we instead store its modifications from another version V_j. For example, we could record that V_i is just V_j but with the 50th tuple deleted. We refer to the information needed to construct version V_i from version V_j as the *delta* from V_j to V_i. The algorithm giving us the delta is called a *differencing algorithm*. The storage cost for recording modifications from V_j, i.e., the size the delta, is denoted by $\Delta_{j,i}$. The recreation cost is the time needed to recreate the recorded version given that V_j has been recreated; this is denoted by $\Phi_{j,i}$.

Thus the storage and recreation costs can be represented using two matrices Δ and Φ: the entries along the diagonal represent the costs for the materialized versions, while the off-diagonal entries represent the costs for deltas. From this point forward, we focus our attention on these matrices: they capture all the relevant information about the versions for managing and retrieving them.

Delta Variants. Notice that by changing the differencing algorithm, we can produce deltas of various types:

- for text files, UNIX-style diffs, i.e., line-by-line modifications between versions, are commonly used;
- we could have a listing of a program, script, SQL query, or command that generates version V_i from V_j;
- for some types of data, an XOR between the two versions can be an appropriate delta; and
- for tabular data (e.g., relational tables), recording the differences at the cell level is yet another type of delta.

Furthermore, the deltas could be stored compressed or uncompressed. The various delta variants lead to various dimensions of problem that we will describe subsequently.

The reader may be wondering why we need to reason about two matrices Δ and Φ. In some cases, the two may be proportional to each other (e.g., if we are using uncompressed UNIX-style diffs). But in many cases, the storage cost of a delta and the recreation cost of applying that delta can be very different from each other, especially if the deltas are

$$\begin{pmatrix} 10000 & 200 & 1000 & -- & -- \\ 500 & 10100 & -- & 50 & 800 \\ -- & 1100 & 9700 & -- & 200 \\ -- & -- & -- & 9800 & 900 \\ -- & -- & -- & 800 & 10120 \end{pmatrix}$$

(i) Δ

$$\begin{pmatrix} 10000 & 200 & 3000 & -- & -- \\ 600 & 10100 & -- & 400 & 2500 \\ -- & 3200 & 9700 & -- & 550 \\ -- & -- & -- & 9800 & 2500 \\ -- & -- & -- & 2300 & 10120 \end{pmatrix}$$

(ii) Φ

Figure 7.2: Matrices corresponding to the example in Figure 1 (with additional entries revealed beyond the ones given by version graph)

stored in a compressed fashion. Furthermore, while the storage cost is more straightforward to account for in that it is proportional to the bytes required to store the deltas between versions, recreation cost is more complicated: it could depend on the network bandwidth (if versions or deltas are stored remotely), the I/O bandwidth, and the computation costs (e.g., if decompression or running of a script is needed).

Example 7.2. *Figure 7.2 shows the matrices Δ and Φ based on version graph in Figure 7.1. The annotation associated with the edge (V_i, V_j) in Figure 7.1 is essentially $\langle \Delta_{i,j}, \Phi_{i,j} \rangle$, whereas the vertex annotation for V_i is $\langle \Delta_{i,i}, \Phi_{i,i} \rangle$. If there is no edge from V_i to V_j in the version graph, we have two choices: we can either set the corresponding Δ and Φ entries to "$-$" (unknown) (as shown in the figure), or we can explicitly compute the values of those entries (by running a differencing algorithm). For instance, $\Delta_{3,2} = 1100$ and $\Phi_{3,2} = 3200$ are computed explicitly in the figure (the specific numbers reported here are fictitious and not the result of running any specific algorithm).*

Discussion. Before moving on to formally defining the basic optimization problem, we note several complications that present unique challenges in this scenario.

- *Revealing entries in the matrix:* Ideally, we would like to compute all pairwise Δ and Φ entries, so that we do not miss any significant redundancies among versions that are far from each other in the version graph. However when the number of versions, denoted n, is large, computing all those entries can be very expensive (and typically infeasible), since this means computing deltas between all pairs of versions. Thus, we must reason with incomplete Δ and Φ matrices. Given a version graph $\mathcal{G}$, one option is to restrict our deltas to correspond to actual edges in the version graph; another option is to restrict our deltas to be between "close by" versions, with the understanding that versions close to each other in the version graph are more likely to be similar. Prior work has also suggested mechanisms (e.g., based on hashing) to find versions that are close to each other [34]. We assume that some mechanism to choose which deltas to reveal is provided to us.

- *Multiple "delta" mechanisms:* Given a pair of versions (V_i, V_j), there could be many ways of maintaining a delta between them, with different $\Delta_{i,j}, \Phi_{i,j}$ costs. For example, we can store a program used to derive V_j from V_i, which could take longer to run (i.e., the recreation cost is higher) but is more compact (i.e., storage cost is lower), or explicitly store the UNIX-style diffs between the two versions, with lower recreation costs but higher storage costs. For simplicity, we pick one delta mechanism: thus the matrices Δ, Φ just have one entry per (i, j) pair. Our techniques also apply to the more general scenario with small modifications.
- *Branches:* Both branching and merging are common in collaborative analysis, making the version graph a directed acyclic graph. In this chapter, we assume each version is either stored in its entirety or stored as a delta from a single other version, even if it is derived from two different datasets. Although it may be more efficient to allow a version to be stored as a delta from two other versions in some cases, representing such a storage solution requires more complex constructs and both the problems of finding an optimal storage solution for a given problem instance and retrieving a specific version become much more complicated. We plan to further study such solutions in future.

Matrix Properties and Problem Dimensions. The storage cost matrix Δ may be symmetric or asymmetric depending on the specific differencing mechanism used for constructing deltas. For example, the XOR differencing function results in a symmetric Δ matrix since the delta from a version V_i to V_j is identical to the delta from V_j to V_i. UNIX-style diffs where line-by-line modifications are listed can either be two-way (symmetric) or one-way (asymmetric). The asymmetry may be quite large. For instance, it may be possible to represent the delta from V_i to V_j using a command like: *delete all tuples with age ¿ 60*, very compactly. However, the reverse delta from V_j to V_i is likely to be quite large, since all the tuples that were deleted from V_i would be a part of that delta. In this chapter, we consider both these scenarios. We refer to the scenario where Δ is symmetric and Δ is asymmetric as the undirected case and directed case, respectively.

A second issue is the relationship between Φ and Δ. In many scenarios, it may be reasonable to assume that Φ is proportional to Δ. This is generally true for deltas that contain detailed line-by-line or cell-by-cell differences. It is also true if the system bottleneck is network communication or I/O cost. In a large number of cases, however, it may be more appropriate to treat them as independent quantities with no overt or known relationship.

For the proportional case, we assume that the proportionality constant is 1 (i.e., $\Phi = \Delta$); the problem statements, algorithms and guarantees are unaffected by having a constant proportionality factor. The other case is denoted by $\Phi \neq \Delta$.

This leads us to identify three distinct cases with significantly diverse properties: (1) **Scenario 7.1**: Undirected case, $\Phi = \Delta$; (2) **Scenario 7.2**: Directed case, $\Phi = \Delta$; and (3) **Scenario 7.3**: Directed case, $\Phi \neq \Delta$.

Objective and Optimization Metrics. Given Δ, Φ, our goal is to find a good storage solution, i.e., we need to decide which versions to materialize and which versions to store as deltas from other versions. Let $\mathcal{P} = \{(i_1, j_1), (i_2, j_2), ...\}$ denote a storage solution. $i_k = j_k$ indicates that the version V_{i_k} is materialized (i.e., stored explicitly in its entirety), whereas a pair $(i_k, j_k), i_k \neq j_k$ indicates that we store a delta from V_{i_k} to V_{j_k}.

We require any solution we consider to be a *valid* solution, where it is possible to reconstruct any of the original versions. More formally, $\mathcal{P}$ is considered a *valid* solution if and only if for every version V_i, there exists a sequence of distinct versions $V_{l_1}, ..., V_{l_k} = V_i$ such that $(i_{l_1}, i_{l_1}), (i_{l_1}, i_{l_2}), (i_{l_2}, i_{l_3}), ..., (i_{l_{k-1}}, i_{l_k})$ are contained in $\mathcal{P}$ (in other words, there is a version V_{l_1} that can be materialized and can be used to recreate V_i through a chain of deltas).

We can now formally define the optimization goals:

- *Total Storage Cost* (denoted $\mathcal{C}$): The total storage cost for a solution $\mathcal{P}$ is simply the storage cost necessary to store all the materialized versions and the deltas: $\mathcal{C} = \sum_{(i,j) \in \mathcal{P}} \Delta_{i,j}$.
- *Recreation Cost for V_i* (denoted $\mathcal{R}_i$): Let $V_{l_1}, ..., V_{l_k} = V_i$ denote a sequence that can be used to reconstruct V_i. The cost of recreating V_i using that sequence is: $\Phi_{l_1,l_1} + \Phi_{l_1,l_2} + ... + \Phi_{l_{k-1},l_k}$. The recreation cost for V_i is the minimum of these quantities over all sequences that can be used to recreate V_i.

Problem Formulations. We now state the problem formulations that we consider in this chapter, starting with two base cases that represent two extreme points in the spectrum of possible problems.

Problem 7.1 (Minimizing Storage). *Given Δ, Φ, find a valid solution $\mathcal{P}$ such that $\mathcal{C}$ is minimized.*

Problem 7.2 (Minimizing Recreation). *Given Δ, Φ, identify a valid solution $\mathcal{P}$ such that $\forall i, R_i$ is minimized.*

The above two formulations minimize either the storage cost or the recreation cost, without worrying about the other. It may appear that the second formulation is not well-defined and we should instead aim to minimize the average recreation cost across all versions. However, the (simple) solution that minimizes average recreation cost also naturally minimizes $\mathcal{R}_i$ for each version.

In the next two formulations, we want to minimize (a) the sum of recreation costs over all versions ($\sum_i \mathcal{R}_i$), (b) the max recreation cost across all versions ($\max_i \mathcal{R}_i$), under the constraint that total storage cost $\mathcal{C}$ is smaller than some threshold β. These problems are relevant when the storage budget is limited.

Problem 7.3 (MinSum Recreation). *Given Δ, Φ and a threshold β, identify $\mathcal{P}$ such that $\mathcal{C} \leq \beta$, and $\sum_i \mathcal{R}_i$ is minimized.*

Problem 7.4 (MinMax Recreation). *Given Δ, Φ and a threshold β, identify $\mathcal{P}$ such that $\mathcal{C} \leq \beta$, and $\max_i \mathcal{R}_i$ is minimized.*

The next two formulations seek to instead minimize the total storage cost $\mathcal{C}$ given a constraint on the sum of recreation costs or max recreation cost. These problems are relevant when we want to reduce the storage cost, but must satisfy some constraints on the recreation costs.

Problem 7.5 (Minimizing Storage(Sum Recreation)). *Given Δ, Φ and a threshold θ, identify $\mathcal{P}$ such that $\sum_i \mathcal{R}_i \leq \theta$, and $\mathcal{C}$ is minimized.*

Problem 7.6 (Minimizing Storage(Max Recreation)). *Given Δ, Φ and a threshold θ, identify $\mathcal{P}$ such that $\max_i \mathcal{R}_i \leq \theta$, and $\mathcal{C}$ is minimized.*

7.2.2 Mapping to Graph Formulation

In this section, we'll map our problem into a graph problem, that will help us to adopt and modify algorithms from well-studied problems such as minimum spanning tree construction and delay-constrained scheduling. Given the matrices Δ and Φ, we can construct a directed, edge-weighted graph $G = (V, E)$ representing the relationship among different versions as follows. For each version V_i, we create a vertex V_i in G. In addition, we create a dummy vertex V_0 in G. For each V_i, we add an edge $V_0 \rightarrow V_i$, and assign its edge-weight as a tuple $\langle \Delta_{i,i}, \Phi_{i,i} \rangle$. Next, for each $\Delta_{i,j} \neq \infty$, we add an edge $V_i \rightarrow V_j$ with edge-weight $\langle \Delta_{i,j}, \Phi_{i,j} \rangle$.

The resulting graph G is similar to the original version graph, but with several important differences. An edge in the version graph indicates a derivation relationship, whereas an edge in G simply indicates that it is possible to recreate the target version using the source version and the associated edge delta (in fact, ideally G is a complete graph). Unlike the version graph, G may contain cycles, and it also contains the special dummy vertex V_0.

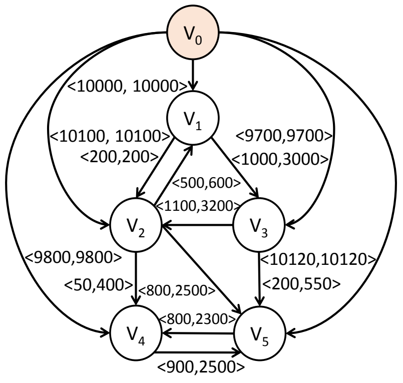


Figure 7.3: Graph G

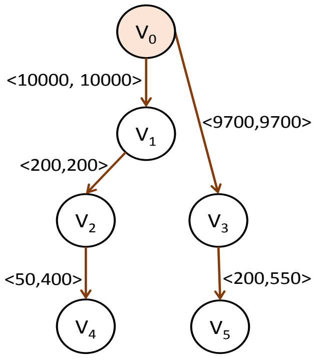


Figure 7.4: Storage Graph G_s

Additionally, in the version graph, if a version V_i has multiple in-edges, it is the result of a user/application merging changes from multiple versions into V_i. However, multiple in-edges in G capture the multiple choices that we have in recreating V_i from some other versions.

Given graph $G = (V, E)$, the goal of each of our problems is to identify a storage graph $G_s = (V_s, E_s)$, a subset of G, favorably balancing total storage cost and the recreation cost for each version. Implicitly, we will store all versions and deltas corresponding to edges in this storage graph. (We explain this in the context of the example below.) We say a storage graph G_s is *feasible* for a given problem if (a) each version can be recreated based on the information contained or stored in G_s, (b) the recreation cost or the total storage cost meets the constraint listed in each problem.

Example 7.3. *Given matrix Δ and Φ in Figure 7.2(i) and 7.2(ii), the corresponding graph G is shown in Figure 7.3. Every version is reachable from V_0. For example, edge (V_0, V_1) is weighted with $\langle \Delta_{1,1}, \Phi_{1,1} \rangle = \langle 10000, 10000 \rangle$; edge $\langle V_3, V_5 \rangle$ is weighted with $\langle \Delta_{3,5}, \Phi_{3,5} \rangle = \langle 800, 2500 \rangle$. Figure 7.4 is a feasible storage graph given G in Figure 7.3, where V_1 and V_3 are materialized (since the edges from V_0 to V_1 and V_3 are present) while V_2, V_4 and V_5 are stored as modifications from other versions.*

After mapping our problem into a graph setting, we have the following lemma.

Lemma 7.1. *The optimal storage graph $G_s = (V_s, E_s)$ for all 6 problems listed above must be a spanning tree T rooted at dummy vertex V_0 in graph G.*

Proof. Recall that a spanning tree of a graph $G(V, E)$ is a subgraph of G that (i) includes all vertices of G, (ii) is connected, i.e., every vertex is reachable from every other vertex, and (iii) has no cycles. Any G_s must satisfy (i) and (ii) in order to ensure that a version V_i can be recreated from V_0 by following the path from V_0 to V_i. Conversely, if a subgraph satisfies

(i) and (ii), it is a valid G_s according to our definition above. Regarding (iii), presence of a cycle creates redundancy in G_s. Formally, given any subgraph that satisfies (i) and (ii), we can arbitrarily delete one from each of its cycle until the subgraph is cycle free, while preserving (i) and (ii).

For Problems 7.1 and 7.2, we have the following observations. A *minimum spanning tree* is defined as a spanning tree of smallest weight, where the weight of a tree is the sum of all its edge weights. A *shortest path tree* is defined as a spanning tree where the path from root to each vertex is a shortest path between those two in the original graph: this would be simply consist of the edges that were explored in an execution of Dijkstra's shortest path algorithm.

Lemma 7.2. *The optimal storage graph G_s for Problem 7.1 is a minimum spanning tree of G rooted at V_0, considering only weights $\Delta_{i,j}$.*

Lemma 7.3. *The optimal storage graph G_s for Problem 7.2 is a shortest path tree of G rooted at V_0, considering only weights $\Phi_{i,j}$.*

7.2.3 ILP Formulation

We present an ILP formulation of the optimization problems described above. Here, we take Problem 7.6 as an example; other problems are similar. Let $x_{i,j}$ be a binary variable for each edge $(V_i, V_j) \in E$, indicating whether edge (V_i, V_j) is in the storage graph or not. Specifically, $x_{0,j} = 1$ indicates that version V_j is materialized, while $x_{i,j} = 1$ indicates that the modification from version i to version j is stored where $i \neq 0$. Let r_i be a continuous variable for each vertex $V_i \in V$, where $r_0 = 0$; r_i captures the recreation cost for version i (and must be $\leq \theta$).

$$\begin{gathered}\textbf{minimize } \Sigma_{(V_i,V_j)\in E} x_{i,j} \times \Delta_{i,j}, \text{ subject to:}\\ \textstyle\sum_i x_{i,j} = 1, \forall j \\ r_j - r_i \geq \Phi_{i,j} \text{ if } x_{i,j} = 1 \\ r_i \leq \theta, \forall i\end{gathered} \tag{7.1}$$

Lemma 7.4. *Problem 7.6 is equivalent to the optimization problem described above.*

Note however that the general form of an ILP does not permit an if-then statement (as in the second constraint in Equation 7.1 above). Instead, we can transform to the general form with the aid of a large constant C. Thus, the second constraint in Equation 7.1 can be expressed as follows:

$$\Phi_{i,j} + r_i - r_j \leq (1 - x_{i,j}) \times C \tag{7.2}$$

Where C is a "sufficiently large" constant such that no additional constraint is added to the model. For instance, C here can be set as $2 * \theta$. On one hand, if $x_{i,j} = 1 \Rightarrow \Phi_{i,j} + r_i - r_j \leq 0$. On the other hand, if $x_{i,j} = 0 \Rightarrow \Phi_{i,j} + r_i - r_j \leq C$. Since C is "sufficiently large", no additional constraint is added.

7.3 COMPUTATIONAL COMPLEXITY

In this section, we study the complexity of the problems listed in Table 7.1 under different application scenarios.

Problem 7.1 and 7.2 Complexity. As discussed in Section 7.2, Problem 7.1 and 7.2 can be solved in polynomial time by directly applying a minimum spanning tree algorithm (Kruskal's algorithm or Prim's algorithm for undirected graphs; Edmonds' algorithm [89] for directed graphs) and Dijkstra's shortest path algorithm respectively. Kruskal's algorithm has time complexity $O(E \log V)$, while Prim's algorithm also has time complexity $O(E \log V)$ when using binary heap for implementing the priority queue, and $O(E + V \log V)$ when using Fibonacci heap for implementing the priority queue. The running time of Edmonds' algorithm is $O(EV)$ and can be reduced to $O(E + V \log V)$ with faster implementation. Similarly, Dijkstra's algorithm for constructing the shortest path tree starting from the root has a time complexity of $O(E \log V)$ via a binary heap-based priority queue implementation and a time complexity of $O(E + V \log V)$ via Fibonacci heap-based priority queue implementation.

Next, we'll show that Problem 7.5 and 7.6 are NP-hard even for the special case where $\Delta = \Phi$ and Φ is symmetric. This will lead to hardness proofs for the other variants.

Triangle Inequality. The primary challenge that we encounter while demonstrating hardness is that our deltas must obey the triangle inequality: unlike other settings where deltas need not obey real constraints, since, in our case, deltas represent actual modifications that can be stored, it must obey additional realistic constraints. This causes severe complications in proving hardness, often transforming the proofs from very simple to fairly challenging.

Consider the scenario when $\Delta = \Phi$ and Φ is symmetric. We take Δ as an example. The triangle inequality, can be stated as follows:

$$|\Delta_{p,q} - \Delta_{q,w}| \leq \Delta_{p,w} \leq \Delta_{p,q} + \Delta_{q,w} \tag{7.3}$$

$$|\Delta_{p,p} - \Delta_{p,q}| \leq \Delta_{q,q} \leq \Delta_{p,p} + \Delta_{p,q} \tag{7.4}$$

where $p, q, w \in V$ and $p \neq q \neq w$. The first inequality states that the "delta" between two versions can not exceed the total "deltas" of any two-hop path with the same starting and ending vertex; while the second inequality indicates that the “delta” between two versions must be bigger than one version’s full storage cost minus another version’s full storage cost. Since each tuple and modification is recorded explicitly when Φ is symmetric, it is natural that these two inequalities hold.

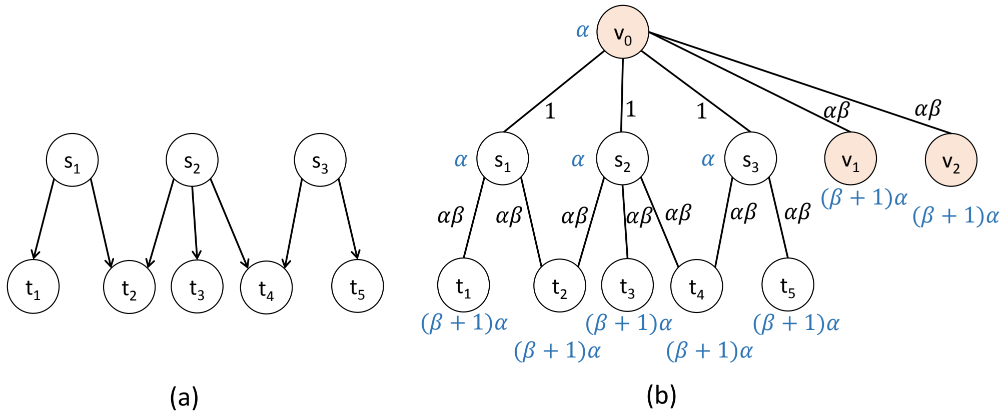


Figure 7.5: Illustration of Proof of Lemma 7.5

Problem 7.6 Hardness. We now demonstrate hardness.

Lemma 7.5. *Problem 7.6 is NP-hard when $\Delta = \Phi$ and Φ is symmetric.*

Proof. Here we prove NP-hardness using a reduction from the set cover problem. Recall that in the set cover problem, we are given m sets $S = \{s_1, s_2, ..., s_m\}$ and n items $T = \{t_1, t_2, ...t_n\}$, where each set s_i covers some items, and the goal is to pick k sets $\mathcal{F} \subset S$ such that $\cup_{\{F \in \mathcal{F}\}} F = T$ while minimizing k.

Given a set cover instance, we now construct an instance of Problem 7.6 that will provide a solution to the original set cover problem. The threshold we will use in Problem 7.6 will be $(\beta + 1)\alpha$, where β, α are constants that are each greater than $2(m + n)$. (This is just to ensure that they are “large”.) We now construct the graph $G(V, E)$ in the following way; we display the constructed graph in Figure 7.5. Our vertex set V is as follows:

- $\forall s_i \in S$, create a vertex s_i in V.
- $\forall t_i \in T$, create a vertex t_i in V.
- create an extra vertex v_0, two dummy vertices v_1, v_2 in V.

We add the two dummy vertices simply to ensure that v_0 is materialized, as we will see later. We now define the storage cost for materializing each vertex in V in the following way:

- $\forall s_i \in S$, the cost is α.
- $\forall t_i \in T$, the cost is $(\beta + 1)\alpha$.
- for vertex v_0, the cost is α.

- for vertex v_1, v_2, the cost is $(\beta + 1)\alpha$.

(These are the numbers colored blue in the tree of Figure 7.5(b).) As we can see above, we have set the costs in such a way that the vertex v_0 and the vertices corresponding to sets in S have low materialization cost, while the other vertices have high materialization cost: this is by design so that we only end up materializing these vertices. Our edge set E is now as follows.

- we connect vertex v_0 to each s_i with weight 1.
- we connect v_0 to both v_1 and v_2 each with weight $\beta\alpha$.
- $\forall s_i \in S$, we connect s_i to t_j with weight $\beta\alpha$ when $t_j \in s_i$, where $\alpha = |V|$.

It is easy to show that our constructed graph G obeys the triangle inequality.

Consider a solution to Problem 7.6 on the constructed graph G. We now demonstrate that that solution leads to a solution of the original set cover problem. Our proof proceeds in four key steps:

Step 1: The vertex v_0 will be materialized, while v_1, v_2 will not be materialized. Assume the contrary—say v_0 is not materialized in a solution to Problem 7.6. Then, both v_1 and v_2 must be materialized, because if they are not, then the recreation cost of v_1 and v_2 would be at least $\alpha(\beta + 1) + 1$, violating the condition of Problem 7.6. However we can avoid materializing v_1 and v_2, instead keep the delta to v_0 and materialize v_0, maintaining the recreation cost as is while reducing the storage cost. Thus v_0 has to be materialized, while v_1, v_2 will not be materialized. (Our reason for introducing v_1, v_2 is precisely to ensure that v_0 is materialized so that it can provide basis for us to store deltas to the sets s_i.)

Step 2: None of the t_i will be materialized. Say a given t_i is materialized in the solution to Problem 7.6. Then, either we have a set s_j where s_j is connected to t_i in Figure 7.5(a) also materialized, or not. Let's consider the former case. In the former case, we can avoid materializing t_i, and instead add the delta from s_j to t_i, thereby reducing storage cost while keeping recreation cost fixed. In the latter case, pick any s_j such that s_j is connected to t_i and is not materialized. Then, we must have the delta from v_0 to s_j as part of the solution. Here, we can replace that edge, and materialized t_i, with materialized s_j, and the delta from s_j to t_i: this would reduce the total storage cost while keeping the recreation cost fixed. Thus, in either case, we can improve the solution if any of the t_i are materialized, rendering the statement false.

Step 3: For each s_i, either it is materialized, or the edge from v_0 to s_i will be part of the storage graph. This step is easy to see: since none of the t_i are materialized, either each s_i has to be materialized, or we must store a delta from v_0.

Step 4: The sets s_i that are materialized correspond to a minimal set cover of the original *problem.* It is easy to see that for each t_j we must have an s_i such that s_i covers t_j, and s_i is materialized, in order for the recreation cost constraint to not be violated for t_j. Thus, the materialized s_i must be a set cover for the original problem. Furthermore, in order for the storage cost to be as small as possible, as few s_i as possible must be materialized (this is the only place we can save cost). Thus, the materialized s_i also correspond to a minimal set cover for the original problem.

Thus, minimizing the total storage cost is equivalent to minimizing k in set cover problem.

Note that while the reduction above uses a graph with only some edge weights (i.e., recreation costs of the deltas) known, a similar reduction can be derived for a complete graph with all edge weights known. Here, we simply use the shortest path in the graph reduction above as the edge weight for the missing edges. In that case, once again, the storage graph in the solution to Problem 7.6 will be identical to the storage graph described above.

Problem 7.5 Hardness: We now show that Problem 7.5 is NP-Hard as well. The general philosophy is similar to the proof in Lemma 7.5, except that we create c dummy vertices instead of two dummy vertices v_1, v_2 in Lemma 7.5, where c is sufficiently large—this is to once again ensure that v_0 is materialized.

Lemma 7.6. *Problem 7.5 is NP-Hard when $\Delta = \Phi$ and Φ is symmetric.*

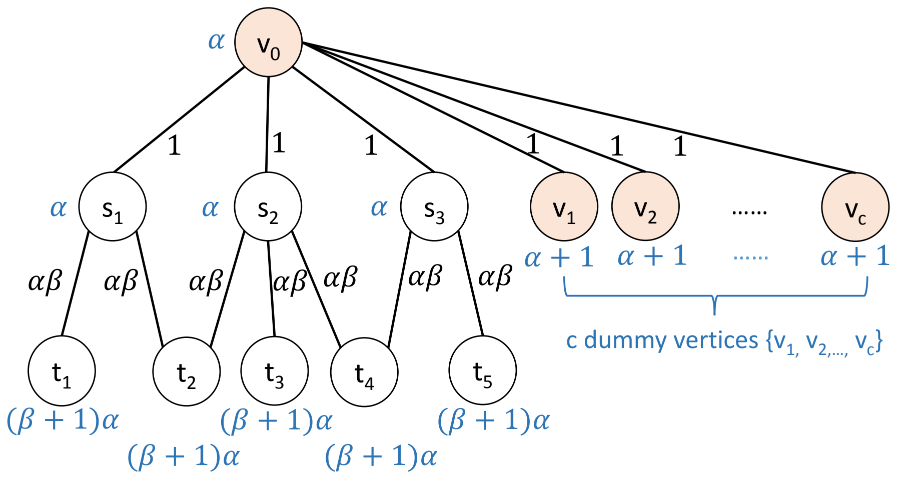


Figure 7.6: Illustration of Proof of Lemma 7.6

Proof. We prove NP-hardness using a reduction from the set cover problem. Recall that in the set cover decision problem, we are given m sets $S = \{s_1, s_2, ..., s_m\}$ and n items

$T = \{t_1, t_2, ...t_n\}$, where each set s_i covers some items, and given a k, we ask if there a subset $\mathcal{F} \subset S$ such that $\cup_{\{F \in \mathcal{F}\}} F = T$ and $|\mathcal{F}| \leq k$.

Given a set cover instance, we now construct an instance of Problem 7.5 that will provide a solution to the original set cover decision problem. The corresponding decision problem for Problem 7.5 is: given threshold $\alpha + (\beta + 1)\alpha n + k\alpha + (m - k)(\alpha + 1) + (\alpha + 1)c$ in Problem 7.5, is the minimum total storage cost in the constructed graph G no bigger than $\alpha + k\alpha + (m - k) + \alpha\beta n + c$.

We now construct the graph $G(V, E)$ in the following way; we display the constructed graph in Figure 7.6. Our vertex set V is as follows:

- $\forall s_i \in S$, create a vertex s_i in V.
- $\forall t_i \in T$, create a vertex t_i in V.
- create an extra vertex v_0, and c dummy vertices $\{v_1, v_2, \ldots, v_c\}$ in V.

We add the c dummy vertices simply to ensure that v_0 is materialized, as we will see later. We now define the storage cost for materializing each vertex in V in the following way:

- $\forall s_i \in S$, the cost is α.
- $\forall t_i \in T$, the cost is $(\beta + 1)\alpha$.
- for vertex v_0, the cost is α.
- for each vertex in $\{v_1, v_2, \ldots, v_c\}$, the cost is $\alpha + 1$.

(These are the numbers colored blue in the tree of Figure 7.6.) As we can see above, we have set the costs in such a way that the vertex v_0 and the vertices corresponding to sets in S have low materialization cost while the vertices corresponding to T have high materialization cost: this is by design so that we only end up materializing these vertices. Even though the costs of the dummy vertices is close to that of v_0, s_i, we will show below that they will not be materialized either. Our edge set E is now as follows.

- we connect vertex v_0 to each s_i with weight 1.
- we connect v_0 to $v_i, 1 \leq i \leq c$ each with weight 1.
- $\forall s_i \in S$, we connect s_i to t_j with weight $\beta\alpha$ when $t_j \in s_i$, where $\alpha = |V|$.

It is easy to show that our constructed graph G obeys the triangle inequality.

Consider a solution to Problem 7.5 on the constructed graph G. We now demonstrate that that solution leads to a solution of the original set cover problem. Our proof proceeds in four key steps:

Step 1: The vertex v_0 will be materialized, while $v_i, 1 \leq i \leq c$ will not be materialized. Let's examine the first part of this observation, i.e., that v_0 will be materialized. Assume the contrary. If v_0 is not materialized, then at least one $v_i, 1 \leq i \leq c$, or one of the s_i must be materialized, because if not, then the recreation cost of $\{v_1, v_2, \ldots, v_c\}$ would be at least

$(\alpha+2)c > (\alpha+1)c+\alpha+(\beta+1)\alpha n+k\alpha+(m-k)(\alpha+1)$, violating the condition (exceeding total recreation cost threshold) of Problem 7.5. However we can avoid materializing this v_i (or s_i), instead keep the delta from v_i (or s_i) to v_0 and materialize v_0, reducing the recreation cost and the storage cost. Thus v_0 has to be materialized. Furthermore, since v_0 is materialized, $\forall v_i, 1 \leq i \leq c$ will not be materialized and instead we will retain the delta to v_0, reducing the recreation cost and the storage cost. Hence, the first step is complete.

Step 2: None of the t_i will be materialized. Say a given t_i is materialized in the solution to Problem 7.5. Then, either we have a set s_j where s_j is connected to t_i in Figure 7.6(a) also materialized, or not. Let us consider the former case. In the former case, we can avoid materializing t_i, and instead add the delta from s_j to t_i, thereby reducing storage cost while keeping recreation cost fixed. In the latter case, pick any s_j such that s_j is connected to t_i and is not materialized. Then, we must have the delta from v_0 to s_j as part of the solution. Here, we can replace that edge, and the materialized t_i, with materialized s_j, and the delta from s_j to t_i: this would reduce the total storage cost while keeping the recreation cost fixed. Thus, in either case, we can improve the solution if any of the t_i are materialized, rendering the statement false.

Step 3: For each s_i, either it is materialized, or the edge from v_0 to s_i will be part of the storage graph. This step is easy to see: since none of the t_i are materialized, either each s_i has to be materialized, or we must store a delta from v_0.

Step 4: If the minimum total storage cost is no bigger than $\alpha+k\alpha+(m-k)+\alpha\beta n+c$, then there exists a subset $\mathcal{F} \subset S$ such that $\cup_{\{F\in\mathcal{F}\}}F = T$ and $|\mathcal{F}| \leq k$ in the original set cover decision problem, and vice versa. Let's examine the first part. If the minimum total storage cost is no bigger than $\alpha+k\alpha+(m-k)+\alpha\beta n+c$, then the storage cost for all $s_i \in S$ must be no bigger than $k\alpha+(m-k)$ since the storage cost for v_0, $\{v_1, v_2, \ldots, v_c\}$ and $\{t_1, t_2, \ldots, t_n\}$ is α, c and $\alpha\beta n$ respectively according to Step 1 and 2. This indicates that at most k $s_i \in S$ is materialized (we let the set of materialized s_i be M and $|M| \leq k$). Next, we prove that each t_j is stored as the modification from the materialized $s_i \in M$. Suppose there exists one or more t_j which is stored as the modification from $s_i \in S - M$, then the total recreation cost must be more than $\alpha+((\beta+1)\alpha n+1)+k\alpha+(m-k)(\alpha+1)+(\alpha+1)c$, which exceeds the total recreation threshold. Thus, we have each $t_j \in T$ is stored as the modification from $s_i \in M$. Let $\mathcal{F} = M$, we can obtain $\cup_{\{F\in\mathcal{F}\}}F = T$ and $|\mathcal{F}| \leq k$. Thus, If the minimum total storage cost is no bigger than $\alpha+k\alpha+(m-k)+\alpha\beta n+c$, then there exists a subset $\mathcal{F} \subset S$ such that $\cup_{\{F\in\mathcal{F}\}}F = T$ and $|\mathcal{F}| \leq k$ in the original set cover decision problem.

Next let's examine the second part. If there exists a subset $\mathcal{F} \subset S$ such that $\cup_{\{F\in\mathcal{F}\}}F = T$ and $|\mathcal{F}| \leq k$ in the original set cover decision problem, then we can materialize each vertex

$s_i \in \mathcal{F}$ as well as the extra vertex v_0, connect v_0 to $\{v_1, v_2, \ldots, v_c\}$ as well as $s_j \in S - \mathcal{F}$, and connect t_j to one $s_i \in \mathcal{F}$. The resulting total storage is $\alpha + k\alpha + (m-k) + \alpha\beta n + c$ and the total recreation cost equals to the threshold. Thus, if there exists a subset $\mathcal{F} \subset S$ such that $\cup_{\{F \in \mathcal{F}\}} F = T$ and $|\mathcal{F}| \leq k$ in the original set cover decision problem, then the minimum total storage cost is no bigger than $\alpha + k\alpha + (m-k) + \alpha\beta n + c$.

Thus, the decision problem in Problem 7.5 is equivalent to the decision problem in set cover problem.

Once again, the problem is still hard if we use a complete graph as opposed to a graph where only some edge weights are known.

Since Problem 7.4 swaps the constraint and goal compared to Problem 7.6, it is similarly NP-Hard. (Note that the decision versions of the two problems are in fact identical, and therefore the proof still applies.) Similarly, Problem 7.3 is also NP-Hard. Now that we have proved the NP-hard even in the special case where $\Delta = \Phi$ and Φ is symmetric, we can conclude that Problem 7.3, 7.4, 7.5, 7.6, are NP-hard in a more general setting where Φ is not symmetric and $\Delta \neq \Phi$, as listed in Table 7.1.

Hop-Based Variants. So far, our focus has been on proving hardness for the special case where $\Delta = \Phi$ and Δ is undirected. We now consider a different kind of special case, where the recreation cost of all pairs is the same, i.e., $\Phi_{ij} = 1$ for all i, j, while $\Delta \neq \Phi$, and Δ is undirected. In this case, we call the recreation cost as the *hop cost*, since it is simply the minimum number of delta operations (or "hops") needed to reconstruct V_i.

The reason why we bring up this variant is that this directly corresponds to a special case of the well-studied *d-MinimumSteinerTree* problem: Given an undirected graph $G = (V, E)$ and a subset $\omega \subseteq V$, find a tree with minimum weight, spanning the entire vertex subset ω while the diameter is bounded by d. The special case of *d-MinimumSteinerTree* problem when $\omega = V$, i.e., the minimum spanning tree problem with bounded diameter, directly corresponds to Problem 7.6 for the hop cost variant we described above. The hardness for this special case was demonstrated by [90] using a reduction from the SAT problem:

Lemma 7.7. *Problem 7.6 is NP-Hard when $\Delta \neq \Phi$ and Δ is symmetric, and $\Phi_{ij} = 1$ for all i, j.*

Note that this proof crucially uses the fact that $\Delta \neq \Phi$ unlike Lemma 7.5 and 7.6; thus the proofs are incomparable (i.e., one does not subsume the other).

For the hop-based variant, additional results on hardness of approximation are known by way of the *d-MinimumSteinerTree* problem [91, 92, 90]:

Lemma 7.8 ([90]). *For any $\epsilon > 0$, Problem 7.6 has no $\ln n$-ϵ approximation unless $NP \subset Dtime(n^{\log \log n})$.*

Since the hop-based variant is a special case of the last column of Table 7.1, this indicates that Problem 7.6 for the most general case is similarly hard to approximate; we suspect similar results hold for the other problems as well. It remains to be seen if hardness of approximation can be demonstrated for the variants in the second and third last columns.

7.4 PROPOSED ALGORITHMS

As discussed in Section 7.2, our different application scenarios lead to different problem formulations, spanning different constraints and objectives, and different assumptions about the nature of Φ, Δ.

Given that we demonstrated in the previous section that all the problems are NP-Hard, we focus on developing efficient heuristics. In this section, we present two novel heuristics: first, in Section 7.4.1, we present LMG, or the Local Move Greedy algorithm, tailored to the case when there is a bound or objective on the *average recreation cost*: thus, this applies to Problems 7.3 and 7.5. Second, in Section 7.4.2, we present MP, or Modified Prim's algorithm, tailored to the case when there is a bound or objective on the *maximum recreation cost*: thus, this applies to Problems 7.4 and 7.6. We present two variants of the MP algorithm tailored to two different settings.

Then, we present two algorithms — in Section 7.4.3, we present an approximation algorithm called LAST, and in Section 7.4.4, we present an algorithm called GitH which is based on Git repack. Both of these are adapted from literature to fit our problems and we compare these against our algorithms in Section 7.5. Note that LAST does not explicitly optimize any objectives or constraints in the manner of LMG, MP, or GitH, and thus the four algorithms are applicable under different settings; LMG and MP are applicable when there is a bound or constraint on the average or maximum recreation cost, while LAST and GitH are applicable when a "good enough" solution is needed. Furthermore, note that all these algorithms apply to both directed and undirected versions of the problems, and to the symmetric and unsymmetric cases.

7.4.1 Local Move Greedy Algorithm

The LMG algorithm is applicable when we have a bound or constraint on the average case recreation cost. We focus on the case where there is a constraint on the storage cost

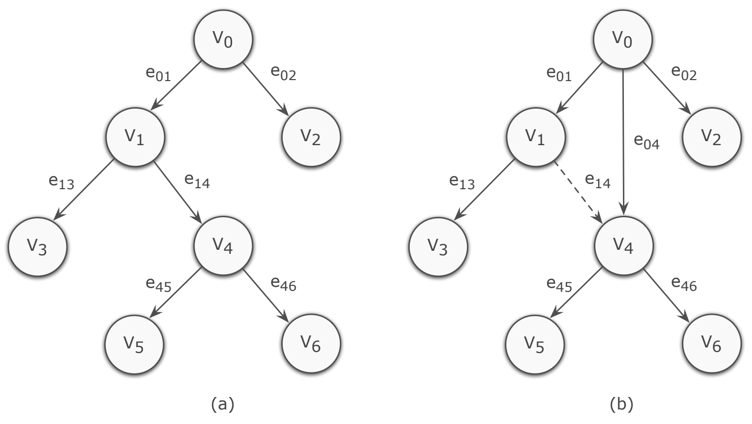


Figure 7.7: Illustration of Local Move Greedy Heuristic

(Problem 7.3); the case when there is no such constraint (Problem 7.5) can be solved by repeated iterations and binary search on the previous problem.

Outline. At a high level, the algorithm starts with the Minimum Spanning Tree (MST) as G_S, and then greedily adds edges from the Shortest Path Tree (SPT) that are not present in G_S, while G_S respects the bound on storage cost.

Detailed Algorithm. The algorithm starts off with G_S equal to the MST. The SPT naturally contains all the edges corresponding to complete versions. The basic idea of the algorithm is to replace deltas in G_S with versions from the SPT that maximize the following ratio:

$$\rho = \frac{\text{reduction in sum of recreation costs}}{\text{increase in storage cost}}$$

This is simply the reduction in total recreation cost per unit addition of weight to the storage graph G_S.

Let ξ consists of edges in the SPT not present in the G_S (these precisely correspond to the versions that are not explicitly stored in the MST, and are instead computed via deltas in the MST). At each "round", we pick the edge $e_{uv} \in \xi$ that maximizes ρ, and replace previous edge $e_{u'v}$ to v. The reduction in the sum of the recreation costs is computed by adding up the reductions in recreation costs of all $w \in G_S$ that are descendants of v in the storage graph (including v itself). On the other hand, the increase in storage cost is simply the weight of e_{uv} minus the weight of $e_{u'v}$. This process is repeated as long as the storage budget is not violated. We explain this with the means of an example.

Example 7.4. *Figure 7.7(a) denotes the current G_S. Node 0 corresponds to the dummy node. Now, we are considering replacing edge e_{14} with edge e_{04}, that is, we are replacing a*

Algorithm 7.1: Local Move Greedy Heuristic

```
   Input  : Minimum Spanning Tree (MST) , Shortest Path Tree (SPT), source vertex $V_0$,
            space budget $W$
   Output : A tree $T$ with weight $\leq W$ rooted at $V_0$ with minimal sum of access cost
1  Initialize $T$ as MST.
2  Let $d(V_i)$ be the distance from $V_0$ to $V_i$ in $T$, and $p(V_i)$ denote the parent of $V_i$ in T. Let
     $W(T)$ denote the storage cost of $T$.
3  while $W(T) < W$ do
4      $(\rho_{max}, e_{SPT}) \leftarrow (0, \emptyset)$
5      foreach $e_{uv} \in \xi$ do
6          compute $\rho_e$
7          if $\rho_e > \rho_{\max}$ then
8              $(\rho_{max}, \bar{e}) \leftarrow (\rho_e, e_{uv})$
9          end
10     end
11     $T \leftarrow T \setminus e_{u'v} \cup e_{uv}$;     $\xi \leftarrow \xi \setminus e_{uv}$
12     if $\xi = \emptyset$ then
13         return $T$
14     end
15 end
```

delta to version 4 *with version* 4 *itself. Then, the denominator of* ρ *is simply* $\Delta_{04} - \Delta_{14}$. *And the numerator is the changes in recreation costs of versions 4, 5, and 6 (notice that 5 and 6 were below 4 in the tree.) This is actually simple to compute: it is simply three times the change in the recreation cost of version 4 (since it affects all versions equally). Thus, we have the numerator of* ρ *is simply* $3 \times (\Phi_{01} + \Phi_{14} - \Phi_{04})$.

Complexity. For a given round, computing ρ for a given edge is $O(|V|)$. This leads to an overall $O(|V|^3)$ complexity, since we have up to $|V|$ rounds, and upto $|V|$ edges in ξ. However, if we are smart about this computation (by precomputing and maintaining across all rounds the number of nodes "below" every node), we can reduce the complexity of computing ρ for a given edge to $O(1)$. This leads to an overall complexity of $O(|V|^2)$ Algorithm 7.1 provides a pseudocode of the described technique.

Access Frequencies. Note that the algorithm can easily take into account access frequencies of different versions and instead optimize for the total weighted recreation cost (weighted by access frequencies). The algorithm is similar, except that the numerator of ρ will capture the reduction in weighted recreation cost.

7.4.2 Modified Prim's Algorithm

Next, we introduce a heuristic algorithm based on Prim's algorithm for Minimum Spanning Trees for Problem 7.6 where the goal is to reduce total storage cost while recreation cost for each version is within threshold θ; the solution for Problem 7.4 is similar.

Outline. At a high level, the algorithm is a variant of Prim's algorithm, greedily adding the version with smallest storage cost and the corresponding edge to form a spanning tree T. Unlike Prim's algorithm where the spanning tree simply grows, in this case, even if an edge is present in T, it could be removed in future iterations. At all stages, the algorithm maintains the invariant that the recreation cost of all versions in T is bounded within θ.

Detailed Algorithm. At each iteration, the algorithm picks the version V_i with the smallest storage cost to be added to the tree. Once this version V_i is added, we consider adding all deltas to all other versions V_j such that their recreation cost through V_i is within the constraint θ, and the storage cost does not increase. Each version maintains a pair $l(V_i)$ and $d(V_i)$: $l(V_i)$ denotes the marginal storage cost of V_i, while $d(V_i)$ denotes the total recreation cost of V_i. At the start, $l(V_i)$ is simply the storage cost of V_i in its entirety.

We now describe the algorithm in detail. Set X represents the current version set of the current spanning tree T. Initially $X = \emptyset$. In each iteration, the version V_i with the smallest storage cost ($l(V_i)$) in the priority queue PQ is picked and added into spanning tree T (line 7-8). When V_i is added into T, we need to update the storage cost and recreation cost for all V_j that are neighbors of V_i. Notice that in Prim's algorithm, we do not need to consider neighbors that are already in T. However, in our scenario a better path to such a neighbor may be found and this may result in an update(line 10-17). For instance, if edge $\langle V_i, V_j \rangle$ can make V_j's storage cost smaller while the recreation cost for V_j does not increase, we can update $p(V_j) = V_i$ as well as $d(V_j)$, $l(V_j)$ and T. For neighbors $V_j \notin T$(line 19-24), we update $d(V_j)$, $l(V_j)$,$p(V_j)$ if edge $\langle V_i, V_j \rangle$ can make V_j's storage cost smaller and the recreation cost for V_j is no bigger than θ. Algorithm 7.2 terminates in $|V|$ iterations since one version is added into X in each iteration.

Example 7.5. *Say we operate on G given by Figure 7.8, and let the threshold θ be 6. Each version V_i is associated with a pair $\langle l(V_i), d(V_i) \rangle$. Initially version V_0 is pushed into priority queue. When V_0 is dequeued, each neighbor V_j updates $< l(V_j), d(V_j) >$ as shown in Figure 7.10 (a). Notice that $l(V_i), i \neq 0$ for all i is simply the storage cost for that version. For example, when considering edge (V_0, V_1), $l(V_1) = 3$ and $d(V_1) = 3$ is updated since recreation cost (if V_1 is to be stored in its entirety) is smaller than threshold θ, i.e., $3 < 6$. Afterwards, version V_1, V_2 and V_3 are inserted into the priority queue. Next, we dequeue V_1 since $l(V_1)$ is*

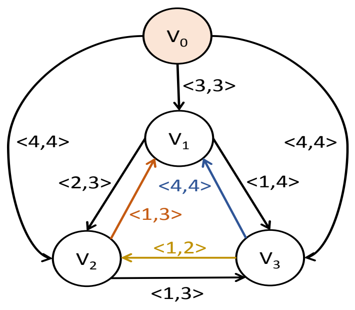


Figure 7.8: Directed Graph G

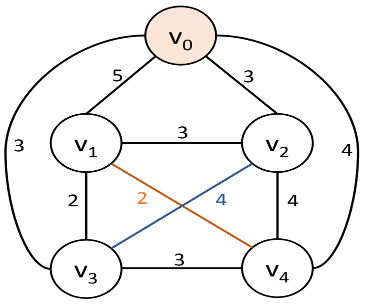


Figure 7.9: Undirected Graph G

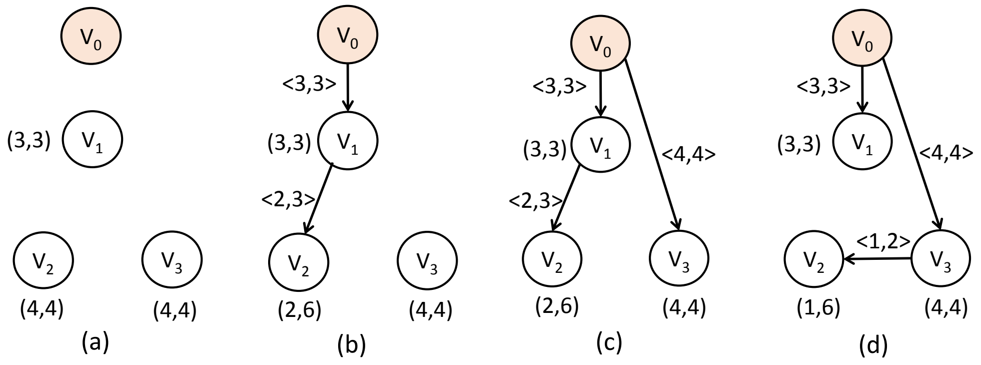


Figure 7.10: Illustration of Modified Prim's algorithm in Figure 7.8

smallest among the versions in the priority queue, and add V_1 *to the spanning tree. We then update* $< l(V_j), d(V_j) >$ *for all neighbors of* V_1*, e.g., the recreation cost for version* V_2 *will be* 6 *and the storage cost will be* 2 *when considering edge* (V_1, V_2)*. Since* $6 \leq 6$, $(l(V_2), d(V_2))$ *is updated to* $(2, 6)$ *as shown in Figure 7.10 (b); however,* $< l(V_3), d(V_3) >$ *will not be updated since the recreation cost is* $3 + 4 > 6$ *when considering edge* (V_1, V_3)*. Subsequently, version* V_2 *is dequeued because it has the lowest* $l(V_2)$*, and is added to the tree, giving Figure 7.10 (b). Subsequently, version* V_3 *are dequeued. When* V_3 *is dequeued from* PQ, $(l(V_2), d(V_2))$ *is updated. This is because the storage cost for* V_2 *can be updated to* 1 *and the recreation cost is still* 6 *when considering edge* (V_3, V_2)*, even if* V_2 *is already in* T *as shown in Figure 7.10 (c). Eventually, we get the final answer in Figure 7.10 (d).*

Complexity. The complexity of the algorithm is the same as that of Prim's algorithm, i.e., $O(|E| \log |V|)$. Each edge is scanned once and the priority queue need to be updated once in the worst case.

Algorithm 7.2: Modified Prim's Algorithm

```
Input   : Graph $G = (V, E)$, threshold $\theta$
Output : Spanning Tree $T = (V_T, E_T)$
1  Let $X$ be the version set of current spanning tree $T$; Initially $T = \emptyset, X = \emptyset$;
2  Let $p(V_i)$ be the parent of $V_i$; $l(V_i)$ denote the storage cost from $p(V_i)$ to $V_i$, $d(V_i)$ denote
     the recreation cost from root $V_0$ to version $V_i$,
3  Initially $\forall i \neq 0$, $d(V_0) = l(V_0) = 0, d(V_i) = l(V_i) = \infty$ ;
4  Enqueue $< V_0, (l(V_0), d(V_0)) >$ into priority queue $PQ$;
5  ($PQ$ is sorted by $l(v_i)$);
6  while $PQ \neq \emptyset$ do
7      $< V_i, (l(V_i), d(V_i)) > \leftarrow$ top($PQ$), dequeue($PQ$);
8      $T = T\cup < V_i, p(V_i) >$, $X = X \cup V_i$;
9      for $V_j \in$ ($V_i$'s neighbors in $G$) do
10         if $V_j \in X$ then
11             if $(\Phi_{i,j} + d(V_i)) \leq d(V_j)$ and $\Delta_{i,j} \leq l(V_j)$ then
12                 $T = T- < V_j, p(V_j) >$;
13                 $p(V_j) = V_i$;
14                 $T = T\cup < V_j, p(V_j) > d(V_j) \leftarrow \Phi_{i,j} + d(V_i)$;
15                 $l(V_j) \leftarrow \Delta_{i,j}$;
16             end
17         end
18         else
19             if $(\Phi_{i,j} + d(V_i)) \leq \theta$ and $\Delta_{i,j} \leq l(V_j)$ then
20                 $d(V_j) \leftarrow \Phi_{i,j} + d(V_i)$;
21                 $l(V_j) \leftarrow \Delta_{i,j}$; $p(V_j) = V_i$;
22                 enqueue(or update) $< V_j, (l(V_j), d(V_j)) >$ in $PQ$;
23             end
24         end
25     end
26 end
```

7.4.3 LAST Algorithm

Here, we sketch an algorithm from previous work [93] that enables us to find a tree with a good balance of storage and recreation costs, under the assumptions that $\Delta = \Phi$ and Φ is symmetric.

Outline. The algorithm starts from a minimum spanning tree and does a depth-first traveral (DFS) over the minimum spanning tree. During the process of DFS, if the recreation cost for a node exceeds the pre-defined threshold (set up front), then this current path is replaced with the shortest path to the node.

Detailed Algorithm. As discussed in Section 7.2.2, balancing between recreation cost and storage cost is equivalent to balancing between the minimum spanning tree and the shortest path tree rooted at V_0. Khuller et al. [93] studied the problem of balancing minimum spanning tree and shortest path tree in an undirected graph, where the resulting spanning tree T has the following properties, given parameter α:

- For each node V_i: the cost of path from V_0 to V_i in T is within α times the shortest path from V_0 to V_i in G.
- The total cost of T is within $(1 + 2/(\alpha - 1))$ times the cost of minimum spanning tree in G.

Even though Khuller's algorithm is meant for undirected graphs, it can be applied to the directed graph case without any comparable guarantees. The pseudocode is listed in Algorithm 7.3.

Let MST denote the minimum spanning tree of graph G and $SP(V_0, V_i)$ denote the shortest path from V_0 to V_i in G. The algorithm starts with the MST and then conducts a depth-first traversal in MST. Each node V keeps track of its path cost from root as well as its parent, denoted as $d(V_i)$ and $p(V_i)$ respectively. Given the approximation parameter α, when visiting each node V_i, we first check whether $d(V_i)$ is bigger than $\alpha \times SP(V_0, V_i)$ where SP stands for shortest path. If yes, we replace the path to V_i with the shortest path from root to V_i in G and update $d(V_i)$ as well as $p(V_i)$. In addition, we keep updating $d(V_i)$ and $p(V_i)$ during depth first traversal as stated in line 4-7 of Algorithm 7.3.

Example 7.6. *Figure 7.11 (a) is the minimum spanning tree (MST) rooted at node V_0 of G in Figure 7.9. The approximation threshold α is set to be 2. The algorithm starts with the MST and conducts a depth-first traversal in the MST from root V_0. When visiting node V_2, $d(V_2) = 3$ and the shortest path to node V_2 is 3, thus $3 < 2 \times 3$. We continue to visit node V_2 and V_3. When visiting V_3, $d(V_3) = 8 > 2 \times 3$ where 3 is the shortest path to V_3 in G. Thus, $d(V_3)$ is set to be 3 and $p(V_3)$ is set to be node 0 by replacing with the shortest path $\langle V_0, V_3 \rangle$ as shown in Figure 7.11 (b). Afterwards, the back-edge $< V_3, V_1 >$ is traversed in MST. Since $3 + 2 < 6$, where 3 is the current value of $d(V_3)$, 2 is the edge weight of (V_3, V_1) and 6 is the current value in $d(V_1)$, thus $d(V_1)$ is updated as 5 and $p(V_1)$ is updated as node V_3. At last node V_4 is visited, $d(V_4)$ is first updated as 7according to line 3-7. Since $7 < 2 \times 4$, lines 9-11 are not executed. Figure 7.11 (c) is the resulting spanning tree of the algorithm, where the recreation cost for each node is under the constraint and the total storage cost is $3 + 3 + 2 + 2 = 10$.*

Complexity. The complexity of the algorithm is $O(|E| \log |V|)$. Given the minimum spanning tree and shortest path tree rooted at V_0, Algorithm 7.3 is conducted via depth first traversal on MST. It is easy to show that the complexity for Algorithm 7.3 is $O(|V|)$. The

Algorithm 7.3: Balance MST and Shortest Path Tree [93]

```
   Input   : Graph $G = (V, E)$, $MST$, $SP$
   Output : Spanning Tree $T = (V_T, E_T)$
1  Initialize $T$ as $MST$. Let $d(V_i)$ be the distance from $V_0$ to $V_i$ in $T$ and $p(V_i)$ be the parent
     of $V_i$ in $T$.
2  while $DFS$ traversal on $MST$ do
3      $(V_i, V_j) \leftarrow$ the edge currently in traversal;
4      if $d(V_j) > d(V_i) + e_{i,j}$ then
5          $d(V_j) \leftarrow (d(V_i) + e_{i,j})$;
6          $p(V_j) \leftarrow V_i$;
7      end
8      if $d(V_j) > \alpha * SP(V_0, V_j)$ then
9          add shortest path $(V_0, V_j)$ into $T$;
10         $d(V_j) \leftarrow SP(V_0, V_j)$;
11         $p(V_j) \leftarrow V_0$;
12     end
13 end
```

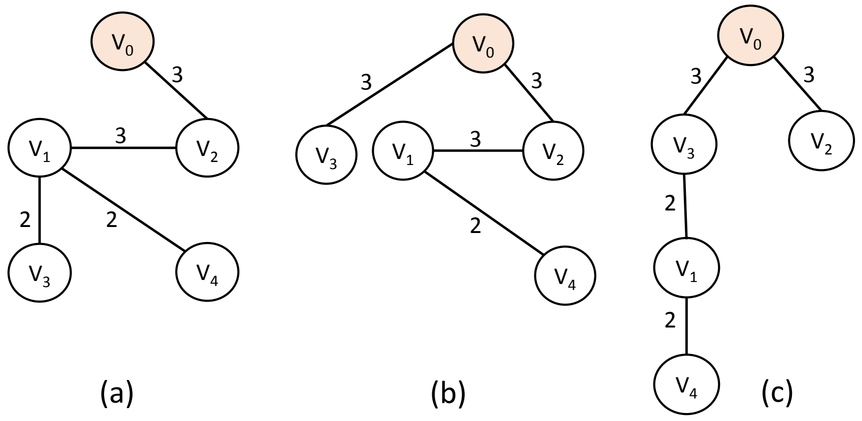


Figure 7.11: Illustration of LAST on Figure 7.9

time complexity for computing minimum spanning tree and shortest path tree is $O(|E| \log |V|)$ using heap-based priority queue.

7.4.4 Git Heuristic

This heuristic is an adaptation of the current heuristic used by Git and we refer to it as GitH. GitH uses two parameters: w (window size) and d (max depth). We consider the versions in an non-increasing order of their sizes. The first version in this ordering is chosen as the root of the storage graph and has depth 0 (i.e., it is materialized). At all times, we maintain a sliding window containing at most w versions. For each version V_i after the first one, let V_l denote a version in the current window. We compute: $\Delta'_{l,i} = \Delta_{l,i}/(d - d_l)$, where d_l is the depth of V_l (thus deltas with shallow depths are preferred over slightly smaller deltas with higher depths). We find the version V_j with the lowest value of this quantity and choose it as V_i's parent (as long as $d_j < d$). The depth of V_i is then set to $d_j + 1$. The sliding window is modified to move V_l to the end of the window (so it will stay in the window longer), V_j is added to the window, and the version at the beginning of the window is dropped.

Git repack

Git uses delta compression to reduce the amount of storage required to store a large number of files (objects) that contain duplicated information. However, git's algorithm for doing so is not clearly described anywhere. An old discussion with Linus has a sketch of the algorithm [94]. However there have been several changes to the heuristics used that don't appear to be documented anywhere.

The following describes our understanding of the algorithm based on the latest git source code [1].

Here we focus on "repack", where the decisions are made for a large group of objects. However, the same algorithm appears to be used for normal commits as well. Most of the algorithm code is in file: `builtin/pack-objects.c`

Step 1: Sort the objects, first by "type", then by "name hash", and then by "size" (in the decreasing order). The comparator is (line 1503):

```
static int type_size_sort(const void *_a, const void *_b)
```

Note the name hash is not a true hash; the `pack_name_hash()` function (`pack-objects.h`) simply creates a number from the last 16 non-white space characters, with the last characters counting the most (so all files with the same suffix, e.g., `.c`, will sort together).

Step 2: The next key function is `ll_find_deltas()`, which goes over the files in the sorted order. It maintains a list of W objects (W = window size, default 10) at all times. For the next object, say O, it finds the delta between O and each of the objects, say B, in the window; it chooses the the object with the minimum value of: `delta(B, O) / (max_depth - depth of B)` where `max_depth` is a parameter (default 50), and depth of B refers to the length of delta chain between a root and B.

The original algorithm appears to have only used `delta(B, O)` to make the decision, but the "depth bias" (denominator) was added at a later point to prefer slightly larger deltas with smaller delta chains. The key lines for the above part:

[1] Cloned from `https://github.com/git/git` on 5/11/2015, commit id: 8440f74997cf7958c7e8ec853f590828085049b8

- line 1812 (check each object in the window):

  ```
  ret = try_delta(n, m, max_depth, &mem_usage);
  ```

- lines 1617-1618 (depth bias):

  ```
  max_size = (uint64_t)max_size * (max_depth - src->depth)
                             / (max_depth - ref_depth + 1);
  ```

- line 1678 (compute delta and compare size):

  ```
  delta_buf = create_delta(src->index, trg->data,
                             trg_size, &delta_size, max_size);
  ```

`create_delta()` returns non-null only if the new delta being tried is smaller than the current delta (modulo depth bias), specifically, only if the size of the new delta is less than `max_size` argument. Note: lines 1682-1688 appear redundant given the depth bias calculations.

Step 3. Originally the window was just the last W objects before the object O under consideration. However, the current algorithm shuffles the objects in the window based on the choices made. Specifically, let $b_1, \dots, b_W$ be the current objects in the window. Let the object chosen to delta against for O be b_i. Then b_i would be moved to the end of the list, so the new list would be: $[b_1, b_2, \dots, b_{i-1}, b_{i+1}, \dots, b_W, O, b_i]$. Then when we move to the new object after O (say O'), we slide the window and so the new window then would be: $[b_2, \dots, b_{i-1}, b_{i+1}, \dots, b_W, O, b_i, O']$. Small detail: the list is actually maintained as a circular buffer so the list doesn't have to be physically "shifted" (moving b_i to the end does involve a shift though). Relevant code here is lines 1854-1861.

Finally we note that git never considers/computes/stores a delta between two objects of different types, and it does the above in a multi-threaded fashion, by partitioning the work among a given number of threads. Each of the threads operates independently of the others.

Complexity. The running time of the heuristic is $O(|V| \log |V| + w|V|)$, excluding the time to construct deltas.

7.5 EXPERIMENTS

We have built a prototype version management system, that will serve as a foundation to full-fledged data versioning management system. The system provides a subset of Git/SVN-like interface for dataset versioning. Users interact with the version management system in a client-server model over HTTP. The server is implemented in Java, and is responsible for

Dataset	DC	LC	BF	LF
Number of versions	100010	100002	986	100
Number of deltas	18086876	2916768	442492	3562
Average version size (MB)	347.65	356.46	0.401	422.79
MCA-Storage Cost (GB)	1265.34	982.27	0.0250	2.2402
MCA-Sum Recreation Cost (GB)	11506437.83	29934960.95	0.9648	47.6046
MCA-Max Recreation Cost (GB)	257.6	717.5	0.0063	0.5998
SPT-Storage Cost (GB)	33953.84	34811.14	0.3854	41.2881
SPT-Sum Recreation Cost (GB)	33953.84	34811.14	0.3854	41.2881
SPT-Max Recreation Cost (GB)	0.524	0.55	0.0063	0.5091

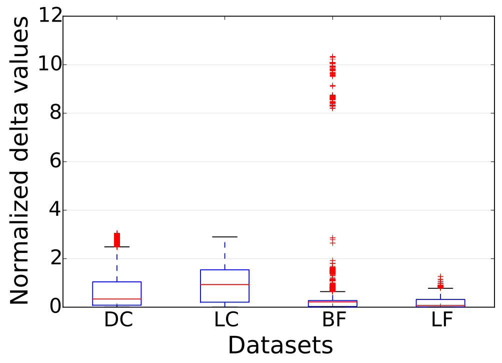


Figure 7.12: Dataset properties and distribution of delta sizes (each delta size scaled by the average version size in the dataset).

storing the version history of the repository as well as the actual files in them. The client is implemented in Python and provides functionality to create (commit) and check out versions of datasets, and create and merge branches. Note that, unlike traditional VCS which make a best effort to perform automatic merges, in our system we let the user perform the merge and notify the system by creating a version with more than one parent.

Implementation. In the following sections, we present an extensive evaluation of our designed algorithms using a combination of synthetic and derived real-world datasets. Apart from implementing the algorithms described above, LMG and LAST require both SPT and MST as input. For both directed and undirected graphs, we use Dijkstra's algorithm to find the single-source shortest path tree (SPT). We use Prim's algorithm to find the minimum spanning tree for undirected graphs. For directed graphs, we use an implementation [95] of the Edmonds' algorithm [89] for computing the min-cost arborescence (MCA). We ran all our experiments on a 2.2GHz Intel Xeon CPU E5-2430 server with 64GB of memory, running 64-bit Red Hat Enterprise Linux 6.5.

7.5.1 Datasets

We use four data sets: two synthetic and two derived from real-world source code repositories. Although there are many publicly available source code repositories with large numbers of commits (e.g., in `GitHub`), those repositories typically contain fairly small (source code) files, and further the changes between versions tend to be localized and are typically very small; we expect dataset versions generated during collaborative data analysis to contain much larger datasets and to exhibit large changes between versions. We were unable to find any realistic workloads of that kind.

Hence, we generated realistic dataset versioning workloads as follows. First, we wrote a *synthetic version generator suite*, driven by a small set of parameters, that is able to generate a variety of version histories and corresponding datasets. Second, we created two real-world datasets using publicly available forks of popular repositories on `GitHub`. We describe each of the two below.

Synthetic Datasets: Our synthetic dataset generation suite takes a two-step approach to generate a dataset that we sketch below. The first step is to generate a version graph with the desired structure, controlled by the following parameters:

- `number of commits,` i.e., the total number of versions.
- `branch interval and probability,` the number of consecutive versions after which a branch can be created, and probability of creating a branch.
- `branch limit,` the maximum number of branches from any point in the version history. We choose a number in [1, `branch limit`] uniformly at random when we decide to create branches.
- `branch length,` the maximum number of commits in any branch. The actual length is a uniformly chosen integer between 1 and `branch length`.

Once a version graph is generated, the second step is to generate the appropriate versions and compute the deltas. The files in our synthetic dataset are ordered CSV files (containing tabular data) and we use deltas based on UNIX-style diffs. The previous step also annotates each edge (u, v) in the version graph with edit commands that can be used to produce v from u. Edit commands are a combination of one of the following six instructions – add/delete a set of consecutive rows, add/remove a column, and modify a subset of rows/columns.

Using this, we generated two synthetic datasets (Figure 7.12):

- **Densely Connected (DC):** This dataset is based on a "flat" version history, i.e., number of branches is high, they occur often and have short lengths. For each version in this data set, we compute the delta with all versions in a 10-hop distance in the version graph to populate additional entries in Δ and Φ.

- **Linear Chain (LC):** This dataset is based on a "mostly-linear" version history, i.e., number of branches is low, they occur after large intervals and have longer lengths. For each version in this data set, we compute the delta with all versions within a 25-hop distance in the version graph to populate Δ and Φ.

Real-world datasets: We use 986 forks of the Twitter Bootstrap repository and 100 forks of the Linux repository, to derive our real-world workloads. For each repository, we checkout the latest version in each fork and concatenate all files in it (by traversing the directory structure in lexicographic order). Thereafter, we compute deltas between all pairs of versions in a repository, provided the size difference between the versions under consideration is less than a threshold. We set this threshold to 100KB for the Twitter Bootstrap repository and 10MB for the Linux repository. This gives us two real-world datasets, Bootstrap Forks (BF) and Linux Forks (LF), with properties shown in Figure 7.12.

7.5.2 Comparison with SVN and Git

We begin with evaluating the performance of two popular version control systems, SVN (v1.8.8) and Git (v1.7.1), using the LF dataset. We create an FSFS-type repository in SVN, which is more space efficient than a Berkeley DB-based repository [96]. We then import the entire LF dataset into the repository in a single commit. The amount of space occupied by the `db/revs/` directory is around 8.5GB and it takes around 48 minutes to complete the import. We contrast this with the naive approach of applying a `gzip` on the files which results in total compressed storage of 10.2GB. In case of Git, we add and commit the files in the repository and then run a `git repack -a -d --depth=50 --window=50` on the repository[2]. The size of the Git pack file is 202 MB although the repack consumes 55GB memory and takes 114 minutes (for higher window sizes, Git fails to complete the repack as it runs out of memory).

In comparison, the solution found by the MCA algorithm occupies 516MB of compressed storage (2.24GB when uncompressed) when using UNIX `diff` for computing the deltas. To make a fair comparison with Git, we use `xdiff` from the LibXDiff library [99] for computing the deltas, which forms the basis of Git's delta computing routine. Using `xdiff` brings down the total storage cost to just 159 MB. The total time taken is around 102 minutes; this includes the time taken to compute the deltas and then to find the MCA for the corresponding graph.

The main reason behind SVN's poor performance is its use of "skip-deltas" to ensure

[2]Unlike `git repack`, `svnadmin pack` has a negligible effect on the storage cost as it primarily aims to reduce disk seeks and per-version disk usage penalty by concatenating files into a single "pack" [97, 98].

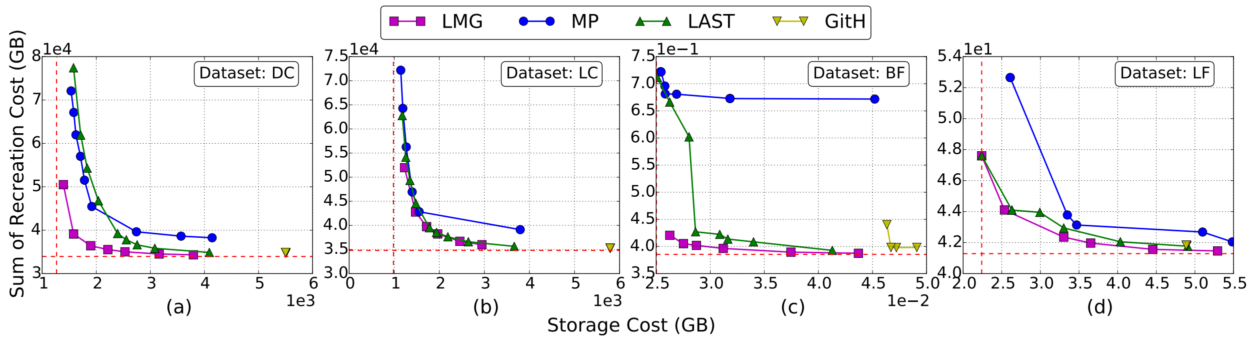


Figure 7.13: Directed case, comparing the storage costs and total recreation costs

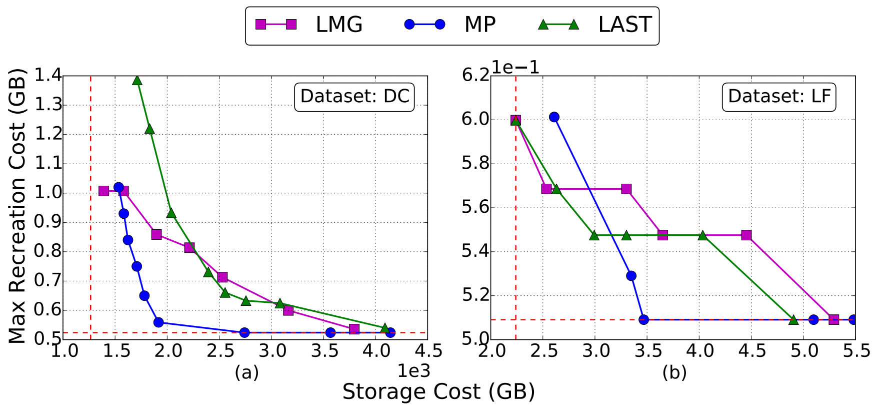


Figure 7.14: Directed case, comparing the storage costs and maximum recreation costs

that at most $O(\log n)$ deltas are needed for reconstructing any version [100]; that tends to lead it to repeatedly store redundant delta information as a result of which the total space requirement increases significantly. The heuristic used by Git is much better than SVN (Section 7.4.4). However as we show later (Fig. 7.13), our implementation of that heuristic (GitH) required more storage than LMG for guaranteeing similar recreation costs.

7.5.3 Experimental Results

Directed Graphs. We begin with a comprehensive evaluation of the three algorithms, LMG, MP, and LAST, on directed datasets. Given that all of these algorithms have parameters that can be used to trade off the storage cost and the total recreation cost, we compare them by plotting the different solutions they are able to find for the different values of their respective input parameters. Figure 7.13(a–d) show four such plots; we run each of the algorithms with a range of different values for its input parameter and plot the storage cost and the total (sum) recreation cost for each of the solutions found. We also show the minimum possible values for these two costs: the vertical dashed red line indicates the min-

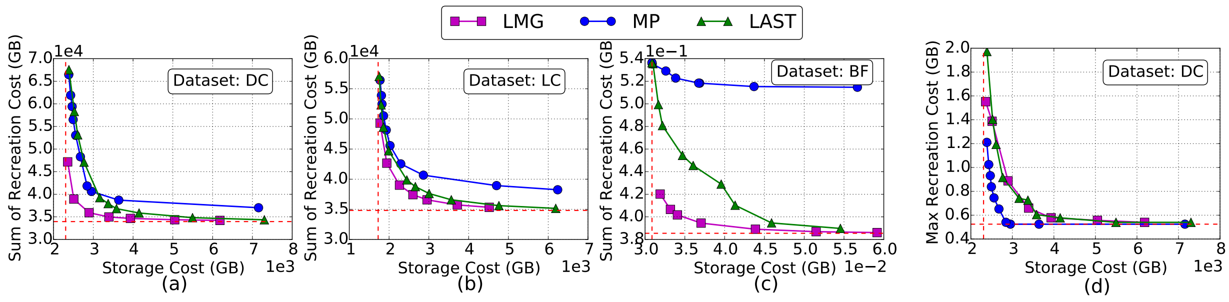


Figure 7.15: Results for the undirected case, comparing the storage costs and total recreation costs (a–c) or maximum recreation costs (d)

imum storage cost required for storing the versions in the dataset as found by MCA, and the horizontal one indicates the minimum total recreation cost as found by SPT (equal to the sum of all version sizes).

The first key observation we make is that, the total recreation cost decreases drastically by allowing a small increase in the storage budget over MCA. For example, for the DC dataset, the sum recreation cost for MCA is over 11 PB (see Table 7.12) as compared to just 34TB for the SPT solution (which is the minimum possible). As we can see from Figure 7.13(a), a space budget of 1.1× the MCA storage cost reduces the sum of recreation cost by three orders of magnitude. Similar trends can be observed for the remaining datasets and across all the algorithms. We observe that LMG results in the best tradeoff between the sum of recreation cost and storage cost with LAST performing fairly closely. An important takeaway here, especially given the amount of prior work that has focused purely on storage cost minimization (Chapter 2), is that: it is possible to construct balanced trees where the sum of recreation costs can be reduced and brought close to that of SPT while using only a fraction of the space that SPT needs.

We also ran GitH heuristic on the all the four datasets with varying window and depth settings. For BF, we ran the algorithm with four different window sizes (50, 25, 20, 10) for a fixed depth 10 and provided the GitH algorithm with all the deltas that it requested. For all other datasets, we ran GitH with an infinite window size but restricted it to choose from deltas that were available to the other algorithms (i.e., only deltas with sizes below a threshold); as we can see, the solutions found by GitH exhibited very good total recreation cost, but required significantly higher storage than other algorithms. This is not surprising given that GitH is a greedy heuristic that makes choices in a somewhat arbitrary order.

In Figures 7.14(a–b), we plot the maximum recreation costs instead of the sum of recreation costs across all versions for two of the datasets (the other two datasets exhibited similar behavior). The MP algorithm found the best solutions here for all datasets, and we also

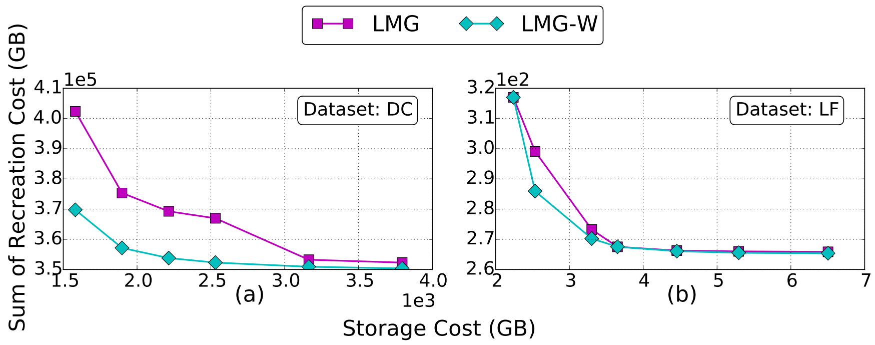


Figure 7.16: Taking workload into account leads to better solutions

observed that LMG and LAST both show plateaus for some datasets where the maximum recreation cost did not change when the storage budget was increased. This is not surprising given that the basic MP algorithm tries to optimize for the storage cost given a bound on the maximum recreation cost, whereas both LMG and LAST focus on minimization of the storage cost and one version with high recreation cost is unlikely to affect that significantly.

Undirected Graphs. We test the three algorithms on the undirected versions of three of the datasets (Figure 7.15). For DC and LC, undirected deltas between pairs of versions were obtained by concatenating the two directional deltas; for the BF dataset, we use UNIX `diff` itself to produce undirected deltas. Here again we observe that LMG consistently outperforms the other algorithms in terms of finding a good balance between the storage cost and the sum of recreation costs. MP again shows the best results when trying to balance the maximum recreation cost and the total storage cost. Similar results were observed for other datasets but are omitted due to space limitations.

Workload-aware Sum of Recreation Cost Optimization. In many cases, we may be able to estimate access frequencies for the various versions (from historical access patterns), and if available, we may want to take those into account when constructing the storage graph. The LMG algorithm can be easily adapted to take such information into account, whereas it is not clear how to adapt either LAST or MP in a similar fashion. In this experiment, we use LMG to compute a storage graph such that the sum of recreation costs is minimal given a space budget, while taking workload information into account. The worload here assigns a frequency of access to each version in the repository using a Zipfian distribution (with exponent 2); real-world access frequencies are known to follow such distributions. Given the workload information, the algorithm should find a storage graph that has the sum of recreation cost less than the index when the workload information is not taken into account (i.e., all versions are assumed to be accessed equally frequently). Figure 7.16 shows the results for this experiment. As we can see, for the DC dataset, taking into account the

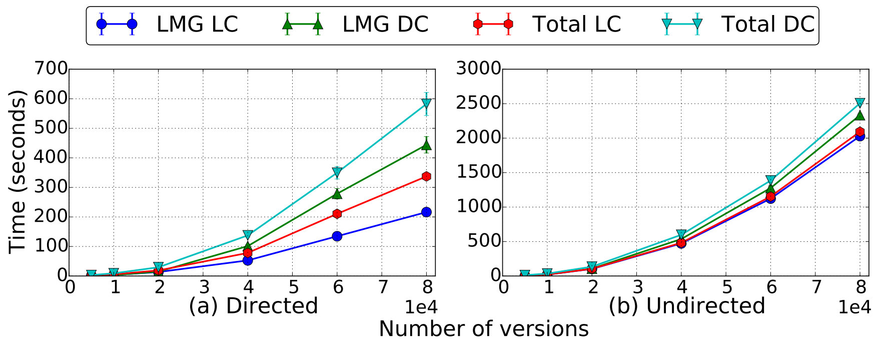


Figure 7.17: Running times of LMG

access frequencies during optimization led to much better solutions than ignoring the access frequencies. On the other hand, for the LF dataset, we did not observe a large difference.

Running Times. Here we evaluate the running times of the LMG algorithm. Recall that LMG takes MST (or MCA) and SPT as inputs. In Fig. 7.17, we report the total running time as well as the time taken by LMG itself. We generated a set of version graphs as subsets of the graphs for LC and DC datasets as follows: for a given number of versions n, we randomly choose a node and traverse the graph starting at that node in breadth-first manner till we construct a subgraph with n versions. We generate 5 such subgraphs for increasing values of n and report the average running time for LMG; the storage budget for LMG is set to three times of the space required by the MST (all our reported experiments with LMG use less storage budget than that). The time taken by LMG on DC dataset is more than LC for the same number of versions; this is because DC has lower delta values than LC (see Fig. 7.12) and thus requires more edges from SPT to satisfy the storage budget.

On the other hand, MP takes between 1 to 8 seconds on those datasets, when the recreation cost is set to maximum. Similar to LMG, LAST requires the MST/MCA and SPT as inputs; however the running time of LAST itself is linear and it takes less than 1 second in all cases. Finally the time taken by GitH on LC and DC datasets, on varying window sizes range from 35 seconds (window = 1000) to a little more than 120 minutes (window = 100000); note that, this excludes the time for constructing the deltas.

In summary, although LMG is inherently a more expensive algorithm than MP or LAST, it runs in reasonable time on large input sizes; we note that all of these times are likely to be dwarfed by the time it takes to construct deltas even for moderately-sized datasets.

Comparison with ILP solutions. Finally, we compare the quality of the solutions found by MP with the optimal solution found using the Gurobi Optimizer for Problem 7.6. We use the ILP formulation from Section 7.2.3 with constraint on the maximum recreation cost (θ), and compare the optimal storage cost with that of the MP algorithm (which resulted

		Storage Cost (GB)				
v15	θ	0.20	0.21	0.22	0.23	0.24
	ILP	0.36	0.36	0.22	0.22	0.22
	MP	0.36	0.36	0.23	0.23	0.23
v25	θ	0.63	0.66	0.69	0.72	0.75
	ILP	2.39	1.95	1.50	1.18	1.06
	MP	2.88	2.13	1.7	1.18	1.18
v50	θ	0.30	0.34	0.41	0.54	0.68
	ILP	1.43	1.10	0.83	0.66	0.60
	MP	1.59	1.45	1.06	0.91	0.82

Table 7.2: Comparing ILP and MP solutions for small datasets, given a bound on max recreation cost, θ (in GB)

in solutions with lowest maximum recreation costs in our evaluation). We use our synthetic dataset generation suite to generate three small datasets, with 15, 25 and 50 versions denoted by v15, v25 and v50 respectively and compute deltas between all pairs of versions. Table 7.2 reports the results of this experiment, across five θ values. The ILP turned out to be very difficult to solve, even for the very small problem sizes, and in many cases, the optimizer did not finish and the reported numbers are the best solutions found by it.

As we can see, the solutions found by MP are quite close to the ILP solutions for the small problem sizes for which we could get any solutions out of the optimizer. However, extrapolating from the (admittedly limited) data points, we expect that on large problem sizes, MP may be significantly worse than optimal for some variations on the problems (we note that the optimization problem formulations involving max recreation cost are likely to turn out to be harder than the formulations that focus on the average recreation cost). Development of better heuristics and approximation algorithms with provable guarantees for the various problems that we introduce are rich areas for further research.

7.6 ADDITIONAL RELATED WORK

We now cover additional related work we didn't cover in Chapter 2.

Diff Mechanism. There is much prior work on compactly encoding differences between two files or strings in order to reduce communication or storage costs. In addition to standard utilities like UNIX `diff`, many sophisticated techniques have been proposed for computing differences or edit sequences between two files (e.g., xdelta [101], vdelta [102], vcdiff [103], zdelta [104]). That work is largely orthogonal and complementary to our work.

Archiving. Buneman et al. [41] proposed an archiving technique where all versions of the data are merged into one hierarchy. An element appearing in multiple versions is stored only once along with a timestamp. This technique of storing versions is in contrast with techniques where retrieval of certain versions may require undoing the changes (unrolling the deltas). The hierarchical data and the resulting archive is represented in XML format which enables use of XML tools such as an XML compressor for compressing the archive. It was not, however, a full-fledged version control system representing an arbitrarily graph of versions; rather it focused on algorithms for compactly encoding a linear chain of versions.

Snapshot Queries. Snapshot queries have recently also been studied in the context of array databases [105, 106] and graph databases [107]. Seering et al. [106] considered the problem of storing an arbitrary tree of versions in the context of scientific databases; their proposed techniques are based on finding a minimum spanning tree (as we discussed earlier, that solution represents one extreme in the spectrum of solutions that needs to be considered). They also proposed several heuristics for choosing which versions to materialize given the distribution of access frequencies to historical versions. Several databases support "time travel" features (e.g., Oracle Flashback, Postgres [108]). However, those do not allow for branching trees of versions. [109] articulates a similar vision to data versioning management; however, they do not propose formalisms or algorithms to solve the underlying data management challenges. In addition, the schema of tables encoded with Flashback cannot change.

In this chapter, we introduced a generalized storage engine for efficient data versioning balancing storage and recreation. In the next chapter, we will relax our final assumption, the "from-scratch" assumption, and focus on lineage inference for dataset versions residing in an existing data repository.

CHAPTER 8: GENERALIZED PROVENANCE MANAGER

Another assumption made in OrpheusDB is that data science teams use OrpheusDB from the very beginning of their project and always register their dataset with complete metadata with the system. In particular, the *provenance manager* in OrpheusDB is responsible for recording the metadata during commits by users. However, in practice, little or no metadata is captured upon each version's generation. In this chapter, we remove the "from-scratch" assumption, and infer the provenance information using a post-processing approach when no such derivation information is available. Specifically, we focus on reverse-engineering the lineage information, based solely on the dataset content. Our goal is to infer the so-called lineage graph among all the versioned datasets, where each node in the lineage graph corresponds to a version and each edge corresponds to the derivation relationship between two versions.

The main contributions of this chapter are given as follows:

- We formally define and analyze the lineage inference problem and propose an end-to-end workflow, titled Relic, to infer the lineage graph, i.e., the derivation relationship among all versions within a working repository (Section 8.2 and 8.3).
- We develop a fine-grained delta metric called *cell-level delta*, and demonstrate that we can capture the derivation edges effectively via cell-level delta together with the minimum description length (MDL) principle (Section 8.4).
- In order to explain the derivation relationship along with each inferred edge, we define an instruction set at the row- and column-level and propose an algorithm to provide a structural explanation using these instructions (Section 8.5).
- In addition to effectiveness, we also work on improving the efficiency of our end-to-end workflow. We propose to employ sketch techniques along with some greedy algorithm to reduce the runtime (Section 8.6).
- We conduct experiments on both synthetic and real workflows, demonstrating the effectiveness and efficiency of Relic. (Section 8.8).

8.1 MOTIVATING EXAMPLE

The impact of data science has been felt across all domains and industries as organizations scramble to find ways to integrate "big data" within their operations. Data scientists often deal with raw data that require several stages of data preparation and feature engineering before it can be used in data analytics or machine learning pipelines. It is typical for scientists to use trial-and-error to refine the outcomes of preparation tasks, such as transformation and feature selection, generating multiple data artifacts as a result [110]. However, in practice, little or no lineage information is recorded upon each artifact's generation, hindering future developmental insights and potentially limiting the processes of dataset sharing and discovery, or even the reproducibility of analytical results [43]. It is often desirable to reconstruct a human-interpretable lineage for such various versions. As demonstrated in a user study from prior work [111], detecting the relationship among datasets can enable users to recall transformations from one dataset version to another, and subsequently help users identify the best dataset for a given task.

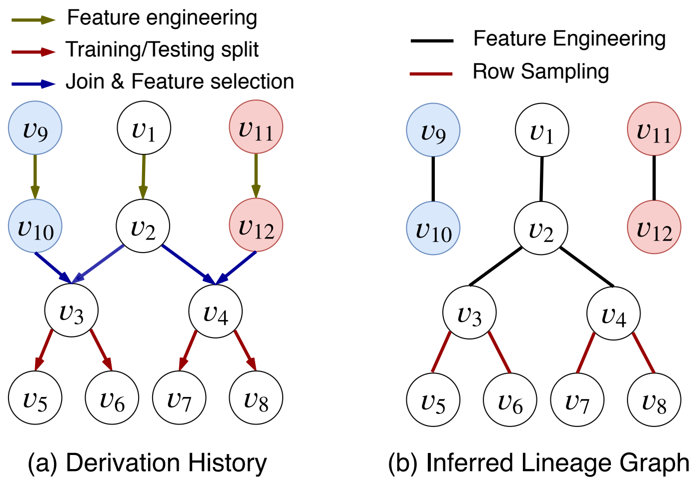

Feature engineering
Training/Testing split
Join & Feature selection
v_9 v_1 v_{11} v_{10} v_2 v_{12} v_3 v_4 v_5 v_6 v_7 v_8
(a) Derivation History
Feature Engineering
Row Sampling
v_9 v_1 v_{11} v_{10} v_2 v_{12} v_3 v_4 v_5 v_6 v_7 v_8
(b) Inferred Lineage Graph

Figure 8.1: Illustration of Lineage Graph

Example 8.1 (Lineage Graph)**.** *Figure 8.1(a) depicts a real data science workflow for flight delay prediction published in Azure AI gallery [112]. Initially, v_1 is a dataset with flight information, v_9 is a dataset with weather information, and v_{11} is a dataset with holiday information. The machine learning task here is to predict flight delay. The stored datasets v_2, v_{10}, and v_{12} are derived from v_1, v_9, and v_{11} respectively by performing some feature engineering operations, while v_3 and v_4 are obtained by joining the derived flight dataset v_2 with the weather dataset v_{10} and the holiday dataset v_{12} respectively. Lastly, v_3 (resp. v_4) is split into training and testing dataset, i.e., v_5 and v_6 (resp. v_7 and v_8).*

As revealed in Example 8.1, a real workflow written by some data scientist, feature engineering and data quality play a critical role in the performance of a machine learning task. Consequently, machine learning practitioners spend a vast amount of time in data curation, transformation, and feature engineering. Typically, data scientists would materialize the derived dataset at each preprocessing stage before feeding data into the machine learning pipeline. The reasons are two-fold: first, the data preparation platform is typically different from the machine learning platform, e.g., Flume in C++ vs. TensorFlow in Python; second, materialization can help eliminate duplicate data preprocessing computations that are common to different machine learning pipelines, such as those exploring various hyperparameters. As a result, different versions are generated as illustrated in Example 8.1 and are scattered across the repository with little lineage information associated with it. Hence, there is no easy way for users to retrospectively understand the evolution of the repository or the project, which motivates our goal of inferring the lineage graph for a set of versions within a working repository. Figure 8.1(b) depicts one possible inferred lineage graph.

Related work. Even though systems like OrpheusDB (Chapter 3-5) and ProvDB [43] can help explicitly capture the derivation relationship across versions, they have substantial barriers to adoption—due to which, most data scientists are manipulating data in a quick-and-dirty manner. Thus, there is still a pressing need for post-processing approaches to infer the relationships among artifacts in a working directory, shared repository, or even data lake.

ReConnect [111] attempts to discover the relationship for a given dataset pair. It first defines a space of relevant relationships, generates the conditions for each relationship based on row and column statistics, and then suggests a relationship for a given dataset pair by examining the conditions. Since the statistics may not be sufficient in determining the relationship, ReConnect asks the user to select a candidate relationship for validation. In order to identify the relationship between all possible pairs in a large collection of datasets, Abdussalam et al. [113] further propose a system, entitled ReDiscover, to automate the relationship discovery process without involving user input. ReDiscover first computes column statistics and then feeds them into machine learning models to predict the relationship. However, both ReConnect and ReDiscover only consider a limited relationship set, i.e., containment, augmentation, complementation, template, and incompatible. In practice, the relationship is usually much more complicated. For instance, a dataset may evolve from another dataset with some old tuples deleted and some new tuples added. In this scenario, ReConnect fails to identify the relationship as it does not exactly correspond to any of the defined relationships—a combination of augmentation and containment.

Another line of work [114, 115, 116, 117] focuses on reverse-engineering SQL queries performed to transform one artifact to another. However, we argue that modern data analytics are usually performed across a variety of platforms, tools, and languages. Besides SQL queries, manual edits, scripts, and programs can also be involved in data curation, transformation, and feature engineering. Thus, SQL may not be a good fit in representing the data difference. In addition, inferring a concise SQL query itself is a very hard problem [118].

We may end up complicating the problem further if we formulate the delta between two artifacts as SQL queries. Furthermore, existing work focuses on a single pair of datasets. Instead, we aim to summarize the relationship among a collection of datasets, which poses scalability challenges since it involves all possible dataset pairs.

Challenges. In this chapter, we explore the problem of inferring the lineage among a collection of datasets under the worst-case scenario, i.e., when relying exclusively on artifact dumps, with little or no metadata available. However, it is not easy to recover the lineage graph both in terms of effectiveness and efficiency. Specifically, the challenges center around *(a)* how to ensure the quality of the inferred lineage graph; *(b)* how to infer the lineage graph in an efficient manner. In order to obtain a high-quality lineage graph, we develop an end-to-end workflow called Relic, with a carefully designed delta metric and structural explanation associated with each inferred edge for better user interpretation of the changes. Furthermore, when users intend to retrospectively explore the repository, it is desirable for Relic to return the inferred lineage graph in a timely manner. To tackle the efficiency issue, we propose to employ sketch-based techniques for inferring the lineage graph. In this way, Relic enables users to quickly explore the lineage of different versions generated during different stages of data preprocessing.

8.2 PROBLEM DEFINITION

The problem studied in this chapter is the following: given a working repository with a collection of datasets, infer the derivation relationship among these datasets purely based on the content in each dataset. Here, the working repository can be a single user's project directory or a shared repository among team collaborators. Let $G = (V, E)$ be the real "true" lineage graph, where each $v \in V$ is an artifact[1] and each $e = (v_i, v_j) \in E$ means that v_j is derived from v_i. Our goal is to derive an inferred lineage graph $G' = (V, E')$, such that E' and E have a large set similarity. Ideally, E' would be exactly the same as E. In the following, we will clarify some of our design choices for this problem, including our focused operation space along each derivation edge, the space of inferred lineage graphs, as well as the quality measure for the inferred lineage graph G'.

Operation Space. A new dataset version can be derived from a base version in various ways, including manual cell-level edits, declarative data manipulation via SQL queries, and general transformations via imperative programs. We envision Relic to work with general transformation operations instead of any DML-specific ones. Similar to previous

[1]We use version, artifact, and dataset interchangeably.

work [119], we also classify operations into three categories, i.e., one-to-one, one-to-many, and many-to-one mappings, based on the record mappings between the base version and the derived version. Notably, most transformations belong to one-to-one mapping category, e.g., sampling, cell-edit, or feature normalization. We call these operations *point-preserving operations*, and are the main focus of this chapter. For other operations such as natural join, aggregate, we will discuss how to handle them in the extension section (Section 8.7).

Directed vs. Undirected. It is easy to see that the real lineage graph G is a directed acyclic graph (DAG). However, we remark that for any edge $e = (v_i, v_j) \in E$, there always exists a corresponding backward edge $\hat{e} = (v_j, v_i)$ and it is often difficult to differentiate between the forward and backward edge (i.e., e and $\hat{e}$) during retrospective lineage inference. For instance, say v_i is derived from v_j by adding some new features in the real lineage graph. However, it is also acceptable to infer that v_j is derived from v_i by applying feature selection operations. Thus, we opt to construct an *undirected* lineage graph G' with an annotated explanation along with each direction.

Quality Measure on G'. As discussed above, the inferred lineage graph G' is undirected. Assume that we have the ground truth, i.e., the real lineage graph G. To compare G' with G, we first convert G into an undirected graph and then compute the Jaccard set similarity between E and E', where each $e \in E$ or $e' \in E'$ is an undirected edge. Specifically, we can use $\theta_{G'} = \frac{|E \cap E'|}{|E \cup E'|}$ as the quality measure for the inferred lineage graph G'. The larger $\theta_{G'}$ is, the more similar G' is to G and the higher quality G' is.

We now formally define the lineage inference problem in Problem 8.1.

Problem 8.1 (Lineage Inference). *Given a collection of tabular dataset versions V, infer the lineage graph $G' = (V, E')$ based on the content of each dataset $v \in V$, maximizing $\theta_{G'}$.*

8.3 END-TO-END WORKFLOW

In order to infer the lineage graph G' purely based on the contents in each version $v \in V$, we propose an end-to-end workflow, called Relic, consisting of four steps: *(a)* profiling; *(b)* pre-clustering; *(c)* edge inference; and *(d)* structural explanation, as depicted in Figure 8.2. Profiling serves as a building block for the following steps. Based on the requirement in step (c), different profiling operators can be applied to each version $v \in V$. For instance, we can apply primary key detection and schema matching in step (a). We will elaborate more in Section 8.4. After profiling, we can conduct pre-clustering based on some heuristic rules such as primary key or schema-based grouping. Step (b) can help constrain the search space for the inferred lineage graph. The core part within this end-to-end workflow is step (c), where we propose to infer the lineage graph by adopting the minimum description length principle and using our carefully designed delta metric between versions. Last but not least, we provide a structural explanation along with each edge using instructions at the row and column granularity. Step (d) enables better user interpretation. Next, we will dive into each step in detail.

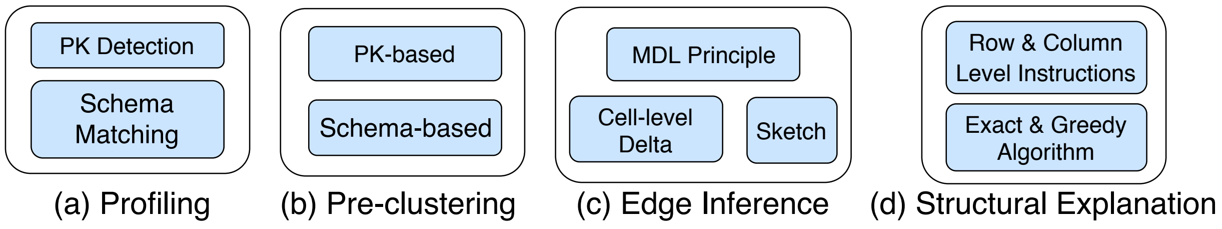


Figure 8.2: End-to-End Workflow of Relic

Profiling. For each version $v \in V$, we can identify its primary key or unique key, providing row-to-row correspondence across versions. In addition, we can perform schema matching across versions. Note that we can plug in any existing profiling operators in this step.

- *Column-to-column correspondence.* Many algorithms [120] have been proposed to address the schema matching problem, either by exploiting the information from the schema (e.g., column name and column type) or content. In particular, we implement schema matching solely based on the column name and column type. That is, when both the column name and column type are the same, we claim that these two columns are matched.

- *Row-to-row correspondence.* There is existing work [121] on identifying all possible primary keys (PK) for each dataset. One natural method is to examine the uniqueness for every single column or combined columns—if the ratio between the cardinality and the size is above some threshold, e.g., 0.9 in our implementation, we say this column set forms a primary key. In our implementation, we perform PK detection on a small sampled dataset and constrain the number of columns in the PK to be no larger than three. If there exists no PK, Relic falls back to a bipartite matching problem as discussed in the extension section (Section 8.7).

Pre-clustering. The search space for an inferred lineage graph is huge. As a first step, we can incorporate some domain knowledge to pre-cluster the datasets based on some heuristic rules, and narrow down the search space. For instance, we can first pre-cluster different versions based on their primary key, infer the lineage graph within each cluster, and then connect across clusters. The reason behind pre-clustering datasets with the same PK is that if two versions have different PKs, they cannot be derived from each other via point-preserving operations. Recall that point-preserving operations are the operations with one-to-one mappings from the base version to the derived version as discussed in Section 8.2. Alternatively, we can also pre-cluster versions based on the schema. The assumption is that content modification happens more often than schema changes. Similar to that in Step (a), other pre-clustering mechanisms can also be plugged into our workflow.

Edge Inference. This is the most crucial step in Relic. Essentially, the goal of lineage inference is to identify the derivation edges and include them in the lineage graph. Thus, we first need a mechanism to describe and quantify the relationship between each version pair. Afterwards, we need to develop a mechanism for selecting edges as our inferred derivation edges E'. To address the first challenge, we propose a cell-based delta metric to quantify the differences between version pairs. For the second challenge, our intuition is based on the minimum description length (MDL) principle by Occam's Razer: a connected lineage graph with a smaller delta score is more likely to capture the derivation relationship correctly. We will discuss the details in Section 8.4.

Structural Explanation. In addition to the delta score for each selected edge E', it is desirable to also present users with the derivation operations for better interpretation. On one hand, Relic targets general transformations instead of any DML (data manipulation language, e.g., SQL) specific derivation operations. This indicates that our inferred derivation operations must be general enough to capture all kinds of different transformations. On the other hand, existing work [118] has demonstrated that it is computationally hard to reverse engineer SQL statements even in a restricted operator space. Thus, we propose a general but simple instruction space with INSERT, DELETE, and UPDATE operators with COLUMN, ROW operands. We will dive into the structural explanation step in Section 8.5.

8.4 EDGES INFERENCE

Given a set of artifacts V, our goal is to construct a lineage graph G' connecting these artifacts. Alternatively, we can frame the lineage inference problem as an *edge selection* problem – which edges among all pairwise edges should be selected in the inferred lineage graph $G' = (V, E')$. As a first step, we should "profile" all pairwise edges (v_i, v_j), where $v_i, v_j \in V$. By profiling, we aim to describe the relationship for each version pair v_i and v_j. Specifically, we propose a cell-level delta metric and compute the delta for all version pairs. Next, we select the edges based on the minimum description length principle.

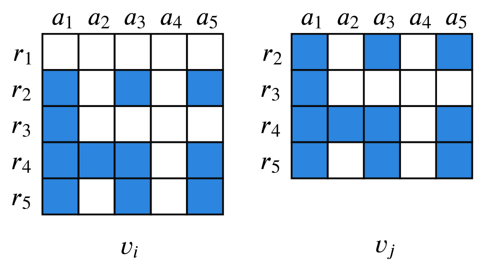
8.4.1 All Pair Delta Computation

For each version pair v_i and v_j, we would like to describe the derivation relationship between them. Practically, a new dataset version can be derived from the base version in various ways such as manual cell-level edits, DML like SQL, and imperative programs. However, post-facts inference of these types of changes are not always apparent and might be even indistinguishable. For example, an analyst engaged in the process of data cleaning might manually correct certain values of a field in a table, or might encode the changes in a single SQL query that updates the values of a field based on some matching predicates. This action could be encoded as individual cell-level changes, or as a single SQL statement. Thus, instead of reverse-engineering the original transformation commands, we propose to represent the derivation relationship in a general and natural form that can cover various transformation approaches. In particular, we use the content difference between v_i and v_j at the cell level, titled cell-level delta, to describe this relationship and measure the difference between two versions.

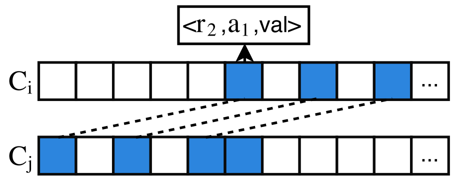
Cell-level Delta Metric. A tabular dataset consists of rows and columns, where each row and column are formed by grouping cells horizontally or vertically. For this discussion, we assume that we have obtained column-to-column correspondence and row-to-row correspondence across dataset versions as described in the profiling step (a) in Section 8.3. A cell is the smallest unit of compound in such 2-dimensional tabular dataset. Thus, we can represent the delta by looking at the modified cells from dataset v_i to v_j. Each cell c_i is uniquely identified by the triple <rowID, columnID, value>, where rowID and columnID is the consolidated row ID and column ID after Step (a) respectively, and value is what is inside this cell. Let C_i be the set of cells in dataset v_i. As shown in Figure 8.3(b), cell $<r_2, a_1$, val> refers to the cell located in r_2 and a_1 of dataset v_i in Figure 8.3(a). The blue cells in Figure 8.3(a) are the common cells between v_i and v_j. The cell-level similarity is defined as the Jaccard similarity between C_i and C_j, while the cell-level delta $\delta_c(v_i, v_j)$ is defined as the complement of the cell-level similarity. Formally, we can compute $\delta_c(v_i, v_j)$ as in Equation 8.1, where $|C_i \cap C_j|$ is the number of common cells between v_i and v_j and $|C_i \cup C_j|$ is the number of total cells in v_i and v_j.

$$\delta_c(v_i, v_j) = 1 - \frac{|C_i \cap C_j|}{|C_i \cup C_j|} \tag{8.1}$$

Computing cell-level deltas are straight-forward. Given two versions v_i and v_j, we first build a hash table with all cells in v_i, and then probe cells in v_j using the built hash table. Thus, the time complexity for computing the cell-level delta is $O(|v_i| + |v_j|)$, where $|v_i|$ is the number of cells in v_i, i.e., $|C_i|$.



(a) Two Dataset Versions



(b) Corresponding Cell Set

Figure 8.3: Cell-level Delta

Justification of Cell-level Delta. Next, we will analytically justify why the cell-level delta is chosen as the delta metric to represent the derivation relationship between two versions.

- *General.* Cell-level delta is able to express the content difference generated by any general transformations, ranging from manual edits, SQL queries, to programs.
- *Intuitive.* Cell-level delta is intuitive to understand and easy to calculate.
- *Informative.* Since a cell is the finest unit in a tabular dataset, cell-level delta can capture all content changes generated by any point-preserving operations[2] without information loss.

On the other hand, if we measure the delta at the column-level by treating each column as a set of values [122, 123], it will be difficult to differentiate versions with similar column domains, thus incurring information loss. For instance, if we apply SAMPLING on the base version, the derived version is likely to share the same domain as the base version for columns with low cardinality. As a result, these two versions are indistinguishable from each other in terms of the column-level delta. In Section 8.8, we also experimentally show that a cell-level delta can achieve better performance in reverse-engineering the lineage graph than column-level delta.

8.4.2 Edge Selection

After computing cell-level delta for all version pairs, the next step is to select edges E' to be included in the inferred lineage graph G'. One naive way is to set a threshold on $\delta_c(v_i, v_j)$ and include all edges that have small delta scores in E'. However, there exists some derivation operations with very small $\delta_c(v_i, v_j)$. Using a naive threshold-based approach

[2]We handle non-point-preserving operations in the post-processing step as discussed in Section 8.7

cannot capture such derivation edges. As a concrete example, SAMPLING and FEATURE SELECTION typically have small delta scores. Instead, we propose to adopt the minimum description length (MDL) principle. The high-level idea is that versions are likely to be derived from versions with similar content and the lineage graph G should have small overall edit distance (i.e., the sum of delta scores) along all edges.

If the lineage graph G is a tree, Problem 8.1 is equivalent to finding the minimum spanning tree based on the MDL principle. Note that such simple tree-based lineage cases are quite common in practice, since most point-preserving operators involve only one base dataset except UNION. When we generalize to the lineage graph case, we can first construct a lineage tree in G and then identify missing edges during post-processing. That is, for versions $v \in V$ with multiple in-coming edges, as a first step RELIC will only identify one base version and leave the others for post-processing.

Lemma 8.1 (Minimum Spanning Tree)**.** *Following the MDL principle, lineage inference is equivalent to finding the minimum spanning tree in a complete graph with node set V, where each edge weight is quantified by the cell-level delta.*

Remark 8.1. If there are multiple clusters after the pre-clustering step (b) in Section 8.3, we can first construct a minimum spanning tree within each cluster; next we treat each cluster as a supernode and construct a minimum spanning tree connecting these supernodes. Furthermore, if any edge is with delta score equalling one (or larger than some user-defined threshold), we exclude that edge from our inferred lineage graph G'.

8.5 STRUCTURAL EXPLANATION

When presenting the inferred lineage graph G' to the user, it is preferable to not just provide the cell-level delta score, but also some explanations along with each edge. However, cell-level instructions can be very verbose due to the fine-grained nature of cells. For instance, “deleting a feature” would correspond to a bunch of “cell deletion”, when using cell-level instructions. Instead, we propose to move to a higher level of abstraction and describe the derivation at the granularity of row and column. We call this *structural explanation.* On the one hand, structural explanation is more succinct and can help reveal some structural or semantic differences between the two datasets. On the other hand, finding the appropriate structural explanation is not as easy as computing cell-level delta. However, compared to expressing the derivation in SQL [124], our proposed structural explanation is more computationally tractable [118].

Structural Explanation. Specifically, from our standpoint, a structural explanation is informally defined to be a sequence of operations that can derive v_j from v_i and has the *smallest* number of operations, where the operation space is {ADD, UPDATE, DELETE} $\times$ {COLUMN, ROW}, i.e., the Cartesian product of the operators and operands. As we can see, the represented operators, i.e., {ADD, UPDATE, DELETE}, are very simple, intuitive, and interpretable. This is analogous to the concept of "edit distance" in string matching. These operations are general enough to capture the differences from any manual edits, data manipulation language operations (DML), or programs. Furthermore, such operators are easy to understand and convey the modification semantics at a high level. In the following, we additionally argue that this designed operation space is very suitable for data science workload. During an iterative data science process, one dataset is evolved from another by applying a combination of data transformation and feature engineering operators. Our column-wise operations, i.e., {(ADD, UPDATE, DELETE) COLUMN}, can represent most feature engineering processes including feature augmentation, encoding, normalization, and selection; while the row-wise operations, i.e., {(ADD, UPDATE, DELETE) ROW}, can represent most data transformation processes including cleaning, sampling, outlier removal, and imputation.

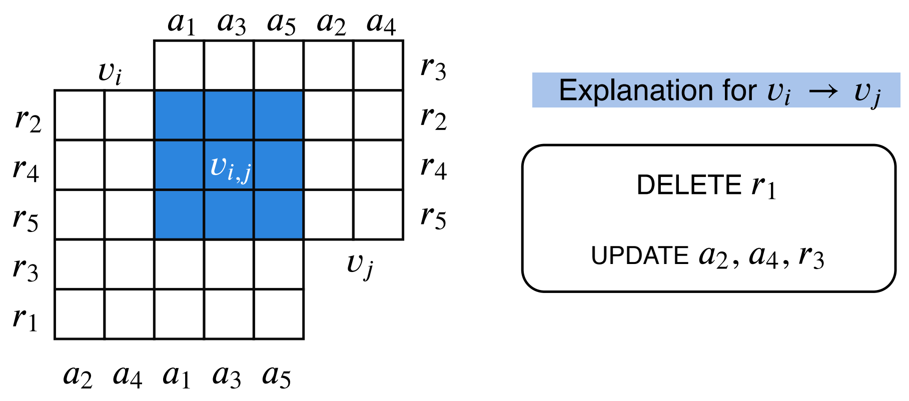


Figure 8.4: Structural Explanation in Figure 8.3

Deriving a structural explanation is not as straightforward as deriving a cell level delta, due to the mix of both column and row operations. Naively, we can represent the delta using only row operations, or using only column operations. However, such explanations may not have the smallest number of operations. Next, we will formally define the structural explanation Δ_R. Since ADD and DELETE operations can be uniquely derived from the row-to-row and column-to-column mappings for each dataset pair, we focus on minimizing the UPDATE operations. As a result, we can simply consider two datasets v_i and v_j with the same set of global column and row IDs. Let $M(v_i, v_j)$ be the indicator matrix representing changes between v_i and v_j, where $M_{r,c}(v_i, v_j) = 1$ if the cell located in row ID r and column ID c is the same in v_i and v_j, otherwise $M_{r,c}(v_i, v_j) = 0$. Next, let R_u and A_u be the selected records and attributes in the UPDATE operations. Due to the imbalance between the number of rows and columns, we further allow different weights for the row and column operations, denoted by w_r and w_c respectively. We formalize the structural explanation discover problem as Problem 8.2. Our goal is to find R_u and A_u such that the weighted sum of the update operations is minimized, as shown in Equation 8.2. The constraint in Equation 8.2 ensures that v_j can be derived from v_i with the sequence of UPDATE operations on R_u and A_u.

Problem 8.2 (Structural Explanation)**.** *Identify rows R_u and columns A_u such that v_j can be derived from v_i with commands* {UPDATE Columns A_u and Rows R_u} *and the weighted sum is minimized. Mathmatically,*

$$\begin{aligned} \delta_u(v_i, v_j) &= \min_{R_u, A_u} \Big(\sum_{r \in R_u} w_r + \sum_{c \in A_u} w_c\Big) \\ s.t.\ & M_{r,c}(v_i, v_j) = 1\ \forall r \notin R_u, c \notin A_u \end{aligned} \tag{8.2}$$

Exact Algorithm. Recall that in Equation 8.2, R_u and A_u are the selected records and attributes in the UPDATE operation. Correspondingly, we denote the unmodified rows and columns as $\bar{R_u}$ and $\bar{A_u}$, i.e., $\bar{R_u} = R \setminus R_u$ and $\bar{A_u} = A \setminus A_u$, where R and A denote the common records and columns in v_i and v_j, respectively. Replace R_u and A_u with $R - \bar{R_u}$ and $A - \bar{A_u}$ in Equation 8.2, we can then obtain the equivalent optimization formulation in Equation 8.3, where the first term $(\sum_{r \in R} w_r + \sum_{c \in A} w_c)$ is a constant.

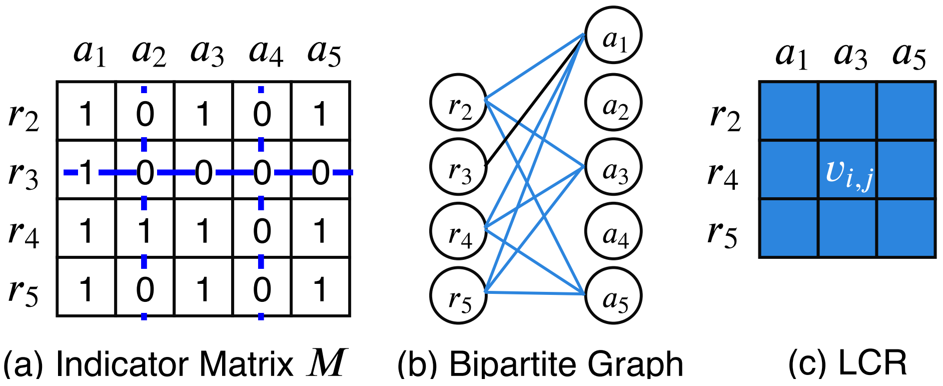
$$\begin{aligned} \delta_u(v_i, v_j) &= \Big(\sum_{r \in R} w_r + \sum_{c \in A} w_c\Big) - \max_{\bar{R_u}, \bar{A_u}} \Big(\sum_{r \in \bar{R_u}} w_r + \sum_{c \in \bar{A_u}} w_c\Big) \\ \text{s.t. } & M_{r,c}(v_i, v_j) = 1\ \ \forall r \in \bar{R_u}, c \in \bar{A_u} \end{aligned} \tag{8.3}$$

Essentially, the problem in Equation 8.3 is to find the row set $\bar{R_u}$ and column set $\bar{A_u}$ with the largest weight, i.e., $(\sum_{r \in \bar{R_u}} w_r + \sum_{c \in \bar{A_u}} w_c)$, under the constraint that each entry in the common rectangle, defined by the row set $\bar{R_u}$ and column set $\bar{A_u}$, has value $M_{r,c} = 1$. We term this problem as *largest common rectangle w.r.t. weighted perimeter*, *LCRP* for short. And we use $|LCRP(v_i, v_j)|$ to denote the weighted sum of the selected rows and columns in LCRP problem. Hence, we can rewrite Equation 8.3 into Equation 8.4

$$\delta_u(v_i, v_j) = \Big(\sum_{r \in R} w_r + \sum_{c \in A} w_c\Big) - |LCRP(v_i, v_j)| \tag{8.4}$$

Let us illustrate LCRP by continuing the example in Figure 8.3, where $w_c = w_r = 1$. First, Figure 8.5(a) is the indicator matrix M for v_i and v_j in Figure 8.3(a). The common rows between v_i and v_j are $R = \{r_2, r_3, r_4, r_5\}$, and the common columns are $A = \{a_1, a_2, a_3, a_4, a_5\}$.

As depicted in Figure 8.5(c), the selected row set $\bar{R_u} = \{r_2, r_4, r_5\}$ and column set $\bar{A_u} = \{a_1, a_3, r_5\}$ form a common rectangle between v_i and v_j, i.e., $M_{r,c} = 1, \forall r \in \bar{R_u}, c \in \bar{A_u}$, and have the largest weighted sum. Thus, Figure 8.5(c) is the LCRP for Figure 8.3(a) with $|LCRP(v_i, v_j)| = 6$.



$$\begin{array}{c|ccccc} & a_1 & a_2 & a_3 & a_4 & a_5 \\ \hline r_2 & 1 & 0 & 1 & 0 & 1 \\ r_3 & 1 & 0 & 0 & 0 & 0 \\ r_4 & 1 & 1 & 1 & 0 & 1 \\ r_5 & 1 & 0 & 1 & 0 & 1 \end{array}$$

Figure 8.5: Indicator Matrix and Bipartite Graph in Figure 8.3

Next, we show that LCRP problem is equivalent to maximum node biclique (MNB) problem in the corresponding bipartite graph. First, we map the indicator matrix M to a bipartite graph $\mathcal{G} = (\mathcal{V}_1, \mathcal{V}_2, \mathcal{E})$, where each row and column correspond to a vertex in $\mathcal{V}_1$ and $\mathcal{V}_2$ respectively. When $M_{r,c} = 1$, there is an edge connecting the corresponding two vertices in $\mathcal{V}_1$ and $\mathcal{V}_2$. Figure 8.5(b) is the bipartite graph corresponding to the indicator matrix in Figure 8.5(a). We can see that each common rectangle with all 1s in the indicator matrix M corresponds to a biclique (blue edges) in the bipartite graph $\mathcal{G}$, and $|LCRP(v_i, v_j)|$ is the same as the weighted node sum in the corresponding biclique. Note that the common rectangle does not require the selected rows or columns to be with continuous IDs, i.e., row and column reordering are allowed in M.

Lemma 8.2 (Maximum node biclique (MNB)). *Problem 8.2 is equivalent to the maximum node biclique (MNB) problem.*

Furthermore, MNB problem is known to be equivalent to the maximum independent set problem in a complemented bipartite graph [125], which can be solved using standard s-t min-cut algorithms [126]. The Ford-Fulkerson algorithm [127] can be used to solve the max-flow problem, and hence min-cut problem based on the max-flow min-cut theorem. In the worst case, the Ford-Fulkerson algorithm takes $O(M \times f)$, where M is the number of edges in the bipartite graph and f is the value of maximal flow.

Lemma 8.3 (Min-cut max-flow Algorithm). *Problem 8.2 can be solved using existing min-cut max-flow algorithms, e.g., Ford-Fulkerson algorithm in polynomial time.*

8.6 ACCELERATING THE WORKFLOW

In the previous sections, we have introduced our proposed workflow Relic. Next, we will discuss how to make this end-to-end workflow faster. First, let us take a look at the execution in each step of the workflow.

- *Step (a).* First, for schema matching, since we only operate at the schema level (column name and type) without looking into the detailed content, the execution can be very fast. As for primary key detection, since we conduct the detection on a sampled dataset with a small size, the running time is essentially negligible.
- *Step (b).* Pre-clustering is based on metadata such as primary key and schema similarity. Thus, step (b) can be completed quickly.
- *Step (c).* This step consists of two parts: the all pair delta computation and edge selection. All pair delta computation is an obvious bottleneck since it involves comparing the contents for each version pair.
- *Step (d).* After obtaining the inferred lineage graph G', we aim to provide structural explanation for each edge $e' \in E'$ using the algorithm described in Section 8.5. This can also be time-consuming due to the expensive max-flow algorithm.

As discussed above, the running time in Step (a) and (b) is negligible. Step (c) involves delta computation for all version pairs, which is quadratic in the number of versions. This step is often the runtime bottleneck for the end-to-end workflow. In particular, if the memory is insufficient to hold all the versions, the I/O access would be time-consuming due to loading (and evicting) data to (from) memory. Thus, we propose to employ existing sketch-based techniques to compute deltas for all version pairs using a smaller footprint. In this way, we are able to hold all versions' sketches in memory, eliminating the expensive I/O time. Compared to Step (c), the runtime in Step (d) is linear in the number of versions. However, deriving a structural explanation is more expensive than calculating cell-level delta. Even though the structural explanation problem can be solved using existing max-flow algorithms, it can be quite expensive in many scenarios. To reduce the runtime, we propose a greedy algorithm with a small time complexity.

Employing Sketching for Step (c). Sketching is a powerful technique for rapidly approximating various statistics (e.g., Jaccard Similarity, Cardinality) [128]. Instead of answering these statistical queries over the original large datasets, we can instead work on sketches that are much smaller, resulting in faster query responses, with a potential loss of accuracy.

Different sketch techniques have been developed in the recent past, e.g., frequency-based sketches [128] and sketches for distinct value queries [128]. In this section, we will discuss how to use an existing sketch technique, i.e., *minhash* [129], for appropriate cell-level delta estimation.

As discussed in Section 8.4, given a version v_i, we can decompose it into its cell set C_i. Then, computing cell-level delta for each version pair v_i and v_j can be essentially transformed to calculating the jaccard similarity between C_i and C_j. Set similarity has been well studied, and minhash is a well-known sketch for approximating Jaccard similarity $JS(C_i, C_j) = \frac{|C_i \cap C_j|}{|C_i \cup C_j|}$. Given a hash function[3] h_k and a set C_i, the minhash is defined as $h_k(C_i) = \min_{c \in C_i} h_k(c)$. An important property of minhash is that the probability that the minhash function produces the same value for C_i and C_j equals the Jaccard similarity between C_i and C_j, i.e., $Pr(h_k(C_i) = h_k(C_j)) = JS(C_i, C_j)$. Based on this property, we can construct a *minhash sketch* of size n for each set C_i, represented by a vector $[h_1(C_i), h_2(C_i), \cdots, h_K(C_i)]$, where each h_k is a hash function. The Jaccard similarity can thus be estimated as described in Equation 8.5.

$$\hat{JS}(C_i, C_j) = \frac{\sum_{k=1}^{K} \mathbb{1}(h_k(C_i) = h_k(C_j))}{n} \tag{8.5}$$

However, applying K hash function to each cell is prohibitive, especially when K is large. BottomK sketch [130] is one technique that has been proposed to tackle this issue, where only one hash function is applied to all cells C_i and the cells with the smallest K hash values are selected as the sketch S_i. We can subsequently estimate the Jaccard similarity using $JS(S_i, S_j)$ in the sketch space.

Greedy Algorithm for Step (d). We can transform Problem 8.2 into a set cover problem, where each column c (or row r) corresponds to a set with its respective weight w_c (or w_r) with this set containing all modified cells in this column (or row). The task is to select sets such that the union of these sets covers all modified cells. In each iteration, we can greedily select the set that covers the largest fraction of modified elements. Such a greedy algorithm leads to a 2-approximation to the exact solution, while the running time is $(m+n)\log(m+n)+\Gamma$, where $(m+n)$ is the number of rows and columns and Γ is the number of modified cells. Compared to the exact solution, the greedy algorithm is, in general, less expensive.

[3]We assume each hash function is corresponding to a random permutation of rows.

8.7 EXTENSIONS

In previous sections, we have talked about how Relic works under certain assumptions. Even though these assumptions hold in most scenarios, in the following we will discuss how to handle the lineage inference problem when relaxing these assumptions.

Row-to-Row Mapping. In Step (a) of Relic, we propose to perform Primary Key detection to obtain row-ro-row mappings across versions. When there exists no PK in each version, we can fall back to other row mapping approaches. Specifically, given two versions v_i and v_j, we first construct a bipartite graph with all records in v_i on one side and all records in v_j on the other side. The edge weight between node $r_k(v_i)$ and node $r_l(v_j)$ is the number of common cells between these two records, where $r_k(v_i)$ and $r_l(v_j)$ refer to record r_k in v_i and record r_l in v_j, respectively. The row-to-row mapping problem between v_i and v_j is essentially a maximum bipartite matching problem in the constructed bipartite graph. This method can handle all general cases, but is much more expensive compared to simple PK detection.

Non-Point-preserving Operations. Even though most transformation operations fall into the category of point-preserving ones, where there is a one-to-one correspondence between rows, there exist two common non-point-preserving operations, i.e., natural join and aggregates. There is existing work [131] aiming to infer join relationships between input and output tables. The high-level idea is that there are (fuzzy) column containment relationships between the join inputs and output. Thus, we can infer the derivation edge with join operators using this column containment [132] property.

8.8 PRELIMINARY EXPERIMENTAL EVALUATION

We experimentally evaluate our proposed end-to-end workflow Relic in terms of both effectiveness and efficiency. In the following, we will study how Relic performs in reconstructing the true lineage graph, followed by the running time comparison with and without using sketching.

8.8.1 Effectiveness of Relic

Dataset. We examined a large number of Jupyter notebooks and extracted the most frequently used operators in Pandas data processing package [133] – ASSIGN, ILOC, NLARGEST, NSMALLEST, SAMPLE, SORT_INDEX, SORT_VALUE, DROP_COLUMNS, DROP_ROWS,

$\lvert V\rvert$	10	10	10	20	20	20	40	40	40	80	80	80
α	1	2	3	1	2	3	1	2	3	1	2	3
RELIC-β	0.86	0.97	0.97	0.81	0.93	0.96	0.81	0.94	0.91	0.80	0.91	0.90
BASELINE-β	0.80	0.73	0.82	0.75	0.81	0.85	0.74	0.83	0.73	0.72	0.82	0.81
RELIC-θ	0.78	0.94	0.94	0.70	0.87	0.92	0.69	0.90	0.83	0.65	0.75	0.81
BASELINE-θ	0.71	0.60	0.72	0.62	0.70	0.75	0.60	0.72	0.58	0.59	0.70	0.69

Table 8.1: Effectiveness of the Inferred Lineage Graph

ADD_ROWS. We then implemented a synthetic dataset generator using these operators. When deriving a new dataset version, we first randomly select a base version and apply a sequence of operations on this base version. The number of operations before each materialization is a user-defined parameter α. The initial dataset had 10000 records and 20 columns.

Baseline. As discussed in Section 8.4, an alternative delta metric is a column-level one as described in an existing paper [122]. This approach works as follows: given two versions v_i and v_j, we first calculate the Jaccard similarity for each corresponding column in v_i and v_j, then sum up the Jaccard similarities across all such column pairs and divide it by the total number of columns in $A_i \cup A_j$, where A_i corresponds to the columns in v_i. We replace our cell-level delta with this aggregated column-wise delta, and use it as our baseline.

Measure. As discussed in Section 8.2, we can measure the quality of G' using the Jaccard similarity $\theta_{G'}$ between E and E'. Alternatively, since both RELIC and the BASELINE return a minimum spanning tree with the same edge size, we can also measure the quality of G' by looking at the percentage of common edges between E' and E, i.e., $\beta_{G'} = \frac{|E \cap E'|}{|E|}$.

For each configuration in Table 8.1, we randomly generate 10 synthetic repositories with $|V|$ versions. We report the average score for RELIC and BASELINE in terms of both β and θ as discussed above. First, we observe that RELIC can recover the lineage graph effectively with an average of β around 0.9, indicating that 90 percent of the derivation edges are recovered. This is a very promising result. Furthermore, we can see that RELIC outperforms BASELINE for all configurations. This is mainly because BASELINE can only capture the derivation difference for some operations, but not all. Last but not least, both RELIC and BASELINE perform worse when $\alpha = 1$ compared to $\alpha = 2$ and 3. The reason is that the versions tend to be very similar and indistinguishable if we materialize the derived version after a single operation. For instance, with operation SORT_INDEX and SORT_VALUE, the base version and the derived version are essentially the same and essentially distinguishable.

		trial1	trial2	trial3	trial4	trial5	trial6	trial7	trial8	trial9	trial10
v40-k100	γ	2.12	2.57	1.94	2.02	2.47	1.15	2.65	2.63	4.91	2.15
v40-k100	β	0.79	0.92	0.82	0.74	0.79	0.74	0.82	0.64	0.79	0.77
v40-k400	γ	0.62	2.52	3.37	3.73	0.80	2.32	3.13	3.84	0.83	1.06
v40-k400	β	0.77	0.77	0.77	0.82	0.77	0.90	0.72	0.85	0.87	0.87

Table 8.2: Effectiveness and Efficiency Comparison With/Without Sketch

8.8.2 Efficiency of Relic

Next, we will illustrate how sketch techniques affect Relic in terms of both efficiency and effectiveness. We run experiments with and without the BottomK sketch on synthetic datasets with 10000 records and 20 columns in the base version, and 40 versions for 10 trials. We measure the speedup ratio γ between Relic without sketch and with sketch along with β for the inferred lineage graph using the sketch, and report the numbers in Table 8.2. For instance, v40-k100 means that there are 40 versions in total, and the sketch size K is 100. On average, Relic with sketch size $K = 100$ achieves 2.5× speedup and β is 0.78; when $K = 400$ Relic achieves 2.2× speedup and β is 0.81 on average. First, we can see that the inferred lineage graph with the sketch has a slightly worse performance for β in terms of the number of common edges with G. However, an average β of around 0.8 is not bad in practice. We also note that in some scenarios Relic with sketch even has slower running time compared to Relic without using the sketch. This is because using sketch typically incurs overhead in hashing each cell, and the average size of all versions can be similar to the sketch size.

Our preliminary experiments indicate the promise of the end-to-end Relic workflow in lineage inference. As next steps, we aim to conduct more experiments while varying the number of versions, records, and sketch size. In addition, we plan to explore some real workflows to evaluate the fine-grained behavior of Relic via real case studies.

CHAPTER 9: CONCLUSION AND FUTURE WORK

In this thesis, we developed OrpheusDB for effective structured data versioning, and relaxed certain assumptions, one at a time, made in OrpheusDB, as a step towards general-purpose versioning. In the following, we will discuss some deficiencies in our work and potential future directions.

Query Optimization. In both OrpheusDB (Chapter 3-5) and the generalized storage engine (Chapter 7), we studied the problem of balancing the storage size and the checkout latency. Even though checkout is a fundamental and important operation, it remains to be seen what are the other operations or queries that are most useful for end-users and how to further optimize for these queries. As readers may remember, in Chapter 3 and 6, we do list some potentially useful queries that can help users reason across versions. However, those queries are mostly envisioned or summarized from informal conversations, biased towards a database audience instead of real end-users. Conduct an extensive survey among data scientists to learn the most frequently asked queries when reasoning across versions would be invaluable. With that information in hand, we can then work on reducing the latency for those queries, e.g., via physical layout design and query optimization.

Diff Primitives. OrpheusDB can support Diff operators, i.e., contrasting one version from the other. However, our current diffing mechanism is very simplistic. Specifically, we identify new and deleted records by examining the two given datasets—a record is deemed to be unmodified only when it exists in both datasets. As a result, a "column normalization" transformation would be interpreted as "deleting all records from the original dataset and inserting all normalized records into the new dataset", which simply does not make any sense. As readers may have noticed, we can potentially improve this Diff command by incorporating the structural explanation as discussed in Chapter 8. However, the underlying question still remains, given two datasets, how should we define the meaning of diff? It seems that different diffing primitives can be meaningful for different scenarios and there is no universal diffing mechanism. Here are a few options:

- [Distributional diff] A common practice during machine learning model serving is to validate each column's distribution alignment with the training data. This is to detect data drift and thus avoid model performance degradation. In such scenarios, distribution-wise differences between the training and serving dataset are desired.
- [Row Diff] Say a dataset is generated from some upstream group of individuals, and say the upstream group decides to relax some of their filter conditions pipeline, re-

sulting in a new dataset version. To analyze the impact from this upstream group, the downstream group would like to understand how the additional data included in the new dataset version looks like. Thus, in this scenario, a summarized row-wise difference is a suitable one.

A natural next step would be to interview various stakeholders, summarize the commonly desired diffing primitives, and provide an efficient diff toolkit catering to users. After all, diff is indeed the most fundamental operation when reasoning between versions.

Towards a Unified System for Data, Code, and Model Versioning. In this thesis, we focused on effective data versioning. However, during the iterative data science workflow, not only does data change across different versions, but also code and models. Additionally, there are strong dependencies between the data, code, and models. Specifically, the data along with code produces a machine learning model. When the model performance is not satisfactory, data scientists would likely to incorporate more data, try different data preprocessing and feature engineering operators, or seek other machine learning algorithms. This would result in a new data version as well as a new code version, which consequently generates a new model. This is a laborious process via trial-and-error and such versioned artifacts are poorly managed in the wild. As a consequence, it is difficult to recall which combinations of data input and machine learning configurations have been tried; to reason about the model performance's relationship with data and code; to study whether the fact that a particular data slice's performance has degraded is due to the incorporation of more data in this slice or the introduction of a different machine learning algorithm. Thus, there is a pressing need for a system to help manage the versioned data, code, and model in a unified manner.

To build such a system, we first need to study the issues data scientists encounter in dealing with various versions of data, code, and models. With these needs in mind, we can then build a unified system for managing data, code, and model. Another thing to keep in mind is that users are reluctant to change their behavior, and thus we need to find a method that is most non-intrusive for users.

REFERENCES

[1] S. Huang, L. Xu, J. Liu, A. J. Elmore, and A. Parameswaran, “Orpheus db: bolt-on versioning for relational databases,” *Proceedings of the VLDB Endowment*, vol. 10, no. 10, pp. 1130–1141, 2017.

[2] L. Xu, S. Huang, S. Hui, A. J. Elmore, and A. Parameswaran, “Orpheusdb: A lightweight approach to relational dataset versioning,” in *Proceedings of the 2017 ACM International Conference on Management of Data*. ACM, 2017, pp. 1655–1658.

[3] A. Chavan et al., “Towards a unified query language for provenance and versioning,” in *TaPP*, 2015.

[4] S. Bhattacherjee et al., “Principles of dataset versioning: Exploring the recreation/storage tradeoff,” *VLDB*, vol. 8, no. 12, pp. 1346–1357, 2015.

[5] http://comments.gmane.org/gmane.comp.version-control.git/189776.

[6] https://git-annex.branchable.com/.

[7] http://caca.zoy.org/wiki/git-bigfiles.

[8] “Liquibase,” http://www.liquibase.org/.

[9] “dbv,” https://dbv.vizuina.com/.

[10] “Dat,” http://datproject.org/.

[11] “Mode,” https://about.modeanalytics.com/.

[12] I. Ahn et al., “Performance evaluation of a temporal database management system,” in *ACM SIGMOD Record*, vol. 15, no. 2. ACM, 1986, pp. 96–107.

[13] R. Snodgrass and I. Ahn, “A taxonomy of time databases,” *ACM Sigmod Record*, vol. 14, no. 4, pp. 236–246, 1985.

[14] R. T. Snodgrass et al., “Tsql2 language specification,” *Sigmod Record*, vol. 23, no. 1, pp. 65–86, 1994.

[15] C. S. Jensen and R. T. Snodgrass, “Temporal data management,” *IEEE Transactions on Knowledge and Data Engineering*, vol. 11, no. 1, pp. 36–44, 1999.

[16] G. Ozsoyoglu et al., “Temporal and real-time databases: A survey,” *TKDE*, vol. 7, no. 4.

[17] A. U. Tansel et al., *Temporal databases: theory, design, and implementation.* Benjamin-Cummings Publishing Co., Inc., 1993.

[18] K. Torp, C. S. Jensen, and R. T. Snodgrass, “Stratum approaches to temporal dbms implementation,” in *IDEAS’98.* IEEE, 1998, pp. 4–13.

[19] C. X. Chen, J. Kong, and C. Zaniolo, "Design and implementation of a temporal extension of sql," in *Data Engineering*, 2003, pp. 689–691.

[20] C. M. Saracco, M. Nicola, and L. Gandhi, "A matter of time: Temporal data management in db2 for z," *IBM Corporation, New York*, 2010.

[21] M. Al-Kateb et al., "Temporal query processing in teradata," in *EDBT'13*.

[22] M. Kaufmann et al., "Timeline index: A unified data structure for processing queries on temporal data in sap hana," in *SIGMOD 2013*, pp. 1173–1184.

[23] F. Färber, N. May, W. Lehner, P. Große, I. Müller, H. Rauhe, and J. Dees, "The sap hana database–an architecture overview." *IEEE Data Eng. Bull.*, vol. 35, no. 1, pp. 28–33, 2012.

[24] M. Kaufmann et al., "Benchmarking bitemporal database systems: Ready for the future or stuck in the past?" in *EDBT*, 2014, pp. 738–749.

[25] K. Kulkarni and J.-E. Michels, "Temporal features in sql: 2011," *ACM Sigmod Record*, vol. 41, no. 3, pp. 34–43, 2012.

[26] G. M. Landau et al., "Historical queries along multiple lines of time evolution," *The VLDB Journal*, vol. 4, no. 4, pp. 703–726, 1995.

[27] B. J. Salzberg and D. B. Lomet, *Branched and Temporal Index Structures.* College of Computer Science, Northeastern University, 1995.

[28] S. Lanka and E. Mays, *Fully persistent B+-trees.* ACM, 1991, vol. 20, no. 2.

[29] L. Jiang, B. Salzberg, D. B. Lomet, and M. B. García, "The bt-tree: A branched and temporal access method." in *VLDB*, 2000, pp. 451–460.

[30] A. Bhardwaj et al., "Datahub: Collaborative data science & dataset version management at scale," *CIDR*, 2015.

[31] M. Maddox et al., "Decibel: The relational dataset branching system," *VLDB*, vol. 9, no. 9.

[32] S. Quinlan and S. Dorward, "Venti: A new approach to archival storage," in *FAST*, 2002.

[33] B. Zhu, K. Li, and R. Patterson, "Avoiding the disk bottleneck in the data domain deduplication file system," in *FAST*, 2008.

[34] F. Douglis and A. Iyengar, "Application-specific delta-encoding via resemblance detection," in *USENIX ATC*, 2003.

[35] Z. Ouyang, N. Memon, T. Suel, and D. Trendafilov, "Cluster-based delta compression of a collection of files," in *WISE*, 2002.

[36] R. Burns and D. Long, "In-place reconstruction of delta compressed files," in *PODC*, 1998.

[37] M. C. Chan and T. Y. Woo, "Cache-based compaction: A new technique for optimizing web transfer," in *INFOCOM*, 1999.

[38] P. Kulkarni, F. Douglis, J. D. LaVoie, and J. M. Tracey, "Redundancy elimination within large collections of files," in *USENIX ATC*, 2004.

[39] J. Paulo and J. Pereira, "A survey and classification of storage deduplication systems," *ACM Comput. Surv.*, vol. 47, no. 1, pp. 11:1–11:30, June 2014.

[40] Y. Ahmad et al., "Dbtoaster: Higher-order delta processing for dynamic, frequently fresh views," *VLDB Endowment*, vol. 5, no. 10, pp. 968–979, 2012.

[41] P. Buneman et al., "Archiving scientific data," *TODS*, vol. 29, no. 1, pp. 2–42, 2004.

[42] P. Buneman, S. Khanna, and W. C. Tan, "Why and Where: A Characterization of Data Provenance," in *ICDT*, 2001, pp. 316–330.

[43] H. Miao, A. Chavan, and A. Deshpande, "Provdb: Lifecycle management of collaborative analysis workflows," in *Proceedings of the 2nd Workshop on Human-In-the-Loop Data Analytics.* ACM, 2017, p. 7.

[44] A. Halevy, F. Korn, N. F. Noy, C. Olston, N. Polyzotis, S. Roy, and S. E. Whang, "Goods: Organizing google's datasets," in *Proceedings of the 2016 International Conference on Management of Data.* ACM, 2016, pp. 795–806.

[45] L. Xu, S. Huang, S. Hui, A. Elmore, and A. Parameswaran, "OrpheusDB: A lightweight approach to relational dataset versioning," in *SIGMOD'17 Demo.*

[46] R. J. Miller et al., "The clio project: managing heterogeneity," *SIGMOD Record*, vol. 30, no. 1, pp. 78–83, 2001.

[47] K. Fisher, D. Walker, K. Q. Zhu, and P. White, "From dirt to shovels: fully automatic tool generation from ad hoc data," in *ACM SIGPLAN Notices*, vol. 43, no. 1. ACM, 2008, pp. 421–434.

[48] D. Szklarczyk et al., "The string database in 2011: functional interaction networks of proteins, globally integrated and scored," *Nucleic acids research.*

[49] G. O. Consortium et al., "Gene ontology consortium: going forward," *Nucleic acids research*, vol. 43, no. D1, pp. D1049–D1056, 2015.

[50] "PostgreSQL9.5," www.postgresql.org/docs/current/static/intarray.html.

[51] "DB2 9.7 array," https://www.ibm.com/support/knowledgecenter/SSEPGG_9.7.0/com.ibm.db2.luw.sql.ref.doc/doc/r0050497.html.

[52] "Array in MySql," https://dev.mysql.com/worklog/task/?id=2081.

[53] B. Salzberg and V. J. Tsotras, “Comparison of access methods for time-evolving data,” *ACM Computing Surveys (CSUR)*, vol. 31, no. 2, pp. 158–221, 1999.

[54] J. W. Lee, J. Loaiza, M. J. Stewart, W.-M. Hu, and W. H. Bridge Jr, “Flashback database,” Feb. 20 2007, uS Patent 7,181,476.

[55] D. Gao, S. Jensen, T. Snodgrass, and D. Soo, “Join operations in temporal databases,” *The VLDB Journal*, vol. 14, no. 1, pp. 2–29, 2005.

[56] S. Jain, D. Moritz, D. Halperin, B. Howe, and E. Lazowska, “Sqlshare: Results from a multi-year sql-as-a-service experiment,” in *Proceedings of the 2016 International Conference on Management of Data*. ACM, 2016, pp. 281–293.

[57] C. De Castro, F. Grandi, and M. R. Scalas, “On schema versioning in temporal databases,” in *Recent advances in temporal databases*, 1995, pp. 272–291.

[58] C. De Castro, F. Grandi, and M. R. Scalas, “Schema versioning for multitemporal relational databases,” *Information Systems*, vol. 22, no. 5, pp. 249–290, 1997.

[59] H. J. Moon et al., “Managing and querying transaction-time databases under schema evolution,” *VLDB*, 2008.

[60] H. J. Moon et al., “Scalable architecture and query optimization fortransaction-time dbs with evolving schemas,” in *SIGMOD 2010*.

[61] A. Quamar, A. Deshpande, and J. Lin, “Nscale: neighborhood-centric large-scale graph analytics in the cloud,” *The VLDB Journal*, pp. 1–26, 2014.

[62] G. Karypis and V. Kumar, “A fast and high quality multilevel scheme for partitioning irregular graphs,” *SISC*, vol. 20, no. 1, pp. 359–392, 1998.

[63] D.-R. Liu and S. Shekhar, “Partitioning similarity graphs: A framework for declustering problems,” *Information Systems*, vol. 21, no. 6, pp. 475–496, 1996.

[64] U. Feige, D. Peleg, and G. Kortsarz, “The dense k-subgraph problem,” *Algorithmica*, vol. 29, no. 3, pp. 410–421, 2001.

[65] G. Karypis et al., “Multilevel k-way hypergraph partitioning,” *VLSI design*, vol. 11, no. 3.

[66] K. A. Kumar et al., “Sword: workload-aware data placement and replica selection for cloud data management systems,” *The VLDB Journal*, vol. 23, no. 6.

[67] C. Date, “A critique of the SQL database language,” *ACM SIGMOD Record*, vol. 14, no. 3, pp. 8–54, 1984.

[68] M. Baker, “De novo genome assembly: what every biologist should know,” *Nature methods*, vol. 9, no. 4, pp. 333–337, 2012.

[69] S. Bhattacherjee, A. Chavan, S. Huang, A. Deshpande, and A. G. Parameswaran, "Principles of dataset versioning: Exploring the recreation/storage tradeoff," in *PVLDB*, 2015.

[70] M. Stonebraker, G. Held, E. Wong, and P. Kreps, "The design and implementation of INGRES," *ACM Transactions on Database Systems (TODS)*, vol. 1, no. 3, pp. 189–222, 1976.

[71] C. Zaniolo, "The database language GEM," in *ACM Sigmod Record*, vol. 13(4). ACM, 1983, pp. 207–218.

[72] P. T. Wood, "Query languages for graph databases," *SIGMOD Rec.*, vol. 41, no. 1, pp. 50–60, Apr. 2012.

[73] R. Snodgrass, "The temporal query language TQuel," *ACM Transactions on Database Systems (TODS)*, vol. 12, no. 2, pp. 247–298, 1987.

[74] H. F. Korth and M. A. Roth, *Query languages for nested relational databases.* Springer, 1989.

[75] J. Widom, "Trio: A System for Integrated Management of Data, Accuracy, and Lineage," in *CIDR*, 2005, pp. 262–276.

[76] A. Marinho, L. Murta, C. Werner, V. Braganholo, S. M. S. d. Cruz, E. Ogasawara, and M. Mattoso, "Provmanager: a provenance management system for scientific workflows," *Concurrency and Computation: Practice and Experience*, vol. 24, no. 13, pp. 1513–1530, 2012.

[77] L. Murta, V. Braganholo, F. Chirigati, D. Koop, and J. Freire, "noworkflow: Capturing and analyzing provenance of scripts," in *Provenance and Annotation of Data and Processes.* Springer, 2014, pp. 71–83.

[78] J. Kim, E. Deelman, Y. Gil, G. Mehta, and V. Ratnakar, "Provenance trails in the Wings/Pegasus system," *Concurrency and Computation: Practice and Experience*, vol. 20, no. 5, pp. 587–597, 2008.

[79] M. Wylot, P. Cudre-Mauroux, and P. Groth, "Executing provenance-enabled queries over web data," in *Proceedings of the 24th International Conference on World Wide Web.* International World Wide Web Conferences Steering Committee, 2015, pp. 1275–1285.

[80] M. K. Anand, S. Bowers, T. Mcphillips, and B. Ludäscher, "Exploring scientific workflow provenance using hybrid queries over nested data and lineage graphs," in *Scientific and Statistical Database Management.* Springer, 2009, pp. 237–254.

[81] M. K. Anand, S. Bowers, and B. Ludäscher, "Techniques for efficiently querying scientific workflow provenance graphs." in *EDBT*, vol. 10, 2010, pp. 287–298.

[82] D. A. Holland, U. J. Braun, D. Maclean, K.-K. Muniswamy-Reddy, and M. I. Seltzer, "Choosing a data model and query language for provenance," in *The 2nd International Provenance and Annotation Workshop.* Springer, 2008.

[83] G. Karvounarakis, Z. G. Ives, and V. Tannen, "Querying data provenance," in *Proceedings of the 2010 ACM SIGMOD International Conference on Management of data.* ACM, 2010, pp. 951–962.

[84] S. Bowers, "Scientific workflow, provenance, and data modeling challenges and approaches," *Journal on Data Semantics*, vol. 1, no. 1, pp. 19–30, 2012.

[85] S. B. Davidson and J. Freire, "Provenance and scientific workflows: challenges and opportunities," in *SIGMOD Conference*, 2008, pp. 1345–1350.

[86] T. J. Green, G. Karvounarakis, and V. Tannen, "Provenance semirings," in *Proceedings of the Twenty-sixth ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems*, ser. PODS '07. New York, NY, USA: ACM, 2007. [Online]. Available: http://doi.acm.org/10.1145/1265530.1265535 pp. 31–40.

[87] S. Bowers, T. M. McPhillips, and B. Ludäscher, "Provenance in collection-oriented scientific workflows," *Concurrency and Computation: Practice and Experience*, vol. 20, no. 5, pp. 519–529, 2008.

[88] J. Freire, D. Koop, E. Santos, and C. T. Silva, "Provenance for computational tasks: A survey," *Computing in Science & Engineering*, vol. 10, no. 3, pp. 11–21, 2008.

[89] R. E. Tarjan, "Finding optimum branchings," *Networks*, vol. 7, no. 1, pp. 25–35, 1977.

[90] G. Kortsarz and D. Peleg, "Approximating shallow-light trees," in *SODA*, 1997.

[91] J. Bar-Ilan, G. Kortsarz, and D. Peleg, "Generalized submodular cover problems and applications," *Theoretical Computer Science*, vol. 250, no. 1, pp. 179–200, 2001.

[92] M. Charikar, C. Chekuri, T.-y. Cheung, Z. Dai, A. Goel, S. Guha, and M. Li, "Approximation algorithms for directed steiner problems," *J. Alg.*, 1999.

[93] S. Khuller, B. Raghavachari, and N. Young, "Balancing minimum spanning trees and shortest-path trees," *Algorithmica*, vol. 14, no. 4, pp. 305–321, 1995.

[94] https://www.kernel.org/pub/software/scm/git/docs/technical/pack-heuristics.txt.

[95] http://edmonds-alg.sourceforge.net/.

[96] http://svn.apache.org/repos/asf/subversion/trunk/notes/fsfs.

[97] http://svnbook.red-bean.com/en/1.8/svn.reposadmin.maint.html#svn.reposadmin.maint.diskspace.fsfspacking.

[98] http://svn.apache.org/repos/asf/subversion/trunk/notes/fsfs-improvements.txt.

[99] http://www.xmailserver.org/xdiff-lib.html.

[100] http://svn.apache.org/repos/asf/subversion/trunk/notes/skip-deltas.

[101] J. MacDonald, "File system support for delta compression," Ph.D. dissertation, UC Berkeley, 2000.

[102] J. Hunt, K. Vo, and W. Tichy, "Delta algorithms: An empirical analysis," *ACM Trans. Softw. Eng. Methodol.*, 1998.

[103] D. Korn and K. Vo, "Engineering a differencing and compression data format," in *USENIX ATC*, 2002.

[104] D. Trendafilov, N. Memon, and T. Suel, "zdelta: An efficient delta compression tool," Tech. Rep., 2002.

[105] E. Soroush and M. Balazinska, "Time travel in a scientific array database," in *ICDE*, 2013, pp. 98–109.

[106] A. Seering, P. Cudre-Mauroux, S. Madden, and M. Stonebraker, "Efficient versioning for scientific array databases," in *ICDE*, 2012.

[107] U. Khurana and A. Deshpande, "Efficient snapshot retrieval over historical graph data," in *ICDE*, 2013, pp. 997–1008.

[108] M. Stonebraker and G. Kemnitz, "The postgres next generation database management system," *Communications of the ACM*, vol. 34, no. 10, pp. 78–92, 1991.

[109] W. Gatterbauer and D. Suciu, "Managing structured collections of community data," in *CIDR 2011, Fifth Biennial Conference on Innovative Data Systems Research, Asilomar, CA, USA, January 9-12, 2011, Online Proceedings*, 2011, pp. 207–210.

[110] A. Bhardwaj, A. Deshpande, A. J. Elmore, D. Karger, S. Madden, A. Parameswaran, H. Subramanyam, E. Wu, and R. Zhang, "Collaborative data analytics with datahub," *Proceedings of the VLDB Endowment*, vol. 8, no. 12, pp. 1916–1919, 2015.

[111] A. Alawini, D. Maier, K. Tufte, and B. Howe, "Helping scientists reconnect their datasets," in *Proceedings of the 26th International Conference on Scientific and Statistical Database Management*. ACM, 2014, p. 29.

[112] "Azure ai gallery," https://gallery.azure.ai/experiments.

[113] A. Alawini, D. Maier, K. Tufte, B. Howe, and R. Nandikur, "Towards automated prediction of relationships among scientific datasets," in *Proceedings of the 27th International Conference on Scientific and Statistical Database Management*. ACM, 2015, p. 35.

[114] G. S. Yilmaz, T. Wattanawaroon, L. Xu, A. Nigam, A. J. Elmore, and A. Parameswaran, “Datadiff: User-interpretable data transformation summaries for collaborative data analysis,” in *Proceedings of the 2018 International Conference on Management of Data*. ACM, 2018, pp. 1769–1772.

[115] W. C. Tan, M. Zhang, H. Elmeleegy, and D. Srivastava, “Reverse engineering aggregation queries,” *Proceedings of the VLDB Endowment*, vol. 10, no. 11, pp. 1394–1405, 2017.

[116] A. Das Sarma, A. Parameswaran, H. Garcia-Molina, and J. Widom, “Synthesizing view definitions from data,” in *Proceedings of the 13th International Conference on Database Theory*. ACM, 2010, pp. 89–103.

[117] Q. T. Tran, C.-Y. Chan, and S. Parthasarathy, “Query by output,” in *Proceedings of the 2009 ACM SIGMOD International Conference on Management of data*. ACM, 2009, pp. 535–548.

[118] T. Wattanawaroon, S. Macke, and A. Parameswaran, “Towards a theory of datadiff: Optimal synthesis of succinct data modification scripts,” *arXiv preprint arXiv:1801.06258*, 2018.

[119] Y. Cui and J. Widom, “Lineage tracing for general data warehouse transformations,” *The VLDB Journal—The International Journal on Very Large Data Bases*, vol. 12, no. 1, pp. 41–58, 2003.

[120] E. Rahm and P. A. Bernstein, “A survey of approaches to automatic schema matching,” *the VLDB Journal*, vol. 10, no. 4, pp. 334–350, 2001.

[121] A. Heise, J.-A. Quiané-Ruiz, Z. Abedjan, A. Jentzsch, and F. Naumann, “Scalable discovery of unique column combinations,” *Proceedings of the VLDB Endowment*, vol. 7, no. 4, pp. 301–312, 2013.

[122] R. C. Fernandez, Z. Abedjan, F. Koko, G. Yuan, S. Madden, and M. Stonebraker, “Aurum: A data discovery system,” in *2018 IEEE 34th International Conference on Data Engineering (ICDE)*. IEEE, 2018, pp. 1001–1012.

[123] R. C. Fernandez, J. Min, D. Nava, and S. Madden, “Lazo: A cardinality-based method for coupled estimation of jaccard similarity and containment,” in *2019 IEEE 35th International Conference on Data Engineering (ICDE)*. IEEE, 2019, pp. 1190–1201.

[124] G. S. Yilmaz, T. Wattanawaroon, L. Xu, A. Nigam, A. J. Elmore, and A. Parameswaran, “Datadiff: User-interpretable data transformation summaries for collaborative data analysis,” in *Proceedings of the 2018 International Conference on Management of Data*, ser. SIGMOD ’18. New York, NY, USA: ACM, 2018. [Online]. Available: http://doi.acm.org/10.1145/3183713.3193564 pp. 1769–1772.

[125] D. S. Hochbaum, “Approximating clique and biclique problems,” *Journal of Algorithms*, vol. 29, no. 1, pp. 174–200, 1998.

[126] M. Stoer and F. Wagner, “A simple min-cut algorithm,” *Journal of the ACM (JACM)*, vol. 44, no. 4, pp. 585–591, 1997.

[127] L. R. Ford and D. R. Fulkerson, “Maximal flow through a network,” in *Classic papers in combinatorics.* Springer, 2009, pp. 243–248.

[128] A. Rajaraman and J. D. Ullman, *Mining of massive datasets.* Cambridge University Press, 2011.

[129] E. Cohen, “Min-hash sketches,” *Encyclopedia of Algorithms*, pp. 1–7, 2008.

[130] E. Cohen and H. Kaplan, “Bottom-k sketches: Better and more efficient estimation of aggregates,” in *ACM SIGMETRICS Performance Evaluation Review*, vol. 35, no. 1. ACM, 2007, pp. 353–354.

[131] D. V. Kalashnikov, L. V. Lakshmanan, and D. Srivastava, “Fastqre: Fast query reverse engineering,” in *Proceedings of the 2018 International Conference on Management of Data.* ACM, 2018, pp. 337–350.

[132] M. Zhang, M. Hadjieleftheriou, B. C. Ooi, C. M. Procopiuc, and D. Srivastava, “On multi-column foreign key discovery,” *Proceedings of the VLDB Endowment*, vol. 3, no. 1-2, pp. 805–814, 2010.

[133] W. McKinney, *Python for data analysis: Data wrangling with Pandas, NumPy, and IPython.* ” O’Reilly Media, Inc.”, 2012.