ABSTRACT

Title of dissertation:	DELTA-BASED STORAGE AND QUERYING FOR VERSIONED DATASETS
	Amit Chavan Doctor of Philosophy, 2018
Dissertation directed by:	Professor Amol Deshpande Department of Computer Science

Data-driven methods and products are becoming increasingly common in a variety of communities, leading to a huge diversity of datasets being continuously generated, modified, and analyzed. An increasingly important consideration for the underlying data management systems is that, all of these datasets and their versions over time need to be stored and queried for a variety of reasons including, but not limited to, reproducibility, collaboration, provenance, auditing, introspective analysis, and backups. However, most solutions today resort to highly ad hoc and manual version management and sharing techniques, that leads to friction when managing collaborative data science workflows, while also introducing inefficiencies.

In this dissertation, we introduce a framework for dataset version management, and address the systems building, operator design, and optimization challenges involved in building a dataset version control system. We describe the various challenges and solutions in the context of our system, called DEX, that we have developed to support increasingly complex version management tasks. We show how to use delta-encoding, a key component in managing redundancy, to provide efficient storage and retrieval for the thousands of dataset versions, and develop a formalism to understand the various trade-offs in a principled manner. We study the *storage–recreation* trade-off in detail and provide a suite of inexpensive heuristics to obtain high-quality solutions under different settings. In order to provide a rich interface to specify version management tasks, we design a new query language, called VQUEL, with the ability to query dataset versions and provenance in a unified manner. We study how assumptions on the delta format can help in the design of a logical algebra, which we then use to execute increasingly complex queries efficiently. A key characteristic of our query execution methods is that the computational cost is primarily dependent on the size and the number of deltas in the expression (typically small), and not the input dataset versions (which can be very large). Finally, we demonstrate the effectiveness of our developed techniques by extensive evaluation of DEX on a mixture of real-world and synthetic datasets.

DELTA-BASED STORAGE AND QUERYING FOR VERSIONED DATASETS

by

Amit Chavan

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

Dedicated to the memories of my dear (late) grandmother, *Kaku*.

Acknowledgments

There are many people whose support, guidance, and friendship have made this thesis possible.

I am forever indebted to my advisor, Amol Deshpande, who has taught me so much and been a great mentor during the last six years. Learning how to do successful research took me many years and I am thankful for the skills that Amol has patiently taught me. He taught me how to communicate ideas more effectively, gave me the freedom to investigate new directions, patiently listened and gave feedback on many of my half-baked ideas, and contributed many ideas to this work.

I am fortunate to have had many mentors on the path towards my Ph.D. I would like to thank Aravind Srinivasan for his support and guidance when I needed it the most. I am indebted to Rajiv Gandhi for introducing me to computer science research and helping me apply to graduate schools.

The staff at the Computer Science Department, particularly, Jennifer Story, have created a wonderful place, and I am happy to be a part of it. I am also grateful to Louiqa Raschid, Daniel Abadi, Mihai Pop, and Samir Khuller for their participation on my proposal and defense committees.

I'd also like to thank my collaborators from whom I've learned so very much. In particular, the work described in Chapter 3 was done jointly with two other PhD students, Souvik Bhattacherjee and Silu Huang. I was primarily responsible for the design of the algorithms, jointly responsible for the theoretical results with Silu, and for the implementation and the experimental evaluation with Souvik. The work described in

Chapter 4 was done jointly with Silu Huang, with equal contributions.

I have deeply enjoyed my interactions and friendships with the members of the the database group and the Computer Science department at UMD. Thank you Souvik Bhattacherjee, Hui Miao, Abdul Quamar, Theodoros Rekatsinas, Manish Purohit, Pan Xu for making graduate study both intellectually stimulating and incredibly enjoyable. I am also lucky to have a wonderful set of friends and housemates. An incomplete list of those who helped me on this path includes Sangeetha Venkatraman, Amey Bhangale, Kartik Nayak, Bhaskar Ramasubramanian, Ramakrishna Padmanabhan, Sudha Rao, and Meethu Malu.

My heartfelt gratitude to my family, without whom I would not have made it this far. My parents provided their love and support throughout my life, and encouraged me to do things the right way, without making any compromises on the quality of work. My sister, Deepti, sometimes believed in my goals and dreams even more than I did.

Lastly, to Shruti, for understanding and supporting me in all my ups and downs during this journey. I'm fortunate to have had her boundless love and her selfless and unwavering support for the last six years. I look forward to our journey ahead, together.

Table of Contents

List of Figures

Chapter 1: Introduction

Data-driven methods and products are becoming increasingly common in a variety of communities, leading to a huge diversity of datasets being continuously generated, modified, and analyzed. Data science teams, in a variety of domains, want to acquire new datasets and perform increasingly sophisticated analysis tasks in order to derive valuable insights. Such efforts are collaborative in nature and can often span multiple teams or organizations, where one team uses datasets generated by another team in their analysis tasks. Moreover, data science tasks are also iterative in nature – data is updated regularly, algorithms are improved, or new approaches are tried out to explore their impact. An update to a dataset, for instance, might lead to updating all the tasks or "pipelines" that depend on it. An increasingly important consideration for the underlying data management systems is that, all of these datasets and their versions over time need to be stored and queried for a variety of reasons including auditing, provenance, transparency, accountability, introspective analysis, and backups [1, 2, 3, 4, 5].

We outline an example data science workflow that highlights the iterative and collaborative aspects of the activity.

Example 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 [6]. 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) $\longrightarrow$ Error correction tools (Quake, Sickle, etc.) $\longrightarrow$ Input analysis, k-mer calculation (KmerGenie) $\longrightarrow$ Assembly tool (SOAPdenovo, ABySS) $\longrightarrow$ 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, mostly with large overlaps between them, and complex derivational dependencies.

We observe that although there is a wealth of data science research addressing stand-alone data analysis issues or building integrated tools for analysis, the dataset management aspects of these tools are poor, requiring data scientists to use ad hoc mechanisms to record and reason about datasets. When people collaborate over data it not uncommon to have hundreds or thousands of versions of collected, curated, and derived datasets, at various degrees of structure (relational/JSON to fully unstructured). Ad hoc solutions to this problem can hardly adapt to satisfy the long term tracking of all such complex data and metadata. Once people start doing data analysis, they need sustainable and scalable tools that allow them to spend little to no effort on maintaining, sharing, and tracking changes to datasets, and instead focus on scientific and knowledge discovery.

In this dissertation, we propose new methods and tools towards the goal of making the management of such datasets hassle free, and address some of the fundamental problems in this space. We build a new dataset version control system (DVCS), to keep track of all the dataset versions and dataset provenance in one place, and making it easy to analyze or query this information. By keeping all this information in one place, we can enable a rich set of functionality that can simplify the lives of data scientists, make it easier to identify and eliminate errors, and decrease the time to obtain actionable insights. A DVCS can keep track of all the datasets and their thousands of versions, storing them compactly, while still being able to retrieve them efficiently, and query them to enable rich introspection capabilities.

The goal of this dissertation is to develop a formalism for managing a large number of dataset versions, i.e., storing and querying them, and designing algorithms that use the formalism to enable efficient execution of a large class of introspection queries over these versions.

1.1 Challenges in building a DVCS

The *first challenge* that we consider in this dissertation, is how to efficiently store and retrieve the thousands of dataset versions. Because of the nature of the tasks that create these versions, not all versions are entirely different from one another, and there is massive redundancy in their contents. If the difference, or *delta*, between two similar versions can be computed, it is possible to save on storage costs by keeping only one version and the delta, instead of two versions. The large number of overlapping versions makes it imperative to exploit such deltas to compactly store all of them. However, this gives rise to the *storage–recreation tradeoff*: 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.

The *second challenge* is that a versioning API that simply provides *put* and *get* facilities is very limiting, and is not a good fit for data scientists to understand and reason about data contained within versions. Tools like git/svn lack sophisticated query interfaces. For example, in the data science scenario mentioned above, one may wish to write a query that finds the intersection of a set of versions, representing, for instance, the final synthesized result of different pipelines. Identifying the design goals of a language that enables users to traverse, compare, and query the version metadata, version history, and the data itself, in a holistic manner, is an important and necessary problem for a DVCS.

The *third challenge* is designing efficient execution algorithms for the tasks mentioned above. For instance, users of a DVCS naturally want tools that help them reason about multiple versions of datasets, and as such, they will pose queries that often access more than one version at a time (e.g., for finding similarities among a set of versions). Existing solutions require users to first *get/checkout*, i.e., reconstruct a specific version of a dataset of a file completely, all versions before running any queries on the data stored within them. This approach is less than ideal particularly when the individual versions are large and the users need to access multiple versions for their analysis task. First, irrespective of the size of the query result, this approach entails creating all the input versions before query processing can begin, resulting in large memory and/or I/O usage. Second, it requires users to maintain another system to assist in executing the queries. Third, this approach fails to exploit the fact that most datasets evolve through changes that are small relative to the dataset sizes, and knowing about these changes can not only help us towards our storage goals, but also enable answering queries efficiently.

1.2 Dissertation Overview and Contributions

Given the challenges described above, the primary goal of this dissertation is to develop a robust framework to model the fundamental problems for dataset version management, and design new techniques and algorithms to enable users to execute rich queries efficiently. We have built a prototye system, called DEX, whose architecture is shown in Figure 1.1. The techniques summarized below serve as building blocks to this architecture. The two main components that this dissertation is focused on are the **Storage Graph Builder** and the **Query Processor** module.

The DEX system is built on top of `git` and has three major components: (a) a set of command line utilities, **DEX CLI**, written in Python, to allow the user to interact with the repository in the form of the standard *add, commit, checkout,* etc., commands (similar to `git`), (b) the **Storage Graph Builder** which decides how best to store the collection of dataset versions, and (c) the **Query Processor**, written in Java, that executes user queries against the compressed representation. `DEX CLI` passes through the version management tasks *not* pertaining to large datasets to `git`; the user may specify a file to

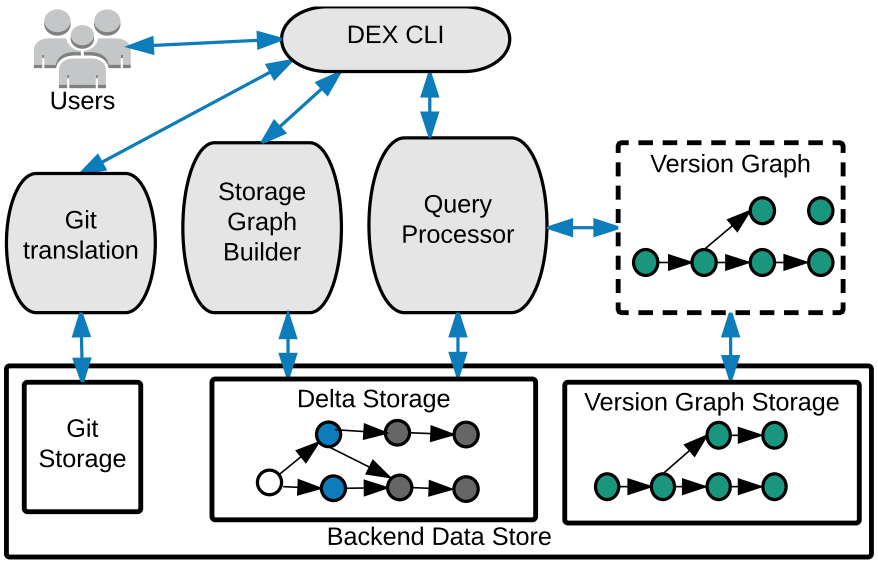


Figure 1.1: System Architecture of DEX

be managed by DEX through a flag to the *add* command, and any tasks pertaining to those files are sent to the Storage Graph Builder (in case of *add* or *commit*) or the Query Processor.

The main parts of the dissertation, as well as the associated chapters and published papers, are summarized below.

1.2.1 Efficient Storage and Retrieval

Given the high overlap and duplication among the datasets, it is attractive to consider using *delta encoding* to store the datasets in a compact manner. Delta encoding is a cornerstone of many *no-overwrite* storage systems that are focused on archiving and maintaining vast quantities of datasets (simply put, a collection of files). Archival and backup systems often store multiple versions or snapshots of large datasets that have significant overlap across their contents using deltas.

At a high level, delta encoding consists of representing a *target* version as the mutation, or delta, from a *source* version content. Typically, source and target versions are selected such that they have a large overlap across their contents and hence their delta is small. Furthermore, the source version itself may be represented as a delta from another version, and so on, creating a "graph" of versions and deltas. The compressed storage is obtained by storing only a few select versions, commonly referred to as *materialized* versions, and deltas (instead of the versions they represent) in this graph, such that it is possible to re-create any version by walking the *path* of deltas starting from a materialized file and ending at the desired file.

However, this gives rise to the *storage–recreation tradeoff*: 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. We illustrate this trade-off via an example.

Example 2 *Figure 1.2(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 1.2, 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 ($V_1 \longrightarrow V_3$) 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 1.2 (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 1.2 (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* $\{(V_1 \longrightarrow V_3 \longrightarrow V_5)\}$ *needs to be accessed in order to retrieve* V_5 *and the recreation cost is* $10000 + 3000 + 550 = 13550 > 10120$.

Figure 1.2 (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).*

Despite the fundamental nature of the storage-retrieval problem, there is surprisingly little prior work on formally analyzing this trade-off and on designing techniques for identifying effective storage solutions for a given collection of datasets. Version Control Systems (VCS) like Git, SVN, or Mercurial, despite their popularity, use fairly simple algorithms underneath, and are known to have significant limitations when managing large datasets [7, 8]. Much of the prior work in literature focuses on a linear chain of versions, or on minimizing the storage cost while ignoring the recreation cost.

We address this problem in detail in Chapter 3. We show that simply using delta compression to minimize storage space leads to very high latencies while retrieving specific datasets. We also show that the delta compression heuristics used by popular VCS'

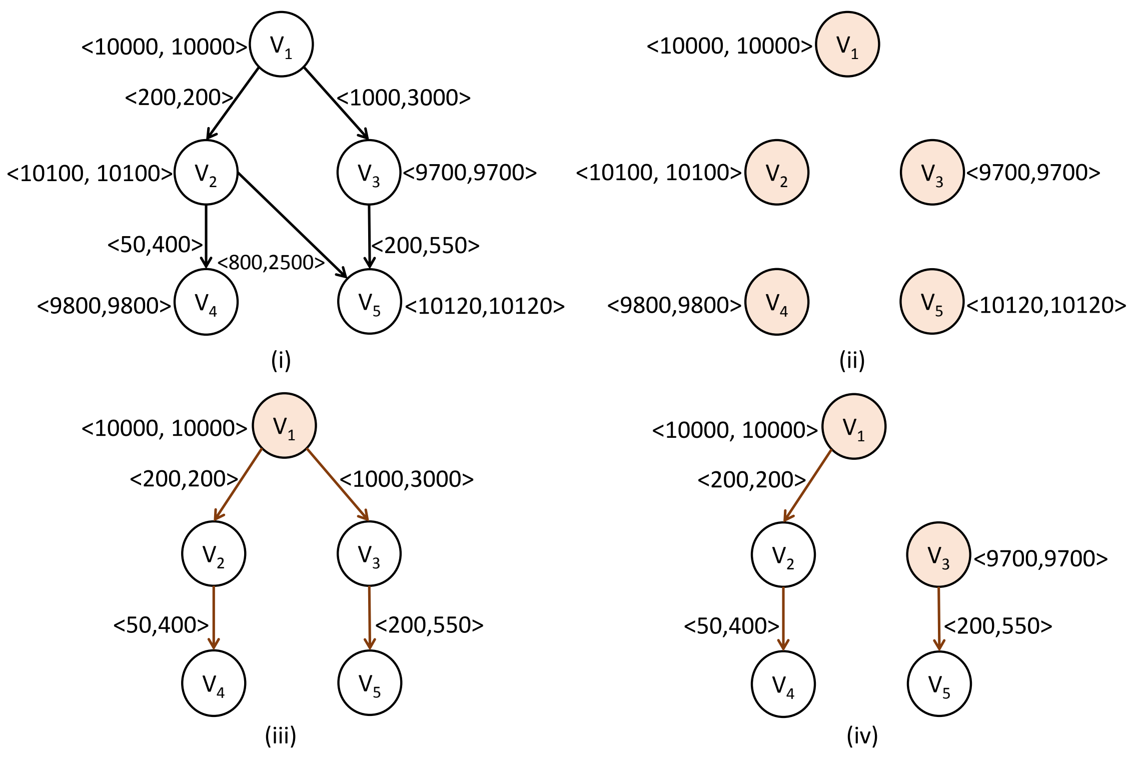


Figure 1.2: (i) A version graph over 5 datasets – annotation $\langle a, b\rangle$ indicates a storage cost of a and a recreation cost of b; (ii, iii, iv) three possible storage graphs

like Git and SVN are ineffective at storing datasets, both in terms of resources consumed and the quality of solution produced. Thus, to address the first challenge mentioned above, we propose novel problem formulations towards understanding the storage and recreation tradeoff in a principled manner. We formulate several optimization problems and show that most variations are NP-Hard. As a result, we design several efficient heuristics that are effective at exploring this trade-off and present an extensive experimental evaluation over several synthetic and real-world workloads demonstrating the effectiveness of our algorithms at handling large problem sizes.

1.2.2 A Language to Query Provenance and Versions in Unified Manner

To be truly useful for collaborative data science, we also need the ability to speficy queries and analysis tasks over the versioning and provenance information in a unified manner. In the Genome assembly example mentioned earlier, there is a wide range of queries that may be of interest. 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.

These examples above 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.

In Chapter 4, we describe a language, called VQUEL, that aims to provide these features. VQUEL is largely a generalization of the Quel language (while also introducing certain syntactic conveniences that Quel does not possess), and combines features from GEM 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 data present in versions, versioning derivation relationships, as well as versioning metadata.

1.2.3 Delta-aware Query Execution

While delta encoding is an effective method to archive large amounts of data in many immutable data stores, many of these stores require users to "check out" complete file/dataset versions in order to manipulate them. In Chapter 5 and Chapter 6, we show that this approach is less than ideal for a variety of rich queries that compare or work with data from across multiple versions, and significant performance benefits can be had if we can make use of the pre-computed deltas during query processing. In particular, in these two chapters, we present a systematic study of the problem of supporting rich analysis queries over delta-oriented storage engines, and describe the Query Execution module of DEX. We focus on the storage design and implementation of DEX for a class of semi-structured datasets that we call `datafiles`, and for a class of basic queries that includes multi-version checkouts, intersections, unions, t-threshold, and single block select-project-join queries.

A `datafile` is a file whose contents can be seen as a *set* of records, i.e., the order of records within a `datafile` is immaterial, and no two records in a `datafile` are identical.

Examples of such files include CSV files, JSON documents, log files, to name a few, and these constitute a large fraction of files in a typical data lake, and hence are particularly suited to our study. A common method to represent a delta between two `datafiles` is to maintain the "deletions" and "additions" of records required to go from one `datafile` to the other. If a `datafile` has additional structure, namely, a user supplied *primary key*, along with details on how to parse a record into its component *fields* or *columns*, we use a more compact *column-based delta* format.

In Chapter 5, we develop a general cost-based optimization framework based on key *algebraic* properties regarding composition of the deltas. The result of this framework is a compact algebraic expression that confines query execution to a small set of deltas. As a side effect, the computational cost is dependent on the size and the number of deltas in the expression (which are typically small) in contrast to the size of the input datasets. We develop optimal algorithms for executing single-file or multi-file checkout queries assuming reasonable restrictions on the evaluation plan search space. We also develop a series of intuitive transformation rules that help simplify the search space for intersection, union, and t-threshold queries, and use them in conjunction with cost-based solutions for base cases, to develop effective search algorithms. We present a comprehensive evaluation against synthetic datasets of varying characteristics. Our results show that our methods perform exceedingly well compared to the baselines, even for simple queries like single-file checkouts.

In Chapter 6, we extend the above framework to execute single block select-project-join (SPJ) queries, such as those that can be formulated in VQUEL, over multiple versions stored in DEX. The key design decision of our approach is to execute the various query operators exactly once for each unique tuple in the set of versions, rather than executing the query once for each version. We propose using a new representaion for tuples, called v-*tuples*, during query processing. In addition to holding information about the fields in a tuple, a v-tuple also stores information about the versions, from the set of queries versions, the respective tuple appears in. We show how to efficiently construct such v-tuples from the delta-encoded representation and present our modifications to the traditional SCAN and JOIN physical operators, in order to process v-tuples efficiently. We present an extensive evaluation of this approach on multiple synthetic datasets. Our results shows that richer query processing is viable in DEX, with significant performance benefits over one version at a time execution.

Delta Representations and Tradeoffs:

A key question in DEX is selecting the *delta variant*, i.e., the particular format/algorithm for computing the delta between two files. This is because different delta formats are appropriate for different types of files: a UNIX-style line-by-line diff is a common delta format for plain text files, while an `XOR` is more suited to numerical array-oriented data. Exploiting the structure in the data, if known, can often lead to better deltas (e.g., for XML [9], or relations [10]). Column-based deltas may be more appopriate when a large number of records are changed slightly, e.g., due to a schema change. Furthermore, a particular delta format may be *directed* or *undirected*: if a delta Δ between source file A and target file B is directed, it may only be used to recreate B given A, and not vice versa. An undirected delta between two files, on the other hand, can accept either file as source and recreate the other.

The desire to execute queries directly on deltas (as we do in this work) brings another dimension to this choice. There is an inherent tension in the amount of information stored in a delta, and our ability to push query execution on to them. In this work, we pick different delta formats depending on the class of queries that the user wishes to support. Supporting richer queries and compact deltas requires us to make assumptions about the `datafile` schema, as follows:

- In Chapter 3, we consider storing and retrieving individual files (in their entirety). As such, we do not require any assumptions or restrictions on the file or delta format, and any suitable delta algorithm can be used. However, we do require the storage cost and the access cost, or their estimates, of the resulting delta.

- In Chapter 5, we consider a class of *set-based* query operators on the past versions. Hence we require that every version of a `datafile`, at commit time, can be separated into a *set* of records. The delta format then is also based on the set of records abstraction, to aid in efficient query execution.

- In Chapter 6, we consider select-project-join queries across a large number of past versions. In order for such queries to make sense, the `datafile`, at commit time, must be separable into records, and every record be separable into its constituent fields/columns. In order to use a compact column-based delta format, we also require the user to specify a column or a set of columns as the primary key in the `datafile`.

Chapter 2: Related Work

This section surveys the state of the art in managing versioned data. Our goal in this section is to give a brief survey of the many decades of work done in this area and to put our contributions in context. We begin with prior work perfomed towards the goal of supporting efficient storage of multiple versions of data followed by a review of recent solutions in the Version Control Systems space. Thereafter, we summarize work done towards building richer interfaces to query such data and to execute those queries efficiently.

2.1 Enabling Multi-Versioned Storage

Relational Database Management Systems. Many database applications require that multiple versions of records be stored and retrieved, and as such, there has been extensive research on temporal and versioned databases and their applications. The effort to satisfy the diverse needs of these applications has led to a number of versioning solutions, e.g., [11, 12, 13, 14, 15, 16]. Much work, especially earlier papers, focused on theoretical foundations, not on practical considerations such as storage efficiency and indexing versioned data. We briefly review some of the work done in this area, and for a detailed survey, we refer the reader to [12]. In addition, extensive bibliographies have also been compiled, see [17] as a starting point.

There was some conceptual temporal database work in 1980s, see [18, 19] ([18] contains references to even earlier work), on developing temporal data models and temporal query languages. The most basic concepts that a relational temporal database is based upon are *valid time* and *transaction time*, considered orthogonal to each other. Valid time denotes the time period during which a fact is true with respect to the real world. Transaction time is the time when a fact is stored in the database.

The first relational database system offering temporal functionality was Postgres [20]. Postgres used R-trees [21] to index historical data, with recent data residing in a B+Tree. This separation is important as a general multi-attribute index like an R-tree has difficulty supporting data that is current and hence does not yet have an end time. The movement of data from the B+tree to the R-tree occurs lazily.

Transactime time functionality has also received some industrial interest, particularly from Oracle [22] and Microsoft. Oracle's FlashBack queries allow the application to access prior transaction time states of the database and to retrieve all the versions of a row between two transaction times. It also allows for "point in time" recovery, i.e., tables and databases can be rolled back to a previous transaction time, discarding all changes after that time; this functionality can be used to deal with bad user transactions. They do not index historical versions, however, so historical version queries must go through current time versions and then search backward "linearly" in time. More recently, Oracle supports "Total Recall" feature for Oracle 11g [23]. Building on FlashBack, Total Recall archive is read only and it supports long time archiving of transaction time versions, including migration of the versions to archival media. A form of compression is supported to reduce the storage cost of retaining more extensive database history.

Immortal DB, which was built into Microsoft SQL Server, integrated a temporal indexing technique called the TSB-tree [14, 15] to provide high performance access and update for both current and historical data.

Buneman et al. [24] 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.

MOLAP systems store data in multidimensional arrays [25] with particular focus on aggregation queries. These systems exploit data structures to efficiently compute rollups. The MOLAP system in [26] supports versions to represent changes to the data sources that should be propagated to the data warehouse periodically. But the versioning system is designed to benefit the concurrency control mechanism in order to minimize contention between query and maintenance transactions.

Scientific Databases. In many fields of science, multidimensional arrays rather than flat tables are standard data types because data values are associated with coordinates in space and time. As a result, many specialized array-processing systems have emerged, e.g, [27, 28]. As noted in [29], an important requirement that scientists have for these systems is the ability to create, archive, and explore different versions of their arrays. None of the temporal database solutions mentioned above are a good fit here because (1) simulating arrays on top of relations can be inefficient [29], and (2) their internal data structures are not specialized for time travel over array data. Hence, a no-overwrite storage manager with efficient support for querying old versions of an array is a critical component of an array database management system. In recent work, Seering et al. [30] presented a disk based versioning system using efficient delta encoding to minimize space consumption and retrieval time in array-based systems.

Other Data Models and Deduplication Schemes Many solutions have been proposed to support multiple versions of complex data, e.g., XML [31], object oriented [32], and spatio-temporal data [33]. Khurana and Deshpande [34] present an approach for managing historical graph data for large information networks, and for executing snapshot retrieval queries on them. Quinlan and Dorward [35] propose an archival "deduplication" storage system that identifies duplicate blocks across files and only stores them once for reducing storage requirements. Zhu et al. [36] present several optimizations on the basic theme. Douglis and Iyengar [37] 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. [38] 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. Similar dictionary-based reference encoding techniques have been used by Chan and Woo [39] to efficiently represent a target web page in terms of additions/modifications to a small number of reference web pages. Burns and Long [40] present a technique for in-place re-construction of delta-compressed files using a graph-theoretic approach. Kulkarni et al. [41] present a more general technique that combines several different techniques to identify similar blocks among a collection files, and use delta compression to reduce the total storage cost (ignoring the recreation costs).

Our Contributions. Most of the work in temporal relational database management systems has focused its efforts on efficiently storing a "linear chain" of versions, unlike our work, which requires supporting an arbitrary DAG of versions. Moreover, unlike our framework, which does not make any assumptions about how the versions were modified, many schemes assume knowledge of the specific records or cells that are updated. The general concept of multi-versioning has also been used extensively in commercial databases to provide snapshot isolation [42]. However, these methods only store enough history to preserve transactional semantics, whereas we preserve all historical branches and derivation relationships to ensure integrity of the version graph. Finally, prior efforts that have looked at the problem of minimizing the total storage cost for storing a collection of related files do not typically consider the recreation cost or the tradeoffs between the two. Their design also does not consider querying data in the past versions using rich query interfaces, such as the ones available in temporal databases.

2.2 Version Control Systems (VCS)

Version Control Systems (VCS) have a long history in Computer Science. A VCS records changes to a file or set of files over time so that any user can recall a specific version later. Versioning techniques such as forward and backward delta encoding and the use of multi-version B-trees have been implemented in various legacy systems. `git` [43] is one of the conventional version control systems and is believed to be faster and more disk efficient than other similar version control systems. The major difference between `git` and any other VCS, such as Subversion (`svn`), Concurrent Versions System (`cvs`) is the way `git` thinks about its data. While most other systems store information as a list of file-based changes, `git` thinks of its data more like a set of snapshots of a miniature filesystem and stores changes at the snapshot level. To be efficient, if a file has not changed, `git` doesn't store the file again, instead, just a link to the previous identical file it has already stored.

Despite their popularity, these systems largely 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-based diffs. These systems are known to have significant limitations when handling large files and large numbers of versions [8]. As a result, a variety of extensions like `git-annex` [44], Git Large File Storage [45], etc., have been developed to make them work reasonably well with large files. These extensions replace large files with text pointers inside Git, while storing the file contents on a remote server like GitHub.com, or Amazon S3.

Our Contributions. The initial design of DEX is modeled after conventional version control software such as `git`. In particular, the concepts of a no-update model and of differencing stored files against each other for more efficient storage have both been explored extensively by such systems. We build upon this work to support a superset of the conventional version control API for large datasets. We show that the underlying algorithms in `git` and Subversion can be extremely inefficient, both in terms of storage used and resource (memory/IO) consumption.

2.3 Query Languages

Abdessalem and Jomier [46] introduce VQL, a language designed for querying data stored in multiversion databases. VQL is based on a first-order calculus and provides users with the ability to navigate through object versions modeled by the database. More recently, there are new temporal constructs pushed in the SQL standard by the main DBMS vendors [47]. This work, however, does not directly apply to our setting because the constructs assume a linear chain of versions — as noted earlier, we could have an arbitrary branching structure of versions.

While there has been substantial work on query languages for provenance, ranging from adapting SQL [48], Prolog [49, 50], SPARQL [51, 52] to specialized languages such as QLP [53, 54], PQL [55], ProQL [56] ([57], [58] 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.

Our Contributions. We design a new query language, called VQUEL, that is capable of querying dataset versions, dataset provenance (e.g., which datasets a given dataset was derived from), and record-level provenance (if available). Our design draws from constructs introduced in the historical Quel [59] and GEM [60] languages, neither of which had a temporal component.

2.4 Query Execution

There are a large number of proposed indexing techniques used for temporal data, e.g., [33, 61, 62]. Salzberg and Tsotras [63] present a comprehensive survey of indexing structures for temporal databases. They also present a classification of different queries that one may ask over a temporal database.

Lomet et al. [14, 15] integrated a temporal indexing technique, the TSB-tree, into Immortal DB (which was built into Microsoft SQL Server) to serve as the core access method. The TSB-tree provides high performance access and update for both current and historical data. Jouini and Jomier [64] studied the problem of efficiently indexing data with "branched evolution". The main contributions here are the extension of temporal index structures to data with branched evolution and an analysis method that estimates the performance of the different index structures and provides guidelines for the selection of the most appropriate one. Soroush and Balazinska [65] present an indexing technique to support "time travel" queries for scientific arrays. A key aspect of their technique is that they can support *approximate* queries that can quickly identify which versions are relevant to a user and return the approximate content of these versions.

Queries in delta-based storage. Delta encoding has been used in a variety of systems to provide trade-offs among time, space, and compression performance, e.g., to reduce data transfer time for text/HTTP objects [66], to reduce access time in a file system [67], to store many versions of the generated artifacts in source code control systems (e.g., `git`) or other types of data [10, 30, 68]. However, the focus of many of the existing delta encoding schemes has been to access the objects in their entirety and to the best of our knowledge, they have not considered the tradeoff between storage and "computability over deltas". Even version control systems that provide functionality to compare multiple objects, e.g., *merge, diff, etc.*, first recreate all required files before operating upon them. Recently, [65] presented an indexing technique to support "time travel" queries for scientific arrays wherein they support *approximate* queries that can quickly identify which versions are relevant to a user and return the approximate content of these versions. However, they did not consider queries that compared the contents of two or more array versions.

Deltas and computing. The concept of making deltas "first-class citizens" was explored in Heraclitus [69]. To support "what-if" scenario analysis, they provided general-purpose constructs for creating, accessing, and combining deltas. In the specific realization of their paradigm for the relational model, deltas are a set of *signed atoms* where the positive atoms correspond to "insertions" and the negative atoms correspond to "deletions". In addition, the deltas have structure and can be manipulated directly by constructs in user programs, e.g., to delete all records satisfying a predicate. In contrast, our use of deltas is at the physical level and not exposed to the users. They do not consider optimizing the different types of queries against a delta storage. Executing queries with hypothetical state updates was also considered in [70]. Here the state updates (or deltas) were allowed to be expressions and the authors considered rewriting such queries into an optimized form based on their novel rules for substituion and the rules for relational algebra. Such rules are however not applicable in our setting. Record-based deltas were also used in [71, 72] to provide the capability of sharing data and updates among different participants. However, they focused on formalizing the semantics of the update exchange process, e.g., mapping updates across schemas and filtering them according to local trust policies, and the challenges introduced therein.

Connections to materialized views. Using pre-computed deltas to answer user queries is, at a high level, similar to the problems that have been considered in the context of materialized view and index selection to speed up query processing. Broadly speaking, research in this area has focused on three issues: (i) determining the search space or class of views to consider for materialization, (ii) choosing a subset of views and indexes to materialize depending on various constraints like storage overhead, maintenance overhead, effectiveness on the query workload, etc., [73, 74] and (iii) quickly determining which views to consider to answer a given query [75, 76]. In our problem setting, set-backed deltas can be considered as a form of materialized views (which can be used to reconstruct base relations), with our work addressing the problem of how to use such views to answer queries. We also design a logical algebra over the specific delta formats, that allows us to combine a large number of deltas to answer one query.

Evaluating set expressions. Several algorithms and data structures have been proposed in literature to solve union, intersection and difference problems on sets [77, 78, 79, 80] by minimizing the number of comparisons required. Although the ordered list representation is the most common, some algorithms also consider representing sets in other data structures, e.g., skip lists [81], machine word-based representations [82], etc., to obtain additional speedup. A comparison of few of these methods is available in [83, 84]. Speeding up set operations is largely orthogonal to our approach and we can make use of some of these techniques as additional operators with the appropriate cost model. As mentioned earlier, in this work, we use an adaptive set intersection algorithm that was shown to have reasonably good performance in [83] without requiring any preprocessing step. The larger problem here, however, is how to efficiently evaluate a set expression consisting of union, intersection and difference. [85] consider evaluating union-intersection expressions in a worst-case efficient way for a non-comparison based model. However, their approach uses hash-based dictionaries, which would require an additional pre-processing step, and it remains an open problem whether their results can be extended to handle set difference. Recently, [86] showed that, for a similar cost model, a union-intersection expression can be rewritten to perform intersections before unions with often a reduced cost. Their approach, however, did not consider rewrites in the presence of set difference.

Chapter 3: Storage–Recreation Tradeoff

3.1 Introduction

In this chapter, we present a formal study of the problem of deciding how to jointly store a collection of dataset versions, provided along with a version or derivation graph. Specifically, we focus on the problem of trading off storage costs and recreation costs in a principled fashion. 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 3.1). 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 for solving this problem cannot be easily adapted to incorporate such access frequencies.

The key contributions of this chapter are 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 3.2). Table 3.1 summarizes the problems and our results. We show that most of the variations of this problem are NP-Hard (Section 3.3).

	Storage Cost	Recreation Cost	Undirected Case, $\Delta = \Phi$	Directed Case, $\Delta = \Phi$	Directed Case, $\Delta \neq \Phi$
Problem 1	minimize $\{\mathcal{C}\}$	$\mathcal{R}_i < \infty, \forall i$	PTime, Minimum Spanning Tree		
Problem 2	$\mathcal{C} < \infty$	minimize $\{\max\{\mathcal{R}_i \mid 1 \leq i \leq n\}\}$	PTime, Shortest Path Tree		
Problem 3	$\mathcal{C} \leq \beta$	minimize $\{\sum_{i=1}^{n} \mathcal{R}_i\}$	NP-hard, LAST Algorithm[†]	NP-hard, LMG Algorithm	
Problem 4	$\mathcal{C} \leq \beta$	minimize $\{\max\{\mathcal{R}_i \mid 1 \leq i \leq n\}\}$		NP-hard, MP Algorithm	
Problem 5	minimize $\{\mathcal{C}\}$	$\sum_{i=1}^{n} \mathcal{R}_i \leq \theta$	NP-hard, LAST Algorithm[†]	NP-hard, LMG Algorithm	
Problem 6	minimize $\{\mathcal{C}\}$	$\max\{\mathcal{R}_i \mid 1 \leq i \leq n\} \leq \theta$		NP-hard, MP Algorithm	

Table 3.1: Problem Variations With Different Constraints, Objectives and Scenarios.

- 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 5.4).

- We implement the proposed algorithms in our prototype DEX system. 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 5.5).

3.2 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.

3.2.1 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 DEX (admitting collaborative data science), $\mathcal{G}$ is a DAG (directed acyclic graph) instead of a linear chain. For example, Figure 1.2 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 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 3 *Figure 3.1 shows the matrices Δ and Φ based on version graph in Figure 1.2. The annotation associated with the edge (V_i, V_j) in Figure 1.2 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

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

(i) Δ

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

(ii) Φ

Figure 3.1: Matrices corresponding to the example in Figure 1.2 (with additional entries revealed beyond the ones given by version graph)

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 [37]. 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. Getting a better understanding of such constructs remains a rich area for future work.

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 1**: Undirected case, $\Phi = \Delta$; (2) **Scenario 2**: Directed case, $\Phi = \Delta$; and (3) **Scenario 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 1 (Minimizing Storage) *Given Δ, Φ, find a valid solution $\mathcal{P}$ such that $\mathcal{C}$ is minimized.*

Problem 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 C is smaller than some threshold β. These problems are relevant when the storage budget is limited.

Problem 3 (MinSum Recreation) *Given Δ, Φ and a th- reshold β, identify $\mathcal{P}$ such that $C \leq \beta$, and $\sum_i \mathcal{R}_i$ is minimized.*

Problem 4 (MinMax Recreation) *Given Δ, Φ and a th- reshold β, identify $\mathcal{P}$ such that $C \leq \beta$, and $\max_i \mathcal{R}_i$ is minimized.*

The next two formulations seek to instead minimize the total storage cost 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 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 C is minimized.*

Problem 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 C is minimized.*

3.2.2 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 $V_0 \longrightarrow V_i$, 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 $V_i \longrightarrow V_j$ 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. 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 4 *Given matrix Δ and Φ in Figure 3.1(i) and 3.1(ii), the corresponding graph G*

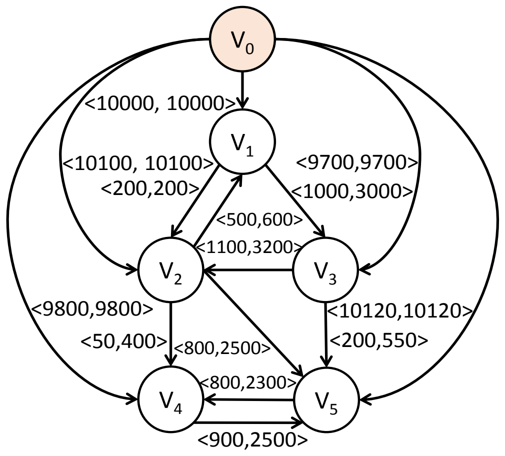


Figure 3.2: Graph G

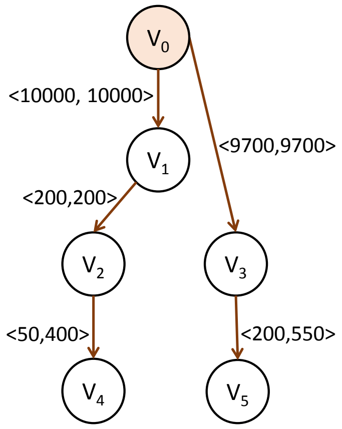


Figure 3.3: Storage Graph G_s

is shown in Figure 3.2. 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 3.3 is a feasible storage graph given G in Figure 3.2, 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 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 1 *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 1 and 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 2 *The optimal storage graph G_s for Problem 1 is a minimum spanning tree of G rooted at V_0, considering only the weights $\Delta_{i,j}$.*

Lemma 3 *The optimal storage graph G_s for Problem 2 is a shortest path tree of G rooted at V_0, considering only the weights $\Phi_{i,j}$.*

3.2.3 ILP Formulation

We present an ILP formulation of the optimization problems described above. Here, we take Problem 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 $\leq \theta$).

minimize $\Sigma_{(V_i,V_j)\in E} x_{i,j} \times \Delta_{i,j}$, subject to:

1. $\sum_i x_{i,j} = 1, \forall j$
2. $r_j - r_i \geq \Phi_{i,j}$ if $x_{i,j} = 1$
3. $r_i \leq \theta, \forall i$

Lemma 4 *Problem 6 is equivalent to the optimization problem described above.*

Proof 2 *First, constraint 1 indicates that each vertex V_i, $1 \leq i \leq n$, has one and only one in coming edge as described in Lemma 1. Constraint 2 indicates that no cycle exists in G_W. This can be proven by contradiction: when there exists a cycle $\{V_{k_1}, \ldots, V_{k_l}, V_{k_1}\}$, we have*

$$r_{k_2} - r_{k_1} \geq \Phi_{k_1,k_2}$$

$$\ldots\ldots$$

$$r_{k_1} - r_{k_l} \geq \Phi_{k_l,k_1}$$

$$\Longrightarrow 0 \geq \sum_{i=1}^{l} \Phi_{k_i,k_{(i+1)\%l}}$$

Thus, constraint 1 and 2 ensures the resulting storage graph G_s is a spanning tree. Constraint 3 corresponds to recreation cost constraint in Problem 6, but note that r_i is not necessarily the recreation cost for version V_i.

First, the solution to Problem 6 fulfils all constraints listed in integer linear programming above by setting $r_i = \mathcal{R}_i$. Thus, $\mathcal{C} \geq \xi$. Then, we prove $\mathcal{C} \leq \xi$ by contradiction. Suppose there exists a solution to the linear programming above such that $\xi < \mathcal{C}$. According to constraint 2, 3 and spanning tree property (we let the path from root V_0 to V_j be

$\{V_{k_1} = V_0, V_{k_2}, \ldots V_{k_l}, V_{k_{(l+1)}} = V_j\}$*):*

$$
\begin{aligned}
& r_j \geq r_{k_l} + \Phi_{k_l,j} \geq \ldots \geq r_0 + \sum_{m=1}^{l} \Phi_{k_m,k_{(m+1)}} \\
& \theta \geq r_j \\
\Longrightarrow\ & \theta \geq r_0 + \sum_{m=1}^{l} \Phi_{k_m,k_{(m+1)}}
\end{aligned}
$$

Thus, the recreation cost for each version V_j is fulfilled. Hence, C is not the minimum storage cost in Problem 6, which contradicts the assumption.

Note however that the general form of an ILP does not permit an if-then statement (as in (2) above). Instead, we can transform to the general form with the aid of a large constant C. Thus, constraint 2 can be expressed as follows:

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

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 $x_{i,j} = 1 \Longrightarrow \Phi_{i,j} + r_i - r_j \leq 0$. On the other hand, if $x_{i,j} = 0 \Longrightarrow \Phi_{i,j} + r_i - r_j \leq C$. Since C is "sufficiently large", no additional constraint is added.

3.3 Computational Complexity

In this section, we study the complexity of the problems listed in Table 3.1 under different application scenarios.

Problem 1 and 2 Complexity. As discussed in Section 3.2, Problem 1 and 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 [87] 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 5 and 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:

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

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

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.

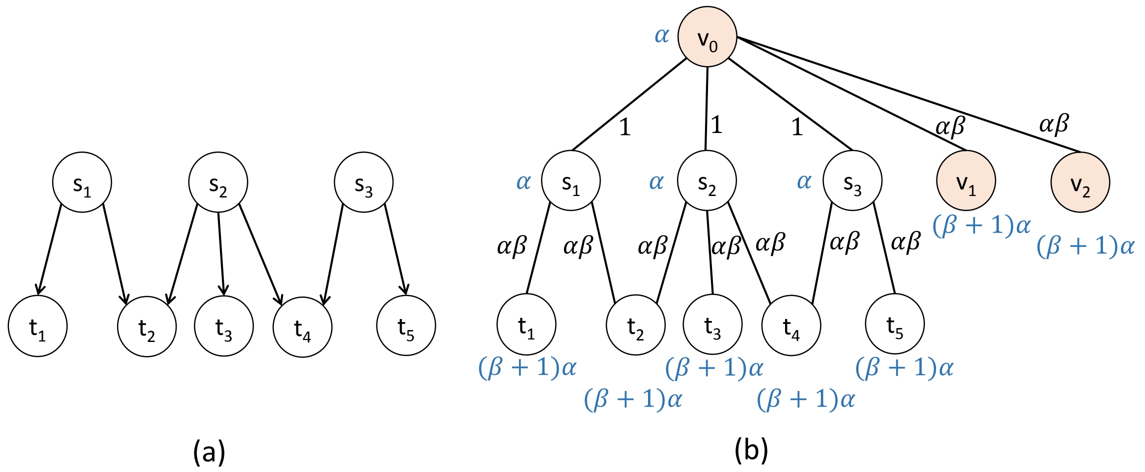


Figure 3.4: Illustration of Proof of Lemma 5

Problem 6 Hardness. We now demonstrate hardness.

Lemma 5 *Problem 6 is NP-hard when $\Delta = \Phi$ and Φ is symmetric.*

Proof 3 *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 6 that will provide a solution to the original set cover problem. The threshold we will use in Problem 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 3.4. 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 3.4(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 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 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 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 6. Then, either we have a set s_j where s_j is connected to t_i in Figure 3.4(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 6 will be identical to the storage graph described above.

Problem 5 Hardness: We now show that Problem 5 is NP-Hard as well. The general philosophy is similar to the proof in Lemma 5, except that we create c dummy vertices instead of two dummy vertices v_1, v_2 in Lemma 5, where c is sufficiently large—this is to once again ensure that v_0 is materialized.

Lemma 6 *Problem 5 is NP-Hard when $\Delta = \Phi$ and Φ is symmetric.*

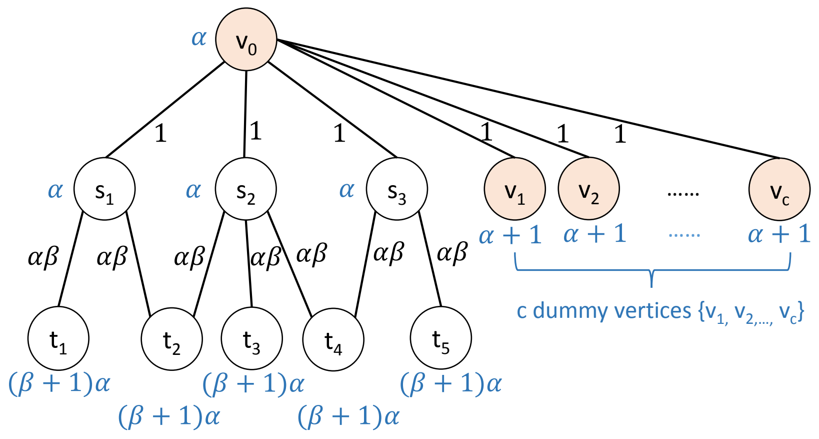


Figure 3.5: Illustration of Proof of Lemma 6

Proof 4 *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 5 that will provide a solution to the original set cover decision problem. The corresponding decision problem for Problem 5 is: given threshold $\alpha + (\beta + 1)\alpha n + k\alpha + (m - k)(\alpha + 1) + (\alpha + 1)c$ in Problem 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 3.5. 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, *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 3.5.) 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 $v_i, 1 \le i \le c$ 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 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 $v_i, 1 \le i \le c$ 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 $v_i, 1 \le i \le c$, or one of the s_i must be materialized, because if not, then the recreation cost of $\{v_1, v_2, \dots, v_c\}$ 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 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, $\forall v_i, 1 \le i \le c$ 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 5. Then, either we have a set s_j where s_j is connected to t_i in Figure 3.5(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 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 4 swaps the constraint and goal compared to Problem 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 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 3, 4, 5, 6, are NP-hard in a more general setting where Φ is not symmetric and $\Delta \neq \Phi$, as listed in Table 3.1.

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 6 for the hop cost variant we described above. The hardness for this special case was demonstrated by [88] using a reduction from the SAT problem:

Lemma 7 *Problem 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 5 and 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 [88, 89, 90]:

Lemma 8 ([88]) *For any $\epsilon > 0$, Problem 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 3.1, this indicates that Problem 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.

3.4 Proposed Algorithms

As discussed in Section 3.2, 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 3.4.1, 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 3 and 5. Second, in Section 3.4.2, 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 4 and 6. We present two variants of the MP algorithm tailored to two different settings.

Then, we present two algorithms — in Section 3.4.3, we present an approximation algorithm called LAST, and in Section 3.4.4, 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 5.5. 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.

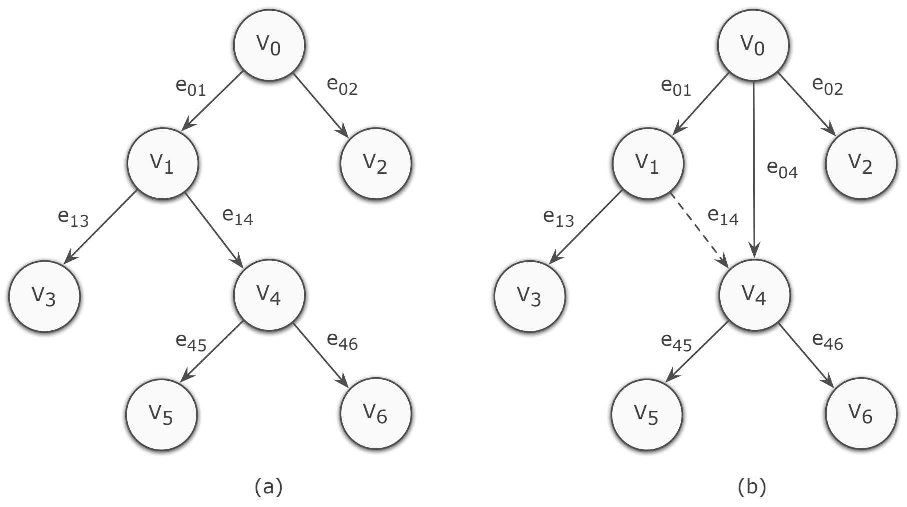


Figure 3.6: Illustration of Local Move Greedy Heuristic

3.4.1 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 (Problem 3); the case when there is no such constraint (Problem 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 5 *Figure 3.6(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 delta to version* 5 *with version* 5 *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. How-

```
Algorithm 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      $(\rho_{max}, e_{SPT}) \longleftarrow (0, \emptyset)$
5      foreach $e_{uv} \in \xi$ do
6          compute $\rho_e$
7          if $\rho_e > \rho_{\max}$ then
8              $(\rho_{max}, \bar{e}) \longleftarrow (\rho_e, e_{uv})$
9          end
10     end
11     $T \longleftarrow T \setminus e_{u'v} \cup e_{uv}$;    $\xi \longleftarrow \xi \setminus e_{uv}$
12     if $\xi = \emptyset$ then
13         return $T$
14     end
15 end
```

ever, 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 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.

3.4.2 Modified Prim's Algorithm

Next, we introduce a heuristic algorithm based on Prim's algorithm for Minimum Spanning Trees for Problem 6 where the goal is to reduce total storage cost while recreation cost for each version is within threshold θ; the solution for Problem 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

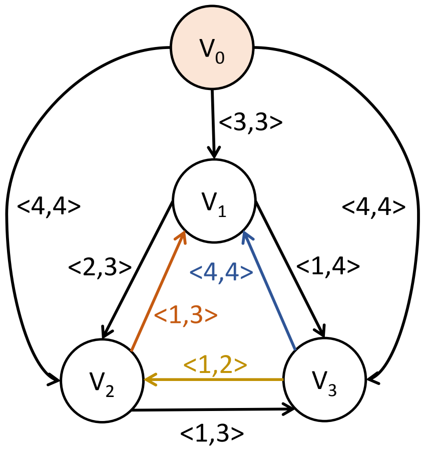


Figure 3.7: Directed Graph G

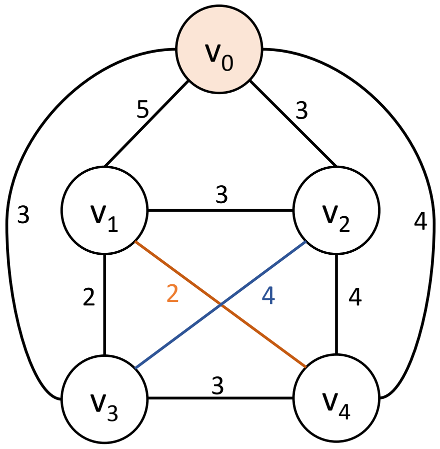


Figure 3.8: Undirected Graph G

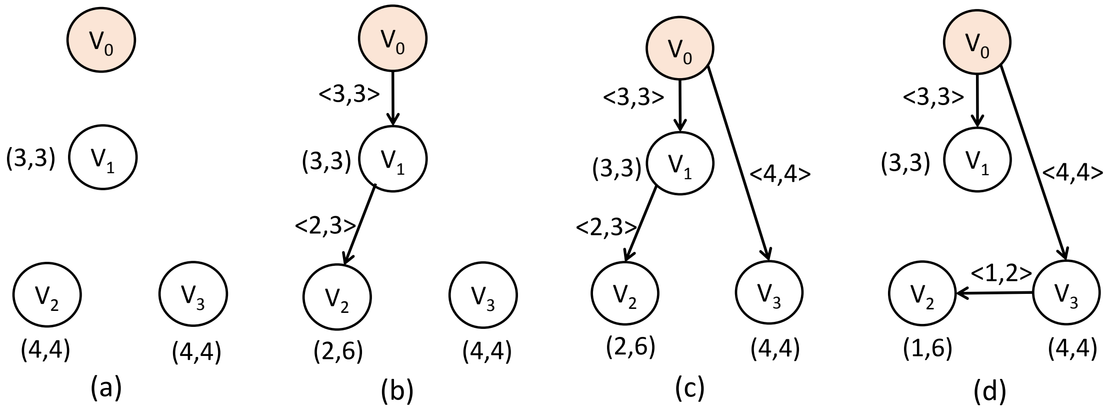


Figure 3.9: Illustration of Modified Prim's algorithm in Figure 3.7

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 2 terminates in $|V|$ iterations since one version is added into X in each iteration.

Example 6 *Say we operate on G given by Figure 3.7, 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 3.9 (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 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 3.9 (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 3.9 (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 3.9 (c). Eventually, we get the final answer in Figure 3.9 (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 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      $< V_i, (l(V_i), d(V_i)) > \longleftarrow$ 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                 $T = T\cup < V_j, p(V_j) > d(V_j) \longleftarrow \Phi_{i,j} + d(V_i)$;
15                 $l(V_j) \longleftarrow \Delta_{i,j}$;
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                 $d(V_j) \longleftarrow \Phi_{i,j} + d(V_i)$;
21                 $l(V_j) \longleftarrow \Delta_{i,j}$; $p(V_j) = V_i$;
22                 enqueue(or update) $< V_j, (l(V_j), d(V_j)) >$ in $PQ$;
23             end
24         end
25     end
26 end
```

3.4.3 LAST Algorithm

Here, we sketch an algorithm from previous work [91] 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 3.2.2, 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. [91] 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 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 3.

Example 7 *Figure 3.10 (a) is the minimum spanning tree (MST) rooted at node V_0 of G in Figure 3.8. 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 3.10 (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 7 according to line 3-7. Since $7 < 2 \times 4$, lines 9-11 are not executed. Figure 3.10 (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 3 is conducted via depth first traversal on MST. It is easy to show that the complexity for Algorithm 3 is $O(|V|)$. The time complexity for computing minimum spanning tree and shortest path tree is $O(|E| \log |V|)$ using heap-based priority queue.

Algorithm 3: Balance MST and Shortest Path Tree [91]

```
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      $(V_i, V_j) \longleftarrow$ the edge currently in traversal;
4      if $d(V_j) > d(V_i) + e_{i,j}$ then
5          $d(V_j) \longleftarrow (d(V_i) + e_{i,j})$;
6          $p(V_j) \longleftarrow V_i$;
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         $d(V_j) \longleftarrow SP(V_0, V_j)$;
11         $p(V_j) \longleftarrow V_0$;
12     end
13 end
```

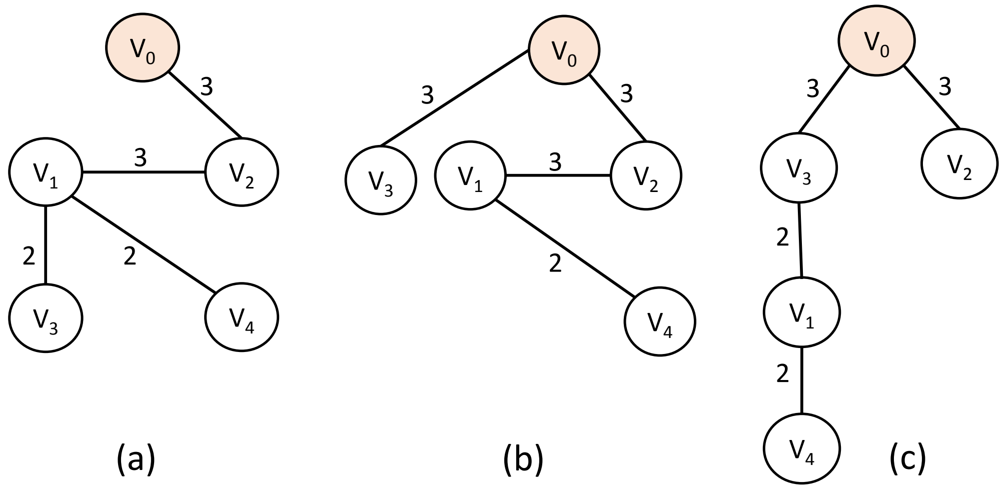


Figure 3.10: Illustration of LAST on Figure 3.8

3.4.4 Git Heuristic

This heuristic is an adaptation of the current heuristic used by Git and we refer to it as GitH. We first describe our understanding of the heuristic used by Git when a user runs `git-repack`, followed by a sketch of GitH.

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 [92]. 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

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

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:

- 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 $b_1, \ldots, b_W$ 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: $[b_1, b_2, \ldots, b_{i-1}, b_{i+1}, \ldots, b_W, O, b_i]$. Then when we move to the new object after O (say O'), we slide the window and so the new window then would be: $[b_2, \ldots, b_{i-1}, b_{i+1}, \ldots, b_W, O, b_i, O']$. 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.

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.

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

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

Table 3.2: Dataset properties

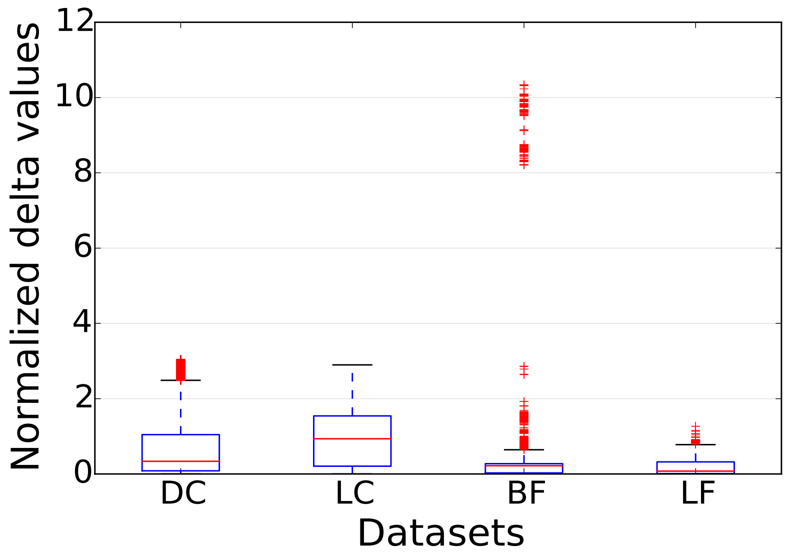


Figure 3.11: Distribution of delta sizes in the datasets (each delta size scaled by the average version size in the dataset)

time to construct deltas.

3.5 Experiments

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 [93] of the Edmonds' algorithm [87] 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.

3.5.1 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[2] takes a two-step approach

[2]Our synthetic dataset generator may be of independent interest to researchers working on version

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 3.2):

- **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

management.

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 lenghts. 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 3.2.

3.5.2 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 that a Berkeley DB-based repository [94]. 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[3]. 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 [97] 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 that at most $O(\log n)$ deltas are needed for reconstructing any version; 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 3.4.4). However as we show later (Fig. 3.12), our implementation of that heuristic (GitH) required more storage than LMG for guaranteeing similar recreation costs.

[3]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" [95, 96].

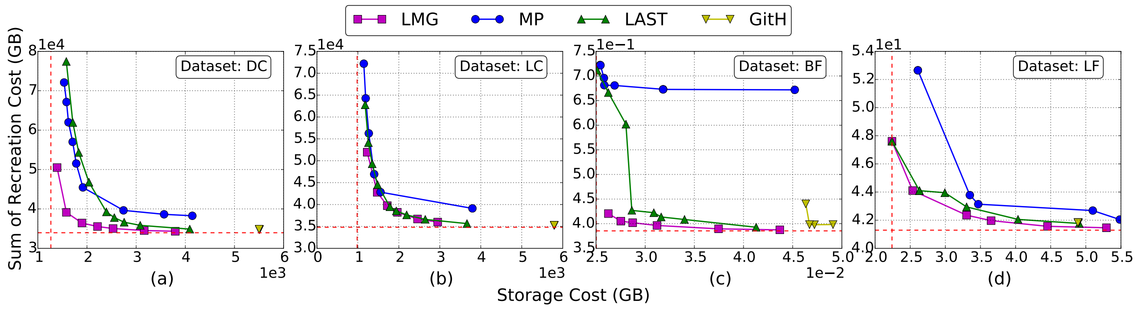


Figure 3.12: Results for the directed case, comparing the storage costs and total recreation costs

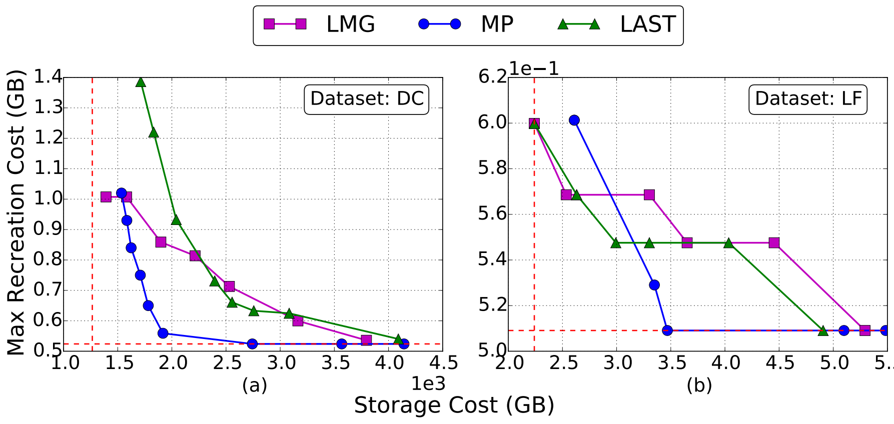


Figure 3.13: Results for the directed case, comparing the storage costs and maximum recreation costs

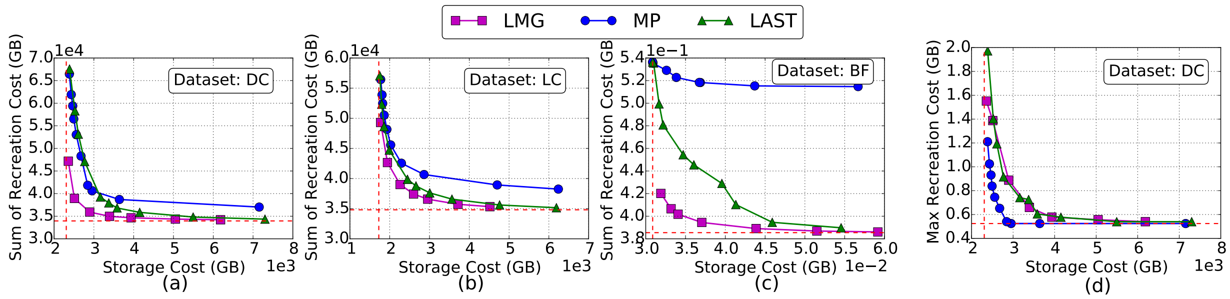


Figure 3.14: Results for the undirected case, comparing the storage costs and total recreation costs (a–c) or maximum recreation costs (d)

3.5.3 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 3.12(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 minimum 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 3.2) as compared to just 34TB for the SPT solution (which is the minimum possible). As we can see from Figure 3.12(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), 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 3.13(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 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.

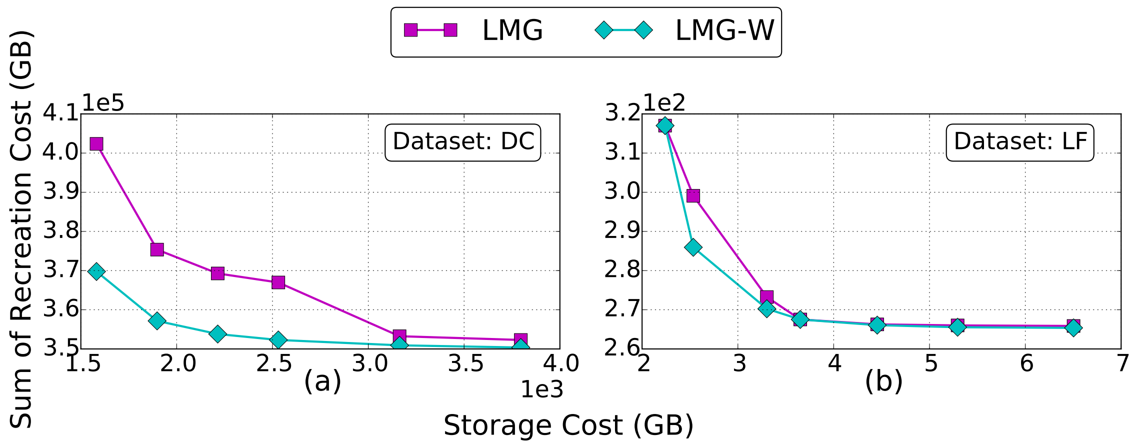


Figure 3.15: Taking workload into account leads to better solutions

Undirected Graphs. We test the three algorithms on the undirected versions of three of the datasets (Figure 3.14). 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 3.15 shows the results for this experiment. As we can see, for the DC dataset, taking into account the 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. 3.16, 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. 3.2) 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

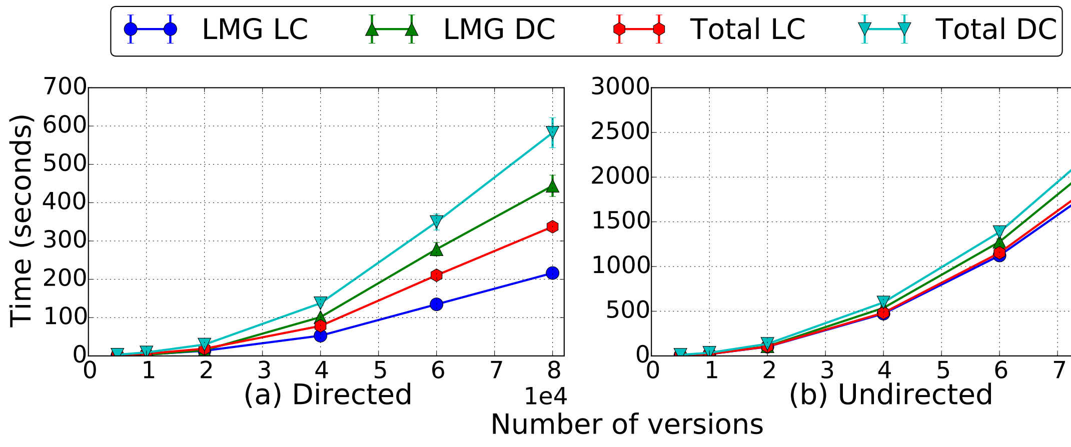


Figure 3.16: Running times of LMG

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 6. We use the ILP formulation from Section 3.2.3 with constraint on the maximum recreation cost (θ), and compare the optimal storage cost with that of the MP algorithm (which resulted 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

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

50 versions denoted by v15, v25 and v50 respectively and compute deltas between all pairs of versions. Table 3.3 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.

Chapter 4: A Unified Query Language for Provenance and Versioning

4.1 Introduction

In this chapter, we present our design of a *version-aware query language*, called VQUEL, capable of querying dataset versions, dataset provenance (e.g., which datasets a given dataset was derived from), and record-level provenance (if available). While `git` and `svn` have proved tremendously useful for collaborative source code management, their versioning API is based a notion of files, not structured records, and as such, is not a good fit for a scenario with a mix of structured and unstructured datasets; the versioning API is also not capable of allowing data scientists to reason about data contained within versions and the relationships between the versions in a holistic manner. VQUEL draws from constructs introduced in the historical Quel [59] and GEM [60] languages, neither of which had a temporal component.

4.2 Preliminaries

Recall that in DEX, a version consists of one or more datasets that are semantically grouped together. Every version is 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

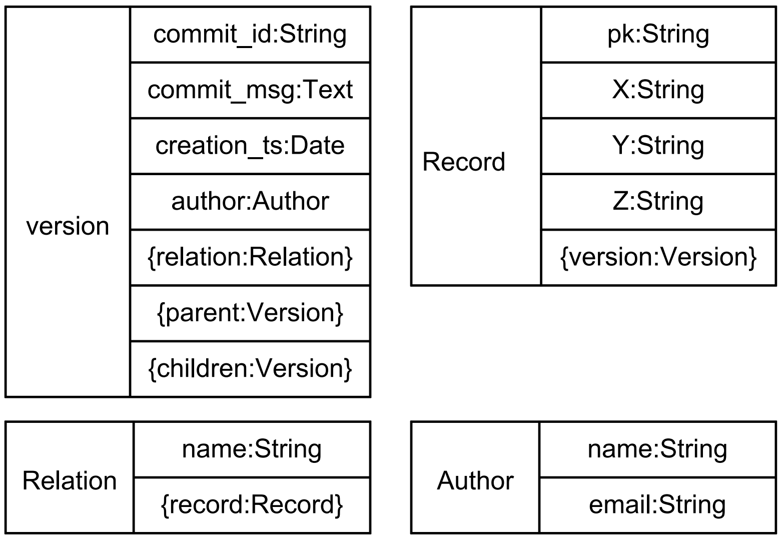


Figure 4.1: Conceptual Data model for VQUEL: the notation "{T}" denotes a set of values of T; fields in the Records entity can be conceptually thought of as a union of all fields across records; other fields and entities (for instance Authors) are not shown to keep the discussion brief; for each entity, entries in the left and right column denote the attribute name and type respectively.

physical data structures are not necessarily immutable, as we described in the previous chapter). 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*.

Figure 4.1 shows a portion of the conceptual data model that we use to write queries against and Figure 4.2 shows an example of a few versions along with the *version graph* connecting them. The data model consists of four essential tables: `Version`, `Relation`, `File`, and `Record`. Additional tables like `Column` and `Author` are required in DEX 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.

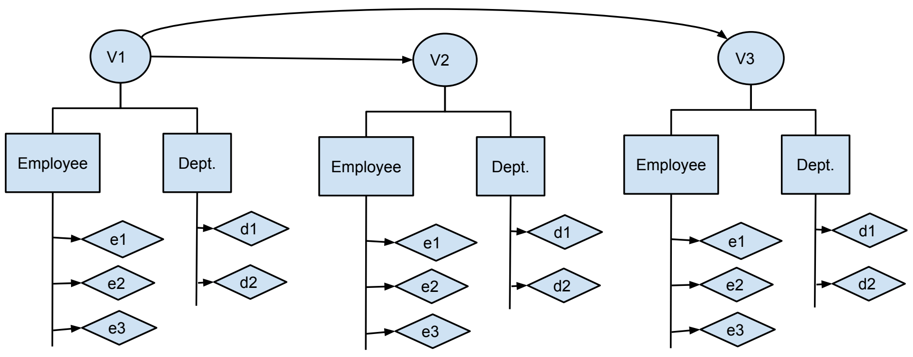


Figure 4.2: An example version graph where circles denote versions; version V1 has two Relations, `Employee` and `Department`, each having a set of records, {E1, E2, E3} and {D1, D2} respectively; version V2 adds new records to both the `Employee` and `Department` relations and also adds a new File, `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.

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 are 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.

4.3 Language Features

VQUEL is largely a generalization of the Quel language (while also introducing certain syntactic conveniences that Quel does not possess), and combines features from GEM 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 4.2, 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 the setting in DEX. For ease of understanding, we first present a version that is clear and easy to understand, but results in longer queries. In Section 4.3.2 we describe additional constructs to make the queries concise.

4.3.1 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 4.3.3. Here, we gradually introduce the constructs of VQUEL as a prelude to the more complex queries combining versioning and data.

Query 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 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 1 and 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 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 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 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
```

4.3.2 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 *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
```

4.3.3 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 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 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 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 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 9 is identical to query 8.

Query 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 9, the result of `count_all` for each group is compared against `100`. Query 10 gives another example.

Query 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 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 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
```

4.3.4 Version graph traversal

VQUEL has three constructs aimed at traversing the version graph. Each of these operate 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 [98]; these combined with the rest of the power of VQUEL enable some fairly challenging queries to be expressed rather easily.

Query 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 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 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)
```

4.3.5 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 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.

Chapter 5: Query Execution I: Set-based Operations

5.1 Introduction

As discussed in Chapter 3, DEX makes use of delta encoding to store past versions of datasets efficiently on disk. Many archival and backup systems, including version control systems like `git`, SVN, etc., often store multiple versions or snapshots of large datasets or files that have significant overlap across their contents using deltas. As a result, there has been significant work on various aspects of delta encoding-based storage systems: computing near-optimal deltas for a variety of data formats [99, 100], quickly finding ideal files to delta from [101], and supporting delta storage in file systems, scientific databases, network transport, etc. [5, 67]. However, existing delta-oriented storage engines offer limited or no support for querying the data stored within them; the primary query type supported by those engines is *checkout*, i.e., reconstructing a specific version of a dataset or a file. With such storage engines becoming efficient and mainstream, there is an increasing desire and opportunity to perform rich analysis queries over the historical information contained within such data stores. The queries of interest include auditing or provenance queries over the datasets (e.g., identify the datasets where a particular property holds), analyzing the evolution of a dataset over time (i.e., temporal analytics), and comparing results of SQL-like queries over different versions of the same dataset (obtained through, e.g., applying different analysis pipelines to the same initial dataset).

However, other delta-oriented storage engines of today require users to "check out" complete file/dataset versions in order to manipulate them. This approach is less than ideal particularly when the individual versions are large and the users need to access multiple versions for their analysis task. In this chapter, we present computationally cheap methods to evaluate a query by *pushing down* query execution to the level of deltas.

5.2 System Overview

We begin with a brief description of the user-facing data model before describing the different types of queries that we support. Thereafter, we describe the system data model, i.e., the physical organization of data, and the primitives used by the system to evaluate the queries.

5.2.1 User Data Model

We recall a few important definitions for ease of exposition. The user data model in DEX has two main abstractions – `datafile`, and *version* – that form the basis of all user interactions.

As mentioned earlier, a **`datafile`** is a file whose contents are interpreted as *set of records.* The user specifies a record separator when a `datafile` is added in the system. Within a `datafile`, we consider a record as an unstructured sequence of bytes. The only constraint we impose, however, is that a `datafile` cannot contain identical records: two records are said to be *identical* if they both have the same sequence of bytes. For instance, textual flat files such as CSV or logs can be seen as containing one record per line.

A **version** is a point-in-time snapshot of one or more `datafiles` typically residing in a directory on the user's file system. A version, identified by a unique ID, is immutable, and can be created at any point in time by any user who has access to the repository.

In addition to `datafiles` and versions, DEX also captures the version-level provenance – derivation and transformation relationships among the set of all versions – in a data structure called the **version graph**. Nodes in a version graph correspond to versions and edges capture relationships such as derivation, branching, transformation, etc, between two versions. Since a version graph is typically much smaller than the `datafile` contents, it can be kept and traversed in memory to identify the versions that are referenced in a query.

We use the following notation to formalize the above discussion. Let $\mathcal{V}$ be the set of all versions. Each version $V \in \mathcal{V}$ contains a finite number of `datafiles`, say, $V = \{A_1, \ldots, A_t\}$. Let $\mathcal{A} = \{A_1, \ldots, A_n\}$ be the set of all `datafiles` across all versions. Note that it is possible for a `datafile` to be present in more than one version – this happens when the said `datafile` is not modified in the respective versions. The set of `datafiles` that appear in a version are kept track of as metadata in the corresponding node of the version graph. Let $A_a = \{r_1, \ldots, r_m\}$ be the set of records contained in `datafile` A_a. As mentioned before, no two records in a `datafile` are identical, i.e., $r_i \neq r_j, \forall r_i, r_j \in A_a$.

5.2.2 Queries

We now describe the semantics of each of the core operations that are the primary focus of this chapter.

Checkout: *Checkouts* are the primary mechanism for reading off older versions of a dataset. Any version or any set of `datafiles` can be checked out, and the result is copied to the location suggested by the user (typically, it will be a directory on the user's machine). When a checkout query is issued, the version graph is consulted to identify the set of `datafiles` that comprise it. Specifically, the checkout operation takes as input a set of $k \geq 1$ `datafiles` $\mathcal{A}_k = \{A_{x_1}, \dots, A_{x_k}\} \subset \mathcal{A}$ and outputs k files, one for each `datafile`. Henceforth, we use the notation $\textsc{Checkout}(\mathcal{A}_k)$ to denote the checkout operation.

Intersect: The *intersect* operation is an important operation when comparing the contents of a `datafile` that was modified across multiple versions. Similar to set intersection, given a set of $k \geq 2$ `datafiles` $\mathcal{A}_k = \{A_{x_1}, \dots, A_{x_k}\} \subset \mathcal{A}$, the intersect operation outputs a single `datafile` containing records that appear in *all* `datafiles` in $\mathcal{A}_k$, i.e., $\{r : r \in A_{x_1} \wedge \cdots \wedge r \in A_{x_k}\}$. We use the notation $I(\mathcal{A}_k)$ to denote the intersect operation.

Union: The *union* operation, denoted by $U(\mathcal{A}_k)$, returns a single `datafile` containing records that appear in *any* of the `datafiles` in $\mathcal{A}_k$, i.e., $\{r : r \in A_{x_1} \vee \cdots \vee r \in A_{x_k}\}$.

***t*-Threshold:** Given as input a set of $k \geq 3$ `datafiles` $\mathcal{A}_k$ and an integer $1 < t < k$, the *t-threshold* operation, denoted by $T_t(\mathcal{A}_k)$, returns a single `datafile` that contains records appearing in *at least t* of the `datafiles` in $\mathcal{A}_k$. This generalizes the above operations – $t = 1$ and $t = k$ correspond to union and intersection respectively.

Although the above set of operations is intended as a starting point for investigating the nascent topic of query processing over deltas, these operations already enable many interesting queries. For example, comparing the results of intersection, union and/or t-threshold across the versions of an evolving dataset can provide insights into the evolution process (e.g., properties of the records that change frequently vs those that remain static). Intersection or t-threshold across the results of different machine learning pipelines on the same input dataset can help us identify which types of records are difficult to predict correctly, which can help an analyst steer the training process. Further, t-threshold can return, for each record, a bitmap indicating the versions to which it belongs; depending on the semantics of the versions being queried, that information could be used for a variety of purposes including correlation analysis, anomaly detection, and visualizations. Finally, if specific analyses of interest are known in advance, materialized views (e.g., projections, results of aggregate queries or joins) can be computed in advance as the dataset versions are ingested; by exploiting the overlaps, these materialized views could be persisted cheaply in the storage engine itself. Although this requires a priori planning, the benefits at the time of querying could be tremendous. Defining and automatically materializing such views remains a rich area for future work.

5.2.3 System Data Model

Next, we discuss the storage graph and the *delta encoding* scheme used in DEX to store the versions of `datafiles` on disk. Thereafter, we describe few properties of the deltas and discuss methods of combining them that will be useful in subsequent sections.

5.2.3.1 Storage Graph

Let $\mathcal{G} = (V, E)$ be a storage graph (see Figure 5.1 for an example). Note that this graph is different from *version graph*, described in section 5.2.1. While the version graph captures derivation or transformation relationships between versions of datasets, the storage graph represents information at the granularity of `datafiles` (encompassing all versions) and is meant to indicate delta relationships between them. Moreover, the storage graph is used by internal query execution routines and, unlike version graph, *is not intended to be exposed to the end user*. The vertex set V of the storage graph captures all unique `datafiles` across all versions, and a special *empty* `datafile`, A_0. Thus, $V = A_0 \cup \mathcal{A}$.

An edge $e(A_i, A_j) \in E$ represents the delta between `datafiles` A_i and A_j, and the edge set E represents the deltas that are chosen to store all `datafiles`. The weight of the edge w_e represents the storage cost (size in bytes) of the delta. For an edge $e(A_0, A_i)$, w_e represents the storage cost of A_i in its entirety (i.e., A_i is **materialized**).

We require that $\mathcal{G}$ be a connected graph so that it is possible to reconstruct any of the `datafiles` in $\mathcal{A}$. Specifically, a path from A_0 to A_i indicates the materialized `datafile` (one following A_0 on the path) and the sequence of deltas to apply in order to recreate A_i. Thus, to store all the `datafiles` in $\mathcal{A}$, it is sufficient to store only the materialized `datafiles` in $\mathcal{G}$ and all the deltas in E.

Prior systems have made use of the storage graph representation [30, 65, 102], albeit with different monikers, to model a delta based solution to store data versions. The storage graph also generalizes the *sequence-of-deltas* model where the versions are ordered according to a certain criteria, e.g., timestamp, file size, etc., and every version except the first is stored as a delta against the previous one. The sequence-of-deltas model, although conceptually simple, has the downside that the retrieval time grows linearly with the number of versions stored. The storage graph representation addresses this limitation by allowing multiple versions to be derived from one version. For instance, if we require that every `datafile` derives 3 `datafiles` not derived by others, we can pack approximately 80K `datafiles` and have a maximum delta sequence of length 10.

5.2.3.2 Set-backed Deltas and Properties

The delta format that we consider in this chapter, called *Set-backed Deltas*, is an undirected delta format, similar to the standard UNIX line-by-line diff. A set-backed delta Δ between a source `datafile` A_i and a target `datafile` A_j, is a set of two `datafiles`, Δ^- and Δ^+, that correspond to "deletions" and "insertions" respectively. Δ^- is the set of records that are present in A_i but not in A_j, while Δ^+ is the set of records that are not present in A_i but present in A_j. Δ can also be used to reconstruct A_i from A_j by exchanging Δ^- and Δ^+.

When using set-backed deltas, we require them to be *consistent* [69], i.e., a delta does not contain the same record in Δ^- and Δ^+. This does not preclude updates to a record, including schema changes, since an update can be recorded as deleting the old record and adding a new record.

Definition 1 (Consistent Delta) *A delta is said to be consistent if* $\Delta^- \cap \Delta^+ = \emptyset$.

Because `datafiles` and deltas are sets, we will often make use of the following three

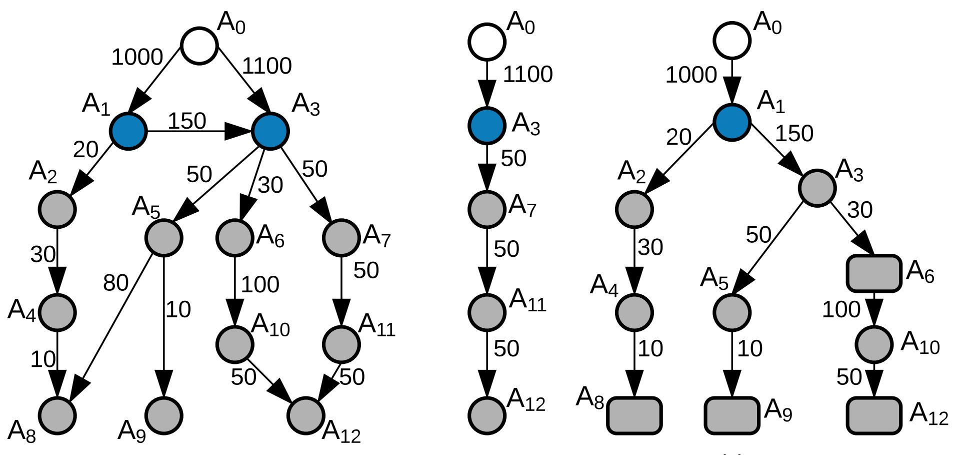


Figure 5.1: (a) A storage graph over `datafiles` $A_1, \ldots, A_{12}$, nodes shaded in blue (A_1, A_3) indicate materialized `datafiles`, edge annotations indicate the disk size of the delta; (b) access tree for $Q(A_{12})$, this is the shortest path from A_0 to A_{12}; (c) access tree for $Q(A_6, A_8, A_9, A_{12})$, this is the minimimum cost Steiner tree for the terminals $\{A_0, A_6, A_8, A_9, A_{12}\}$

standard operations on sets – union (∪), intersection (∩) and difference (−). Continuing the example, when we use Δ to construct A_j from A_i we call this operation **patching** A_i using Δ, and denote it as $A_j = A_i \oplus \Delta$.

Definition 2 (Patch) $A_i \oplus \Delta = (A_i - \Delta^-) \cup \Delta^+$

Observation 1 *If Δ is consistent, $A_i \oplus \Delta = (A_i - \Delta^-) \cup \Delta^+ = (A_i \cup \Delta^+) - \Delta^-$.*

Next, we describe another important property of set-deltas, called *contraction*. Intuitively, delta contraction corresponds to combining two deltas into a single delta such that the new delta has the same effect as applying the individual deltas. Formally, if A_1, A_2, A_3 are three `datafiles` and $\Delta_1 = \Delta(A_1, A_2), \Delta_2 = \Delta(A_2, A_3)$, we use the patch operator as before to represent contraction as follows,

Definition 3 (Delta Contraction) $\Delta = \Delta_1 \oplus \Delta_2$, *where,*

$$\Delta^- = (\Delta_1^- - \Delta_2^+) \cup \Delta_2^-; \qquad \Delta^+ = (\Delta_1^+ - \Delta_2^-) \cup \Delta_2^+ \tag{5.2.1}$$

Although delta contraction, as defined above, can be applied to two arbitrary deltas, the result is well-defined only if the target `datafile` of Δ_1 is same as the source `datafile` of Δ_2. The result Δ has the same source `datafile` as Δ_1 and derives the target `datafile` of Δ_2.

This definition can be generalized to a sequence of deltas: the contraction of a sequence of deltas $\Delta_1, \ldots, \Delta_m$ is the result of the operation $\Delta_1 \oplus \cdots \oplus \Delta_m$.

Given the above properties, we can infer that:

Observation 2 *If Δ_1 and Δ_2 are consistent, then their contraction, $\Delta = \Delta_1 \oplus \Delta_2$, is also consistent.*

Observation 3 *The patch operation is associative, i.e.,* $(\Delta_1 \oplus \Delta_2) \oplus \Delta_3 = \Delta_1 \oplus (\Delta_2 \oplus \Delta_3)$.

Although some of these observations might seem straightforward, formalizing them is crucial to argue the correctness of the transformations that we do later.

5.3 Query Execution Preliminaries

We begin with a more formal treatment of the query optimization problem, with first discussing the optimization metrics of interest and introducing the two-phase optimization approach that we take. We then briefly discuss the issues of cost and cardinality estimation and the search space of query evaluation plans.

Given a query, $Q(\mathcal{A}_k)$ where Q is one of $\{\textsc{Checkout}, I, U, T_t\}$ (section 5.2.2) against a given storage graph $\mathcal{G}$, there are two somewhat independent stages in the overall query execution. First, we need to identify all the relevant `datafiles` and deltas in $\mathcal{G}$ that are necessary to execute $Q(\mathcal{A}_k)$. We refer to this problem as finding an *access tree* of $Q(\mathcal{A}_k)$, and describe it in detail in section 5.3.2.

Second, given an access tree, we need to devise an efficient *evaluation plan*, that describes exactly what operations are used to compute the result of $Q(\mathcal{A}_k)$. This plan is represented as a *delta expression*: an algebraic expression where the operands are `datafiles` and deltas from the storage graph $\mathcal{G}$, and the operations are patch and primitive set operations. During this stage, we also consider the problem of finding a good ordering of evaluating the different operations in the delta expression. We describe the techniques for each query $Q \in \{\textsc{Checkout}, I, U, T_t\}$ in Section 5.4.

5.3.1 Optimization Metrics

To be able to develop a systematic cost-based approach to query execution, we first need to identify appropriate *optimization metrics* and *cost models*. It is unfortunately difficult to develop a single cost metric that captures the costs of the two stages discussed above, which also makes it hard to do joint optimization across them. Because the backend store is likely to be relatively expensive to access (we expect it to be distributed in general), we would like to minimize the amount of data that is read from the backend store; this also reduces the network I/O. Once the data has been gathered, however, the different ways to evaluate a query can have very different CPU costs and wall-clock time. Hence, for the second phase, we would prefer to use a metric that tracks the CPU cost.

We adopt a two-phase approach in DEX inspired by this. We first find the best "access tree" that minimizes the total amount of data that needs to be read (in bytes) from the backend store. In other words, we identify the set of `datafiles` and deltas that have the smallest total size, that are sufficient to reconstruct the required `datafiles`. We then search for the best evaluation plan according to a cost model that estimates the CPU resources needed by the plan. We discuss the specifics in further detail in Section 5.3.4 when we discuss the operator implementations.

We do not explicitly account for disk access costs during the second phase for several reasons. First, although the overall storage graph and the delta sizes in total are expected to be very large, the access tree for any given query is typically much smaller and the deltas constituting that will typically fit in the memory of a powerful machine. More importantly, most of our algorithms (Section 5.3.4) access the deltas sequentially (while reading and writing), and thus even if the deltas were disk resident (or intermediate results needed to be written to disk), the CPU and/or the memory bandwidth is still the main bottleneck. One exception here is binary search or gallop search (that an intersection operation might employ) where our approach might underestimate the cost of an intersection in case of extreme skew. However, our cost estimation procedure can be easily modified to account for that case. Moreover, the deltas are typically stored in a compressed fashion on disk, thereby making it necessary to uncompress them by reading them once into memory, and further making the overall computation CPU-bound.

5.3.2 Access Tree

Given a query $Q(\mathcal{A}_k)$, an access tree, $\mathcal{G}_Q = (V_Q, E_Q)$ is a subgraph of $\mathcal{G}$ such that: (i) $A_0 \cup \mathcal{A}_k \subseteq V_Q \subseteq V$, and (ii) $\mathcal{G}_Q$ is a tree, i.e., a connected graph with no cycles.

The first condition implies that all `datafiles` required by the query are part of the access tree. The second condition ensures that we have a *valid* and *minimal* solution: (i) Valid: because $\mathcal{G}_Q$ is connected, there exists at least one path between A_0 and A_{x_i}, which denotes the materialized `datafile` and the sequence of deltas to apply to reconstruct A_{x_i}, (ii) Minimal: because $\mathcal{G}_Q$ is a tree, for every $A_{x_i} \in \mathcal{A}_k$, $\mathcal{G}_Q$ contains exactly one path from A_0 to A_{x_i}.

We define the *cost* of an access tree as the sum of weights of all edges in it, i.e., $C(\mathcal{G}_Q) = \sum_{e \in E_Q} w_e$. When the edge weights correspond to the sizes of the deltas, this definition captures the cost metric mentioned above. To address the problem of identifying the least cost access tree, we consider two cases, $k = 1$ and $k > 1$. We refer to these as *single* `datafile` *access* and *multiple* `datafile` *access* respectively.

Single `datafile` Access: When $k = 1$, $\mathcal{A}_1 = \{A_{x_1}\}$. Any A_0 to A_{x_1} path in $\mathcal{G}$ satisfies the conditions of an access tree. Thus, finding the least cost access tree amounts to finding the shortest path between A_0 and A_{x_1}, and we use the classical Dijkstra's algorithm.

Multiple `datafile` Access: When $k > 1$, the problem of finding a low cost access tree is equivalent to finding a *Steiner Tree* [103]. Here, the set of nodes $A_0 \cup \mathcal{A}_k$ act as terminals and our objective is to find a minimum cost Steiner tree that contains all of them. This problem is $\mathcal{APX}$-Hard, i.e., arbitrarily good approximations cannot be achieved in polynomial time (unless $\mathcal{P} = \mathcal{NP}$). In this work, we use the classical 2-

approximation algorithm, which finds a tree with cost *at most* 2 times the optimal.

Example 8 *Consider the query* CHECKOUT$(\{A_6, A_8, A_9, A_{12}\})$ *on the storage graph in Figure 5.1(a). Figure 5.1(c) shows the least cost access tree for this query.*

5.3.3 Search Space

Cost-based optimization requires us define the search space of potential, equivalent plans. The search space that we use in this work revolves around two equivalences: (i) associativity of the *patch* operation, and (ii) De Morgan's laws for set theory. We can thus generate equivalent evaluation plans by repeatedly applying those equivalence rules. Unfortunately the number of different evaluation plans is very large, even with just the first rule (Section 5.4.1). Unlike relational query optimization, the set of potential intermediate results is not easy to define either, and thus this problem does not seem amenable to dynamic programming-style algorithms used there. We instead take a hybrid approach where we use a series of heuristic transformation rules, based on De Morgan's laws, to simplify the expressions, and use a dynamic programming-based algorithm (that exploits the associativity of patch) to optimize the sub-expressions in the simplified expression.

Apart from generating alternative query expressions using logical equivalence rules, it is also possible to expand the search space of candidate plans by considering the impact of physical access structures on the data, e.g., secondary indexes. For instance, B-Trees on datafiles or deltas can be helpful when records are filtered on some attribute, bloom filters on deltas can help in evaluating queries like set difference, and so on. Additional considerations also arise when a join result is required across multiple versions – the delta chains for the different sets of datafiles (corresponding to the different relations) may not be "aligned" and the access tree selection will have to consider possibility of "joint" optimizations. Understanding this search space further, especially for richer queries involving joins and aggregates, remains a rich area for future work.

5.3.4 Cost and Cardinality Estimation

The cost of executing any of the set operations mentioned so far depends on the physical `datafile` format and the specific implementation of the operation. Since there exist several implementations for the set operations, there exist several cost functions. In DEX, the primary method of storing a `datafile` is *clustered storage*. In this method, records are stored in a sorted manner based on a suitable derived key (e.g., SHA1). There are several algorithms for evaluating a set expression between two or more operands based on this storage format and we outline our choices next alongwith their respective cost. To keep the discussion simple, we describe algorithms and their respective cost functions when all input data for a specific operation fits in memory and there is no paging of intermediate results to disk. Even if some deltas are large enough to require using disk, most of the algorithms below access the deltas sequentially and thus can be used with small modifications. We note that our optimization algorithms are largely agnostic to the specific choices for operator implementations, and can be used as long as the costs of the operations can be estimated.

Intersection: To compute the intersection of l `datafiles`, $A_1, \ldots, A_l$, we use an *adaptive* algorithm introduced in [83] called Small Adaptive (SA). SA first sorts the set of input `datafiles` according to their size. For each element in the smallest `datafile`, SA performs a *gallop search* on the second smallest `datafile`. A gallop search consists of two stages. In the first stage, we determine a range in which the element would reside if it were in the `datafile`. This range is found by identifying the first exponent j such that the element at 2^j is greater than the searched element. In the second stage, a binary search is performed in the range $(2^{j-1}, 2^j)$ to find if the element exists. If found, a new gallop search is performed in the remaining $l - 2$ `datafiles` to determine if the element is present in the intersection, otherwise a new search is performed. After this step, each `datafile` has an examined range (from the beginning to the position returned by the current gallop search) and an unexamined range. SA then selects two `datafiles` with the smallest unexamined range and repeats the process until one of the `datafiles` has been fully examined.

Because intersections only make sets smaller, as the algorithm progresses with several sets, the time to do each intersection effectively reduces. In particular, as pointed to in [83], the algorithm benefits largely if the set sizes vary widely, and performs poorly if the set sizes are all roughly the same. Since one gallop search takes $O(\log i)$ time, where i is the index where the element would be in the `datafile`, we can model the worst case cost of intersection as,

$$C_\cap(A_1, \dots, A_l) = l|A_1|\log(|A_l|/|A_1|), \tag{5.3.1}$$

where A_1 and A_l are the smallest and largest `datafiles` respectively.

Union: To take a union of l `datafiles` $\{A_1, \ldots, A_l\}$, we perform a linear scan over all lists to merge them, and output the result.

$$C_{\cup}(A_1, \ldots, A_l) = |A_1| + \cdots + |A_l|. \tag{5.3.2}$$

Set Difference: To compute the set difference $A_1 - A_2$, we choose the better among the following two based on input sizes: perform a linear scan over both `datafiles` and use a merging algorithm, or for each element in A_1, perform a gallop search on A_2, including the element in the output if the search fails. This can be captured using the cost function,

$$C_{-}(A_1, A_2) = \min\{|A_1| + |A_2|, |A_1| \log(|A_2|/|A_1|)\}. \tag{5.3.3}$$

Patch: This is a binary operation where the two inputs are either (i) a `datafile` (M) and a delta (Δ), or (ii) two deltas (Δ_1 and Δ_2). In the first case, the output `datafile` can be computed by performing one linear scan over each of M, Δ^+ and Δ^- and evaluating Definition 2, making the cost function,

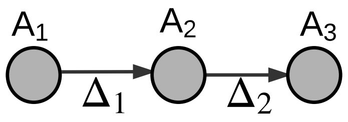
$$C_{\oplus}(M, \Delta) = |M| + |\Delta|. \tag{5.3.4}$$

Typically, $|M| > |\Delta^-|$, and we use the linear scan approach to compute the set difference. In the second case, the output Δ can be computed by evaluating Definition 3. Note that the `datafiles` of Δ_2 are scanned twice, once to compute Δ^+ and once to compute Δ^-.

Thus, the cost function is given as,

$$C_{\oplus}(\Delta_1, \Delta_2) = |\Delta_1| + 2|\Delta_2|. \tag{5.3.5}$$

Cardinality Estimation: Because we restrict the search space as discussed in Section 5.3.3, we require intermediate result size estimates only when two deltas are patched.



Let $\Delta = \Delta_1 \oplus \Delta_2$, where Δ_1 and Δ_2 are deltas between three `datafiles` as above. Let $x = |\Delta^-|$ and $y = |\Delta^+|$. We want to estimate x and y. By definition, Δ is a consistent delta between A_1 and A_3. Therefore, $|A_3| = |A_1| - x + y$. Since $|A_1|$ and $|A_3|$ are known, we can estimate x from y, or vice versa.

From Definition 3 we can obtain intervals for both x and y as,

$$x \in \left[\max(0, |\Delta_1^-| - |\Delta_2^+|) + |\Delta_2^-|, |\Delta_1^-| + |\Delta_2^-|\right],$$
$$y \in \left[\max(0, |\Delta_1^+| - |\Delta_2^-|) + |\Delta_2^+|, |\Delta_1^+| + |\Delta_2^+|\right].$$

We estimate the quantity with the smaller interval, where the value is chosen uniformly at random from the corresponding interval.

5.4 Query Execution Algorithms

Next we present a series of algorithms for cost-based optimization for each of the different query types.

5.4.1 Checkout Queries

Let $\textsc{Checkout}(\mathcal{A}_k)$ and $\mathcal{G}_Q$ denote a checkout query and its access tree resp. We first consider the case when $k = 1$ (single `datafile` checkout) followed by the case $k > 1$ (multiple `datafile` checkout).

5.4.1.1 Single `datafile` Checkout

Recall that the access tree $\mathcal{G}_Q$, when $k = 1$, is the shortest path from A_0 to A_{x_1} in $\mathcal{G}$. The delta expression for single `datafile` checkout is therefore, of the form, $\mathcal{Q} : M \oplus \Delta_1 \oplus \Delta_2 \oplus \cdots \oplus \Delta_m$, where M is the materialized `datafile`.

Evaluation Algorithms: Since the $\oplus$ operation is associative, we can evaluate $\mathcal{Q}$ in multiple ways by changing the placement of "parentheses". For instance, one method is to evaluate the expression from left-to-right, i.e., $\mathcal{Q} : (((M \oplus \Delta_1) \oplus \Delta_2) \oplus \cdots \oplus \Delta_m)$. Alternately, we can evaluate the expression from right-to-left, or in any arbitrary fashion that repeatedly combines two operands at a time, until we are left with the result. These evaluation methods, in general, will have varying costs. The total number of evaluation orders is equivalent to the classical problem of counting the number of ways of associating m applications of a binary operator, and is given by the $(m-1)$th Catalan number, which is $\Omega(4^m/m^{3/2})$.

Note that a greedy algorithm that iteratively combines two deltas having the least cost is not always the optimal strategy.

Example 9 *Consider the expression,* $\mathcal{Q} : \Delta_1 \oplus \Delta_2 \oplus \Delta_3$, *where the deltas are such that* $|\Delta_1| = x, |\Delta_2| \simeq |\Delta_3| = y, x \ll y$. *Intuitively, the deltas* Δ_2, Δ_3 *are larger compared to* Δ_1 *and* *they are such that they almost "undo" each other. The greedy algorithm will pick the plan* $(\Delta_1 \oplus \Delta_2) \oplus \Delta_3$ *with estimated cost* $\approx 2x + 5y$*, while the optimal plan* $\Delta_1 \oplus (\Delta_2 \oplus \Delta_3)$ *has cost* $\approx x + 2y + 2\varepsilon$*, where* $\varepsilon = |\Delta_2 \oplus \Delta_3|$.

For sake of completeness, we have reproduced the classical dynamic programming algorithm to select the (estimated) best evaluation order below. We call this the *path contraction (PC)* algorithm. We use PC extensively in subsequent sections to determine the best evaluation order to combine a sequence of deltas. The runtime of PC is $\Theta(m^3)$ where m is the number of deltas.

The input to PC is a materialized `datafile`, say M, followed by a sequence of deltas, say $\Delta_1, \ldots, \Delta_m$ and the output is the `datafile` represented by the corresponding delta expression, $M \oplus \Delta_1 \oplus \cdots \oplus \Delta_m$. The algorithm uses the property that if the optimal solution splits the contraction of a path of length m into two sub-paths, then the contraction of each of the two sub-paths must be optimal (otherwise, we can improve the solution for the sub-paths to enhance the overall solution). For $0 \leq i \leq j \leq m$, let $C[i, j]$ denote the minimum cost to contract the sequence $\Delta_i, \ldots, \Delta_j$, $D[i, j]$ denote the estimated size of the corresponding intermediate result, and $S[i, j]$ denote how to best split the sequence. For notational convenience only, if M is present, $\Delta_0 = M$. The pseudocode of PC is shown in Algorithm 4.

As discussed in Section 5.3.3, we have syntactically restricted the space of alternative evaluation plans for checkout by only considering the associativity of the patch operation. Although additional transformations could be used to expand the search space, we could not identify any such transformation rules for checkout that were effective outside of pathological cases.

Algorithm 4: Path Contraction (`PC`)

```
Input  : A materialized datafile $M$ (optional); $\Delta_1, \dots, \Delta_m$
Result: Minimum cost to evaluate $M \oplus \Delta_1 \oplus \dots \Delta_m$
1 for $l \longleftarrow 2$ to $m$ do
2     for $i \longleftarrow 0$ to $m - l + 1$ do
3         $j \longleftarrow i + l - 1$
4         $C[i, j] \longleftarrow \min_{i \le k < j} C[i, k] + C[k + 1, j] + C_{\oplus}(D[i, k], D[k + 1, j])$
5         $S[i, j] \longleftarrow \arg\min_{i \le k < j} C[i, k] + C[k + 1, j] + C_{\oplus}(D[i, k], D[k + 1, j])$
          /* Update estimated size of $\Delta_i \oplus \cdots \oplus \Delta_j$ */
6         $D[i, j] \longleftarrow$ EstimateSize($D[i, S[i, j]], D[S[i, j] + 1, j]$)
7     end
8 end
9 return $C[1, m]$ (final cost) and $S$ (splitting markers)
```

5.4.1.2 Multiple `datafile` Checkout

Since $\mathcal{G}_Q$ is a tree, there exists exactly one path from A_0 to each $A_i \in \mathcal{A}_k$. Let $\rho(A_i)$ denote the sequence of deltas on this path. A straightforward method to evaluate the query is to consider the delta expression for each A_i based on the deltas in $\rho(A_i)$ and use `PC` to get the optimal execution order. However, doing so does not take into account the opportunity for shared computation. Specifically, two or more paths may share sub-expressions and we end up evaluating a sub-expression multiple times if we consider each path independently. We illustrate this with the help of an example.

Example 10 *Consider the query* CHECKOUT(A_6, A_8, A_9, A_{12}) *and the access tree in Fig-*

ure 5.1(c). We write one expression for each of A_6, A_8, A_9 and A_{12} respectively, as follows,

$$
\begin{aligned}
\mathcal{Q}\ :\ & A_1 \oplus \Delta(A_1, A_2) \oplus \Delta(A_2, A_4) \oplus \Delta(A_4, A_8);\\
& A_1 \oplus \Delta(A_1, A_3) \oplus \Delta(A_3, A_5) \oplus \Delta(A_5, A_9);\\
& A_1 \oplus \Delta(A_1, A_3) \oplus \Delta(A_3, A_6);\\
& A_1 \oplus \Delta(A_1, A_3) \oplus \Delta(A_3, A_6) \oplus \Delta(A_6, A_{10}) \oplus \Delta(A_{10}, A_{12})
\end{aligned}
$$

Note that if we evaluate each of these independently, based on how the parenthesization is performed, we will evaluate $A_1 \oplus \Delta(A_1, A_3)$ thrice, or $\Delta(A_1, A_3) \oplus \Delta(A_3, A_6)$ twice.

Evaluation Algorithms: The above can be seen as the problem of how to plan the execution of a batch of queries, where each query is a single `datafile` checkout, analogous to multi-query optimization. The goal here is to design a strategy that recognizes the possibilities of shared computation so that we can re-use the result of sub-expressions to the extent possible in order to obtain a globally optimal evaluation plan. To that effect, we develop a dynamic programming algorithm, called *tree contraction (TC)*, to select the best evaluation plan after accounting for shared computation.

At a high level, `TC` breaks up the problem into two questions: how do we decide which sub-expressions to share and how do we best parenthesize each (sub-)expression? We already know how to compute the solution for the latter using `PC`. Therefore, we begin with applying `PC` for all $\rho(A_i)$, $A_i \in \mathcal{A}_k$. However, apart from the best solution for the complete expression, we also return the following information about each path $\rho(A_i)$, all of which is computed by `PC` during its execution. Let $\Delta_1, \dots, \Delta_m$ be the sequence of deltas on the path $\rho(A_i)$ and i, j be indices such that $0 \le i \le j \le m$. Then we return: (i) the minimum cost to contract the sequence $\Delta_i, \dots, \Delta_j$, denoted by $C[i, j]$, (ii) the estimated size of the intermediate delta, denoted by $D[i, j]$, and (iii) the split marker, indicating how to best split the the sequence, denoted by $S[i, j]$. We re-use these estimates of intermediate delta sizes and partial contraction costs wherever the corresponding sub-expressions are considered for sharing. To decide which sub-expression to share, we simply enumerate all possibilities for the sub-expression starting from the root of the access tree (this is done in line 5). The rest of the problem is solved recursively where $\mathcal{L}$ is the sequence of sub-expressions that are considered to be shared. Finally, we also need to account for the possibility of not sharing any sub-expression. The time complexity of `TC` is $O(m^3)$ where m is the number of deltas in the access tree.

The input to `TC` is an access tree $\mathcal{G}_Q$ and the output is the set of `datafiles` $\mathcal{A}_k$. The pseudocode of `TC` is shown in Algorithm 5 and we explain it with the help of Figure 5.2. First, note that if $\mathcal{G}_Q$ has more than one materialized `datafile`, then we can consider the sub-trees rooted at each materialized file independently. Figure 5.2(a) shows an access tree to checkout A_1, A_2, A_3. Let M be the root of this access tree. Starting from M, let B_i be the first node that has $l > 1$ children. Let $B_0, \dots, B_{i-1}$ be the intermediate nodes on the $M - B_i$ path. Let $\mathcal{G}_Q(B_j)$, $0 \le j < i$, be the access tree rooted at B_j (this tree is equivalent to deleting the nodes $M, B_0, \dots, B_{j-1}$ from $\mathcal{G}_Q$). For example, Figure 5.2(b) shows two components: (i) the path $\rho(B_{i-1})$ (above), and (ii) the access tree $\mathcal{G}_Q(B_{i-1})$ (below). Let `split`($\mathcal{G}_Q$) denote the operation that splits $\mathcal{G}_Q$ at B_i into l access trees, one for each child of B_i. This is showin in Figure 5.2(c). Let `split-par`($\mathcal{G}_Q$) denote the operation that splits $\mathcal{G}_Q$ at B_i into l access trees, one for each child of B_i, but this time preserving the parent sequence of deltas in each split. This is shown in Figure 5.2(d).

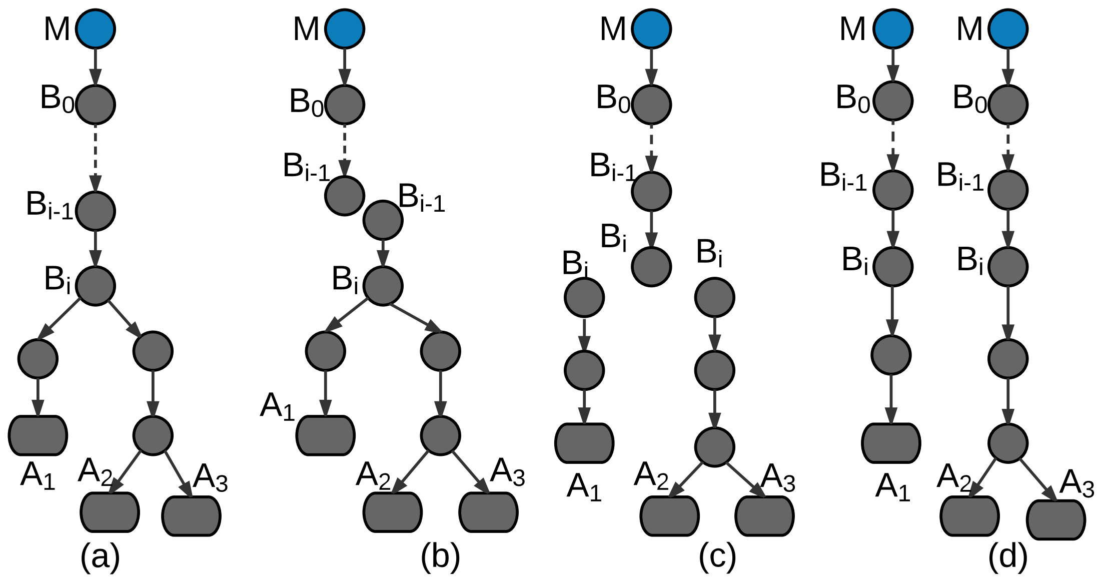


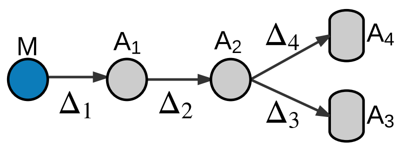
Figure 5.2: An instance of the Tree Contraction algorithm; (a) is an access tree for the query CHECKOUT(A_1, A_2, A_3).

Algorithm 5: Tree Contraction

```
   Input : Access Tree $\mathcal{G}_Q$
1  Apply PC for each $\rho(A_i), A_i \in \mathcal{A}_k$. Memoize $C[], S[]$ and $D[]$.
2  return Best-Subexp($\{\}, \mathcal{G}_Q$)
3  Procedure Best-Subexp($\mathcal{L}, \mathcal{G}_Q$)
       Input : Deltas, $\mathcal{L}$; Access tree, $\mathcal{G}_Q$
4      if $\mathcal{G}_Q$ is a path then return Best solution for $\{\mathcal{L} \cup \mathcal{G}_Q\}$
5      forall $0 \leq j < i$ do
6          $\Delta_{\rho(B_j)} \longleftarrow$ estimated delta for the sequence $\rho(B_j)$
7          $\mathcal{L}' = \mathcal{L} \cup \Delta_{\rho(B_j)}$
8          cost_g $\longleftarrow$ Best-Subexp($\mathcal{L}', \mathcal{G}_Q(B_j)$)
9      $\Delta_{\rho(B_i)} \longleftarrow$ estimated delta for the sequence $\rho(B_i)$
10     $\mathcal{L}' = \mathcal{L} \cup \Delta_{\rho(B_i)}$
11     cost_g $\longleftarrow \sum_{\mathcal{G}'_Q \in \texttt{split}(\mathcal{G}_Q)}$ Best-Subexp($\mathcal{L}', \mathcal{G}'_Q$)
12 cost_g $\longleftarrow \sum_{\mathcal{G}'_Q \in \texttt{split-par}(\mathcal{G}_Q)}$ Best-Subexp($\mathcal{L}, \mathcal{G}'_Q$)
13 return Best cost_g
```

Before we conclude this discussion, it will be helpful to understand, as the following example shows, why a simple greedy strategy of always sharing the largest possible expression (from left to right) is not always optimal.

Example 11 *Consider the following access tree to checkout A_3 and A_4.*



The instance is constructed such that $\Delta_2, \Delta_3, \Delta_4$ are large (say, $y \approx |\Delta_2| \approx |\Delta_3| \approx |\Delta_4|$) and Δ_3, Δ_4 "undo" most of the changes done by Δ_2. Δ_1 is a small independent set of changes, say, $x = |\Delta_1|, x \ll y$. The greedy strategy will force us to share $\Delta' = \Delta_1 \oplus \Delta_2$ and $|\Delta'| \approx x + y$. Thus the cost of the greedy strategy is $\approx 3x + 8y$. On the other hand, evaluating $\Delta_1 \oplus (\Delta_2 \oplus \Delta_3), \Delta_1 \oplus (\Delta_2 \oplus \Delta_4)$ incurs a cost $\approx 2x + 6y$.

5.4.2 Intersection Queries

Given an intersect query $I(\mathcal{A}_k)$ and its access tree $\mathcal{G}_Q$, a straightforward method, that we treat as a baseline, is to first use `TC` to perform Checkout($\mathcal{A}_k$) followed by the intersection. This approach, however, only considers the associativity of patch and the sharing of sub-expressions in order to find a good evaluation order. We now develop a set of *transformation rules* on the access tree that allow us to compute *partial intersection results* using only the deltas. Since a delta between two `datafiles` already captures a notion of difference between them, we leverage this information and avoid redundant computation while finding the intersection.

The transformation rules are based on identifying two simple structures in the access tree $\mathcal{G}_Q$, called *line* and *star* (Figure 5.3). In each figure, we use boxes to denote datafiles in $\mathcal{A}_k$ and circles to denote other datafiles. Also, if a box or circle is filled, it denotes a materialized datafile.

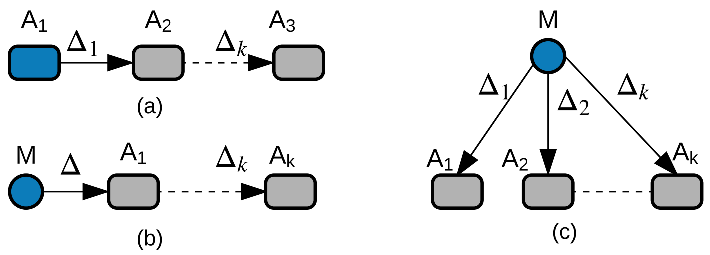


Figure 5.3: (a) A line of two or more datafiles; (b) A line when the materialized datafile M is not a part of query input; (c) A star.

Line Access Trees: Consider the query $I(A_1, A_2)$ with the datafiles as arranged in Figure 5.3(a). Here, A_1 is the materialized datafile while A_2 is stored as a delta from A_1. It is easy to see that:

$$R = A_1 \cap A_2 = A_1 - \Delta_1^-.$$

In general, for the query $I(\mathcal{A}_k)$ with the datafiles as arranged in Figure 5.3(a), the result R is computed as,

$$R = I(\mathcal{A}_k) = A_1 - (\Delta_1^- \cup \cdots \cup \Delta_{k-1}^-) \tag{5.4.1}$$

Note that the above equality does not hold if there are other datafiles in $\mathcal{G}_Q$ even if $\mathcal{G}_Q$ is a line. We use this equality to introduce our first transformation rule that "reduces" the deltas in a line structure to a single delta that gives the result for the intersect query. Conceptually, this reduced delta acts as a delta between two datafiles: the same materialized datafile as in the line and a (new) datafile representing the intersection result.

T1:– If $\Delta_1, \ldots, \Delta_{k-1}$ are the deltas in the line, then the reduced delta, Δ_l, for the intersect query is composed as,

$$\Delta_l^- = \Delta_1^- \cup \Delta_2^- \cup \cdots \cup \Delta_{k-1}^-; \quad \Delta_l^+ = \emptyset$$

This transformation rule significantly reduces the amount of data that needs to be subsequently processed.

To handle the case when the materialized `datafile` is not a part of the line, as in Figure 5.3(b), we use a two-step approach. First, assuming that A_1 is the materialized `datafile` and we can use rule **T1** to compute the reduced delta Δ_l. Second, we can contract Δ and Δ_l since they share the `datafile` A_1. The result is computed as $R = M \oplus \Delta \oplus \Delta_l$.

Next, we discuss how to evaluate eq. (5.4.1). Consider the identity, $X - (Y \cup Z) = (X - Y) - Z$, for three sets X, Y, Z. If $|Y|, |Z| \ll |X|$, observe that $X - (Y \cup Z)$ will often have less cost than $(X - Y) - Z$. Intuitively, if we do the set difference first, then $|X - Y|$ will be comparable to $|X|$ and will end up being scanned again. Specifically, under the cost model stated in section 5.3.4, when $|X| > 3\max(|Y|, |Z|)$, performing $X - (Y \cup Z)$ will result in a reduced cost. We therefore use the following greedy heuristic when evaluating eq. (5.4.1).

H1:– Let $\mathcal{L} = \{\Delta_1^-, \ldots, \Delta_{k-1}^-\}, R = M$. We iteratively perform the following until $\mathcal{L}$ is empty: let Δ' be the largest size delta in $\mathcal{L}$. If $|R| > 3|\Delta'|$, we replace the largest two deltas in $\mathcal{L}$ by their union; else, we set $R = R - \Delta'$.

Star Access Trees: Consider the query $I(A_1, A_2)$ with the `datafiles` as arranged in

Figure 5.3(c). Here, M is the materialized `datafile` and A_1 and A_2 are stored as deltas from M. We have that:

$$R = A_1 \cap A_2 = (M - (\Delta_1^- \cup \Delta_2^-)) \cup (\Delta_1^+ \cap \Delta_2^+)$$

To see why, recall that Δ_i^- indicates the set of records to be removed from M to get A_i. Hence, no record in Δ_i^- can be a part of the intersection result. Additionally, new records (that do not exist in M) can be added only if they belong to all of Δ_i^+.

In general, for the query $I(\mathcal{A}_k)$ with the `datafiles` as arranged in Figure 5.3(c), the result R is computed as,

$$R = I(\mathcal{A}_k) = \left(M - (\cup_{i=1}^k \Delta_i^-)\right) \cup (\cap_{i=1}^k \Delta_i^+) \tag{5.4.2}$$

The result R is written in terms of the materialized `datafile` M. This leads us to our second tarnsformation rule that "reduces" the deltas in a star structure to a single delta that gives the result for the intersect query. Conceptually, this reduced delta acts as a delta between M and the intersection result.

T2:– If $\Delta_1, \ldots, \Delta_k$ are the deltas in the star, then the reduced delta Δ_s, for the intersect query is composed as,

$$\Delta_s^- = \cup_{i=1}^k \Delta_i^-; \quad \Delta_s^+ = \cap_{i=1}^k \Delta_i^+$$

We use **H1** to evaluate Equation (5.4.2). Since none of Δ_i^+ can help reduce intermediate result sizes, the intersection of Δ_i^+s can be done independently. Finally, we also make the following observation.

Observation 4 *Δ_l and Δ_s are consistent.*

Arbitrary Access Trees: We develop an algorithm, called *Contract and Reduce (C&R)*, that puts the above two techniques together for arbitrary access trees. With minor modifications, the same algorithm can be used for other types of queries, and hence we describe its general form. The pseudocode for C&R is shown in Algorithm 6.

Starting with a query $Q \in \{I, U, T_t\}$, and its access tree $\mathcal{G}_Q$ as inputs, C&R iteratively evaluates partial delta expressions, effectively reducing the size of $\mathcal{G}_Q$. Each iteration of the algorithm has two phases: *contract* phase and *reduce* phase. In the contract phase, we identify all *maximal continuous* delta paths: a path where all nodes, except the start and end node, have exactly 2 neighbors, and none of the intermediate nodes is a part of $\mathcal{A}_k$. Each path should be of length > 2 and be the longest possible. Every such path is then contracted to a single delta using PC. Specifically, if $\Delta_1, \dots, \Delta_u$ is the sequence of deltas on the path between two nodes A_x and A_y in $\mathcal{G}_Q$, we use PC to find the best order to evaluate $\Delta_\rho = \Delta_1 \oplus \cdots \oplus \Delta_u$, execute the operations, and replace the sequence by the delta Δ_ρ between A_x and A_y.

In the reduce phase, we find all lines and stars in $\mathcal{G}_Q$ and reduce them according to the appropriate transformation rules – **T1/T3** for lines and **T2/T4/T5** for stars. Each transformation takes as input two or more deltas, either in a line or star configuration, and replaces them by a single delta. Note that if all paths are contracted, and number of deltas in $\mathcal{G}_Q$ is more than 1, there will at be at least one reduction to be performed.

The algorithm ends when there is only one delta remaining in $\mathcal{G}_Q$. At this point, we simply apply the delta to the materialized `datafile` in $\mathcal{G}_Q$ and return the result. We illustrate the behaviour of the algorithm with the help of an example.

Example 12 *Consider the query* $I(A_6, A_8, A_9, A_{12})$ *with access tree as in Figure 5.4(a). In the first iteration, during the contract phase, we compute the deltas: (1)* $\Delta_{c1} = \Delta_1 \oplus \Delta_3 \oplus \Delta_6$*, (2)* $\Delta_{c2} = \Delta_4 \oplus \Delta_7$*, and (3)* $\Delta_{c3} = \Delta_8 \oplus \Delta_9$ *(Figure 5.4(b)). In the reduce phase where we reduce* A_6 *and* A_{12} *arranged in a line (Figure 5.4(c)). This reduction puts* A_9 *and* $Q(A_6, A_{12})$ *in a star, which is then reduced as in Figure 5.4(d)). In the next iteration, during the contract phase, we compute* $\Delta_{c4} = \Delta_2 \oplus \Delta_{s1}$ *(Figure 5.4(e))). In the reduce phase, we reduce the star* A_8 *and* $Q(A_6, A_9, A_{12})$ *which leaves the access tree with a single delta.*

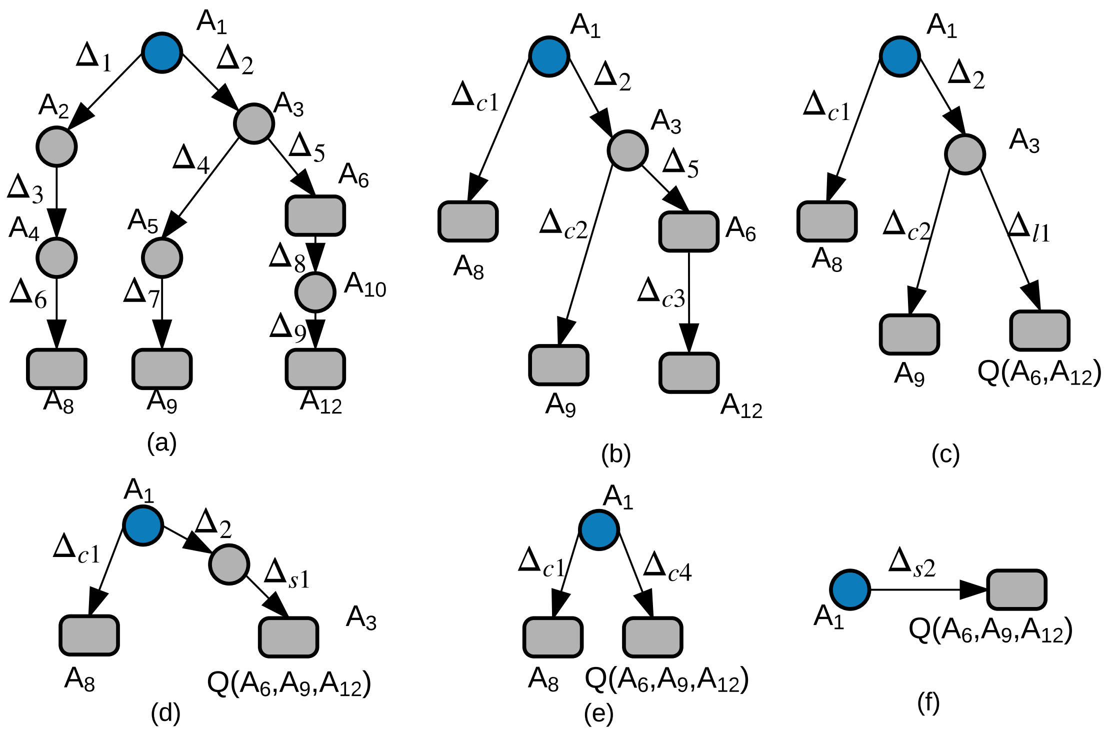


Figure 5.4: Access tree during the progress of `C&R`

Algorithm 6: Contract and Reduce (C&R)

```
   Data: Query $Q \in \{I, U, T_t\}$ and access tree $\mathcal{G}_Q$
1  while $\mathcal{G}_Q$ contains more than one delta do
2      Contract Phase:
3          $P \longleftarrow$ list of maximal continuous delta paths of length > 2 in $\mathcal{G}_Q$
4          Contract all paths in $P$ using PC and update $\mathcal{G}_Q$
5      if $\mathcal{G}_Q$ becomes a line/star then
6          return $R \longleftarrow$ result using H1
7      Reduce Phase:
8          $L \longleftarrow$ list of line structures in $\mathcal{G}_Q$
9          Reduce each line in $L$ based on $Q$, i.e., apply one of T1/T3, and update $\mathcal{G}_Q$
10         $S \longleftarrow$ list of star structures in $\mathcal{G}_Q$
11         Reduce each star in $S$ based on $Q$, i.e., apply one of T2/T4/T5, and update
             $\mathcal{G}_Q$
   /* At this point $\mathcal{G}_Q$ contains one delta.                          */
12 return $R \longleftarrow Root(\mathcal{G}_Q) \oplus \Delta$
```

5.4.3 Union

In this section, we give transformation rules for *line* and *star* for the query $U(\mathcal{A}_k)$. We can then use C&R with the mentioned rules to evaluate arbitrary access tree structures.

Line: Consider the query $U(\mathcal{A}_k)$ with the datafiles as arranged in Figure 5.3(a). Then: $R = A_1 \cup (\cup_{i=1}^{k} \Delta_i^+)$.

The transformation rule for a line can therefore be stated as,

T3:– If $\Delta_1, \dots, \Delta_{k-1}$ are the deltas in the line, then the reduced delta, Δ_l, for the union query is composed as,

$$\Delta_l^- = \emptyset; \quad \Delta_l^+ = \cup_{i=1}^{k} \Delta_i^+$$

Star: If the datafiles are arranged as shown in in Figure 5.3(c), we have that: $R = U(\mathcal{A}_k) = \left(M - (\cap_{i=1}^{k} \Delta_i^-)\right) \cup (\cup_{i=1}^{k} \Delta_i^+)$.

To see why, since Δ_i^- indicates the set of records to be removed from M to get A_i, if a record is absent in the union, it must have been present in all Δ_i^-. New records that are added in any Δ_i^+ are a part of the union result. Then:

T4:– If $\Delta_1, \dots, \Delta_k$ are the deltas in the star, then the reduced delta Δ_s, for the union query is composed as,

$$\Delta_s^- = \cap_{i=1}^k \Delta_i^-; \quad \Delta_s^+ = \cup_{i=1}^k \Delta_i^+$$

We conclude this section by mentioning that similar to the intersection case, Δ_l and Δ_s, for the union query, are consistent.

5.4.4 *t*-Threshold

In order to evaluate a t-threshold query $T_t(\mathcal{A}_k)$, we make use of multiset-backed deltas during intermediate query execution, instead of the set-backed deltas that we have been using so far. This introduces two important issues during the execution of C&R that are the main focus of this section. First, we need to re-define the semantics of delta contraction in this new setting. Second, a line cannot be reduced in a straightforward manner as before. We begin with some definitions, describe the transformation rule for a star, and then discuss each of two issues in detail.

A multiset, unlike a set, allows multiple instances of any of its elements. We represent a multiset as $A = \{(r, c) : c \in N_{\geq 1}\}$, where r is an element and $N_{\geq 1}$ is the set of natural numbers. The number c is referred to as the multiplicity of r. A set is a multiset with all multiplicities as 1.

Consider the following two operations concerning multisets.

Definition 4 (Multiset Union) *Multiset union, denoted by* $A = A_1 \uplus A_2$*, returns a multi-*

set containing elements that occur either in A_1 or A_2, where A_1 and A_2 are either multisets or sets. The multiplicity of an element in A is the sum of its multiplicites in A_1 and A_2.

Definition 5 (Multiset Restrict) *If A is a multiset, $A_{c\leq p}$ is the <u>set</u> of elements in A with multiplicity at most p. Similarly, $A_{c\geq p}$ is the <u>set</u> of elements with multiplicity at least p.*

We now describe how to evaluate $T_t(\mathcal{A}_k)$ when $\mathcal{G}_Q$ is a star.

Star: Consider the query $R = T_3(\mathcal{A}_4)$, i.e., find all records that appear in at least 3 of $\{A_1, A_2, A_3, A_4\}$, when they are arranged in a star (Figure 5.3(c)). Suppose the delta Δ is such that $\Delta^- = \Delta_1^- \uplus \Delta_2^- \uplus \Delta_3^- \uplus \Delta_4^-$ and $\Delta^+ = \Delta_1^+ \uplus \Delta_2^+ \uplus \Delta_3^+ \uplus \Delta_4^+$. Here, Δ^- and Δ^+ are multisets, and Δ is a multiset-backed delta. Then:

$$R = T_3(\mathcal{A}_4) = (M - \Delta^-_{c\geq 2}) \cup \Delta^+_{c\geq 3}$$

To understand why, consider a record $(r, c) \in \Delta^-$. This indicates that $r \in M$ and c of 4 deltas, $\{\Delta_1^-, \Delta_2^-, \Delta_3^-, \Delta_4^-\}$, ask to delete r. So long as $c \geq 2$, r will absent from at least 2 of $\{A_1, A_2, A_3, A_4\}$. Similarly, consider a record $(s, c) \in \Delta^+$. This indicates that $s \notin M$ and c of 4 deltas, $\{\Delta_1^-, \Delta_2^-, \Delta_3^-, \Delta_4^-\}$, ask to add s. So long as $c \geq 3$, k will be present in at least 3 of $\{A_1, A_2, A_3, A_4\}$.

More generally, the transformation rule for a star can be stated as below.

T5:– If $\Delta_1, \ldots, \Delta_k$ are the deltas in the star, then the reduced delta Δ_s, for the t-threshold query is composed as,

$$\Delta_s^- = \uplus_{i=1}^k \Delta_i^-; \quad \Delta_s^+ = \uplus_{i=1}^k \Delta_i^+$$

We note the following: (i) With every multiset-backed delta, we keep an integer value $2 \leq p(\Delta) \leq k$ which indicates the number of `datafiles` in $\mathcal{A}_k$ that are reduced by this delta. For instance, $p(\Delta) = 4$, in the previous example. In general, the access tree will have several disjoint stars (or lines) which are reduced at different times in the evaluation process. We discuss how $p(\Delta)$ is used shortly. (ii) The deltas $\Delta^-_{c \geq k-t+1})$ and $\Delta^+_{c \geq t}$ are consistent.

We now describe the semantics of the patch operation in the presence of multiset deltas.

Delta Contraction: Algorithm 7 computes the result of $\Delta = \Delta_x \oplus \Delta_y$, where Δ_y is a multiset delta. Note that due to the nature of `C&R`, only the last delta in any sequence of deltas can be a multiset delta. The result delta Δ is also a multiset delta.

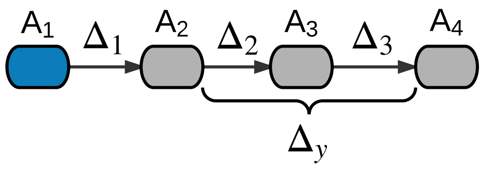
The main idea here is to preseve semantics of two values: (i) the multiplicity c of a record r, and (ii) the number of `datafiles` that are reduced due to the delta, $p(\Delta)$. Since this is a patch operation, we can simply set $p(\Delta) \longleftarrow p(\Delta_y)$. Recall that a record $(r', c') \in \Delta^-_y$ indicates that c' of $p(\Delta_y)$ deltas ask us to remove r'. Now consider a record $r \in \Delta^+_x$. If $r \notin \Delta^-_y$, then we can add $(r, p(\Delta_y))$ to Δ^+. However, if $r \in \Delta^-_y$, then we need to "fix" the multiplicity of r, i.e., add $(r, p(\Delta_y) - c)$ to Δ^+, where c is the multiplicity of r in Δ^-_y. The other case is similar.

We use Algorithm 7 in place of the patch operator defined in section 5.3.4 when one of the operands is a multiset delta. Its estimated cost is modeled as, $C_\oplus(\Delta_x, \Delta_y) = |\Delta_x| + 2|\Delta_y|$, and we can use `PC` during the contract phase as before.

Line: Consider the query, $R = T_2(\mathcal{A}_4)$, with the `datafiles` as shown below.

Algorithm 7: Patch operation for multiset-based deltas

```
Data: Set-backed delta $\Delta_x$, and multiset-based delta $\Delta_y$
Result: Multiset delta $\Delta = \Delta_x \oplus \Delta_y$
1  Initialize $\Delta \longleftarrow \Delta_y, p \longleftarrow p(\Delta) \longleftarrow p(\Delta_y)$
2  for $r \in \Delta_x^+$ do
3      if $r \in \Delta^-$ then
4          Remove $(r, c)$ from $\Delta^-$, Add $(r, p - c)$ to $\Delta^+$
5      else
6          Add $(r, p)$ to $\Delta^+$
7  for $r \in \Delta_x^-$ do
8      if $r \in \Delta^+$ then
9          Remove $(r, c)$ from $\Delta^+$, Add $(r, p - c)$ to $\Delta^-$
10     else
11         Add $(r, p)$ to $\Delta^-$
12 return $\Delta$
```



Consider a record r such that $r \in \Delta_1^+$, and $r \in \Delta_3^-$. Although r is present in two opposite deltas, $r \in R$. More generally, in the case of t-threshold queries, simply knowing whether a record is in Δ_i^- or Δ_i^+ is not sufficient to conclude if it is present in the result. We also require knowledge of the "position" of the delta containing the record on the line. Alternately, we can reduce the line by considering deltas in right-to-left order, by the following simple modification to Algorithm 7. Suppose that we know how to contract $\Delta_2 \oplus \Delta_3$ to obtain a multiset delta Δ_y as shown. We show how to modify Algorithm 7 to compute $\Delta = \Delta_1 \oplus \Delta_y$. The central idea is again to set record multiplicites and $p(\Delta)$ correctly. Note that $p(\Delta_y) = 2$, as it reduces A_3 and A_4. Since, Δ is also meant to reduce A_2, we set $p(\Delta) = p(\Delta_y) + 1$ (line 1). Consider a record $r \in \Delta_1^+$. If $r \notin \Delta_y^-$, then we can add $(r, p(\Delta_y) + 1)$ to Δ^+ (line 6). On the other hand, if $r \in \Delta_y^-$, we add $(r, p(\Delta_y) - c + 1)$ to Δ^+

(line 4) where c is the multiplicity of r in Δ_y^-. The other case is similar.

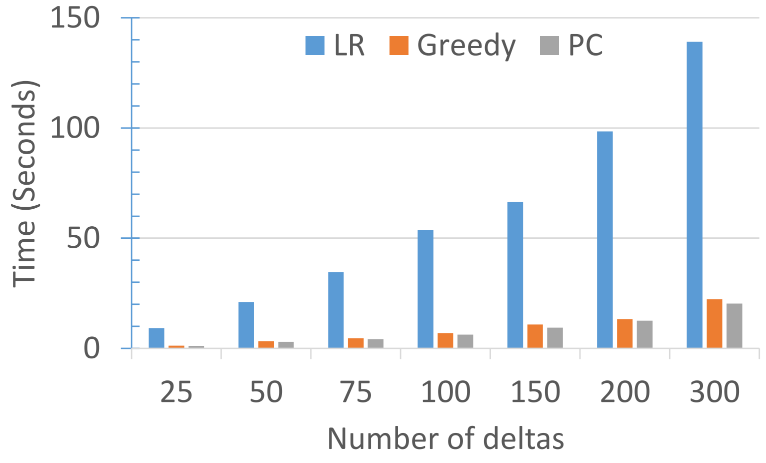


Figure 5.5: Effect of varying $\#\Delta$ when $|\Delta| = 5\%$

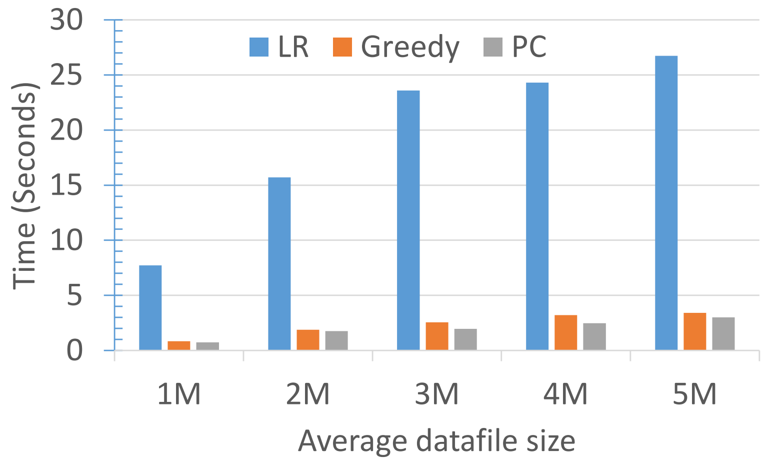


Figure 5.6: Effect of varying $|A|$

5.5 Experimental Evaluation

In this section, we present a comprehensive evaluation of our DEX prototype. The key takeaway from our study is that, pushing down computation to the deltas can lead to signifincant savings, an order-of-magnitude in many cases. Surprisingly, even for a single `datafile` checkout, we see large benefits in the computational time. We also show,

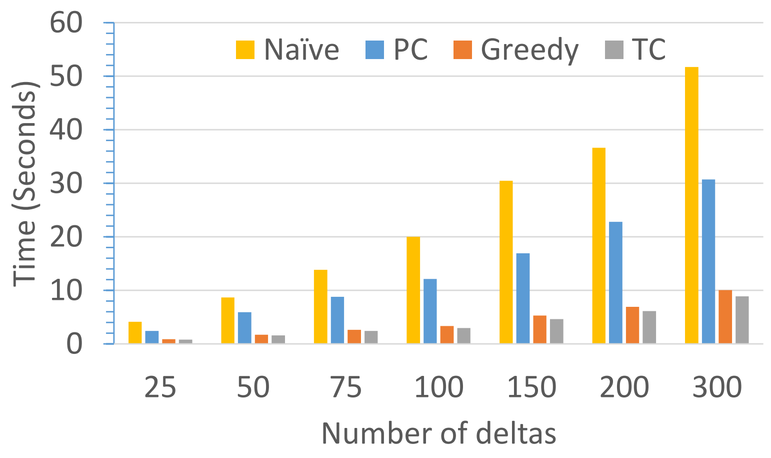


Figure 5.7: Effect of varying #Δ when |Δ| = 5%

through an illustrative experiment (Section 5.5), that using auxiliary data structures like bitmaps can increase the benefits many-fold, indicating that this is a rich direction for future work.

All experiments were conducted on a single machine with Intel Core i7-4790 CPU (3.60 GHz, 8MB L3 cache), 32GB of memory, running Ubuntu 16.04 and OpenJDK 64-bit server JVM (ver. 1.8.0_111). Our choice to write the query processor in Java was primarily based on getting quick development time while still being reasonably performant on large datasets. While using a low-level language (e.g., C) will reduce the absolute query execution times, it will not change our primary objective which is to measure relative speedup of our techniques compared to the baseline. All time measurements are recorded as wall-clock time. Unless otherwise stated, to measure response time, we run each query 10 times and consider the median. To account for the adaptive performance of some of the set operations, we repeat the above on 25 datasets with identical properties (described next) and report the median. As discussed in Section 5.3.4, our computations are CPU bound, and we did not find an appreciable difference in warm cache vs cold cache settings; for consistency, we report results for a warm cache setting.

Datasets: Lacking access to real-world versioned datasets with sufficient and varied structure, we instead developed a synthetic data generator to generate datasets with very different characteristics for a wide variety of parameter values. This enables us to carefully study the performance of our techniques in various settings. Formally, every experiment setting is characterized by a 4-tuple, $\langle T, |A|, |\Delta|, \#\Delta \rangle$, where $|A|$ and $|\Delta|$ refer to the average number records in a `datafile` and average size of the deltas in the dataset (as a percentage of $|A|$). T denotes the shape of the access tree that is used, and is one of: *line-shaped* (l), *star-shaped* (s) and *line-and-star* (ls); and $\#\Delta$ refers to the number of deltas in the access tree. All records are 64-byte randomly generated strings.

Parameter	**Explanation**	**Values**
k	Query size	2, 4, 6, 8, 10
$\lvert A\rvert$	Average `datafile` size	1 million(M), 2M, 3M, 4M, 5M
$\lvert\Delta\rvert$	Average delta size	1%, 2%, 3%, 4%, 5%
$\#\Delta$	Number of deltas	10, 25, 50, 75, 100
T	Shape of access tree	Line (l), Star (s), Line-and-star (ls)

Table 5.1: Possible values of parameters characterizing a synthetic dataset

Single `datafile` Checkout: We begin with evaluating the performance of `PC`, i.e., Algorithm 4, against two heuristics for the case of single `datafile` checkout. Figure 5.5 shows the median response time of this analysis (in milliseconds) on the vertical axis, and the horizontal axis is the number of deltas ($\#\Delta$) in the expression. The other parameters of the dataset are fixed at $\langle T = l, |A| = 3M, |\Delta| = 5\% \rangle$.

The `LR` heuristic simply evaluates the delta expression from left-to-right starting with the materialized `datafile`. This is the standard heuristic used in prior delta-based

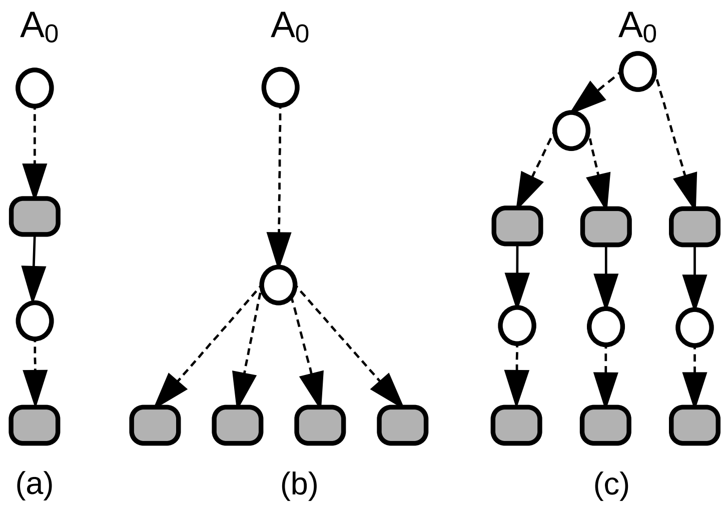


Figure 5.8: Access tree shapes; (a) Line, (b) Star, (c) Line-and-star

storage engines, like `git`. On the other hand, the GREEDY heuristic iteratively patches two operands having the least estimated cost.

We observe that in each instance, PC performs better than GREEDY which performs better than LR. Specifically, we note up to **7.0-8.8X** improvement in median response times when comparing PC with LR and up to 14% improvement when comparing with GREEDY. The performance gap between LR and the other methods also increases slightly as the number of deltas goes up. This is because the left input of every patch operation in LR has a large size, in contrast to both GREEDY and PC, that "balance" their inputs in a cost-based manner. Also, because we assume that every record in a `datafile` is equally likely to be modified and there is no set of "hot" records, i.e., records that are modified often, we observe that the intermediate result sizes continue to grow in GREEDY and PC as well. We observe similar trends for other delta sizes and omit their results.

Next, we study the effect of varying average `datafile` size on the response times. Figure 5.6 shows the result of this study on the dataset $\langle T = l, |\Delta| = 1\%, \#\Delta = 100\rangle$ when

$|A|$ is varied from 1 million records to 5 million records. In this case, we observe a **8.9-10.5X** speedup when compared to LR, with the GREEDY solution being approximately close to PC.

Finally, although PC has cubic time complexity in the number of deltas, the solutions it finds are, in all cases, better than alternatives even after taking optimization time into consideration. When $\#\Delta = 100$, the average time to find the optimal solution was 1.2ms.

Multiple `datafile` Checkout: We now evaluate the time taken to checkout $k = 8$ `datafiles` on the dataset $\langle T = ls, |\Delta| = 5\%, |A| = 1M \rangle$. We evaluate TC, i.e., Algorithm 5, by comparison against three approaches. The NAIVE approach simply performs a checkout of each `datafile` independently using LR. The second approach uses PC to checkout individual `datafiles`. Both these approaches do not take into account sharing of intermediate results. The third approach, called GREEDY, shares the results of the largest sub-expressions as much as it can (e.g., for two `datafiles`, the result of the expression from the root of the access tree to their lowest common ancestor is always shared).

Figure 5.7 reports the median checkout time (in seconds) as the number of deltas ($\#\Delta$) in the access tree is varied. We observe that overall GREEDY and TC have similar response times and TC performs slightly better than GREEDY in each case (between $7.2 - 10.8\%$ improvement). Also, when compared to NAIVE, we observe a **5.1–6.8X** improvement in median response time.

The average optimization time when $\#\Delta = 300$ was 18.4ms.

Intersect: In the following set of experiments, we compare the running time of evalu-

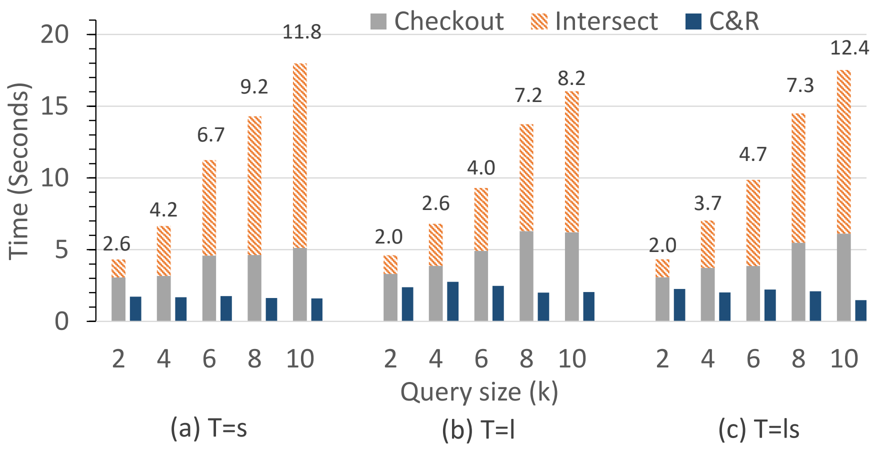


Figure 5.9: Effect of access tree structure when $|\Delta| = 1\%, \#\Delta = 100$

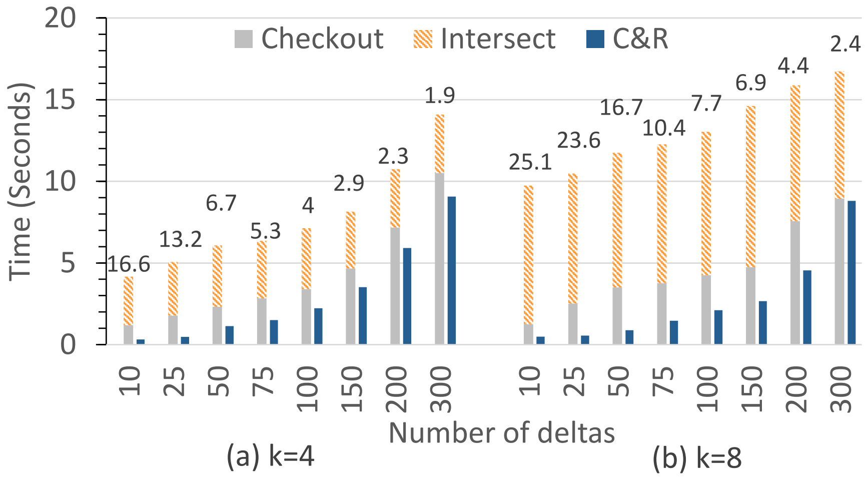


Figure 5.10: Effect of query size when $|\Delta| = 1\%$

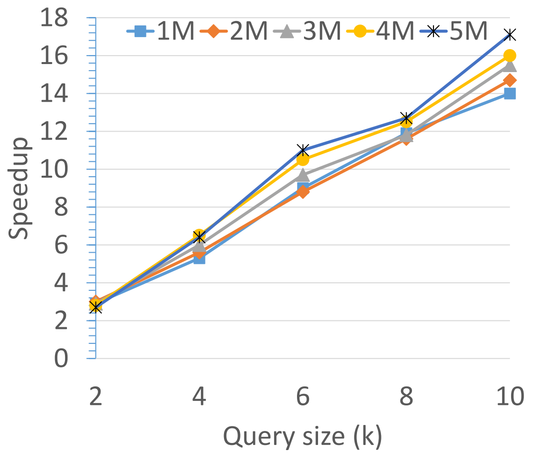


Figure 5.11: Intersect – Effect of $|A|$

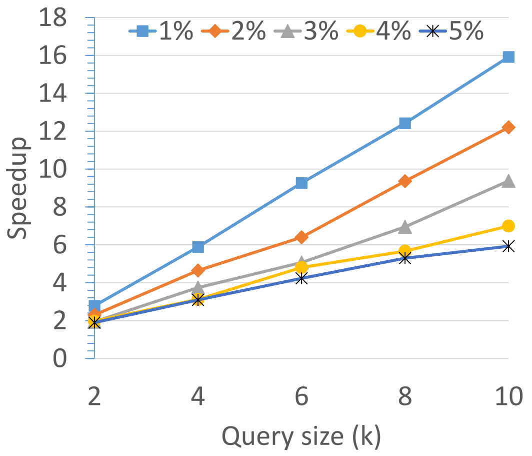


Figure 5.12: Intersect – Effect of $|\Delta|$

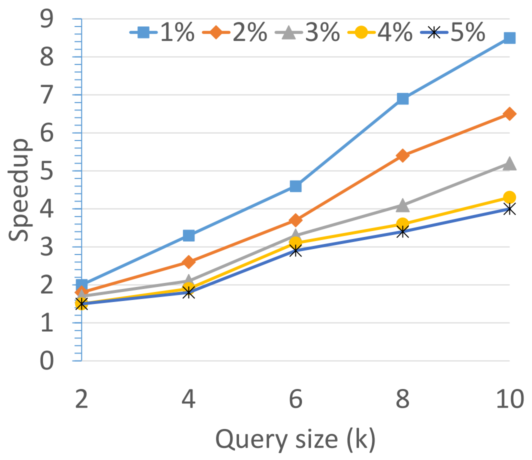


Figure 5.13: Union – Effect of $|\Delta|$

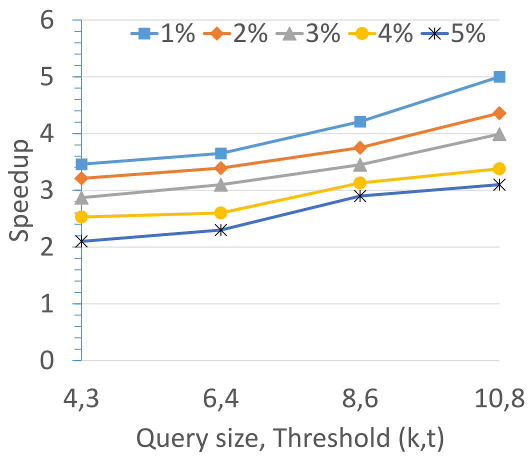


Figure 5.14: t-Thres. – Effect of $|\Delta|$

ating $I(\mathcal{A}_k)$ using two algorithms. The baseline approach simply performs a checkout of all the `datafiles` in $\mathcal{A}_k$ using `TC`, followed by their intersection. The second approach measures the performance of `C&R`, i.e., Algorithm 6.

Because `C&R` makes decisions based on the shape of the access tree, we first study the effect of varying the shape of the access tree on intersect performance. Figure 5.9 shows the median response time against the query size for the three types of access trees: line, star, and line-and-star. The other parameters of the dataset are $\langle |A| = 3M, |\Delta| = 1\%, \#\Delta = 100\rangle$. The numbers on top of each bar indicate the speedup obtained. We note speedups of upto **12X** when using `C&R`. The speedup obtained for $T = l$ is smaller than others primariliy due to the shape of the access tree – in a line, the smallest path between root and a query `datafile` cannot be reduced using any of the tranformation rules and must be contracted using `PC`.

In the next experiment, reported in Figure 5.10 we study the effect of varying the number of deltas in the access tree of $I(\mathcal{A}_k)$. Here, we use the dataset $\langle T = ls, |A| = 3M, |\Delta| = 1\%\rangle$ and vary $\#\Delta$; we report the results for $k = 4, 8$. As we can see, our techniques are particularly effective, giving a speedup upto **16X** and **25X**, for $k = 4$ and $k = 8$ respectively. The speedup decreases as the number of deltas increases primarily due to larger intermediate delta sizes.

Figure 5.11 shows the speedup obtained when the average `datafile` size is varied between 1M and 5M records; other dataset parameters are $\langle T = ls, |\Delta| = 1\%, \#\Delta = 50\rangle$. We observe that our techniques show significant benefit, obtaining upto **17X** speedup. Further, we note that `datafile` size does not affect `C&R` to a large degree.

Figure 5.12 reports the speedup obtained when the average delta size in the dataset, $|\Delta|$, is varied between 1% and 5% of the average `datafile` size; other dataset parameters are $\langle T = ls, |A| = 3M, \#\Delta = 50\rangle$. We note a speedup of **2.8-16X** when $|\Delta| = 1\%$ that decreases gradually to **2-6X** when $|\Delta| = 5\%$. This confirms our hypothesis that if the deltas between the `datafiles` are small, significant improvements can be obtained by using the deltas in query execution in a more direct manner. When the deltas get large, the intermediate result sizes grow too, which results in a reduced speedup.

Union: The results for $U(\mathcal{A}_k)$ are similar to the intersection case although with smaller speedup values. We report one such result in Figure 5.13: the effect of varying query size (k) for datasets with different average delta size $|\Delta|$. The other parameters of the dataset are $\langle T = ls, \#\Delta = 50, |A| = 3M\rangle$. We note a speedup of **1.6-8.6X** when $|\Delta| = 1\%$ that decreases gradually to **1.5-4.1X** when $|\Delta| = 5\%$.

t-Threshold: We use the adaptive algorithm of [104] as a baseline for our t-threshold experiments. Similar to adaptive set intersection, this algorithm uses gallop search in order to find the position of an element r in a set. Moreover, it maintains a min heap of size $k - t + 1$, containing at most one element per set, in order to select a "good" element to probe other sets during each iteration. Figure 5.14 reports the effects of varying (k, t) across datasets with different delta sizes. The other parameters of the dataset are $\langle T = ls, \#\Delta = 50, |A| = 3M\rangle$. We observe a speedup of **3.5-5X** when $|\Delta| = 1\%$ that gradually reduces as the delta size increases. When $|\Delta| = 5\%$, we report a speedup of **2.1-3.1X**. The overall speedup in this case is less than that obtained in the intersection or union query because unlike the two, the size of the intermediate results does not decrease when transforming lines and stars.

Experiments with Bitmap Deltas: We have also built support for a filtered index to answer intersection and union queries, and we show the results for an illustrative experiment. Akin to a relational database, a filtered index in DEX is suited to answer queries that always select from a finite "universal" set of records. In this case, we can encode a set of records using a bitmap, where the order of records is determined by their SHA1 value. The index creation step creates a bitmap of size $|\mathcal{K}|$ for each materialized `datafile` and two bitmaps for each delta in the storage graph. We can then use the bitwise AND($\wedge$), OR($\vee$) and NOT($\neg$) operations to compute set intersection, union and difference. In this experiment, we use a compressed bitmap library called *roaring bitmaps* [105] . Figure 5.15 shows the effect of index size on the intersect query. Here, we measure the speedup vs query size for index size ranging from 500K-3M records. As expected, for small universal sets, we get largely improved speedup ratios (up to **1200X**). With large universe sizes, there is however a penalty incurred when selecting the records themselves given the bitmap information.

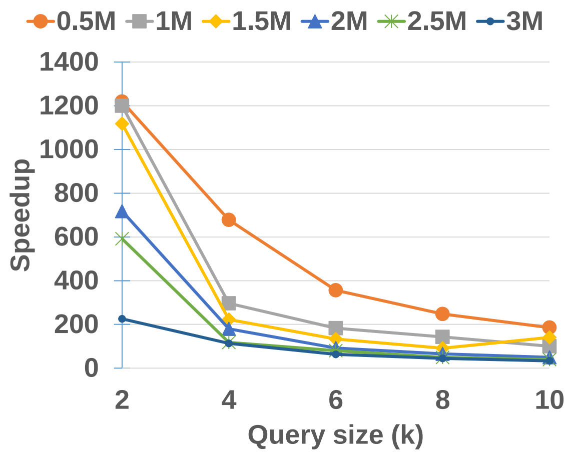


Figure 5.15: Effect of bitmap size

5.5.1 Comparisons with Temporal Indexing

In this section, we present a comparison of our approach with the temporal indexing techniques by Buneman et al. [24], and discuss how we reimplemented and compared against their approach.

Buneman et al. [24] proposed an archiving technique based on identifying changes to (keyed) records across versions, specifically temporal versions of hierarchical data, that are then merged into one hierarchy represented in XML format. Because they also compared against a diff-based storage solution, we present a brief comparison to highlight the respective strengths and weaknesses of the two strategies. Broadly stated, their scheme, henceforth referred to as BA, merges all hierarchical elements across versions into one hierarchy by identifying an element by its key and storing it only once, along with the sequence of version timestamps where the respective element appears. Answering a checkout query thus requires scanning the entire archive, and using the intervals to decide which elements belong to the answer. We reimplemented their technique in our framework, using either sorted lists or bitmaps to store the sequence of version ids where an element appears. We describe this next.

We represent a dataset of records as a one-level hierarchical document with all the records as children of the root node. When merging two datasets into a single archive, we identify the common records and only store them once. Due to the nonlinear nature of "version ids" (unlike timestamps) in our problem setting, we tried two different implementations to keep the set of version ids for an element/record: (1) a sorted list or (2) a bitmap. In the sorted list implementation, the version ids associated with every record are stored in a sorted array and during retrieval, we use binary search to decide if the record is present in the desired version. In the bitmap implementation, a bitmap of size equal to the number of versions in the archive is used with each record to indicate the versions that the record is present in. We use the roaring bitmap library [105] to store these bitmaps. During retrieval, a simple scan through the archive can retrieve any version. We note that it is not clear how to extend some of the optimizations in Buneman et al., most notably "timestamp trees", that depend on the linearity of timestamps, to the nonlinear nature of version ids in a decentralized versioning/data lake scenario.

Buneman et al. compared the performance of their archiver against two approaches based on deltas: (i) "cumulative diff", where every version is stored as a delta against a common (typically first) version, and (ii) "incremental diff", or "sequence-of-deltas", where every version is stored as a delta against the previous version, resulting in a line storage graph. However, cumulative diff had a large space overhead [24], and incremental diff results in large checkout times due to long delta chains.

We consider single ($k = 1$) and multiple ($k = 4$) `datafile` checkout on the dataset $\langle |\Delta| = 5\%, |A| = 1M \rangle$. Additionally, for `BA`, we vary the number of `datafiles` (N) in the archive as $N = 10, 50, 100, 250, 500, 1000$. The bitmap implementation gives superior performance (up to 19%) for $N = 100$ onwards and we use that to report checkout time, while for $N = 10, 50$, we use the sorted list implementation. As noted previously, we can pack approximately $N = 80K$ `datafiles` in a storage graph (with certain constraints) and get a delta chain of size at most 10 to checkout any single `datafile`; therefore, we set $\#\Delta = 10, 25$ for fair comparison.

	DEX; #Δ		BA; archive size (N)					
	10	25	10	50	100	250	500	1000
$k = 1$	125	550	97	388	659	1286	2677	5362
$k = 4$	263	483	144	596	1110	2484	5160	9550

Table 5.2: Median checkout time (ms) in DEX and `BA`

As we can see, `BA` performs better than a sequence-of-deltas approach for checkout queries. When $k = 1$ storing $N = 50$ `datafiles` gives better response times in `BA` than storing $N = 25$ `datafiles` in a sequence-of-deltas approach (demonstrated by $k = 1, \#\Delta = 25$). However, checkout times for `BA` increase rapidly as the archive size grows, and DEX is vastly superior to `BA` under more reasonable assumptions about the storage graph (in the context of a versioning/data lake scenario, it is not clear how to extend some of the optimizations in [24] that depend on the linearity of timestamps).

In short, the main difference between DEX and the sequence-of-deltas approach (that [24] primarily compared against) is that we assume that the storage graph is constructed using a technique that avoids very long delta chains (e.g., the methods presented in Chapter 3, "skip links"-based approach [65], techniques that balance storage and retrieval costs [106], greedy heuristic used by `git`, etc.). We further note that `BA` suffers from three major limitations: (i) the entire archive must be read even when checking out a single version, (ii) adding a new version requires an expensive merge operation that scans the entire archive (unlike a delta-oriented storage engine where only a single delta may be added), and (iii) decentralization is much more difficult in BA (in theory one could maintain multiple archives and merge them periodically, but we are not aware of any work that has attempted that).

Chapter 6: Query Execution II: Declarative Queries

6.1 Introduction

The preceding chapter described a query processing architecture for a simple class of queries that compared data from multiple versions. As we showed, even these simple queries exhibit interesting and unexplored computational challenges and the benefits of optimizing their execution can be tremendous (orders-of-magnitude in many cases). In this chapter, we take a step further and describe an architecture to execute a much richer class of queries from the VQUEL language described in Chapter 4. Specifically, we consider the query optimization challenges of executing a single block select-project-join (SPJ) query on multiple versions.

Our motivation regarding studying this SQL-like API is that tabular data formats (CSV/TSV) are extremely common in data science workflows. Oftentimes analyzing and inspecting past dataset versions can involve executing rich diff queries to understand how subsets of data have changed across versions. For example, a user might need to slice a few columns from a CSV file present in multiple past versions, as a result of updates from different sources, and only consider those records that appear in a majority of the versions. She may also want to do the same for a dataset that is logically a "join" of the records from two or more CSV files. Currently, users either use a combination of *nix utilities like `sort`, `uniq`, `cut`, etc., to accomplish these tasks, or they may use a number of open source tools such as Miller [107], csvkit [108], textql [109], etc., that provide SQL-like API to quickly wrangle and work with table-like data formats. However, these ad hoc approaches suffer from the same limitations as discussed before – simply checking out the files in every version and evaluating the user-specified query is prone to a lot of wasted work, especially because of the overlap between the versions.

The key idea behind processing such rich queries in DEX is that we execute the query plan exactly once, regardless of the number of versions. Each record/tuple that is processed in the query plan is actually a data structure called a *v-tuple*. Apart from storing data about the different columns required by the query, a v-tuple also keeps track of a *version list*, or *version bitmap*, of the versions (among the subset of the queried versions) that the tuple appears in. The potential performance benefit of this approach is that the physical query processing operators can operate in batch across versions that are encoded for every tuple in the respective tuple bundle. For example, if a query is to be run on 50 versions, and if a column c has value 100, then a selection operator corresponding to $c = 100$ can filter out tuples across all 50 versions in one operation, possibly resulting in a 50-fold reduction in the number of tuples that have to be moved across the query plan.

This chapter presents the design of a query processing engine that we have built for executing SPJ queries over multiple versions. In summary, our contributions are as follows.

1. We propose a new framework to execute single block select-project-join (SPJ)

queries over multiple versions stored in a delta-based system. The key design decision of our approach is to execute the various query operators once for each unique tuple in the set of versions, rather than executing the query once for each version.

2. We propose using a new representation for tuples, called v-tuple, during query processing. In addition to the regular fields of a tuple, a v-tuple also has information about which versions (among the set of queried versions) the respective tuple appears in. We develop algorithms to efficiently create such v-tuples from the delta representation and to join/aggregate them.

3. We implement our techniques in DEX using Apache Calcite, which is an open source, highly customizable engine for parsing and planning queries on data in a variety of formats. Specifically, we add new operators to Calcite for retrieving data (Scan) and processing data (Filter/Join).

4. We extensively evaluate the performance of our methods on multiple synthetic datasets. Our results show that DEX makes richer query processing viable in a dataset version control system, with significant benefits over version-at-a-time execution.

6.2 System Design

We review a few terms before describing our extensions to DEX to enable declarative query processing. Conceptually, the execution engine evaluates a query Q over a number of existing versions, say k, and outputs the result with each record being annotated with the respective version(s) it appears in. Recall that a `datafile` is a file whose contents are interpreted as a *set of records*. A *version* is a point-in-time snapshot of one or more `datafiles` typically residing in a directory on the user's file system. A version is identified by a unique id, is immutable, and can be created by any user who has access to the repository. A *version graph* captures the version-level provenance that includes the derivation and transformation relationships, and metadata about the versions themselves. Nodes in a version graph correspond to versions, and edges capture relationships such as derivation, branching, transformation, etc., between two versions. Both nodes and edges have metadata that can be used to allow writing rich queries over the entire repository. In this chapter, however, we only make use of the node metadata to identify the relevant versions during query processing.

6.2.1 Schema Specification

Since our focus is on executing queries on table-like data, when a `datafile` is added to the system, we also require the user to submit a schema file that specifies how to parse the text-based format into rows and columns. By default, each line is processed as an array of columns, all values being of type `string`. The schema file can specify the data type for individual columns. Finally, the schema file must also designate a column, or a group of columns, as the *primary key*, that must contain a unique value that can be used to identify each and every row of the `datafile` uniquely. At commit time, both intial commit, and any subsequent commits after updates, the `datafile` is converted into Apache Parquet format for physical storage, as described in Section 6.2.3. Additionally, a Parquet file can also be given as a input during commit time. We require such rich schema information regarding the `datafiles` primarily to support *column deltas*, as discussed next.

6.2.2 Delta format

As noted before, the choice of the delta format has significant implications on what operations/queries can be run on it. When deciding on a delta format between two versions of a `datafile`, we had the following considerations:

1. the format should be compact,
2. creating and using the delta during query processing should be efficient, and
3. we should be able to read and work with parts of the delta that are essential.

Since every record is identified by a primary key, we can meet all of the above criteria by capturing changes to individual record fields as outlined next.

Suppose R_1 and R_2 are two versions of a `datafile` R, with every record having the schema $\langle k, c1, c2, \ldots, ck\rangle$, where k is the primary key of R and $c1, c2, \ldots, ck$ are k columns. The delta (Δ) between them contains records of the form $\langle k, [c1_1, c1_2], [c2_1, c2_2], \ldots, [ck_1, ck_2]\rangle$, where k is the primary key of the record that is different in one or more columns in the two versions, and every ci_1 or ci_2 is the respective value in R_1 or R_2, in the column where the record differs, or the special value "$\perp$", if the column values are identical. We record the value of the column in both versions, and therefore, this is an undirected delta format.

Figure 6.1 gives an example of this format. R_1, R_2, R_3 are three versions of a `datafile` R, Δ_1 is the delta between R_1 and R_2, and Δ_2 is the delta between R_2 and R_3. Consider the record with primary key 1 in R_1 and R_2, i.e., $R_1 : \langle 1, 100, a\rangle$ and $R_2 : \langle 1, 100, b\rangle$. In Δ_1, the delta record for this primary key is therefore, $\Delta_1 : \langle 1, [\perp, \perp], [a, b]\rangle$. Similarly, we have delta records for the primary keys $2, 3, 5$, but not for 4 as the record is identical in the two versions. Note that for every record in the delta, there is at least one column where the values are not $[\perp, \perp]$.

Next, we discuss how we compute the deltas between two versions of the same `datafile`. The differencing procedure works by finding records in the source and target `datafiles` that are logical "pairs". The schema-defined primary key is used to pair up records in the source and target `datafiles`. For a record pair, any differences in the content is output as a new record in the delta having the schema described above. Such pairs are considered as *updates*. Records in the source `datafile` that could not be paired are considered as *deletes*, and we output one delta record for each such source record. Similarly, records in the target `datafile` that could not be paired are output as *inserts*.

6.2.3 Physical Representation

We now describe how the `datafiles` and deltas are persisted on disk. Once a `datafile` is committed in DEX, we parse it according to the schema file supplied and store it in the Apache Parquet format [110]. Apache Parquet is a state-of-the-art, open source columnar file format offering both high compression and high scan efficiency. Parquet is a PAX-like [111] format optimized for large data blocks and is vastly more

R_1

k	c1	c2
1	100	a
2	10	z
3	5	c
4	25	d
5	90	e

R_2

k	c1	c2
1	100	b
2	20	z
3	8	d
4	25	d
5	100	e

R_3

k	c1	c2
1	100	b
2	40	f
3	5	d
4	25	d
5	100	e

$\Delta_1 = \Delta(R_1, R_2)$

k	c1	c2
1	[⊥, ⊥]	[a,b]
2	[10,20]	[⊥, ⊥]
3	[5,8]	[c,d]
5	[90,100]	[⊥, ⊥]

$\Delta_2 = \Delta(R_2, R_3)$

k	c1	c2
2	[20,40]	[z,f]
3	[8,5]	[⊥, ⊥]

$\Delta_3 = \Delta(R_1, R_3)$

k	c1	c2
1	[⊥, ⊥]	[a,b]
2	[10,40]	[z,f]
3	[⊥, ⊥]	[c,d]
5	[90,100]	[⊥, ⊥]

Figure 6.1: R_1, R_2, R_3 are three versions of a `datafile` R. k is the primary key column. Three deltas are shown, Δ_1 is the delta between R_1 and R_2, Δ_2 is the delta between R_2 and R_3, and Δ_3 is the delta between R_1 and R_3.

efficient than text-based formats like CSV during storage and query processing [112]. This format is also suited to store the deltas, because it results in a more compact representation when only a small number of columns are modified across versions.

6.2.4 Discussion

Input data may come in the form of CSV/TSV/JSON formats which are text-based. Currently user has to ensure that the schema file is correct and there are no errors during the conversion process. Automatic schema inference may be added later. In addition, such conversion can also be done in a post-hoc manner. If it is expected that new `datafiles` may benefit from richer query processing, a separate process can be run to convert the existing binary data to Parquet format by specifying a schema file.

6.3 Query Execution

In this section, we describe the query processing ideas underlying our prototype implementation. We make heavy use of terminology and algorithms from Section 5.3 and Section 5.4, and extend the ideas behind delta contraction, and line and star reductions to be applicable in this setting. Thereafter, we outline the design of other physical operators to execute the rest of the query.

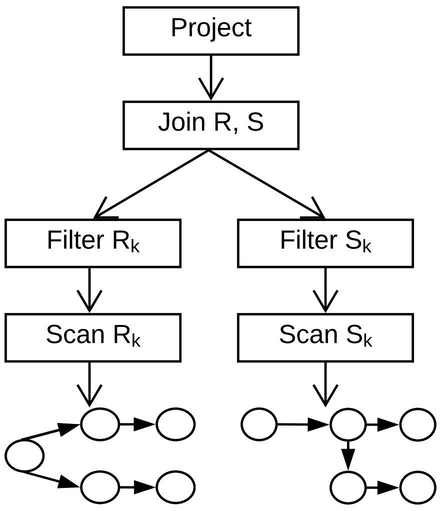
Query execution follows the same two-phase optimization approach that we described in Section 5.3. Suppose we denote the query as $Q(\mathcal{R}_k)$, where Q denotes the SPJ part of the query, and $\mathcal{R}_k = \{R_1, R_2, \dots, R_k\}$ is the set of `datafiles` required by the query. For simplicity of notation, we assume that all of $\mathcal{R}_k$ are versions of a same `datafile` R. However, in the the presence of a join, $\mathcal{R}_k$ will contain versions of multiple `datafiles`, and the SCAN step described in detail in Section 6.3.2, is modified to account for each `datafile` separately.

The *storage graph*, $\mathcal{G}$, described in detail in Section 5.2.3.1, indicates the delta decisions that have been made when storing all versions of a `datafile`. In the first phase of query execution, we identify all the relevant `datafiles` and deltas in $\mathcal{G}$ that are necessary to execute $Q(\mathcal{R}_k)$. This is the problem of finding an access tree of $Q(\mathcal{R}_k)$, and we have described this in detail in Section 5.3.2.

In the second phase, we map the logical SPJ query on to a directed acyclic graph (DAG) of the physical operators described in the next few sections. Since in this work, we address single-block queries, the mapping from the logical plan to the physical operators is straightforward, and an example is shown in Figure 6.2. The primary difference

```
on-versions(V1, V2, ..., Vk)
select R.a, S.z
from R join S on R.b = S.x
where R.c = "..." and S.y = "..."
```

(a)



(b)

Figure 6.2: (a) Example query on k versions, (b) physical plan to execute the query.

from the perspective of a classical query execution engine is that the physical operators described below are modified to create and process v-tuples instead of tuples. At the lowest step of the DAG, we have the Scan operator (Section 6.3.2) taking as input the respective access trees (for the two `datafiles` R and S in the query). The output of the Scan operator is a set of v-tuples for each `datafile`. Thereafter, the Filter operator applies all relevant predicates in the query. The Join operator (Section 6.3.3) reads the two sets of v-tuples and outputs a single set of v-tuples corresponding to the join result. The projection step removes any fields not needed by the query.

6.3.1 v-tuples

A v-tuple r, generated either as a result of the Scan step, or any subsequent steps in the query processing pipeline, contains data about the relevant columns required by the query, and a *version bitmap*, indicating the versions where the tuple is present. Figure 6.3

shows a set of v-tuples, which are the output of the SCAN step on three versions of the relation R, shown earlier in Figure 6.1. The tuple with the primary key 4 is unchanged in all three versions, and hence it has the bitmap [111]. On the other hand, tuples with primary keys 2 and 3 are different in all three versions, and hence we have three v-tuples corresponding to each.

In principle, v-tuples are similar to the concept of "tuple bundles" in the Monte Carlo Database System (MCDB) [113]. MCDB is a prototype relational database designed to allow an analyst to attach arbitrary stochastic models to a database, thereby specifying, in addition to the ordinary relations, "random" relations that contain uncertain data. A tuple bundle, like a v-tuple, encapsulates the instantiations of a tuple over a set of many Monte Carlo iterations, with the goal of operating in batch across all Monte Carlo iterations. However, due to the nature of the application, the specific structure of the data inside tuple bundles is different from v-tuples, and we require new physical operators to generate them efficiently.

6.3.2 SCAN Operator

The SCAN operation is the workhorse of the query processing phase. The input to a SCAN operation is the access tree of the `datafile` and a list of versions in the query. The output is a set of v-tuples across all the versions requested by the query.

Suppose $\mathcal{R}_k = \{R_1, R_2, \ldots, R_k\}$ are versions of a `datafile` R. A naive implementation of the SCAN operator would be to first perform a mutiple `datafile` checkout (Section 5.4.1.2) of all of $\mathcal{R}_k$, followed by a grouping step that brings common records to-

k	c1	c2	[v_1 v_2 v_3]
1	100	a	[1 0 0]
1	100	b	[0 1 1]
2	10	z	[1 0 0]
2	20	z	[0 1 0]
2	40	f	[0 0 1]
3	5	c	[1 0 0]
3	8	d	[0 1 0]
3	5	d	[0 0 1]
4	25	d	[1 1 1]
5	90	e	[1 0 0]
5	100	e	[0 1 1]

Figure 6.3: v-tuples for R_1, R_2, R_3

gether, to create a set of v-tuples. However, this scheme is likely to be inefficient, as it involves materializing close to k copies of most of the tuples, only to merge all tuples together (if most of the `datafile` does not change). When k is large, about 50–100 in our experiments, this scheme becomes a major performance bottleneck.

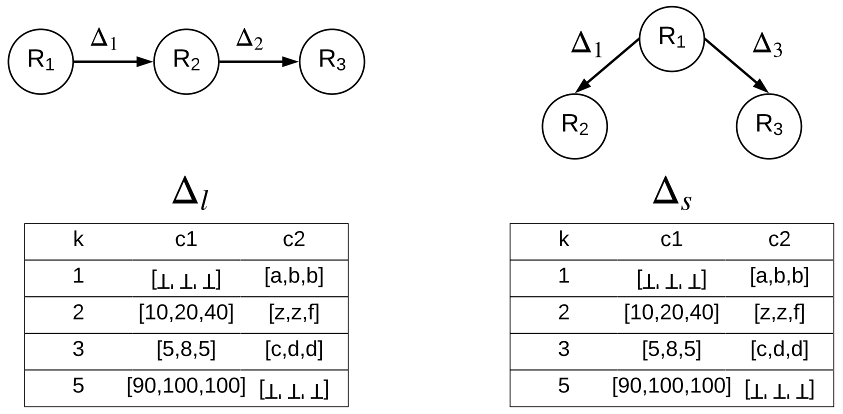
We therefore use a different strategy, one that works with the deltas as much as possible. We use the **Contract and Reduce (`C&R`)** algorithm, described in detail in Section 5.4.2 and in Algorithm 6 to generate v-tuples efficiently. Recall that the `C&R` algorithm takes as input an access tree and iteratively applies a set of transformations to generate the query result efficiently. In order for the algorithm to be applicable in this setting, we need to address three important issues, that are the focus of the rest of the section.

First, we need to define the semantics of delta contraction for the delta format that we use. The delta contraction step takes two deltas, say, Δ_1 between R_1 and R_2, and Δ_2 between R_2 and R_3, and combines them to create a single delta, say, Δ_3 between R_1 and R_3. This operation is useful when R_2 is not required in the query. Second, we need to define reduction rules for *line* and *star* structures, as defined in Section 5.4.2, for our delta format. Due to the additional structure in the deltas, we extend the delta format to incorporate the result of reducing multiple deltas arranged in line/star configurations. This delta format is similar to the one described in Section 6.2.2 earlier, but instead of keeping two values per column in a reduced delta Δ, we keep $2 \leq p(\Delta) \leq k$ values, where $p(\Delta)$ indicates the number of versions in $\mathcal{R}_k$ that are reduced by this delta. Third, we describe how to to apply, or patch, the "last" delta to the materialized `datafile` in order to generate all the v-tuples.

6.3.2.1 Delta Contraction

Suppose we want to compute $\Delta = \Delta_1 \oplus \Delta_2$, where Δ_1 and Δ_2 are defined as above (see Figure 6.1 for an example). If a primary key k' occurs in only one of Δ_1 or Δ_2, it is included as is in Δ. This indicates that the record was modified in only one of the versions. If a primary key k is present in both Δ_1 and Δ_2, it indicates that the record was modified in both versions, and we resolve it as follows. For each column ci, of primary key k, appearing in the deltas, we have three possible scenarios:

- $ci = [\bot, \bot]$ in both Δ_1 and Δ_2,
- $ci = [\bot, \bot]$ in Δ_1 and $ci = [v, w]$ in Δ_2, or vice versa, or
- $ci = [x, y]$ in Δ_1 and $ci = [y, z]$ in Δ_2,



Δ_l

k	c1	c2
1	[⊥, ⊥, ⊥]	[a,b,b]
2	[10,20,40]	[z,z,f]
3	[5,8,5]	[c,d,d]
5	[90,100,100]	[⊥, ⊥, ⊥]

(a) Line

Δ_s

k	c1	c2
1	[⊥, ⊥, ⊥]	[a,b,b]
2	[10,20,40]	[z,z,f]
3	[5,8,5]	[c,d,d]
5	[90,100,100]	[⊥, ⊥, ⊥]

(b) Star

Figure 6.4: Line and star transformations for the SCAN operation

where v, w, x, y, z are values from the domain of ci. In the first case, we can infer that the value in column ci has not changed between R_1 and R_3. Hence we set $ci = [\perp, \perp]$ in Δ. In the second case, we can infer that column value changed in one of Δ_1 or Δ_2, and we set $ci = [v, w]$ in Δ. In the third case, if $x \neq z$, we set $ci = [x, z]$ in Δ, otherwise we set $ci = [\perp, \perp]$. After all columns of a key k have been processed and set appropriately in Δ, we check if there is at least one column where the value is not $[\perp, \perp]$. If no such column exists, k can be removed from Δ.

6.3.2.2 Line/Star Structures

We first describe the transformation rule in the case of a line. Figure 6.4(a) shows an example where three `datafiles`, R_1, R_2, R_3, are arranged in a line, and our goal is to generate v-tuples for all three. Similar to Section 5.4.4, we keep additional information with each reduced delta. We first set $p(\Delta_l) = 3$, where Δ_l is the reduced delta. In general $p(\Delta_l) = p(\Delta_1) + p(\Delta_2) - 1$, where $p(\Delta) = 2$ for a delta between two `datafiles`. This value keeps track of the number of `datafiles` reduced by the delta. Every column ci in Δ_l is an array of $p(\Delta_l)$ values. The main idea here is to preserve the values (in the array) of the modified columns in every version, and infer values, whenever possible, to columns that are $\perp$ in the Δ. There are two cases to consider when computing these values for a primary key k.

Case 1: k is present in only one of Δ_1 or Δ_2. This is the case with primary keys $k = 1, 5$ in our example. Next, there are two scenarios. First, when a column ci contains the special value $\perp$, ci in Δ_l can simply be set to an array of all $\perp$. For instance, $c1$ for $k = 1$ and $c2$ for $k = 5$. Second, when the column contains values from the domain of ci. We discuss the case when k is present in Δ_1. For instance, in Δ_1 column $c2$ for $k = 1$, contains the values $[a, b]$. $c2$ in Δ_l is then set as follows: the first $p(\Delta_1) = 2$ values are taken from Δ_1 and copied to ci in Δ_l, i.e., $[a, b, \perp]$, and the remaining $p(\Delta_2) - 1$ values are set to the last value in Δ_1, i.e., $[a, b, b]$. We can complement the above rule to account for the case when k is present in Δ_2 instead.

Case 2: k is present in both Δ_1 or Δ_2. This is the case with primary keys $k = 2, 3$ in our example. Next, there are three scenarios similar to the ones we discussed for delta contraction above.

1. ci is all $\perp$ in both Δ_1 and Δ_2, then ci can be set to all $\perp$ in Δ_l

2. ci is blank in one of Δ_1 or Δ_2, then the $\perp$ values can be set to the appropriate value as in **Case 1** above, depending on whether the $\perp$ value is in Δ_1 or Δ_2 (e.g., $c2$ for $k = 2, 3$)

3. ci has values in both Δ_1 and Δ_2, then ci in Δ_l is a simply a concatenation of the

values in Δ_1 and Δ_2 (accounting the last value in Δ_1 and first value in Δ_2 only once, since they are guaranteed to be the same).

The transformation rule in the case of a star is similar to the rules for line above, with the only difference being the value that is copied to ci in Δ_s, when either one of both of Δ_1 and Δ_2 contain values (not $\perp$) in ci. In the case of a star, it is the first value (this value will be the same in both Δ_1 and Δ_2) in the column ci. Figure 6.4(b) gives an example when R_1, R_2, R_3 are arranged in a star, and Δ_s is the reduced delta of Δ_1 and Δ_3.

6.3.2.3 Applying Delta to a Materialized `datafile`

The final step in `C&R` algorithm is to apply the last delta to the materialized file in order to generate all the v-tuples. Continuing our example from before, we want to obtain the v-tuples in Figure 6.3 as a result of the operation $R_1 \oplus \Delta_l$ in the example of Figure 6.4(a).

We perform a linear scan of the materialized file, R_1, and the delta, Δ_l, resolving records based on their primary key. For every primary key k in R_1, we output between 1 and $p(\Delta_l) = 3$ v-tuples. If a primary key k does not appear in the delta, it indicates that record k has not been modified, and it is output as a single v-tuple with the bitmap of $p(\Delta_l)$ 1s. For example, the record with primary key $k = 4$ does not appear in Δ_l, and hence is output as $(4, 25, d, [111])$. On the other hand, if a primary key k appears in the delta, we consider every column ci where the values are not $\perp$ and generate the v-tuples by resolving one column at a time. Specifically, consider the record with primary key $k = 3$ in Δ_l, i.e, $(3, [5, 8, 5], [c, d, d])$. We first resolve the column $c1$. The goal here is to identify all repeated values in $c1$ and output intermediate tuples, such that there is one tuple, per repeated value in $c1$. For instance, the value 5 is repeated in versions 1 and 3, after resolving, the partial result is $\{(3, 5, [c, \perp, d]), (3, 8, [\perp, d, \perp])\}$. Next, we resolve $c2$. For the first tuple in the intermediate result, $(3, 5, [c, \perp, d])$, there are no duplicate values in $c2$, and since this is the last column to be resolved, we can output two v-tuples, $\{(3, 5, c, [100]), (3, 5, d, [001])\}$. Similarly, for the second tuple $(3, 8, [\perp, d, \perp])$, there are no duplicate values in $c2$, and we can output $\{(3, 8, d, [010])\}$.

6.3.3 JOIN Operator

In this section, we describe hash join-based algorithms to perform the equi-join operation. Our choice to study hash-based methods, as opposed to sort-based methods, was influenced by the observation that in general, they produce results faster than sort-based methods and have a smaller memory footprint [114]. A detailed study of the criteria when sort-based algorithms become competitive in the context of DEX remains an area for future work.

6.3.4 Simple Hash Join

We first adapt the canonical hash join algorithm to our setting. Suppose the two inputs to be joined are R and S, such that $|R| < |S|$, and both R and S are sets of v-tuples. The algorithm has a build phase and a probe phase. At the start of the build phase, it allocates memory for the hash table. It then reads a v-tuple $r \in R$, hashes on the join key of r using a pre-defined hash function $h(\cdot)$, and writes the v-tuple r into the

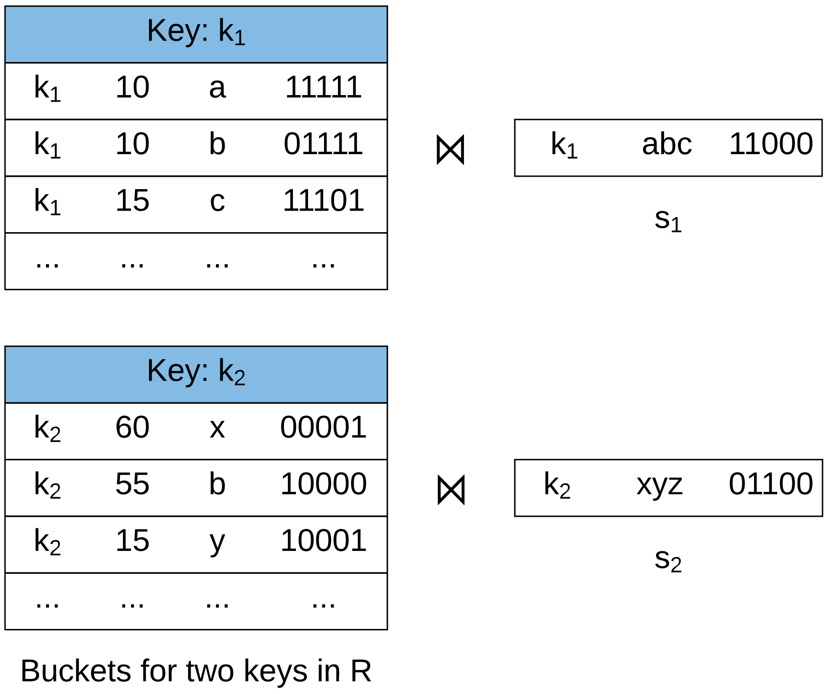


Figure 6.5: Buckets in simple hash join

corresponding bucket. The build phase is completed when all the R v-tuples have been stored in the hash table.

During the probe phase, each v-tuple $s \in S$ is hashed on the join key using the same hash function $h(\cdot)$, and the correct hash bucket for the join key is identified (after accounting for hash collisions). For each v-tuple $r_h(s)$ in the bucket, we perform a logical AND operation on the the bitmap of $r_h(s)$ and s, and if non-zero, the concatenated v-tuple $r_h(s) \cdot s$ is output, with the new bitmap.

Figure 6.5 shows an example with two hash buckets of R, for keys k_1 and k_2. During probe phase, the bitmap in s_1 is ANDed with every tuple in bucket for key k_1 to find join matches. This example also illustrates a source of inefficiency in our simple adaptation. Note that the v-tuples in the bucket for key k_2 have mostly 0s in the bitmap. During the probe phase, when a tuple s_2 matches on the join key, most of the bitmap AND operations result in bitmaps with all 0s. We address this limitation next.

6.3.5 Version-aware Hash Join

The main bottleneck in the simple hash join method described above is the case when a bucket contains a large number of v-tuples having bitmaps of mostly 0s. When probing for matches of a tuple s, the algorithm ends up scanning the entire bucket, only to find few matches. We propose a different bucket design to improve probe efficiency by exploiting the sparsity in such buckets. Figure 6.6 shows the important elements of the new bucket design.

The bucket consists of two pieces. Suppose the bitmap in the v-tuples is of size k.

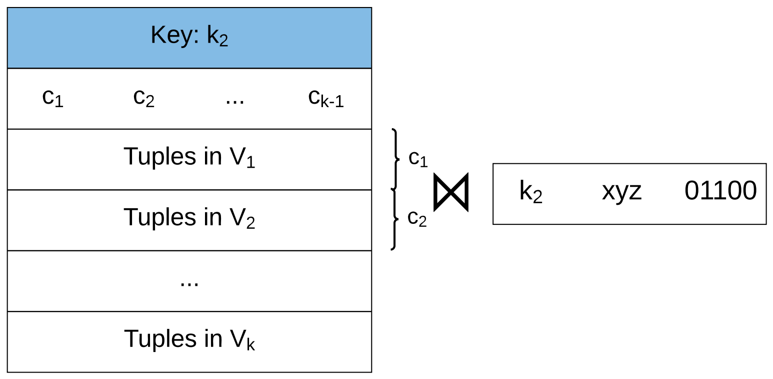


Figure 6.6: Improved bucket structure for sparse keys

The first piece is an array of $k - 1$ integer values, that serve as a pointer to the tuples for k versions into the bucket. The second piece is the tuples in the input R, partitioned by version. That is all tuples in V_1 are stored first, followed by all tuples in V_2, and so on. During the build phase, the bitmap of every v-tuple $r \in R$ is inspected to check the versions r that appears in. If r appears in $j \leq k$ versions, we create j copies of r, sans the bitmap, and store them at the correct partition in the bucket. Once the build phase is completed, the bucket is scanned once to compute the counts/pointers in the first piece. During the probe phase, the bitmap of tuple s is inspected to identify the versions that s appears in, and the array of pointers in the bucket is used to lookup the relevant tuples in the bucket. The primary benefit of this approach is when for a large number of r, $j << k$. Although in the build phase, we use more memory for such buckets, the probe phase can be done efficiently by looking up only the required versions, instead of a linear scan of all entries in the bucket.

6.3.6 Other Operators

6.3.6.1 Filter Operator

The Filter operator is identical to selection in a classical database system. The filter predicate is applied to every v-tuple and if it evaluates to true, then the tuple is output as is to the next step.

6.3.6.2 Project Operator

Similar to the selection operation, the project operation works with a tuple bundle at a time, only keeping the respective columns and the version list for each tuple bundle intact. Note that this also preserves multiset semantics of the projection operation.

6.4 Evaluation

We now evaluate the benefits of using the `C&R` algorithm for the Scan operator and the new bucket design for the hash join operation for the Join operator. Our goal is to quantify the performance of both techniques compared to the respective baseline approaches. For the Scan operation, where the input is k `datafiles`, the baseline approach is the multiple `datafile` checkout operation described in Section 5.4.1.2, followed by a k-way merge to construct the v-tuples. For the Join operation, the baseline approach is the simple hash join method described in Section 6.3.4 above.

All experiments were conducted on a single machine with Intel Core i7-4790 CPU (3.60 GHz, 8MB L3 cache), 32GB of memory, running Ubuntu 16.04 and OpenJDK 64-

bit server JVM (ver. 1.8.0_111). All time measurements are recorded as wall-clock time. Unless otherwise stated, to measure response time, we run each query 10 times and consider the median.

6.4.1 Datasets

Lacking access to real-world versioned datasets with sufficient and varied structure, we instead developed a synthetic data generator, described in detail in Section 5.5, to generate datasets with different characteristics for a wide variety of parameter values. This enables us to carefully study the performance of our techniques in various settings. We describe the modifications that we made to the dataset generation process next.

Every dataset is characterized by a 3-tuple, $\langle |A|, |\Delta|, \#\Delta \rangle$, where $|A|$ and $|\Delta|$ refer to the average number records in a `datafile` and average number of records in the deltas (as a percentage of $|A|$), respectively. $\#\Delta$ refers to the number of deltas in the access tree. A record has four columns – a primary key column of 64-bit `integer` type, and three 64-byte `string` columns. When creating a new version of a `datafile` from an existing version, the dataset generator assigns probabilities, according to the Zipfian distribution (with exponent 2), to every record. This assignment indicates how likely a record is to be modified when creating the new version. In a record, the column to be modified is chosen at random from the three `string` columns.

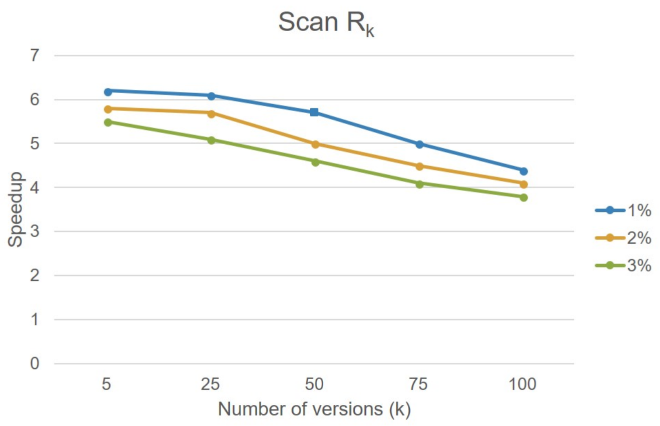


Figure 6.7: SCAN performance over varying delta sizes; dataset parameters: $|R| = 3M, \#\Delta = 50$.

6.4.2 Results

We first compare the running time of SCAN using the two methods. The input to the SCAN operator is an access tree over k versions, say of a `datafile` R, and the output is a set of v-tuples. The first method simply performs a multiple `datafile` checkout of k versions, followed by a k-way merge. The second method uses the `C&R` algorithm, by applying the transformation and reduction rules described in Section 6.3.2. Figure 6.7 shows the speedup obtained by using `C&R` when the average delta size is varied between $1\%, 2\%, 3\%$, with other dataset parameters set to $|R| = 3M, \#\Delta = 50$. As we can see, `C&R` is between **4X–6X** effective when the delta size is small, i.e., 1%. The speedup decreases as the size of the deltas increases primarily due to larger intermediate delta sizes.

Next, Table 6.1 reports the running time of the two hash join methods when performing an equi-join operation. The first method, SHJ, denotes the simple hash join approach described in Section 6.3.4, and the second method, VHJ, denotes the version-aware hash join approach described in Section 6.3.5. We perform an equijoin on v-tuples

k	SHJ (sec)	VHJ (sec)	Speedup
5	3.3	3.3	0%
25	19.4	11.6	61%
50	49.8	29.6	68%
75	103.2	59.7	73%
100	146.6	81.9	79%

Table 6.1: Join performance of two hash join methods over varying number of input versions R, S; dataset parameters: $|R| = 100K, |S| = 1M, |\Delta| = 1\%, \#\Delta = 50$

representing k versions of two `datafiles`, R and S, where $|R| = 100K, |S| = 1M$. During the build phase, VHJ determines when to use the new bucket design, depending on the sparsity of the bucket for a primary key. Specifically, there are two parameters to compute when deciding whether a bucket is sparse or not. The first parameter, b, is the sparsity of the bitmap in a v-tuple. This is measured relative to the size of the bitmap, i.e., the number of versions k, and a v-tuple is said to be sparse when its b value is less than a specified threshold, $\bar{b}$. The second parameter, n, is the relative number of sparse v-tuples in the bucket, and the bucket is said to be sparse, when its n value is greater than a specified threshold, $\bar{n}$. Table 6.1 reports the runtime (in seconds) and the speedup obtained when we set $\bar{b} = 5\%$ and $\bar{n} = 90\%$. When there are only 5 versions across which to perform a join, no buckets satisfy the sparsity criteria, and hence the run times are identical. We note speedup of 61% – 79%, increasing as the number of versions in the join increases, when using a different bucket design for sparse buckets.

Our initial set of experiments thus shows significant benefits of the v-tuples approach in evaluating richer declarative queries on multiple versions. They indicate that we can both create v-tuples efficiently, by taking advantage of the delta representation when reading the input, and that we can perform richer operations like join at acceptable overheads. As the next step, we plan to add additional operations, such as aggregation and user defined functions, to further extend the class of queries that DEX can support.

Chapter 7: Conclusions

In this dissertation, we introduced a framework for building a dataset version control system. We presented a theoretical model for capturing and reasoning about the tradeoffs that arise in the management of thousands of dataset versions. We demonstrated that if the versions are largely overlapping in their contents, by using delta-encoding, it is possible to both store and retrieve them effectively. Instead of applying delta-encoding in an ad hoc manner, we studied different algorithms that helped us navigate the storage-recreation tradeoff in a principled fashion. We also showed that it is possible to execute a variety of queries over multiple versions of past dataset versions, without having to first retrieve all in their entirety.

The technical contributions of this dissertation are (i) a formal study of the dataset versioning problem that considers the trade off between storage cost and recreation cost in different manners, and provides a collection of polynomial time algorithms for finding good solutions for large problem sizes, (ii) a cost based optimization framework along with a set of transformation rules that, based on the algebraic properties of the deltas, finds efficient methods to evaluate checkout (retrieval), intersection, union, and t-threshold queries over multiple versions, and (iii) a new approach to efficiently execute single block select-project-join queries over multiple data versions. We also introduced the DEX system, that demonstrates how the techniques in this dissertation can lead to significantly improved performance compared to ad hoc techniques and version management systems prevalent today.

Bibliography

[1] James Cheney, Stephen Chong, Nate Foster, Margo Seltzer, and Stijn Vansummeren. Provenance: A future history. In *OOPSLA*, pages 957–964. ACM, 2009.

[2] Daniel J Weitzner, Harold Abelson, Tim Berners-Lee, Joan Feigenbaum, James Hendler, and Gerald Jay Sussman. Information accountability. *Communications of the ACM*, 51(6):82–87, 2008.

[3] R. K. L. Ko, P. Jagadpramana, M. Mowbray, S. Pearson, M. Kirchberg, Q. Liang, and B. S. Lee. Trustcloud: A framework for accountability and trust in cloud computing. In *IEEE World Congress on Services*, July 2011.

[4] Pat Helland. Immutability changes everything. In *CIDR*, 2015.

[5] João Paulo and José Pereira. A survey and classification of storage deduplication systems. *ACM Computing Surveys*, 2014.

[6] Monya Baker. De novo genome assembly: what every biologist should know. *Nature methods*, 9(4):333–337, 2012.

[7] `http://git.kernel.org/cgit/git/git.git/tree/Documentation/technical/pack-heuristics.txt`, .

[8] `http://comments.gmane.org/gmane.comp.version-control.git/189776`, .

[9] Sebastian Rönnau, Jan Scheffczyk, and Uwe M. Borghoff. Towards XML Version Control of Office Documents. In *Proc. of the ACM Symposium on Document Engineering*, pages 10–19, 2005.

[10] Michael Maddox, David Goehring, Aaron J. Elmore, Samuel Madden, Aditya G. Parameswaran, and Amol Deshpande. Decibel: The relational dataset branching system. *PVLDB*, 9(9):624–635, 2016.

[11] James Clifford, Curtis Dyreson, Tomás Isakowitz, Christian S Jensen, and Richard Thomas Snodgrass. On the semantics of "now" in databases. *ACM Transactions on Database Systems (TODS)*, 22(2):171–214, 1997.

[12] Gultekin Özsoyoğlu and Richard T Snodgrass. Temporal and real-time databases: A survey. *IEEE Trans. on Knowl. and Data Eng.*, pages 513–532, 1995.

[13] Abdullah Uz Tansel, James Clifford, Shashi Gadia, Sushil Jajodia, Arie Segev, and Richard Snodgrass. *Temporal databases: theory, design, and implementation.* Benjamin-Cummings Publishing Co., Inc., 1993.

[14] David Lomet, Roger Barga, Mohamed F Mokbel, German Shegalov, Rui Wang, and Yunyue Zhu. Immortal DB: Transaction time support for SQL server. In *SIGMOD*, pages 939–941, 2005.

[15] David Lomet, Mingsheng Hong, Rimma Nehme, and Rui Zhang. Transaction time indexing with version compression. In *PVLDB*, pages 870–881, 2008.

[16] David B. Lomet, Alan Fekete, Rui Wang, and Peter Ward. Multi-version concurrency via timestamp range conflict management. In *IEEE 28th International Conference on Data Engineering (ICDE 2012), Washington, DC, USA (Arlington, Virginia), 1-5 April, 2012*, pages 714–725, 2012. doi: 10.1109/ICDE.2012.10. URL `http://dx.doi.org/10.1109/ICDE.2012.10`.

[17] Yu Wu, Sushil Jajodia, and X Sean Wang. Temporal database bibliography update. In *Temporal Databases: research and practice*, pages 338–366. Springer, 1998.

[18] Richard Snodgrass. The temporal query language tquel. *ACM Trans. Database Syst.*, 12(2):247–298, June 1987. ISSN 0362-5915. doi: 10.1145/22952.22956. URL `http://doi.acm.org/10.1145/22952.22956`.

[19] Richard T Snodgrass, Santiago Gomez, and L Edwin McKenzie Jr. Aggregates in the temporal query language tquel. *Knowledge and Data Engineering, IEEE Transactions on*, 5(5):826–842, 1993.

[20] Michael Stonebraker, Lawrence A Rowe, and Michael Hirohama. The implementation of postgres. *IEEE Transactions on Knowledge & Data Engineering*, (1):125–142, 1990.

[21] Antonin Guttman. R-trees: A dynamic index structure for spatial searching. In *SIGMOD*, pages 47–57, 1984.

[22] Using Oracle Flashback Technology. `https://docs.oracle.com/cd/B28359_01/appdev.111/b28424/adfns_flashback.htm`, . Accessed: May 04, 2016.

[23] Oracle Total Recall with Oracle Database 11g Release 2. `http://www.oracle.com/technetwork/database/focus-areas/storage/total-recall-whitepaper-171749.pdf`, . Accessed: November 11, 2016.

[24] Peter Buneman, Sanjeev Khanna, Keishi Tajima, and Wang Chiew Tan. Archiving scientific data. *ACM Trans. Database Syst.*, 29:2–42, 2004. doi: 10.1145/974750.974752. URL `http://doi.acm.org/10.1145/974750.974752`.

[25] Torben Bach Pedersen and Christian S Jensen. Multidimensional database technology. *Computer*, 34(12):40–46, 2001.

[26] Heum-Geun Kang and Chin-Wan Chung. Exploiting versions for on-line data warehouse maintenance in molap servers. In *Proceedings of the 28th International Conference on Very Large Data Bases*, VLDB '02, pages 742–753. VLDB Endowment, 2002. URL `http://dl.acm.org/citation.cfm?id=1287369.1287433`.

[27] P. Baumann, A. Dehmel, P. Furtado, R. Ritsch, and N. Widmann. The multidimensional database system rasdaman. In *Proceedings of the 1998 ACM SIGMOD International Conference on Management of Data*, SIGMOD '98, pages 575–577, New York, NY, USA, 1998. ACM. ISBN 0-89791-995-5. doi: 10.1145/276304.276386. URL `http://doi.acm.org/10.1145/276304.276386`.

[28] Paul G. Brown. Overview of scidb: Large scale array storage, processing and analysis. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of Data*, SIGMOD '10, pages 963–968, New York, NY, USA, 2010. ACM. ISBN 978-1-4503-0032-2. doi: 10.1145/1807167.1807271. URL `http://doi.acm.org/10.1145/1807167.1807271`.

[29] Michael Stonebraker, Jacek Becla, David J. DeWitt, Kian-Tat Lim, David Maier, Oliver Ratzesberger, and Stanley B. Zdonik. Requirements for science data bases and scidb. In *CIDR 2009, Fourth Biennial Conference on Innovative Data Systems Research, Asilomar, CA, USA, January 4-7, 2009, Online Proceedings*, 2009. URL `http://www-db.cs.wisc.edu/cidr/cidr2009/Paper_26.pdf`.

[30] Adam Seering, Philippe Cudré-Mauroux, Samuel Madden, and Michael Stonebraker. Efficient versioning for scientific array databases. In *ICDE*, pages 1013–1024, 2012.

[31] Shu-Yao Chien, Vassilis J Tsotras, Carlo Zaniolo, and Donghui Zhang. Efficient complex query support for multiversion xml documents. In *Advances in Database Technology—EDBT 2002*, pages 161–178. Springer, 2002.

[32] Anders Björnerstedt and Christer Hultén. Object-oriented concepts, databases, and applications. chapter Version Control in an Object-oriented Architecture, pages 451–485. ACM, New York, NY, USA, 1989. ISBN 0-201-14410-7. doi: 10.1145/63320.66513. URL `http://doi.acm.org/10.1145/63320.66513`.

[33] Marios Hadjieleftheriou, George Kollios, Vassilis J Tsotras, and Dimitrios Gunopulos. Efficient indexing of spatiotemporal objects. In *Advances in Database Technology—EDBT 2002*, pages 251–268. Springer, 2002.

[34] U. Khurana and A. Deshpande. Efficient snapshot retrieval over historical graph data. In *Data Engineering (ICDE), 2013 IEEE 29th International Conference on*, pages 997–1008, April 2013. doi: 10.1109/ICDE.2013.6544892.

[35] Sean Quinlan and Sean Dorward. Venti: A new approach to archival storage. In *FAST*, volume 2, pages 89–101, 2002.

[36] Benjamin Zhu, Kai Li, and R Hugo Patterson. Avoiding the disk bottleneck in the data domain deduplication file system. In *Fast*, volume 8, pages 1–14, 2008.

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

[38] Zan Ouyang, Nasir Memon, Torsten Suel, and Dimitre Trendafilov. Cluster-based delta compression of a collection of files. In *WISE*, 2002.

[39] Mun Choon Chan and Thomas YC Woo. Cache-based compaction: A new technique for optimizing web transfer. In *INFOCOM*, 1999.

[40] Randal C Burns and Darrell DE Long. In-place reconstruction of delta compressed files. In *Proceedings of the seventeenth annual ACM symposium on Principles of Distributed Computing*, pages 267–275. ACM, 1998.

[41] Purushottam Kulkarni, Fred Douglis, Jason D. LaVoie, and John M. Tracey. Redundancy elimination within large collections of files. In *USENIX ATC*, 2004.

[42] Dan RK Ports and Kevin Grittner. Serializable snapshot isolation in postgresql. *Proceedings of the VLDB Endowment*, 5(12):1850–1861, 2012.

[43] Scott Chacon and Ben Straub. Pro Git Book. `https://git-scm.com/book/en/v2`. Accessed: May 04, 2016.

[44] Git-Annex. `https://git-annex.branchable.com/`, . Accessed: May 08, 2016.

[45] Git Large File Storage. `https://git-lfs.github.com/`, . Accessed: May 08, 2016.

[46] Talel Abdessalem and Geneviève Jomier. Vql: A query language for multiversion databases. In *Database Programming Languages*, pages 160–179. Springer, 1997.

[47] Temporal Tables. `https://msdn.microsoft.com/en-us/library/dn935015.aspx`. Accessed: May 04, 2016.

[48] Jennifer Widom. Trio: A system for integrated management of data, accuracy, and lineage. *Technical Report*, 2004.

[49] Anderson Marinho, Leonardo Murta, Cláudia Werner, Vanessa Braganholo, Sérgio Manuel Serra da Cruz, Eduardo Ogasawara, and Marta Mattoso. Provmanager: a provenance management system for scientific workflows. *Concurrency and Computation: Practice and Experience*, 24(13):1513–1530, 2012.

[50] Leonardo Murta, Vanessa Braganholo, Fernando Chirigati, David Koop, and Juliana Freire. noworkflow: Capturing and analyzing provenance of scripts. In *Provenance and Annotation of Data and Processes*, pages 71–83. Springer, 2014.

[51] Jihie Kim, Ewa Deelman, Yolanda Gil, Gaurang Mehta, and Varun Ratnakar. Provenance trails in the Wings/Pegasus system. *Concurrency and Computation: Practice and Experience*, 20(5):587–597, 2008.

[52] Marcin Wylot, Philippe Cudre-Mauroux, and Paul Groth. Executing provenance-enabled queries over web data. In *Proceedings of the 24th International Conference on World Wide Web*, pages 1275–1285. International World Wide Web Conferences Steering Committee, 2015.

[53] Manish Kumar Anand, Shawn Bowers, Timothy Mcphillips, and Bertram Ludäscher. Exploring scientific workflow provenance using hybrid queries over nested data and lineage graphs. In *Scientific and Statistical Database Management*, pages 237–254. Springer, 2009.

[54] Manish Kumar Anand, Shawn Bowers, and Bertram Ludäscher. Techniques for efficiently querying scientific workflow provenance graphs. In *EDBT*, volume 10, pages 287–298, 2010.

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

[56] Grigoris Karvounarakis, Zachary G Ives, and Val Tannen. Querying data provenance. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of data*, pages 951–962. ACM, 2010.

[57] Shawn Bowers. Scientific workflow, provenance, and data modeling challenges and approaches. *Journal on Data Semantics*, 1(1):19–30, 2012.

[58] Susan B Davidson and Juliana Freire. Provenance and scientific workflows: challenges and opportunities. In *Proceedings of the 2008 ACM SIGMOD international conference on Management of data*, pages 1345–1350. ACM, 2008.

[59] Michael Stonebraker, Gerald Held, Eugene Wong, and Peter Kreps. The design and implementation of INGRES. *ACM Transactions on Database Systems (TODS)*, 1(3):189–222, 1976.

[60] Carlo Zaniolo. The database language GEM. In *ACM Sigmod Record*, volume 13(4), pages 207–218. ACM, 1983.

[61] Bruno Becker, Stephan Gschwind, Thomas Ohler, Bernhard Seeger, and Peter Widmayer. An asymptotically optimal multiversion b-tree. *The VLDB Journal—The International Journal on Very Large Data Bases*, 5(4):264–275, 1996.

[62] Christian Plattner, Andreas Wapf, and Gustavo Alonso. Searching in time. In *Proceedings of the 2006 ACM SIGMOD international conference on Management of data*, pages 754–756. ACM, 2006.

[63] Betty Salzberg and Vassilis J. Tsotras. Comparison of access methods for time-evolving data. *ACM Comput. Surv.*, 31(2):158–221, June 1999. ISSN 0360-0300. doi: 10.1145/319806.319816. URL `http://doi.acm.org/10.1145/319806.319816`.

[64] Khaled Jouini and Geneviève Jomier. Indexing multiversion databases. In *Proceedings of the Sixteenth ACM Conference on Conference on Information and Knowledge Management*, CIKM '07, pages 915–918, New York, NY, USA, 2007. ACM. ISBN 978-1-59593-803-9. doi: 10.1145/1321440.1321574. URL `http://doi.acm.org/10.1145/1321440.1321574`.

[65] Emad Soroush and Magdalena Balazinska. Time travel in a scientific array database. In *ICDE*, pages 98–109, 2013.

[66] Jeffrey C. Mogul, Fred Douglis, Anja Feldmann, and Balachander Krishnamurthy. Potential benefits of delta encoding and data compression for http. In *SIGCOMM*, pages 181–194, 1997.

[67] Josh MacDonald. File system support for delta compression. 2000.

[68] Anant P. Bhardwaj, Souvik Bhattacherjee, Amit Chavan, Amol Deshpande, Aaron Elmore, Samuel Madden, and Aditya Parameswaran. DataHub: Collaborative Data Science & Dataset Version Management at Scale. In *CIDR*, 2015.

[69] Shahram Ghandeharizadeh, Richard Hull, and Dean Jacobs. Heraclitus: Elevating deltas to be first-class citizens in a database programming language. *ACM Trans. Database Syst.*, 21(3):370–426, 1996. ISSN 0362-5915. doi: 10.1145/232753.232801. URL `http://doi.acm.org/10.1145/232753.232801`.

[70] Timothy Griffin and Richard Hull. A framework for implementing hypothetical queries. In *SIGMOD*, pages 231–242, 1997.

[71] Nicholas E. Taylor and Zachary G. Ives. Reconciling while tolerating disagreement in collaborative data sharing. In *SIGMOD*, pages 13–24, 2006. ISBN 1-59593-434-0. doi: 10.1145/1142473.1142476. URL `http://doi.acm.org/10.1145/1142473.1142476`.

[72] Todd J. Green, Grigoris Karvounarakis, Zachary G. Ives, and Val Tannen. Update exchange with mappings and provenance. In *PVLDB*, pages 675–686, 2007. ISBN 978-1-59593-649-3. URL `http://dl.acm.org/citation.cfm?id=1325851.1325929`.

[73] Sanjay Agrawal, Surajit Chaudhuri, and Vivek R. Narasayya. Automated Selection of Materialized Views and Indexes in SQL Databases. In *PVLDB*, pages 496–505, 2000. URL `http://dl.acm.org/citation.cfm?id=645926.671701`.

[74] Zohreh Asgharzadeh Talebi, Rada Chirkova, Yahya Fathi, and Matthias Stallmann. Exact and inexact methods for selecting views and indexes for OLAP performance improvement. In *EDBT*, pages 311–322, 2008. doi: 10.1145/1353343.1353383. URL `http://doi.acm.org/10.1145/1353343.1353383`.

[75] Alon Y. Halevy. Answering queries using views: A survey. *The VLDB Journal*, 10(4):270–294, 2001. ISSN 0949-877X. doi: 10.1007/s007780100054. URL `http://dx.doi.org/10.1007/s007780100054`.

[76] Jonathan Goldstein and Per-Åke Larson. Optimizing queries using materialized views: A practical, scalable solution. In *SIGMOD*, pages 331–342, 2001. ISBN 1-58113-332-4. doi: 10.1145/375663.375706. URL `http://doi.acm.org/10.1145/375663.375706`.

[77] Zvi Galil and Giuseppe F. Italiano. Data structures and algorithms for disjoint set union problems. *ACM Comput. Surv.*, pages 319–344, 1991. ISSN 0360-0300. doi: 10.1145/116873.116878. URL `http://doi.acm.org/10.1145/116873.116878`.

[78] Frank K. Hwang and Shen Lin. A simple algorithm for merging two disjoint linearly ordered sets. *SIAM Journal on Computing*, 1(1):31–39, 1972.

[79] Erik Demaine, Alejandro López-Ortiz, and J. Munro. Adaptive Set Intersections, Unions, and Differences. In *SODA*, pages 743–752, 2000. URL `http://dl.acm.org/citation.cfm?id=338219.338634`.

[80] Ricardo Baeza-Yates. A fast set intersection algorithm for sorted sequences. In *Annual Symposium on Combinatorial Pattern Matching*, pages 400–408. Springer, 2004.

[81] Peter Sanders and Frederik Transier. Intersection in integer inverted indices. In *Proceedings of the Meeting on Algorithm Engineering & Expermiments*, pages 71–83, 2007.

[82] Bolin Ding and Arnd Christian König. Fast set intersection in memory. *PVLDB*, pages 255–266, 2011. ISSN 2150-8097. doi: 10.14778/1938545.1938550. URL `http://dx.doi.org/10.14778/1938545.1938550`.

[83] Erik D. Demaine, Alejandro López-Ortiz, and J. Ian Munro. Experiments on adaptive set intersections for text retrieval systems. In *ALENEX*, 2001.

[84] Jérémy Barbay, Alejandro López-Ortiz, Tyler Lu, and Alejandro Salinger. An experimental investigation of set intersection algorithms for text searching. *ACM Journal of Experimental Algorithmics*, 2010.

[85] Philip Bille, Anna Pagh, and Rasmus Pagh. Fast evaluation of union - intersection expressions. In *ISAAC*, pages 739–750, 2007.

[86] Taesung Lee, Jin-Woo Park, Sanghoon Lee, Seung-won Hwang, Sameh Elnikety, and Yuxiong He. Processing and optimizing main memory spatial-keyword queries. *PVLDB*, 9(3):132–143, 2015.

[87] Robert Endre Tarjan. Finding optimum branchings. *Networks*, 7(1):25–35, 1977.

[88] Guy Kortsarz and David Peleg. Approximating shallow-light trees. In *SODA*, 1997.

[89] Judit Bar-Ilan, Guy Kortsarz, and David Peleg. Generalized submodular cover problems and applications. *Theoretical Computer Science*, 250(1):179–200, 2001.

[90] Moses Charikar, Chandra Chekuri, To-yat Cheung, Zuo Dai, Ashish Goel, Sudipto Guha, and Ming Li. Approximation algorithms for directed steiner problems. *Journal of Algorithms*, 33(1):73–91, 1999.

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

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

[93] `http://edmonds-alg.sourceforge.net/`.

[94] `http://svn.apache.org/repos/asf/subversion/trunk/notes/fsfs`, .

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

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

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

[98] Peter T. Wood. Query languages for graph databases. *SIGMOD Rec.*, 41(1):50–60, April 2012. ISSN 0163-5808.

[99] Eugene W Myers. An O(ND) difference algorithm and its variations. *Algorithmica*, pages 251–266, 1986.

[100] Erik D. Demaine, Shay Mozes, Benjamin Rossman, and Oren Weimann. An optimal decomposition algorithm for tree edit distance. *ACM Trans. Algorithms*, 6 (1):2:1–2:19, December 2009. ISSN 1549-6325. doi: 10.1145/1644015.1644017. URL `http://doi.acm.org/10.1145/1644015.1644017`.

[101] Fred Douglis and Arun Iyengar. Application-specific delta-encoding via resemblance detection. In *USENIX Annual Technical Conference, General Track*, pages 113–126, 2003.

[102] Git Packfiles. `https://git-scm.com/book/en/v2/Git-Internals-Packfiles`, . Accessed: February 15, 2017.

[103] Richard M Karp. Reducibility among combinatorial problems. In *Complexity of computer computations*. Springer, 1972.

[104] Jérémy Barbay and Claire Kenyon. Deterministic algorithm for the t-threshold set problem. In *Algorithms and Computation*, pages 575–584. Springer, 2003.

[105] Samy Chambi, Daniel Lemire, Owen Kaser, and Robert Godin. Better bitmap performance with roaring bitmaps. *Software: Practice and Experience*, 2015.

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

[107] Miller. `http://johnkerl.org/miller/doc/`. Accessed: April 15, 2018.

[108] csvkit. `https://csvkit.readthedocs.io/en/1.0.3/`. Accessed: April 15, 2018.

[109] textql. `https://github.com/dinedal/textql`. Accessed: April 15, 2018.

[110] `http://parquet.apache.org/documentation/latest/`.

[111] Anastassia Ailamaki, David J. DeWitt, Mark D. Hill, and Marios Skounakis. Weaving relations for cache performance. In *VLDB 2001, Proceedings of 27th International Conference on Very Large Data Bases, September 11-14, 2001, Roma, Italy*, pages 169–180, 2001. URL `http://www.vldb.org/conf/2001/P169.pdf`.

[112] Marcel Kornacker, Alexander Behm, Victor Bittorf, Taras Bobrovytsky, Casey Ching, Alan Choi, Justin Erickson, Martin Grund, Daniel Hecht, Matthew Jacobs, Ishaan Joshi, Lenni Kuff, Dileep Kumar, Alex Leblang, Nong Li, Ippokratis Pandis, Henry Robinson, David Rorke, Silvius Rus, John Russell, Dimitris Tsirogiannis, Skye Wanderman-Milne, and Michael Yoder. Impala: A modern, open-source SQL engine for hadoop. In *CIDR 2015, Seventh Biennial Conference on Innovative Data Systems Research, Asilomar, CA, USA, January 4-7, 2015, Online Proceedings*, 2015. URL `http://cidrdb.org/cidr2015/Papers/CIDR15_Paper28.pdf`.

[113] Ravi Jampani, Fei Xu, Mingxi Wu, Luis Leopoldo Perez, Chris Jermaine, and Peter J. Haas. The monte carlo database system: Stochastic analysis close to the data. *ACM Trans. Database Syst.*, 36(3):18:1–18:41, 2011. doi: 10.1145/2000824.2000828. URL `http://doi.acm.org/10.1145/2000824.2000828`.

[114] Spyros Blanas and Jignesh M. Patel. Memory footprint matters: efficient equi-join algorithms for main memory data processing. In *ACM Symposium on Cloud Computing, SOCC '13, Santa Clara, CA, USA, October 1-3, 2013*, pages 19:1–19:16, 2013. doi: 10.1145/2523616.2523626. URL `http://doi.acm.org/10.1145/2523616.2523626`.