ABSTRACT

Title of dissertation: EFFICIENT LAYOUTS AND ALGORITHMS
FOR MANAGING VERSIONED DATASETS

Souvik Bhattacherjee
Doctor of Philosophy, 2018

Dissertation directed by: Professor Amol Deshpande
Department of Computer Science

Version Control Systems were primarily designed to keep track of and provide control over changes to source code and have since provided an excellent way to combat the problem of sharing and editing files in a collaborative setting. The recent surge in data-driven decision making has resulted in a proliferation of datasets elevating them to the level of source code which in turn has led the data analysts to resort to version control systems for the purpose of storing and managing datasets and their versions over time. Unfortunately existing version control systems are poor at handling large datasets primarily due to the underlying assumption that the stored files are relatively small text files with localized changes. Moreover the algorithms used by these systems tend to be fairly simple leading to suboptimal performance when applied to large datasets. In order to address the shortcomings,

a key requirement here is to have a Dataset Version Control System (DVCS) that will serve as a common platform to enable data analysts to efficiently store and query dataset versions, track changes to datasets and share datasets between users at ease.

Towards this goal, we address the fundamental problem of designing storage layouts for a wide range of datasets to serve as the primary building block for an efficient and scalable DVCS. The key problem in this setting is to compactly store a large number of dataset versions and efficiently retrieve any specific version (or a collection of partial versions). We initiate our study by considering storage-retrieval trade-offs for versions of unstructured dataset such as text files, blobs, etc. where the notion of a partial version is not well-defined. Next, we consider array datasets, i.e., a collection of temporal snapshots (or versions) of multi-dimensional arrays, where the data is predominantly represented in single precision or double precision format. The primary challenge here is to develop efficient compression techniques for the hard-to-compress floating point data due to the high degree of entropy. We observe that the underlying techniques developed for unstructured or array datasets are not well suited for more structured dataset versions – a version in this setting is defined by a collection of records each of which is uniquely addressable. We carefully explore the design space for build-

ing such a system and the various storage-retrieval trade-offs, and discuss how different storage layouts influence those trade-offs. Next, we formulate several problems trading off the version storage and retrieval cost in various ways and design several offline storage layout algorithms that effectively minimize the storage costs while keeping the retrieval costs low. In addition to version retrieval queries, our system also provides support for record provenance queries. Through extensive experiments on large datasets, we demonstrate that our proposed designs can operate at the scale required in most practical scenarios.

EFFICIENT LAYOUTS AND ALGORITHMS FOR MANAGING VERSIONED DATASETS

by

Souvik Bhattacherjee

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

Advisory Committee:
Professor Amol Deshpande, Chair/Advisor
Professor Richard Marciano, Dean's Representative
Professor Alan Sussman
Professor Peter J. Keleher
Professor Hector Corrada Bravo

Dedication

To Baba, Maa, Piyana & Anahita.

Acknowledgments

To begin with, I would like to express my sincere gratitude to my advisor, Amol Deshpande, for providing me with the necessary guidance to successfully complete my dissertation. Over the years, his constant encouragement and constructive criticism has instilled in me the confidence to evolve and mature as a researcher. Not only has he introduced me to highly challenging, yet relevant problems but has also trusted in me by allowing me the latitude to pursue my own research ideas. I am greatly indebted to him for supporting my graduate studies and I am fortunate to have him as my advisor.

I wish to profoundly thank Alan Sussman for his valuable guidance especially during the PStore project at UMD and sharing his expertise in HPC systems research. I would also cherish our candid conversations which greatly helped me in dealing with various issues that any graduate student has to go through.

I would like to extend my sincere thanks to my collaborators in the DATAHUB project: Anant Bhardwaj, Amit Chavan, Aaron J. Elmore, Silu Huang, Samuel Madden and Aditya Parameswaran, for their valuable contribution. The work described in Chapter 3 was done jointly with two other PhD students, Amit Chavan and Silu Huang with joint responsibility for all the aspects of the work.

I would like to acknowledge my committee members Hector Corrada Bravo, Pete Keleher and Richard Marciano for agreeing to be in my thesis committee and for their valuable time in reviewing my dissertation. I am greatly indebted to my mentors Mohammad Sadoghi at IBM T. J. Watson Research Centre, Ravishankar Ramamurthy and Badrish Chandramouli at Microsoft Research for providing valuable guidance during my internships. Working with them helped me to deepen my understanding as well as increase the breadth of my knowledge.

I am greatly indebted to my friends who helped me survive through this PhD tenure. Thanks to my colleagues in the database lab - Abdul, Amit, Ashwin, Hui, Jayanta, Kostas, Theo and Udayan, for making the place lively and productive. I am also fortunate to have met a bunch of cheerful and motivated individuals over these years. Thanks to Agniv, Ankit, Debdipta, Dipankar, Rishov, Siddharth, Soham, Soumyadeep, Sunandita, Tanmoy, Udit and Upamanyu. I would also like to thank two most wonderful couples: Shrin & Geetanjali and Sayan & Esha, for accompanying us on numerous weekend sojourns. Outside of the UMD campus, I had the privilege of having friends like Anamitra, Pratyay, Sayak and Syamantak, who had provided motivation and support to maintain my tempo throughout this journey.

Finally, I would like to mention those people who are most important to me and to whom I dedicate this dissertation. To my late father, for making me the person that I am today. I wish he was here today to witness this moment. To my mother, for her unsung sacrifices and for inspiring me to realize my dreams. To my wife, Piyana, for her love and constant support and for keeping me sane over the past few years. Thank you for your unwavering belief in me, for standing by me through thick and thin but most of all, thank you for being my best friend. Above all, my beautiful daughter Anahita, my delightful bundle of joy for invoking hope, optimism and altruism in me and making me a more responsible and better person.

Table of Contents

List of Tables

List of Figures

Chapter 1: Introduction

The proliferation of data from an increasing number of highly diverse sources ranging from social networks to healthcare, business transactions to climate data and the numerous applications that reap benefits from it has undoubtedly made it an invaluable commodity. As such this commodity needs to be harnessed properly in order to extract the tremendous business value or derive rich scientific knowledge that is inherent within it. This in turn has led to an increase in data-driven decision making in a large number of areas including, but not limited to, online advertising, credit scoring, financial trading, chronic disease forecasting and treatment, fraud detection, product recommendation and so on [38, 68]. This data-driven decision making process has also resulted in an explosion in the number of datasets that are generated in the process within organizations that are treated at par with source code due to their inherent value [45].

Due to the huge and diverse nature of these datasets, researchers, domain experts and "data science" teams across multiple domains tend to collaborate to harvest the knowledge out of it. Consider for example, the Human Brain Project (HBP)[1] launched in October, 2013 and aimed to put in place a cutting-edge research infrastructure for brain research, cognitive neuroscience and brain-inspired computing. The HBP Consortium has 116 partners from a wide range of European organizations and brings together experts in areas of neuroscience, computing and medicine to advance the state-of-the-art. It is beyond question that a project of this dimension that involves participation of several institutions and across multiple disciplines would require a collaboratory platform that enables individuals or teams to share, exchange, collaborate on code and datasets.

While data science algorithms that mine the data for extracting valuable insights are important, managing the datasets is also critical for supporting data science activities. There are well-defined collaborative tools that allow teams to manage and collaborate on source-code such as Git, SVN or Mercurial; however none of these tools are well-suited for managing, sharing and tracking changes to large *datasets* [5, 6]. This has resulted in teams resorting to storing data in file systems and using ad-hoc techniques for managing the data. In order to deal with the difficulties, a key requirement here is to have a Dataset Version Control System (DVCS), that may serve as a platform for efficiently storing versions

[1] https://www.humanbrainproject.eu/

of data, capture modifications succinctly, identify differences between versions, share the datasets and be able to execute different types of queries at ease. In addition to these, such a system should also be able to support auditing and provenance tracking. Further, the system needs to be able to handle datasets that range from hundreds of MBs to several TBs and consist of several thousand versions. In this dissertation, we focus on developing techniques that are suited for handling diverse data types, ranging from unstructured types such as documents, images, etc., to more structured datasets such as CSV datasets, array datasets, etc. Finally, to handle the scale that we envision and simultaneously support efficient queries, designing workload-aware storage layouts is important that serve as the backbone of our DVCS.

1.1 Motivating Scenarios

In this section, we present practical scenarios that motivate us to design distinct layouts for storing different types of datasets compactly and answering frequently issued queries on those datasets, efficiently.

[Intermediate Result Datatsets] For most organizations dealing with large volumes of diverse datasets, a common scenario is that many datasets are repeatedly analyzed in slightly different ways, with the intermediate results stored for future use. Often, we find that the intermediate results are the same across many pipelines (e.g., a *PageRank* computation on the Web graph is often part of a multi-step workflow). Often times, the datasets being analyzed might be slightly different (e.g., results of simple transformations or cleaning operations, or small updates), but are still stored in their entirety. There is currently no way of reducing the amount of stored data in such a scenario: there is massive redundancy and duplication, and often the computation required to recompute a given version from another one is small enough to not merit storing a new version.

[Data Science Dataset Versions] In the absence of any dataset management tools, a group of data scientists working on a dataset curation workflow, may need to make modifications to the datasets for cleansing, corrections, adding annotations, etc., at various stages. Not every modification is applied in-place as it is also important to checkpoint the data at various stages of the workflow for the purposes of accountability. This results in the generation of new versions throughout the data curation process. Moreover, every time another team of data scientists wishes to work on a particular version of the dataset, they make a private copy and perform modifications to it. As a result, there is massive redundancy and duplication across these copies, and there is a need to minimize these storage costs while keeping these versions easily retrievable.

[Array Dataset Versions] The increasing volumes of array data generated by geospatial and temporal data such as satellite imagery, high-resolution climate simulations, telescope imagery has triggered the need for understanding and developing efficient techniques for array data management. For example, a single day of simulated time for mesoscale climate modeling can generate 30 GB of data while an entire ensemble simulation requiring the equivalent of several thousand years of simulated time can generate more than 30 PB of data [54]. Similarly, the Large Synoptic Survey Telescope (LSST) will generate 10s to 100s of petabyte a year of imagery and derived data [10]. Although enormous computational power can be applied to run the simulations themselves, *storing* the data that is generated during these simulations, and later *querying* it during subsequent offline analysis, can be a major challenge, especially with the trend toward very high resolution simulations. In many cases, the inability to offload the data onto a storage device in a timely manner leads domain scientists to throw away much of the data that is generated, maybe by sampling at a lower resolution, or by storing only a subset of the simulation variables, or by summarizing in various ways. Since it is difficult to evaluate the long-term importance of any specific data products at simulation time, none of these options is very attractive, and often simulations need to be re-run when deficiencies in the stored data are revealed.

[Keyed Dataset Versions] A healthcare provider who wants to perform different types of diagnostic and prognostic analytics may need to continuously maintain and analyze Electronic Health Records (EHRs) of thousands to millions of patients, where each EHR is identified uniquely by a primary key, e.g. SSN of patient. The EHR dataset is continuously changing through addition/deletion of new patient EHRs and updates to existing ones. For many practical reasons, results of applying any analytics are usually stored in the same EHR documents. Data analysts usually target a particular group of people when running analytical tasks in order to minimize the number of variables, e.g., people between age 50 - 60, belonging to a given ethnicity, with certain other characteristics, etc. As a result the number of updates per version usually remains restricted to a small percentage w.r.t the total pool of patients. Different teams of data scientists, with different goals, may be tweaking, training, and applying predictive models to those documents at the same time. Because of decentralized nature of the updates and increased use of collaborative analytics, the resulting version histories are mostly "branched". For accountability and debugging, it is essential that the precise details and provenance of all of those steps are maintained; e.g., an analyst must be able to clearly identify which versions of the EHRs were used to train a particular model, or which models were used to derive a specific individual prediction. It is also necessary for them to retrieve all or a subset of past versions of patients to analyze them for insights. Further, looking up a patient history from the point it enters their system is a very common query for them. The EHR schemas also evolve continuously when new data points that correspond to non-existing attributes are added in the form of new medical tests or measurements to a subset of the EHRs. Given the scale of the data, continuously evolving and semi-structured schema, and a desire to support distributed collaboration, key-value stores are often a natural option for storing such data (an extraction step to convert from the highly normalized relational databases where the original data is stored is quite common).

1.2 Challenges in Designing Storage Layouts for Versioned Datasets

Designing storage layouts constitute an important problem in the design of a DVCS. An ideal storage layout in a DVCS must not only reduce the query execution time of the most frequently issued (or popular) queries in the system but also bring down the storage cost by exploiting redundancies in the dataset. Furthermore, due to different types of (versioned) datasets that a DVCS is expected to handle, a particular storage layout that may be appropriate for a given type of dataset may not be suitable at all for another type of dataset. In what follows, we present three challenges that arise in the context of managing and designing storage layouts for three different types of datasets and the versions derived from it.

Challenge 1. As demonstrated in the first two scenarios above, there is a high degree of overlap among these datasets that makes it necessary to exploit the redundancy in order to store them compactly. However, it is easy to see that more compactly we store the datasets, more time we spend in reconstructing the versions and vice-versa. The first challenge, which we consider in this dissertation, is understanding the *storage–recreation tradeoff*. We illustrate this trade-off via an example.

Example 1. *Figure 1.1(i) displays a version graph, indicating the derivation relationships among 5 versions of a dataset. 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 1.1, V_1 is associated with $\langle 10000, 10000 \rangle$, indicating that V_1's storage cost and recreation cost are both 10000 when stored in its entirety (we note that these two are typically measured in different units – see the*

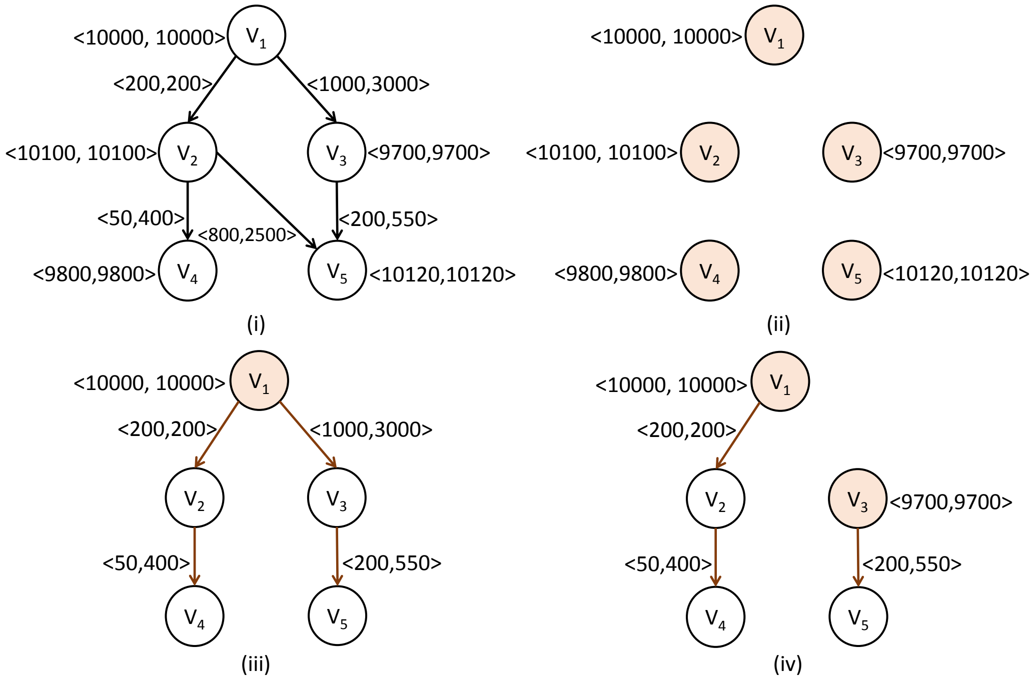


Figure 1.1: (i) A version graph over 5 datasets – annotation $\langle a, b\rangle$ indicates a storage cost of a and a recreation cost of b; (ii), (iii), (iv) three possible storage graphs.

second challenge below); the edge $(V_1 \rightarrow V_3)$ *is annotated with* $\langle 1000, 3000\rangle$*, where* 1000 *is the storage cost for* V_3 *when stored as the modification from* V_1 *(we call this the* delta *of* V_3 *from* V_1*) and* 3000 *is the recreation cost for* V_3 *given* V_1*, i.e, the time taken to recreate* V_3 *given that* V_1 *has already been recreated.*

One naive solution to store these datasets would be to store all of them in their entirety (Figure 1.1 (ii)). In this case, each version can be retrieved directly but the total storage cost is rather large, i.e., $10000 + 10100 + 9700 + 9800 + 10120 = 49720$*. At the other extreme, only one version is stored in its entirety* *while other versions are stored as modifications or deltas to that version, as shown in Figure 1.1 (iii). The total storage cost here is much smaller (*$10000 + 200 + 1000 + 50 + 200 = 11450$*), but the recreation cost is large for* V_2, V_3, V_4 *and* V_5*. For instance, the path* $\{(V_1 \rightarrow V_3 \rightarrow V_5)\}$ *needs to be accessed in order to retrieve* V_5 *and the recreation cost is* $10000 + 3000 + 550 = 13550 > 10120$.

Figure 1.1 (iv) shows an intermediate solution that trades off increased storage for reduced recreation costs for some version. Here we store versions V_1 *and* V_3 *in their entirety and store modifications to other versions. This solution also exhibits higher storage cost than solution (ii) but lower than (iii), and still results in significantly reduced retrieval costs for versions* V_3 *and* V_5 *over (ii).*

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 layouts for a given collection of datasets. 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.

Challenge 2. As demonstrated by the third scenario above, reducing the volume of the datasets after it has been generated is an important challenge that needs to be addressed. Moreover, since floating point data is the most prevalent type of data generated by scientific simulations, it is important to consider compression techniques tailored for such data due to its high degree of entropy, especially in the lower order bytes of the mantissa (fraction) part [19]. Querying the data after it has been stored is important for purposes of analyzing the data – the dominant form of queries on arrays are range slicing queries i.e. queries that read a hyper-rectangle from one or a collection of array versions. As a result, while it is important to pursue aggressive techniques that reduce the volume of data, it is equally important to design storage layouts for array data that minimize the query latencies for the aforementioned queries.

An array dataset is essentially a collection of temporal snapshots (or versions) of multi-dimensional arrays. Since arrays are homogenous entities, each cell in an array stores data of the same type. Therefore, each cell of a collection of multi-dimensional array snapshot may contain a temperature measurement, recorded at a particular point in 3D space at a certain point in time or generated as a result of climate simulation model. Using the relational model for arrays turns out be inefficient [81], thereby requiring a different data model for efficient representation. Further, simple delta-based encoding would turn out to be inefficient for answering range queries as the entire array version version has to be retrieved for returning a portion of the array that falls within the query range. Therefore the second challenge here is to design the storage layout for array datasets with floating point data that would facilitate efficient range queries on the data.

Challenge 3. The techniques developed for unstructured and array datasets (i.e. unstructured text data, blobs, records with no primary keys) are not well-suited for keyed datasets that comprise a significant part of the data ecosystem, as we observe subsequently. We define a version in this setting to be a collection of records each having a primary key that may be used to uniquely identify a specific record. For simplicity, consider a chain of *deltas* between versions where a delta between two versions captures the information needed to transform one version into another. Each delta may have new records, updates to existing records or records that are marked as deleted. To access versions towards the end of the delta-chain we need to apply all the deltas one after another from the base version until we reach the queried version. Thus the number of records fetched is proportional to the distance of the version from the base version. However, we observe that it suffices to store the records within the versions *only once* along with a map that stores the version to primary key information. In this case, for any version query we need to access exactly the same number of records that a version contains. Motivated by this observation, we store the records only once in a key-value store (KVS). Therefore, if we need to retrieve a version, we need to consult a map that has all the keys of records that belong to a version.

The third challenge thus arises in the context of designing efficient storage layouts for datasets with keyed records. Now consider the final scenario presented above where versions consists of on the order of a hundred thousand records; querying the KVS for each version recreation on the order of the number of records in it turns out to be an expensive operation. An alternate approach is to issue range queries to the KVS for retrieving the versions. However the client needs a version to record map for issuing the queries. Unfortunately the size of these maps may be on the order of a few Gigabytes and may turn out to be a burden on the part of the client. To avoid this bottleneck, we put multiple records together into a *chunk* and maintain a map per chunk that indicates the versions that each record belongs to. In addition to this map that is maintained on a per chunk basis, we maintain another map that indicate the chunks in which records of a version belongs to. This approach reduces the number of requests to the KVS by several orders of magnitude depending on the average number of records per chunk. The latter map, which is of few KBs, resides with the user and is consulted during version recreation.

In order to validate our claim, we performed a simple experiment using Apache Cassandra [1] as the underlying KVS. Each version in the dataset has around

Chunk size	Time (in secs.)
1	65.415
10	14.175
100	3.098
1000	1.072
10000	0.562

Table 1.1: Query performance for different chunk sizes; version size $\sim$100K records

100K records and there are a total of 1 million unique records stored in the KVS and each record occupies around 100 bytes. The query here is to reconstruct a version implying that we need to retrieve around 100K records for every version reconstruction query from the KVS. In the naive setting, we maintain a chunk of unit size and issue around 100K requests to the KVS. In comparison, we create larger sized chunks using a random assignment of records to chunks. As a result, we need to retrieve more number of chunks than exactly required to recreate a version. However the overhead of retrieving additional chunks and scanning through them to extract the records is significantly less. Table 1.1 illustrates the performance of the different chunking strategies described above.

Thus the challenge of designing an efficient storage layout for keyed records reduces to the problem of partitioning records into chunks, such that *span* of a version query is minimized. We define span (of a version) to be the number of queries to the underlying KVS.

Also note that there may be small differences between two different versions of a record (e.g., only a single attribute may be updated in a document). One way to exploit this overlap is to store the two versions of the record together in a "compressed" fashion, with specific compression technique chosen according to the data properties (e.g., one may store "deltas" (differences) between the two records, or use an off-the-shelf compression tool that in effect does the same thing). Such compression, however, negatively impacts the query performance and restricts the data placement opportunities. Therefore another challenge here is to partition the records to minimize the span in the presence of record compression.

This dissertation focuses on identifying and addressing the fundamental challenges involved in the design of a DVCS for efficient and scalable dataset version management. Towards this goal, we have focussed primarily on designing efficient storage layouts for both structured and unstructured datasets that serve as the backbone in the pursuit of building scalable and efficient DVCS.

1.3 Contributions and Dissertation Organization

In this dissertation, we address the aforementioned challenges in a systematic manner. Our primary goal here is to evaluate the underlying requirements of a DVCS and come up with appropriate design layouts for storing and querying both unstructured and structured data.

1.3.1 Layouts for Unstructured Datasets

The high degree of redundancy present in the datasets, that are mostly unstructured, resulting from collaboratory efforts motivates us to use *delta* encoding to store them compactly. Delta encoding is a technique that only stores the *changes* that have occurred to a target version with respect to a source version, instead of storing the entire target version. In general, the source and the target versions are so chosen such that their delta is small, which in other words imply that they have a large amount of overlap between them. Thus it is possible to form a weighted graph where nodes correspond to versions and the edge between two versions exist if the delta between them have been computed. The size of the delta (in bytes) forms the weight of the edge between the versions. In order to represent the *materialized* versions, we add an edge between every version to an empty root version and the weight of the edge is basically the size of the materialized version. To obtain the minimum storage solution, we must choose the appropriate edges in this graph that minimize the overall cost of the resulting tree. Using the resulting tree, it is possible to reconstruct any version of interest by applying the deltas in the path from a materialized version to the desired version.

However using delta compression to minimize storage space may lead to very high latencies while retrieving specific versions. We also demonstrate that the compaction heuristics used by popular VCS' like Git and SVN are ineffective at storing datasets, both in terms of resources consumed and the quality of solution produced. More importantly, they focus only on reducing the storage footprint of the datasets involved while ignoring the effect of the compaction strategy on query performance. 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 for a majority of the variations which are shown to be NP-Hard [23]. 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.

Our contributions here are as follows:

1) 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.

2) We provide two light-weight algorithms: 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. We also demonstrate that our algorithms outperform the compaction algorithms used by popular VCS' like Git and SVN due to their ineffectiveness at storing datasets, both in terms of resources consumed and the quality of solution produced.

3) We have built a prototype system where we implement the proposed algorithms. We present an extensive experimental evaluation of these algorithms over several synthetic and real-world workloads demonstrating the effectiveness of our algorithms at handling large problem sizes.

A detailed description of the contributions listed above can be found in Chapter 3.

1.3.2 Layouts for Array Datasets

To address the challenge of reducing the data volume after it has been generated but before storing it, we can use sophisticated compression techniques that exploit the high spatial and temporal correlation that exists in the data generated by scientific simulations. While the conventional compression techniques such as as zlib [18] and lzo [11] are efficient for compressing the data, the overall compression ratio may be enhanced by using them in conjunction with some preprocessing techniques, such as computing deltas, which makes the data amenable to better compression. Further, it is essential to select the best compression procedure given a particular dataset, instead of using a fixed, preselected compression technique. Although better compression ratios reduce space on disk, they may result in higher compression and/or decompression times. For example, zlib does a very good job of compressing a dataset, however it loses out to lzo in terms of compression/decompression speed. Therefore, it is essential to provide users with the flexibility of choosing the appropriate compression technique(s) based on their individual requirements. However, both zlib and lzo are not suitable for compressing floating point data, which is the most predominant type of data resulting from scientific simulations necessitating the need for development of compression techniques for such data. Moreover, the full precision of floating point data may not always be required for some applications, e.g., visualizations, where truncation of the lower order mantissa bits can be tolerated. As a result, we may not be required to retrieve all the bytes of floating point data for query processing. Thus we can perform a byte-wise partitioning of the floating point data before storing on disk.

Before the data is stored, it is partitioned and then compressed. The execution begins by analyzing a sample of the data to be stored in an offline mode and chooses the appropriate compression scheme suited for the given dataset. Thereafter the multi-dimensional simulation data is partitioned along three different dimensions: (1) temporal, (2) spatial, and (3) bytewise. Partitioning the multidimensional data along both temporal and spatial dimensions is a technique employed for alleviating dimension dependency, resulting in low latency range queries.. In addition to single-level array partitioning, we implement a two-level array partitioning technique and propose a variant of the technique when the effect of compression is taken into consideration. Another dimension of partitioning is bytewise, motivated by the observation that the full precision of the data is not required in many offline analysis and visualization tasks. Therefore the data is partitioned along byte boundaries, with the added benefit that the compression *efficiency* for this storage strategy is significantly better than traditional compression techniques for floating point data.

Overall, our contributions are as follows:

1) We design and implement PSTORE, a framework for managing array datasets that supports a suite of compression schemes and selects the best compres-

sion plan based on the nature of the data. This procedure is carried out in an offline mode where a representative sample of the data selected for ingestion is analyzed and a compression plan is selected.

2) We integrate both byte-wise partitioning of floating point data and partitioning along data array dimensions to provide maximum flexibility in terms of accessing the desired data elements.

3) We propose a two-level partitioning/chunking strategy in the context of compression and show that it is better off (w.r.t the query response time) compared to a single-level chunking with compression.

4) For long running range queries, where several data chunks are accessed, it is beneficial to hide I/O latencies by overlapping them with CPU processing time. Since file accesses are often sequential, we can process one chunk at a time and overlap the I/O access for the next chunk with CPU processing of the current chunk.

A detailed description of PSTORE can be found in Chapter 4.

1.3.3 Layouts for Datasets with Keys

To address the final challenge, instead of using delta encoding, we elevate records within the versions to first class citizens as every version can be expressed in terms of their constituent records. Although the presence of a *primary key* in each record makes it unique within a version, the same is not true across versions. Since a record may be unchanged from one version to the next, to be able to refer to a specific record within a specific version, we use a *composite key*: $\langle$primarykey, versionid$\rangle$, where the second part refers to the *version-id* of the version where the record was created. This allows us to uniquely reference records within a global address space. We chose to use version-id of the appropriate version instead of an auto-incremented value as the latter introduces additional difficulties in a decentralized setting without any obvious benefits. Our objective here is to be able to retrieve any version with the help of an index that maintains the version to composite key mapping. To be able to deploy our solution in a distributed environment, we choose to store the records in a key-value store.

Due to the aforementioned issues, this problem must be solved by explicitly creating *chunks* of records, where records belonging to the same set of versions are grouped together. The primary challenge here now is deciding how to partition the records into chunks to maximize the query performance. To address the challenge at hand, we formally define the problem and draw connections to other existing problems in literature. We show that the problem maps to an existing problem and is NP-Hard. As a next step, we propose several heuristics that effectively use the information to partition the record space efficiently. Thereafter we focus on a more general version of the aforementioned problem that attempts to compress different versions of the record together while simultaneously trying to achieve a good partitioning of the records. Through extensive experimentation over multiple datasets, we demonstrate that our proposed techniques can operate at the scale required in most practical scenarios.

Our key contributions are as follows:

1) We systematically explore the design space for supporting versioning as a first-class construct in distributed key-value stores and present a detailed analysis of the different trade-offs and how different baselines fare with respect to those.

2) We propose a flexible system architecture, that supports the key desiderata through use of *chunking*.

3) We design novel offline partitioning algorithms that exploit how the versions

relate to each other to identify good chunking strategies. We also present an online algorithm to keep the partitioning and the indexes up-to-date as new versions are committed.

4) We have built a working prototype, RSTORE, on top of the Apache Cassandra key-value store, which we use to validate our design decisions. We expect that RSTORE, like many NoSQL stores, will primarily be deployed in a distributed environment; however, it can also be used in a local cluster.

We describe this in detail in Chapter 5.

Chapter 2: Literature Survey

In this chapter, we present a survey of the literature in managing versioned data. We begin with a discussion of prior work in multi-versioned data management in relational database systems, array databases, XML and graph databases and present the key differences with our work. Deduplication of data in storage of archival data and backup systems is an important area of research that is related to eliminating redundancy and storing multiple versions of data compactly. We present a comprehensive literature survey and compare the techniques with the techniques we develop in this dissertation. Thereafter, we provide a survey of version control systems, that were primarily designed to keep track of and manage changes to source code. In this dissertation, our primary goal is to design storage layouts for version control systems for datasets. Next, we review cloud-based data systems, and in particular, key-value stores that serves as the backbone of RStore, our system for managing multi-versioned keyed documents. We also review different types of partitioning techniques that we have used for array datasets and keyed datasets, for minimizing the query latencies on these datasets. Finally, we conclude the literature survey with floating point data compression techniques; floating point data is the predominant type of data generated in scientific simulations and compressing such data due to its high degree of entropy is challenging and constitutes an important topic for survey.

2.1 Multi-Version Storage Systems

2.1.1 Relational Database Management Systems.

An important requirement of a large number of database systems is to store and retrieve multiple versions of records which led to extensive research in the area of temporal and versioned databases [36, 57, 58, 59, 65, 82]. Most of this work focuses on managing a linear chain of versions and retrieving a version as of a specific time point (called *snapshot* queries). There also has been work in the area of temporal databases for handling data evolving with time [76, 77]. There are several databases that support "time travel" features that allow users to retrieve historical snapshots. Postgres [80] was the among the first to offer temporal functionality. Thereafter, there were many databases from the industry that supported this feature [9, 15, 17].

2.1.2 Scientific Databases

The inadequacy of current commercial relational database management systems has triggered the need for developing specialized data management systems for catering to science applications [81]. The requirements include a 1) nested array data model, 2) science-specific primitive operations, 3) no-overwrite storage, 4) time-travel, to name a few. This has resulted in many specialized multidimensional array-processing systems [21, 28] with support for querying old versions of data. Recent work by Seering et al. [74] considered the problem of storing an arbitrary tree of versions in the context of scientific databases. Their proposed techniques are based on finding storage layouts for compactly storing dataset versions. Soroush and Balazinska [78] present a storage manager for versioned arrays and supports extraction of sub-arrays of a version or multiple sub-arrays across a range of versions. It also supports approximate version selection and historical queries to speed-up exploration of versioned arrays.

2.1.3 XML and Graph Databases.

There has been a lot of work on providing versioning support and compactly storing graph and XML data. Buneman et al. [30] 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. There has been prior work on compressing XML data [55] and providing versioning support [34]. Tools like XyDiff [37], X-Diff [84] have contributed towards comparing XML documents.

Most of the prior work on compressing graphs looked into neighborhood queries on the compressed graphs [20, 25, 35]. Khurana and Deshpande [51] present an approach for managing historical graph data for large information networks, and for executing snapshot retrieval queries on them. LLAMA is a graph storage and analysis system that allows temporal analysis on multiple snapshots which are constructed with each new batch of updates; the base snapshot is represented using compressed row storage format and new snapshots are constructed using copy-on-write approach [60]. ImmortalGraph [62] is a system for storing and analyzing temporal graphs that optimize in-memory organizations of temporal graphs by sharing storage and computation to ensure faster graph query processing.

2.1.4 Other Deduplication Schemes.

The area of automatic elimination of duplicate data in storage systems has witnessed a lot of prior work. Key techniques from this area are primarily employed for use in archival and backup systems. Douglis and Iyengar [39] 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. [64] 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 techniques have been used by Seering et al. [74] for compactly storing versions of array datasets. Burns and Long [31] present a technique for in-place re-construction of delta-compressed files using a graph-theoretic approach. Kulkarni et al. [52] 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). There is also work on file content deduplication by *indexing* that employs hashes (or signatures) for identifying similar blocks of data [27, 69]. The technique works by first identifying similar chunks of data across files or documents by computing a set of fingerprints for each chunk and then comparing the number of common fingerprints to assess the similarity. These fingerprints are then used to build indexes. Thus each block of chunk is stored once and each document can be represented by a collection of signatures. We refer the reader to a recent survey [66] for a more comprehensive coverage of this line of work.

Our work in Chapter 3 and Chapter 5 focuses on managing versions of data that can form an arbitrary DAG whereas prior work on temporal relational databases was restricted to managing a linear chain of versions. Further prior work on deduplication systems has primarily looked into the problem of minimizing the total storage cost as they were mostly used for archival purposes. We consider the problem of minimizing both storage and recreation cost by trading off one with the other.

2.2 Version Control Systems (VCS)

Version Control Systems (VCS) were primarily designed to keep track of and provide control over changes to source code (e.g. Git, SVN, Mercurial). It provides an excellent way to combat the problem of sharing files in a collaborative setting. They follow a no-overwrite storage model that enables users to retrive any version created at any point in time. Despite their popularity, these systems largely use fairly simple algorithms underneath and are known to have significant limitations when working with large files [5]. Many of the limitations stem from the assumption that the files being stored are relatively small text files with localized changes; data science applications, on the other hand, typically feature a range of large datasets from unstructured text files to structured database tables and the change patterns tend to be more complex. As a result, a variety of extensions like git-annex [7], Git Large File Storage [8], etc., have been developed to make them work reasonably well with large files. However these extensions are simple workarounds that replace large files with text pointers inside Git, while storing the actual file on a remote server. There has been some recent work on building a version control system specifically geared towards handling relational tables [61] and addresses some of the storage issues that arise while storing and querying the tables.

In Chapter 3, we build a dataset version control system that is inspired from the conventional version control system, such as Git. Version control systems are built upon the idea of a no-overwrite model and exploit the differences between neighboring versions of the files for compact storage. Through extensive experiments, we demonstrate that the underlying algorithms employed by Git and SVN are inefficient when dealing with large files both in terms of storage and resource consumption. We also show that these algorithms are primarily aimed at reducing the storage costs without considering the recreation cost of the versions.

2.3 Cloud-based Data Systems

Most cloud-based data systems including key-value stores primarily focus on providing efficient support for storing and retrieving data at the record level. Some of them provide support for additional features such as allowing range queries [1, 33, 67] however it would be difficult for them to support range queries on versioned datasets in the absence of special indexes. Although there is no full-fledged support for storing and querying multiple versions of the same record in these existing systems, there is some discussion about providing support for some *naive* form of versioning using the existing APIs in these systems. For example, the following online material [3, 14] describes how to implement versioning feature in Couchbase [2] and MongoDB [13]. The techniques described are similar and advocates storing the previous versions of the record in a separate shadow *collection* before overwriting it with the updated value. A version number property (an `int32` called `_version`) is added to the document to keep record of the different versions. The major downside of this approach as described is that records cannot be updated in batches and the older versions are more expensive to retrieve. Moreover it is not clear if they support compressing multiple versions of the same record together. These cloud-based systems also uses partitioning to distribute data across multiple nodes (or partitions). However in most of the cases, this is a simple hash-based partitioning scheme and is orthogonal to the sophisticated record-partitioning schemes that we employ to improve query performance.

In Chapter 5, we build a system to store and retrieve multiple versions of the same record as they evolve over time and use a key-value store for storing "chunks" of records. We show that none of the queries that we intend to support, i.e., full version retrieval, partial version retrieval or record evolution query, can be supported easily using the basic functionalities provided by a key-value store. Moreover, we employ sophisticated record partitioning schemes to minimize query latency, which is not present in any of the existing key-value stores.

2.4 Data Partitioning

The problem of partitioning data items (or records) for minimizing the query latencies has been addressed previously in [53] by mapping it to a hypergraph partitioning problem, however the problem setting was completely different from the data partitioning problem that we address in Chapter 5. The hypergraph in the previous setting had data items or tuples as vertices and the hyperedges correspond to a query in the workload that requests those data items. In Chaper 5, we define the vertices in the graph to be the records that constitute versions and a hyperedge connects all records that belong to a given version. The number of records in a version according to the current problem definition is at least two to three orders of magnitude larger than in the former setting, which makes the current problem significantly different. The main challenge in designing these algorithms arises due to the large sizes of the hyperedges and as a result it becomes difficult to get good partitions using standard hypergraph partitioning tools such as hMETIS [49]. In this work, we employ sophisticated partitioning schemes that partition records into different "chunks" by taking cues from the associated version graph for minimizing the total query span.

Prior work related to chunking multidimensional data has considered both single-level and two-level chunking for array storage [70, 75, 79]. In the two-level chunking scheme, the proposed techniques resort to different combinations of both regular and irregular tiling. However they do not consider the effect of compression while chunking the array. In Chapter 4, we propose a variant of two-level chunking that takes into consideration the effect of compression. The problem of tuning the chunk shape and size for a given query workload has also been considered previously [63, 71]. We show that the earlier formulations for computing the optimal chunk size can be modified to take into consideration the effect of compression.

2.5 Floating-point compression

Not all datasets are amenable to delta compression and therefore there is a need to select the best possible compression scheme for a given dataset. Moreover, the delta encoding techniques we refer to do not consider floating point numbers explicitly, which is the most prevalent type of data generated by scientific simulations and stored in multi-dimensional array snapshots. Due to the high degree of entropy present in the low-order mantissa bits, delta encoding between two floating point numbers does not result in low magnitude values which can then be stored using fewer of bits. Schendel et al. [73] propose the ISOBAR system which does a byte-wise partitioning of floating point numbers and preprocesses the bytes to identify compressible and hard-to-compress bytes. They observe that it may not be effective to compress all the bytes as some of the bytes may be incompressible due to the high amount of randomness present in them. In Chapter 4, we extend their technique to improve the delta encoding performance for floating point numbers. This partitioning strategy comes with the added benefit of being able to query the data approximately. Not all applications require full precision data and if the user specifies a relative error bound, it is possible to avoid retrieving all the bytes from the disk which makes query processing more efficient [48].

Other compression techniques for floating point numbers have been proposed, such as `fpzip` [56] and FPC [32] that are also used widely in scientific database applications. These techniques are based on context modeling applications, and use predictors for predicting the next value based on the values seen earlier in a sequence. The predicted value is then XORed with the actual value and the leading zero bytes are compressed. The performance of these techniques may degrade due to the use of predictors for predicting the next value. However for data generated by scientific simulations, one may do away with these predictors by using the next temporal or spatial value in an array since the values are highly correlated and are usually very close to the previous value. In Chapter 4, we extend FPC in our tFPC and sFPC compression methods, which essentially XORs neighboring temporal or spatial values, respectively.

Chapter 3: Storage Layouts for Unstructured Datasets

In this chapter, we present a principled study of the problem of designing efficient storage layouts for datasets where we do not assume any structure, while respecting the constraints involved in retrieving the versions. Here, we begin with a description of the data and the query model, followed by some preliminaries necessary for understanding the problem formulations. Next, we provide a description of the algorithms for constructing storage layouts for those datasets and conclude with experimental evaluation.

3.1 Data and Query Model

Data Model. A version in this setting is stored and retrieved in terms of *deltas*, where a delta from version V_j to V_i is defined as the information needed to construct version V_i from version V_j. For example, we could record that V_i is just V_j but with the 50th tuple deleted. The algorithm giving us the delta is called a *differencing algorithm*. Deltas need to be applied to a full version in order to construct another version, therefore, some versions are stored in their entirety; such versions are said to be *materialized*.

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 a DVCS (admitting collaborative data science), $\mathcal{G}$ is a DAG (directed acyclic graph) instead of a linear chain. For example, Figure 1.1 represents a version graph $\mathcal{G}$, where V_2 and V_3 are derived from V_1 separately, and then merged to form V_5.

Query Model. In a collaborative setting with large datasets, the query workload consist of retrieving an entire version. A version in this setting is assumed to be a data file whose contents may be unstructured text or binary data. Note that the notion of a partial version in this setting is not well-defined since there is no way to refer to a particular subset of the file.

3.2 Preliminaries and Problem Overview

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. As described earlier in Section 3.1, 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 $\Phi_{i,i}$.

- Store a "delta" from V_j: 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.

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

(i) Δ

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

(ii) Φ

Figure 3.1: Matrices corresonding to the example in Figure 1 (with additional entries revealed beyond the ones given by version graph)

Example 2. *Figure 3.1 shows the matrices Δ and Φ based on version graph in Figure 1.1. The annotation associated with the edge (V_i, V_j) in Figure 1.1 is essentially $\langle \Delta_{i,j}, \Phi_{i,j} \rangle$, whereas the vertex annotation for V_i is $\langle \Delta_{i,i}, \Phi_{i,i} \rangle$. If there is no edge from V_i to V_j in the version graph, we have two choices: we can either set the corresponding Δ and Φ entries to "$-$" (unknown) (as shown in the figure), or we can explicitly compute the values of those entries (by running a differencing algorithm). For instance, $\Delta_{3,2} = 1100$ and $\Phi_{3,2} = 3200$ are computed explicitly in the figure (the specific numbers reported here are fictitious and not the result of running any specific algorithm).*

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

- *Revealing entries in the matrix:* Ideally, we would like to compute all pairwise Δ and Φ entries, so that we do not miss any significant redundancies among versions that are far from each other in the version graph. However when the number of versions, denoted n, is large, computing all those entries can be very expensive (and typically infeasible), since this means computing deltas between all pairs of versions. Thus, we must reason with incomplete Δ and Φ matrices. Given a version graph $\mathcal{G}$, one option is to restrict our deltas to correspond to actual edges in the version graph; another option is to restrict our deltas to be between "close by" versions, with the understanding that versions close to each other in the version graph are more likely to be similar. Prior work has also suggested mechanisms (e.g., based on hashing) to find versions that are close to each other [39]. 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.

- *Branches:* Both branching and merging are common in collaborative analysis, making the version graph a directed acyclic graph. In this paper, we assume each version is either stored in its entirety or stored as a delta from a single other version, even if it is derived from two different datasets. Although it may be more efficient to allow a version to be stored as a delta from two other versions in some cases, representing such a storage solution requires more complex constructs and both the problems of finding an optimal storage solution for a given problem instance and retrieving a specific version become much more complicated. We plan to further study such solutions in future.

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

3.2.1 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.

3.2.2 Problem Formulations

We now state the problem formulations that we consider in this paper, 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 $\mathcal{C}$ 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 threshold β, identify $\mathcal{P}$ such that $\mathcal{C} \leq \beta$, and $\sum_i \mathcal{R}_i$ is minimized.*

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

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

Problem 5 (Minimizing Storage(Sum Recreation))**.** *Given Δ, Φ and a threshold θ, identify $\mathcal{P}$ such that $\sum_i \mathcal{R}_i \leq \theta$, and $\mathcal{C}$ is minimized.*

Problem 6 (Minimizing Storage(Max Recreation))**.** *Given Δ, Φ and a threshold θ, identify $\mathcal{P}$ such that $\max_i \mathcal{R}_i \leq \theta$, and $\mathcal{C}$ is minimized.*

Table 3.1 summarizes the problem formulations for the various dimensions described above.

	Storage Cost	Recreation Cost	Undirected Case, $\Delta = \Phi$	Directed Case, $\Delta = \Phi$	Directed Case, $\Delta \neq \Phi$
P1	$\min\{\mathcal{C}\}$	$\mathcal{R}_i < \infty, \forall i$	PTime, Minimum Spanning Tree		
P2	$\mathcal{C} < \infty$	$\min\{\max\{\mathcal{R}_i \mid 1 \leq i \leq n\}\}$	PTime, Shortest Path Tree		
P3	$\mathcal{C} \leq \beta$	$\min\{\sum_{i=1}^{n} \mathcal{R}_i\}$	NP-hard, LAST Algorithm†	NP-hard, LMG Algorithm	
P4	$\mathcal{C} \leq \beta$	$\min\{\max\{\mathcal{R}_i \mid 1 \leq i \leq n\}\}$		NP-hard, MP Algorithm	
P5	$\min\{\mathcal{C}\}$	$\sum_{i=1}^{n} \mathcal{R}_i \leq \theta$	NP-hard, LAST Algorithm†	NP-hard, LMG Algorithm	
P6	$\min\{\mathcal{C}\}$	$\max\{\mathcal{R}_i \mid 1 \leq i \leq n\} \leq \theta$		NP-hard, MP Algorithm	

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

3.2.3 Mapping to Graph Formulation

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

The resulting graph G is similar to the original version graph, but with several important differences. An edge in the version graph indicates a derivation relationship, whereas an edge in G simply indicates that it is possible to recreate the target version using the source version and the associated edge delta (in fact, ideally G is a complete graph). Unlike the version graph, G may contain cycles, and it also contains the special dummy vertex V_0. 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 3. *Given matrix Δ and Φ in Figure 3.1(i) and 3.1(ii), the corresponding graph G 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*

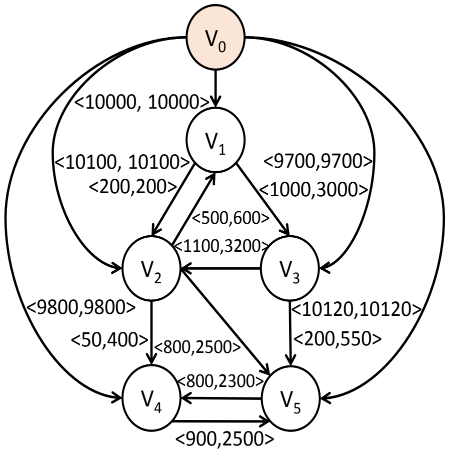


Figure 3.2: Graph G

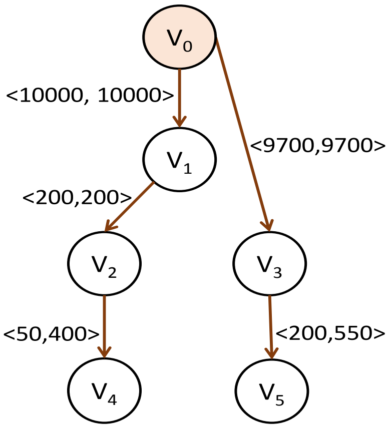


Figure 3.3: Storage Graph G_s

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 lemmas.

Lemma 1. *The optimal storage graph $G_s = (V_s, E_s)$ for all 6 problems in Table 3.1 must be a spanning tree T rooted at dummy vertex V_0 in graph G.*

Proof. Recall that a spanning tree of a graph $G(V, E)$ is a subgraph of G that (i) includes all vertices of G, (ii) is connected, i.e., every vertex is reachable from every other vertex, and (iii) has no cycles. Any G_s must satisfy (i) and (ii) in order to ensure that a version V_i can be recreated from V_0 by following the path from V_0 to V_i. Conversely, if a subgraph satisfies (i) and (ii), it is a valid G_s according to our definition above. Regarding (iii), presence of a cycle creates redundancy in G_s. Formally, given any subgraph that satisfies (i) and (ii), we can arbitrarily delete one from each of its cycle until the subgraph is cycle free, while preserving (i) and (ii). □

For Problems 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 $\Phi_{i,j}$.*

3.3 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 the problems discussed above are NP-Hard [23], we now focus on developing efficient heuristics. In this section, we present two novel heuristics: first, in Section 3.3.1, 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 3.3.2, 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 3.3.3, we present an approximation algorithm called LAST, and in Section 3.3.4, 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 3.4. 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,

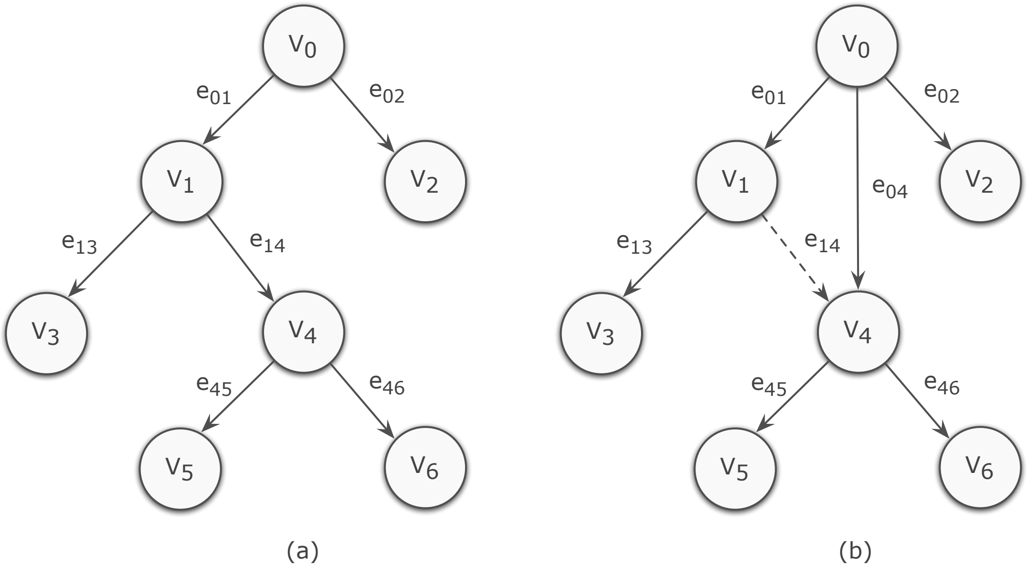


Figure 3.4: Illustration of Local Move Greedy Heuristic

and to the symmetric and unsymmetric cases.

3.3.1 Local Move Greedy Heuristic

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 4. *Figure 3.4(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. However, if we are smart about this computation (by precomputing and maintaining across all rounds the number of nodes “below” every node), we can reduce the complexity of computing ρ for a given edge to $O(1)$. This leads to an overall complexity of $O(|V|^2)$ Algorithm 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

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
Initialize $T$ as MST.
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 $T$.
while $W(T) < W$ do
    $(\rho_{max}, e_{SPT}) \leftarrow (0, \emptyset)$
    foreach $e_{uv} \in \xi$ do
        compute $\rho_e$
        if $\rho_e > \rho_{\max}$ then
            $(\rho_{max}, \bar{e}) \leftarrow (\rho_e, e_{uv})$
        end
    end
    $T \leftarrow T \setminus e_{u'v} \cup e_{uv};\quad \xi \leftarrow \xi \setminus e_{uv}$
    if $\xi = \emptyset$ then
        return $T$
    end
end
```

the numerator of ρ will capture the reduction in weighted recreation cost.

3.3.2 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. Furthermore, we maintain at all stages the recreation cost of all nodes present in T to be within the threshold θ

Detailed Algorithm. At all stages, the invariant maintained by the algorithm is that the recreation cost of all versions in T is bounded within θ. 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 using pseudocode. Set X represents the current version set of the current spanning tree T in Algorithm 2. 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

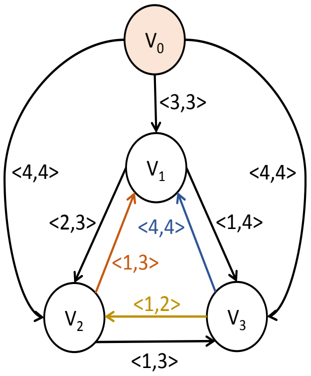


Figure 3.5: Directed Graph G

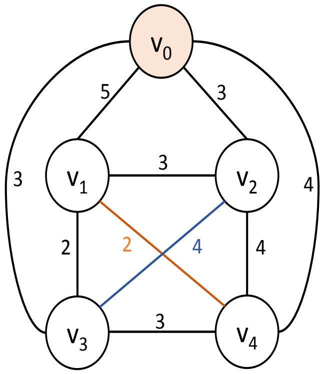


Figure 3.6: Undirected Graph G

V_i is added into T, we need to update the storage cost and recreation cost for all V_j that are neighbors of V_i. Notice that in Prim's algorithm, we do not need to consider neighbors that are already in T. However, in our scenario a better path to such a neighbor may be found and this may result in an update (line 10-17). For instance, if edge $\langle V_i, V_j \rangle$ can make V_j's storage cost smaller while the recreation cost for V_j does not increase, we can update $p(V_j) = V_i$ as well as $d(V_j)$, $l(V_j)$ and T. For neighbors $V_j \notin T$(line 19-24), we update $d(V_j)$, $l(V_j)$,$p(V_j)$ if edge $\langle V_i, V_j \rangle$ can make V_j's storage cost smaller and the recreation cost for V_j is no bigger than θ. Algorithm 2 terminates in $|V|$ iterations since one version is added into X in each iteration.

Example 5. *Say we operate on G given by Figure 3.5, 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*

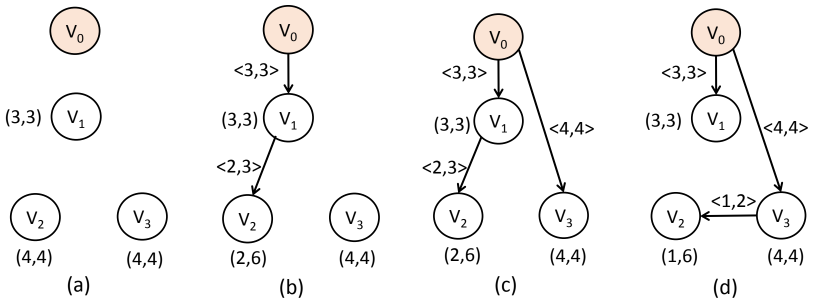


Figure 3.7: Illustration of Algorithm 2 in Figure 3.5

$< l(V_j), d(V_j) >$ as shown in Figure 3.7 (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 3.7 (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 3.7 (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 3.7 (c). Eventually, we get the final answer in Figure 3.7 (d).*

Complexity. The complexity for Algorithm 2 is the same as that for 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.

3.3.3 LAST Algorithm

Here, we sketch an algorithm from previous work [50] 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 3.2.3, 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. [50] studied the problem

Algorithm 2: Modified Prim's Algorithm

```
Input  : Graph $G = (V, E)$, threshold $\theta$
Output: Spanning Tree $T = (V_T, E_T)$
Let $X$ be the version set of current spanning tree $T$; Initially $T = \emptyset, X = \emptyset$;
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 $V_i$; Initially
  $\forall i \neq 0, d(V_0) = l(V_0) = 0, d(V_i) = l(V_i) = \infty$ ;
Enqueue $< V_0, (l(V_0), d(V_0)) >$ into priority queue $PQ$;
($PQ$ is sorted by $l(v_i)$);
while $PQ \neq \emptyset$ do
    $< V_i, (l(V_i), d(V_i)) > \leftarrow$ top($PQ$), dequeue($PQ$);
    $T = T \cup < V_i, p(V_i) >$, $X = X \cup V_i$;
    for $V_j \in$ ($V_i$'s neighbors in $G$) do
        if $V_j \in X$ then
            if $(\Phi_{i,j} + d(V_i)) \leq d(V_j)$ and $\Delta_{i,j} \leq l(V_j)$ then
                $T = T - < V_j, p(V_j) >$;
                $p(V_j) = V_i$;
                $T = T \cup < V_j, p(V_j) > d(V_j) \leftarrow \Phi_{i,j} + d(V_i)$;
                $l(V_j) \leftarrow \Delta_{i,j}$;
            end
        end
        else
            if $(\Phi_{i,j} + d(V_i)) \leq \theta$ and $\Delta_{i,j} \leq l(V_j)$ then
                $d(V_j) \leftarrow \Phi_{i,j} + d(V_i)$;
                $l(V_j) \leftarrow \Delta_{i,j}$; $p(V_j) = V_i$;
                enqueue(or update) $< V_j, (l(V_j), d(V_j)) >$ in $PQ$;
            end
        end
    end
end
```

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 6. *Figure 3.8 (a) is the minimum spanning tree (MST) rooted at node* V_0 *of* G *in Figure 3.6. 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 3.8 (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 3.8 (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 [50]

```
Input   : Graph $G = (V, E)$, $MST$, $SP$
Output: Spanning Tree $T = (V_T, E_T)$
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 $T$.
while DFS traversal on $MST$ do
    $(V_i, V_j) \leftarrow$ the edge currently in traversal;
    if $d(V_j) > d(V_i) + e_{i,j}$ then
        $d(V_j) \leftarrow (d(V_i) + e_{i,j})$;
        $p(V_j) \leftarrow V_i$;
    end
    if $d(V_j) > \alpha * SP(V_0, V_j)$ then
        add shortest path $(V_0, V_j)$ into $T$;
        $d(V_j) \leftarrow SP(V_0, V_j)$;
        $p(V_j) \leftarrow V_0$;
    end
end
```

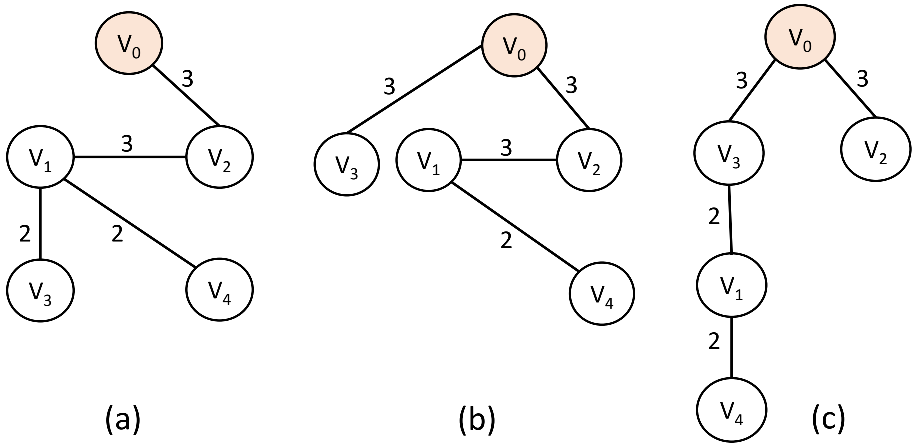


Figure 3.8: Illustration of LAST on Figure 3.6

3.3.4 Git Heuristic

This heuristic is an adaptation of the current heuristic used by Git and we refer to it as GitH. GitH uses two parameters: w (window size) and d (max depth).

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

Git 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. 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

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

of: `delta(B, O) / (max_depth - depth of B)` where `max_depth` is a parameter (default 50), and depth of B refers to the length of delta chain between a root and B.

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

- 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.

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

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

Figure 3.9: Dataset properties and distribution of delta sizes (each delta size scaled by the average version size in the dataset).

3.4 Experimental Evaluation

We have built a prototype version management system, that will serve as a foundation to DATAHUB [22], a system for facilitating collaborative data science. The system provides a subset of Git/SVN-like interface for dataset versioning. Users interact with the version management system in a client-server model over HTTP. The server is implemented in Java, and is responsible for storing the version history of the repository as well as the actual files in them. The client is implemented in Python and provides functionality to create (commit) and check out versions of datasets, and create and merge branches. Note that, unlike traditional VCS which make a best effort to perform automatic merges, in our system we let the user perform the merge and notify the system by creating a version with more than one parent.

Implementation. In the following sections, we present an extensive evaluation of our designed algorithms using a combination of synthetic and derived real-world datasets. Apart from implementing the algorithms described above, LMG and LAST require both SPT and MST as input. For both directed and undirected graphs, we use Dijkstra's algorithm to find the single-source shortest path tree (SPT). We use Prim's algorithm to find the minimum spanning tree for undirected graphs. For directed graphs, we use an implementation [12] of the Edmonds' algorithm [83] 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.

3.4.1 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 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

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

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 3.9):

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

- **Linear Chain (LC):** This dataset is based on a "mostly-linear" version history, i.e., number of branches is low, they occur after large intervals and have longer 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

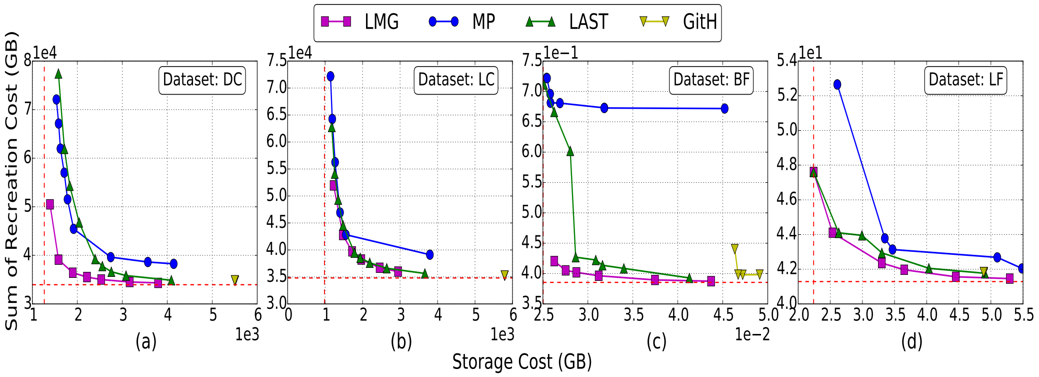


Figure 3.10: Results for the directed case, comparing the storage costs and total recreation costs

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 3.9.

3.4.2 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 the 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 3.10(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 3.9) as compared to just 34TB for the SPT solution (which is the minimum possible). As we can see from Figure 3.10(a), a space budget of 1.1× the

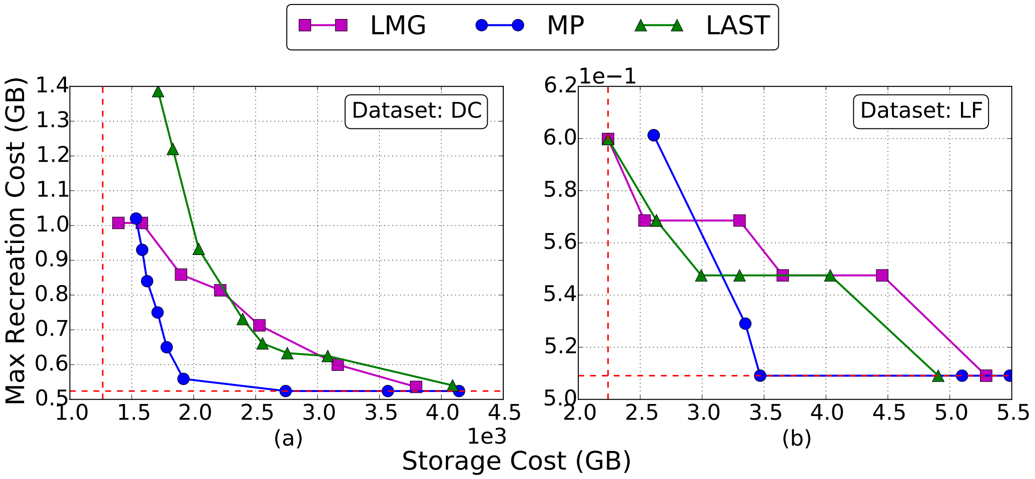


Figure 3.11: Results for the directed case, comparing the storage costs and maximum recreation costs

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 (Section 2), is that: it is possible to construct balanced trees where the sum of recreation costs can be reduced and brought close to that of SPT while using only a fraction of the space that SPT needs.**

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

In Figures 3.11(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.

Undirected Graphs. We test the three algorithms on the undirected versions of three of the datasets (Figure 3.12). For DC and LC, undirected deltas between pairs of versions were obtained by concatenating the two directional deltas; for the

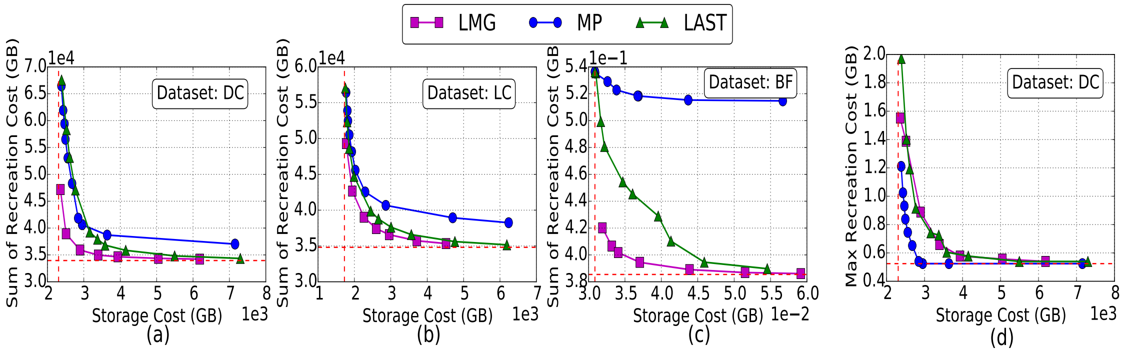


Figure 3.12: Results for the undirected case, comparing the storage costs and total recreation costs (a–c) or maximum recreation costs (d)

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 for brevity.

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

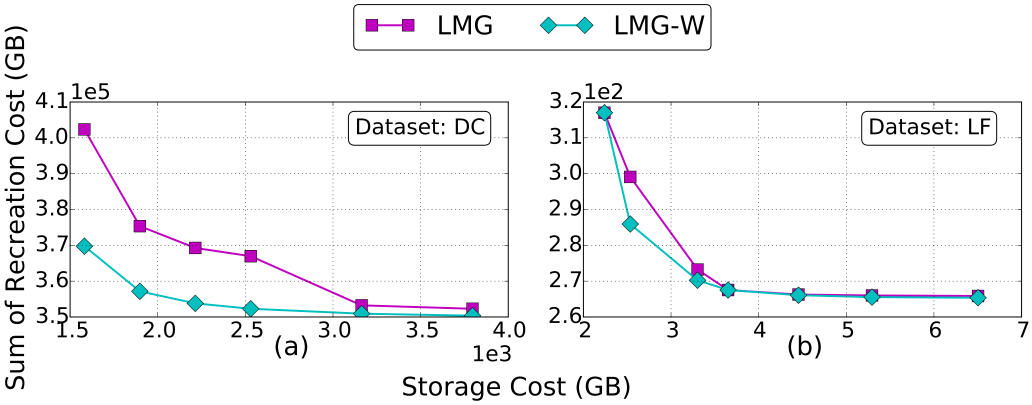


Figure 3.13: Taking workload into account leads to better solutions

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 3.13 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. 3.14, 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. 3.9) and thus requires more edges from SPT to satisfy the storage budget.

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

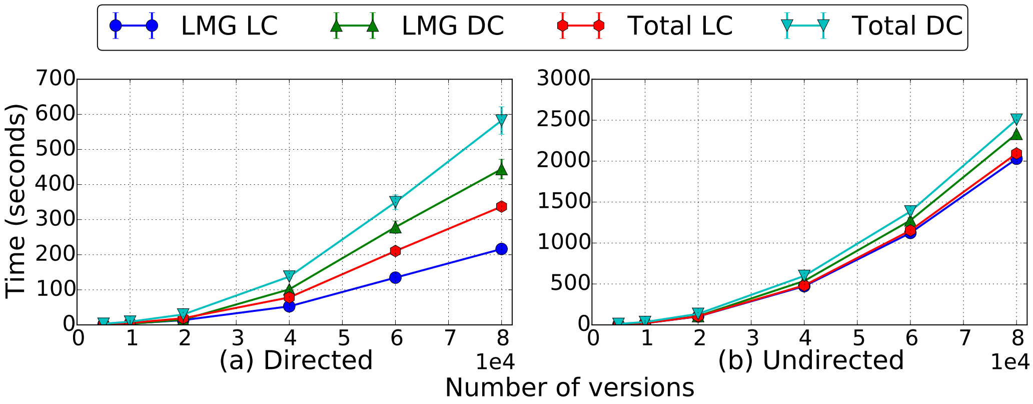


Figure 3.14: Running times of LMG

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.

3.5 Conclusion

In this chapter, we use delta compression to store unstructured datasets compactly due to the high overlap and duplication among the datasets, where some datasets or versions are stored as modifications from other datasets. Such delta compression however leads to higher latencies while retrieving specific datasets. We studied the trade-off between the storage and recreation costs in a principled manner, by formulating several optimization problems that trade off these two in different ways. We also presented several efficient algorithms that are effective at exploring this trade-off. Using extensive experimental evaluation we show that our techniques not only outperform existing version control systems but also capable of handing large problem sizes.

Chapter 4: PSTORE: Storage Layouts for Array Datasets

As described in the Section 1.3.2, the techniques developed in Chapter 3 are not suitable for answering range queries that require fine grained access to the array cells. In this chapter, we present our PSTORE framework for managing and querying large volumes of array datasets. PSTORE is an end-to-end framework containing two modules, namely, 1) a data ingestion module and 2) a query processing module for managing array datasets. We begin with a description of the data and the query model, followed by the detailed description of the two modules. The data ingestion module may execute in parallel with the simulation that is generating the data, and performs both partitioning and compression of the data. The data generated by the simulation is transferred to the data ingestion module one *snapshot* at a time, as the snapshots are progressively generated (a snapshot typically corresponds to a time step in a physical simulation) which are subsequently chunked and compressed. Thereafter, the data ingestion module sends the compressed data chunks to the storage system as they are produced, and may have to wait to gather enough data before compression. The compressed chunks are also inserted into an index structure for querying the data efficiently. The query processing module runs offline in a separate computing environment decoupled from the data ingestion module.

4.1 Data and Query Model

Data Model. The primary unit of storage and retrieval in PSTORE is a cell in a multi-dimensional array snapshot. Arrays are homogenous entities, each cell in an array stores data of the same type. For ease of explanation, we assume the array dataset stores the result of a single simulation variable. Each multi-dimensional array snapshot, storing the result of a given simulation, usually has the same dimensions. Therefore, every snapshot corresponding to that particular simulation variable occupies the same size. Every array cell can be uniquely identified by the spatial and temporal coordinates (or version-id). Since these arrays are the result of a simulation, therefore, the resulting version graph is a linear chain.

Query Model. PSTORE primarily supports array *slicing* queries; given range of spatial and temporal coordinates, the query returns a hyper-rectangle from one or a collection of array versions. Floating point data is the most prevalent type of data generated by scientific simulations. Further, not all applications require full precision of data or are able to tolerate reduced precision. Therefore, given a relative error bound, and the range of spatial and temporal coordinates, the query returns the appropriate number of bytes of a floating point data, that *approximates* the actual value of the data.

4.2 Compression Schemes

We describe the different compression schemes supported by our framework. Unlike most prior work (e.g.,[78]), we do not rely on a single compression scheme, and instead determine the compression scheme based on the structure of the data, or the requirements of the user. The data to be compressed is analyzed in an offline mode to determine the best scheme, in accordance with the user preference or the overall efficiency (based on both compression ratio and compression/decompression time). The simulation data that is currently compressed is analyzed periodically to ensure that we are still using the best compression scheme even when the distribution of the simulation data changes. We now discuss the different compression schemes implemented in PSTORE.

Bytewise Compression (bwc): This compression technique is similar to the one described by Schendel et al. in a recent work [73]. Data is partitioned into columns of bytes and the compressible bytes are identified by computing the byte value frequency distribution. The compressible bytes are then compressed using a backend compression algorithm such as `zlib` [18] or `lzo` [11]. We use `zlib` to obtain better compression ratios, and `lzo` to achieve faster compression and decompression rates as required. However unlike in [73], we do not compress and store all the compressible bytes together, as that adds considerable overhead to reconstruct the data due to required reshuffling of the bytes. Further, for approximate query processing where only a contiguous subset of the bytes are required, it is wasteful to decompress those bytes that are not required for answering the given query. As an exception to this strategy, only the two most significant bytes of a variable are stored together regardless of whether both are compressible or not. The reasoning behind this exception will be discussed in Section 4.3.2.

Bytewise-XOR Compression (bwcXOR): For scientific simulation datasets, there is a high degree of spatial and/or temporal correlation between neighboring array elements. As a result, the magnitude of the difference between two adjacent spatial or temporal data elements in the dataset may be small. Determining such correlations is essentially a preprocessing step towards better compressibility. The method for computing the difference between variables that we choose is XOR, rather than subtraction, as it yields a higher compression ratio both with two's complement integer and sign-magnitude floating point number representations [32]. After the data is partitioned into columns of bytes and the compressible byte columns are identified, we apply the XOR operation between the bytes of two different variables located spatially or temporally adjacent to each other. However, the XOR operation is not applied between incompressible byte columns as those are highly entropic and the resultant XOR value between two entropic bytes is also highly entropic. This is especially true for lower order bytes in a floating point variable where the bit randomness is often very high. The XORed byte columns are then compressed using one of the backend compression algorithms. All the snapshots except the last snapshot in a given data chunk are XORed in a temporal XOR operation. A spatial XOR operation can be applied on the last snapshot to further enhance the overall compression or it may be left unaltered, depending on the degree of spatial correlation between the variables and/or the amount of compression/throughput required. This snapshot is used for retrieving the prior snapshots by applying the XOR operation in the reverse direction, a process referred to as *unrolling* hereafter. For retrieving the first snapshot, all prior snapshots needs to unrolled first and hence this technique comes with an overhead due to this unrolling process. We have designed this compression technique as an alternate compression scheme compared to the ISOBAR technique [73].

FPC: FPC is a compression technique developed for compressing 64-bit floating point data [32]. FPC predicts values by sequentially using two predictors (`fcm` [72] and `dfcm` [44]) and selects the value closer to the actual value. Thereafter, FPC performs an XOR between the two values and encodes the leading zero bytes of the result using three bits. The scheme uses an additional bit to specify which of the two predictors was used for prediction. The resulting 4-bit code and the non-zero residual bytes are written to the output. We observe that in the context of climate or simulation datasets, the value from the predictor in FPC can be replaced with an adjacent spatial or temporal value (due to the high degree of correlation between neighboring elements in these datasets). The removal of the predictor from the FPC algorithm speeds up its execution. In addition, the single bit of storage which is needed by the predictor is no longer required. We also extend FPC for single precision floating point data as well. For 32-bit floating point values, we assign two bits for counting the leading zero bytes, although there are five different possibilities (0 to 4). In the context of our climate datasets, we have observed that the count of four leading zeroes occurs least frequently. As a result, all XOR results with four leading zero bytes are treated the same as values with only three leading zero bytes, with the fourth zero byte emitted as part of the output. Our framework supports two different versions of FPC, sFPC and tFPC to denote XORing along spatial and temporal dimensions, respectively.

Other schemes: In addition to the compression schemes described above, our framework also implements the naive compression algorithms that apply zlib or lzo over the data. As an alternative to this approach, an XOR of the variables along the spatial or temporal dimensions can be performed followed by the application of one of the backend compression algorithms.

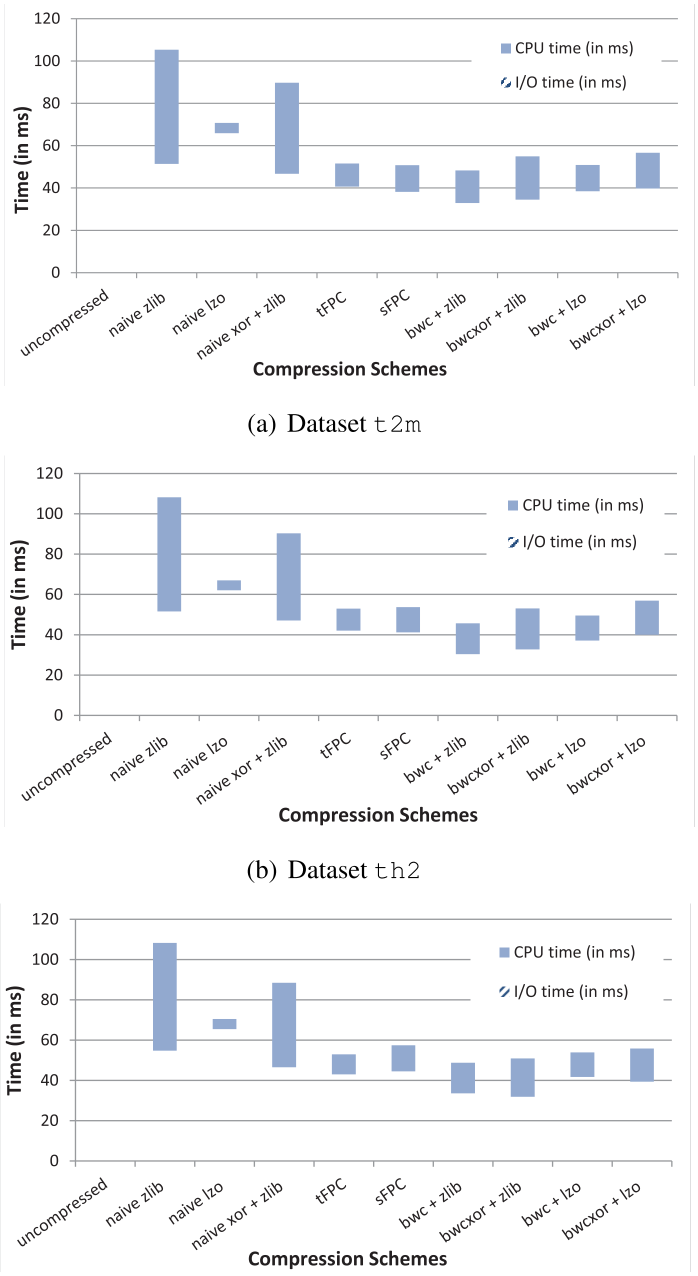
Compression Method	**t2m**		**th2**		**psfc**	
	CR	% gain	CR	% gain	CR	% gain
naive zlib	1.325	0.00	1.305	0.00	1.279	0.00
naive lzo	0.996	-24.83	0.996	-23.68	0.996	-22.12
naivexor+zlib	1.375	3.77	1.371	5.05	1.474	15.25
tFPC	1.510	13.96	1.509	15.63	1.682	31.51
sFPC	1.615	21.89	1.631	24.98	1.529	19.55
bwc+zlib	**1.823**	**37.58**	**1.833**	**40.46**	1.823	42.53
bwcxor+zlib	1.798	35.70	1.804	38.24	**1.880**	**46.99**
bwc+lzo	1.686	27.25	1.696	29.96	1.643	28.46
bwcxor+lzo	1.669	25.96	1.676	28.43	1.776	38.86

Table 4.1: Performance comparison of compression ratios between different compression schemes (compression ratio (CR) and % improvement relative to zlib (% gain) for each dataset). The best scheme for each dataset is highlighted.

Experimental comparison and discussion: Table 4.1 presents compression ratios and their percentage improvement over the naive approach for three different datasets `t2m`, `th2` and `psfc`. The data is obtained from a simulation of the CWRF climate model [4]. A detailed description of the data is provided in Section 4.5.

First, we consider the use of a difference operator (i.e., XOR) as a pre-processing step. The use of the difference operator between correlated data values for better compressibility has been advocated in [78] for compressing snapshots in a scientific database system. They propose the use of variable-length delta encoding and subsequently using run-length encoding for compressing the bitmasks. The work targets compressing large number of zeroes and the small magnitude differences generated in the process of delta encoding. However, the technique may not be suitable for compressing floating point values, since taking the differences between two floating point values does not always result in small bit differences in values due to the way in which floating point numbers are actually represented. Further, the number of zeroes after delta encoding is comparatively small. We must keep in mind that a high degree of correlation between adjacent variables does not necessarily result in a zero in most cases for floating point numbers, due to the highly entropic low-order mantissa bits. We note from Table 4.1 that even selectively applying a difference operator between compressible bytes (*bwcXOR*) does not always turn out to be profitable when measuring compression ratio. The compression ratios of *bwc* for the datasets `t2m` and `th2` are better than *bwcXOR*. In addition, the decompression cost for schemes employing a difference operator for enhancing compressibility is higher than those without them. However, we observe that *bwcXOR* outperforms *bwc* for the dataset `psfc` and therefore justifies the inclusion of *bwcXOR* in the suite of compression schemes in the framework. Among all the compression schemes, the *bwc* schemes with or without the difference operator turn out to be the best for floating point data when measuring compression ratios. Our primary intent in this analysis is to establish that the use of the difference operator in scientific datasets, especially for datasets having a high degree of correlation, may not always turn out to be beneficial in terms of improving the compression ratio. Therefore care must to taken to choose the compression scheme selectively, rather than relying on a single compression scheme.

Better compression ratios for the data especially in a setting that involves huge amounts of simulation data being generated is an absolute necessity. In addition to reducing storage requirements, better compression also reduces the bandwidth requirement for transmitting the data to storage, or put another way, enables the data to be delivered faster from the generation site to the storage nodes. However, the stored data needs to be analyzed (or queried) later, hence efficient retrieval of the



(a) Dataset `t2m`

(b) Dataset `th2`

(c) Dataset `psfc`

Figure 4.1: Total execution time for data retrieval with different compression schemes, for three datasets. The datasets are described in Section 4.5.

data is as important as developing better compression techniques for storage efficiency. Therefore we also need to incorporate query response time while choosing a compression technique for a given dataset. In Figure 4.1, we show the total execution time, which includes the I/O time (time taken to retrieve the data from the disk) and the CPU time (which includes the time to decompress the data, unroll the data when we perform an XOR operation and reshuffle the data for the *bwc*) for different compression schemes that PSTORE supports. We also demonstrate the I/O time for uncompressed data; the CPU time is zero in this case as we do not compress or perform an XOR on it. Since compression efficiency is dependent on the input data size, usually with larger data chunks resulting in better compression ratios, we perform the experiments on a data chunk size of around 3 MB. We determined this value empirically by experimenting with the same datasets that we use in the current experiment and observe the 3 MB value to be similar to that determined in previous studies [46, 73, 85]. We observe that disk I/O time constitutes a small percentage of the overall execution time and as a result compression ratio plays only a small role in reducing the overhead of data retrieval. Instead we pay a high price for decompressing the data and therefore care should be taken in choosing the appropriate compression scheme if compression ratio is not the only priority. We note that for `t2m` the disk space savings due to *bwc* + `zlib` (compression scheme with the highest compression ratio) is 45.15% compared to 40.69% when we use lzo in combination with *bwc* while the latter scheme performs decompression faster. Therefore an application that does not desire much space savings but does require fast query response time might want to select the latter scheme. The best compression scheme for the psfc dataset results in 46.81% disk savings compared to 39.14% when *bwc*+lzo is used. However the overall execution time for *bwcXOR*+lzo is around 2.5× higher than that of the other scheme due to cost of the unrolling operation. This implies that although using XOR may lead to higher compression ratios, there is a heavy price to pay when querying data compressed by this scheme. We emphasize that although lzo may not be comparable to zlib in terms of compression efficiency, it still proves to be a useful backend compression scheme when query throughput is important.

4.3 Data Partitioning

In this section, we describe the different data partitioning techniques that are employed in our PSTORE framework. We first describe partitioning along dimensions (which include both spatial and temporal dimensions) followed by bytewise partitioning. The former partitioning technique alleviates dimension dependency whereas the latter is useful for achieving better compression ratios and for answering certain types of queries.

4.3.1 Partitioning along Dimensions

The multidimensional data is partitioned (or *chunked*) regularly across both temporal and spatial dimensions, where all partitions are assigned an almost equal number of elements. Regular chunking of multidimensional data has been shown to be an effective partitioning technique for many types of array operations [79]. If available, we use workload information to choose the chunk size and shape; such workload-aware chunking can lead to significant speedups for range queries [71]. Sarawagi et al. [71] showed that the average number of block fetches for a given access pattern can be minimized by choosing the shape of a chunk appropriately.

A block B is defined as the unit of transfer used by the file system for data movement to and from the storage device. The shape of a chunk is specified by the tuple $(c_1, c_2, \ldots, c_n)$, where c_i is the length of the ith dimension of the multidimensional chunk. A probability is assigned to each query access pattern independent of the actual position of occurrence in the array and the positions are assumed to be uniformly distributed across the entire domain. Therefore access patterns can be represented as $\{(P_i, s_{i1}, s_{i2}, \ldots, s_{in}) : 1 \leq i \leq k\}$ where k is the number of different classes of queries and P_i is the probability of occurrence of the

ith class. Queries in this case are specified by an n-dimensional hypercube with only lengths of accesses in each dimension. The problem with the formulation for average number of block fetches by Sarawagi et al. [71] is that in some query instances the computed number of blocks to be fetched is exactly one less than the actual number of blocks to be fetched. That error is amplified due to the multiplication of factors across dimensions. Thus the error is significant if it is made for the majority of the dimensions of a given class. Otoo et al. [63] therefore modify the objective function as follows:

$$\sum_{i=1}^{k} P_i \prod_{j=1}^{n} \left(\frac{s_{ij} - 1}{c_j} + 1 \right) \tag{4.1}$$

In order to minimize the amount of additional data fetched from the disk, the chunk shape must satisfy the constraint:

$$\prod_{i=1}^{n} c_i \leq B \tag{4.2}$$

The goal in that prior work was to choose the chunk shape satisfying Eq. 4.2 that minimizes Eq. 4.1.

However, we note that the effect of compression has not been taken into account in earlier work, in computing the optimal chunk shape for array storage. We want to compress the data before it is stored in secondary storage to reduce the storage footprint on disk and also to maximize the disk bandwidth utilization. If the data has to be transferred over a network to/from the storage nodes, compression helps to reduce the transfer time as well. The compression ratio of standard compression algorithms like `zlib` also starts to degrade if the chunks contain too few bytes; let us denote such a threshold by B_c. This threshold may be different for different types of data and different algorithms, but can be easily learned given a sample dataset and an algorithm. Above this threshold, the compression ratio usually stabilizes to a fixed ratio ρ. To guarantee a good compression ratio, we place the following constraint on the chunk shape:

$$\prod_{i=1}^{n} c_i \geq B_c \tag{4.3}$$

At the same time, we want the compressed chunk to fit within a multiple of the disk block size B:

$$\frac{\prod_{i=1}^{n} c_i}{\rho} \leq mB \tag{4.4}$$

The threshold B_c is usually a multiple of the block size B and thus $B_c \leq m'B$

where m, m' are positive integers. We then have the following relation:

$$m'B \leq \prod_{i=1}^{n} c_i \leq m\rho B \tag{4.5}$$

where $m' \leq m$. Our intent here is to show how the constraints can be modified to incorporate the effect of compression into the optimization process. It is then not difficult to compute the optimal dimensions using the procedure outlined in [63].

The bytewise precision partitioning technique results in a compressed and an uncompressed data chunk corresponding to the compressible and incompressible bytes in a floating point data set. For best performance, this implies that there should be two different chunking strategies, one for the compressed bytes and another for the uncompressed ones, tuned according to the query workload. However, supporting two different chunking strategies would require maintaining two separate index structures which might prove to be a costly overhead. Another problem with this approach is that it requires two separate chunking strategies to chunk two different data representations. As a result, the number of snapshots in a chunk for the compressed data may be different from that required for the uncompressed data, which would require buffering and could slow down the overall chunking process. For these reasons, we do not chunk the data differently in the current implementation, nevertheless it would be an interesting study for future work to compare performance between these two alternatives.

We must keep in mind that with larger chunk sizes there is a higher price for decompression. Instead we can extend the idea of a *two-level chunking scheme* [79] by not compressing the entire chunk, but first partitioning into sub-chunks and then compressing the sub-chunks separately. We write the entire chunk to the disk. With this design, we do not have to decompress the entire chunk but only those sub-chunks that are required to answer the query. With this design, we just need to ensure that the sub-chunks satisfy Eq. 4.3 and the whole chunk satisfies Eq. 4.4, maximizing both compression ratio and I/O performance.

4.3.2 Bytewise Partitioning

In many applications, the full precision of the data may not be needed. For example, the lower order mantissa bits of a floating point number may be truncated during some types of visualization applications, since the human eye may not be capable of perceiving such fine differences. The IEEE 754 standard [19] for floating point arithmetic represents single precision values as a single sign bit, 8 exponent bits and 23 mantissa bits. Representing double precision values requires a single sign bit, 11 exponent bits and 52 mantissa bits. The mantissa bits in a floating point number represent a fractional component in the overall value and the lower order bits each contribute an exponentially smaller value to the overall magnitude of the number as we move from the higher to the lower order bits. This implies that discarding the lower order bytes in a floating point number (where the mantissa is stored) introduces less error compared to discarding higher order bytes. In contrast, truncation by discarding lower order bytes is not a feasible option for integer data due to the loss of significant bits that is not mitigated by multiplication by an exponent as for floating point data [48].

Table 4.2 presents the maximum relative error produced based on the number of mantissa bits retained for both single and double precision floating point data. Due to the small maximum relative errors introduced due to truncation of mantissa bits, it might suffice for applications to only retrieve the higher order k bytes corresponding to the amount of error the application can tolerate. This format of data access also requires partitioning the values in bytewise fashion. Therefore bytewise partitioning of values serves the dual purpose of enabling precision level partitioning and enhancing the compressibility of the data. While partitioning the data bytewise we always store the higher order two bytes together as they contain the sign and the exponent bits and truncation of exponent bits would introduce unacceptably high error rates. Also due to the expected high degree of correlation in the data, the higher order bits (which include the sign, exponent and higher order mantissa bits) of adjacent variables are likely to be similar. As the exponent bits in both single and double precision numbers span the first two higher order bytes, it is beneficial to store them together to enhance the compressibility of the data. However, partitioning the numbers bytewise requires reconstructing them when retrieving the data, which will introduce some overhead. When partial precision data is retrieved, the missing bytes are replaced with a fixed pattern as defined by the user.

4.4 Query Processing

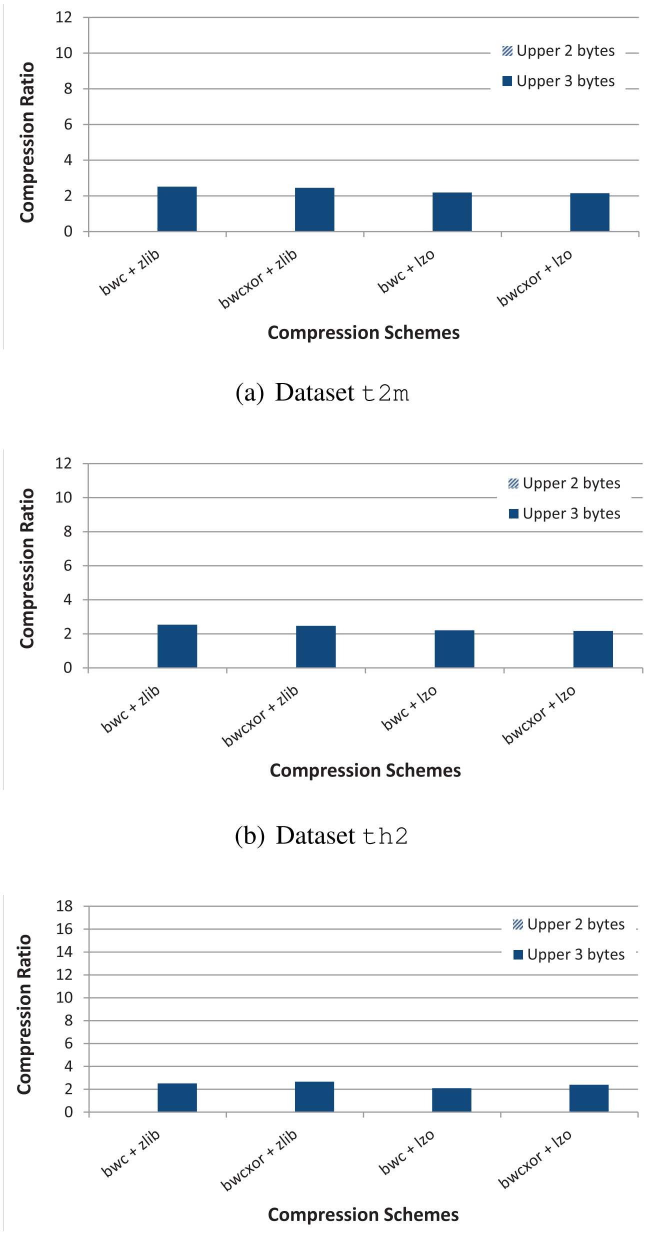
We support range queries in the query processing module of PSTORE. For retrieving the data for a range selection query, PSTORE determines the overlapping chunks in the query range and locates the data chunks on disk using the index. After the chunks are retrieved from disk, overlap with the sub-chunks within every chunk is determined from the query region. The overlapping sub-chunks are then extracted by decompressing them (if they were compressed in the data ingestion state) after determining their location from the header information that was stored along with the each chunk.

Decompression is an expensive operation and results in CPU processing time that can be even more than the time for I/O operations. We reduce this overhead by parallelizing the decompression. Sub-chunk decompression can be parallelized as every sub-chunk can be decompressed independently once a chunk is retrieved from disk. However, this process does not always lead to a linear speedup with the number of CPUs available. This is because not all the sub-chunks need to be extracted from a chunk, as they may not overlap with the query region. Further, since the process of sub-chunk extraction is overlapped with the chunk retrieval from disk (as described in Section 4.4.2), the overall processing time decreases with increasing parallelism until the CPU processing time becomes equal to the I/O processing time.

Significant Bytes	**Max. Error% (SP)**	**Max. Error% (DP)**
2	2.6e-1	3.1e0
3	1.0e-3	1.2e-2
4	-	4.8e-5
5	-	1.9e-7
6	-	7.3e-10
7	-	2.8e-12

Table 4.2: Maximum relative error due to reduced precision of IEEE 754 single and double precision floating point numbers

For approximate query processing, the application retrieves fewer bytes from disk, since the target application is able to tolerate lower precision. So the overall processing time should be lower compared to retrieving all the bytes for each



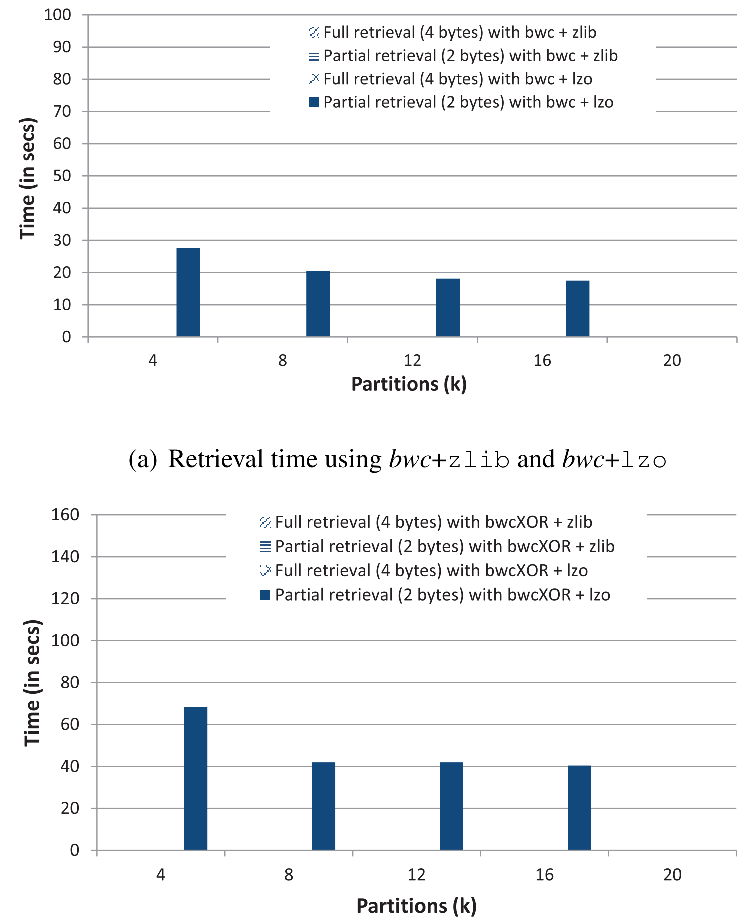
(a) Dataset `t2m`

(b) Dataset `th2`

(c) Dataset `psfc`

Figure 4.2: Compression Ratio for different values of byte-precision for single precision datasets

data element from disk. Moreover, for partial precision data retrieval, usually the higher order bytes for the data elements are retrieved, which are generally more compressible than the lower order bytes. As a result the I/O operations are less expensive than when retrieving all (or only lower order) bytes of the data elements.



(a) Retrieval time using *bwc*+`zlib` and *bwc*+`lzo`

(b) Retrieval time using *bwcXOR*+`zlib` and *bwcXOR*+`lzo`

Figure 4.3: Partial (upper 2 bytes) retrieval time vs. full (4 bytes) retrieval time for different compression schemes for dataset `T`

4.4.1 Two-level chunking with compression

Larger chunks help amortize disk seek overhead but pose a problem when the query region is a small subset of the chunk. We do not have random access to individual elements inside a chunk and therefore it is wasteful to process additional elements when the actual query region is a small contiguous fraction of the entire chunk. Smaller chunks however, reduce the average processing time for accessing an element in a chunk but introduces overheads from additional disk seeks. Two-level chunking seeks to balance the two factors. A larger chunk is split into regular sized sub-chunks so that the overall processing overhead is minimized. The larger chunks are the units of disk I/O while the smaller chunks form the unit of array processing. Two-level chunking has been studied previously [70, 75, 79], but without including the effects of compression. There are several strategies we may choose to apply when using two-level chunking with compression. One strategy is to compress the chunks before writing them to disk. The drawback of this approach is that in the case of range queries, it is seldom the case that one needs to access all the data elements contained in the larger chunk. As a result, we must pay the cost of decompression for the additional data that is not needed. As we observed previously, decompression is a relatively costly operation, therefore it might be beneficial to compress the smaller sub-chunks individually and then combine them into a chunk before writing to the disk. Two-level chunking also opens up parallelization opportunities by allowing us to process the sub-chunks in parallel and thus speed up query processing. In our implementation, we follow a regular chunking scheme for both chunks and sub-chunks, as this chunking strategy has been shown to yield the best performance compared to irregular chunking schemes [79] and determine the optimal chunking layout empirically.

4.4.2 Chunk prefetching

For long running range queries, where multiple chunks may be accessed, we can improve performance by hiding I/O latencies by overlapping them with CPU processing time. Since a chunk access is sequential, we want to process one chunk at a time and overlap the I/O access of the next chunk with CPU processing of the current chunk. With two-level chunking, we can further process the sub-chunks in an embarrassingly parallel fashion as all the computations related to each sub-chunk can be executed independently. This is advantageous because we require large chunks to amortize the I/O time and simultaneously the presence of a large number of sub-chunks in a chunk enables higher throughput.

4.5 Experimental Results

We evaluate the performance of the different components that constitute PSTORE. We use a real dataset for this purpose which is based on the Regional Earth System Model (RESM) to provide climate and environmental information for a wide range of end users with drastically different data demands. RESM is based on the state-of-the-art regional climate model CWRF [4] that predicts mesoscale climate processes, including atmosphere, hydrology, crop, soil, air and water quality and their interactions. The dataset contains numerous variables each of which records measurements of parameters such as temperature, pressure, relative humidity, wind velocity, actual biogas emissions, CO concentration, etc. The measurement is either performed on a region of space defined by a 2D grid or a 3D grid and is recorded periodically over a fixed time duration. Each such grid is termed a *snapshot* at a particular instance in time. We perform our evaluation on both 3D and 4D datasets, which are described below.

3D dataset: Each snapshot in this dataset is a 138×195 array recorded over a 3 hour interval, so there are 8 snapshots per day. Currently, we have simulation data for one month, which has a total 240 snapshots for all the variables. We used three variables (i) `t2m`: air temperature at 2m, (ii) `th2`: potential temperature at

2m, and (iii) psfc: surface pressure, for the purpose of experimentation with 3D datasets.

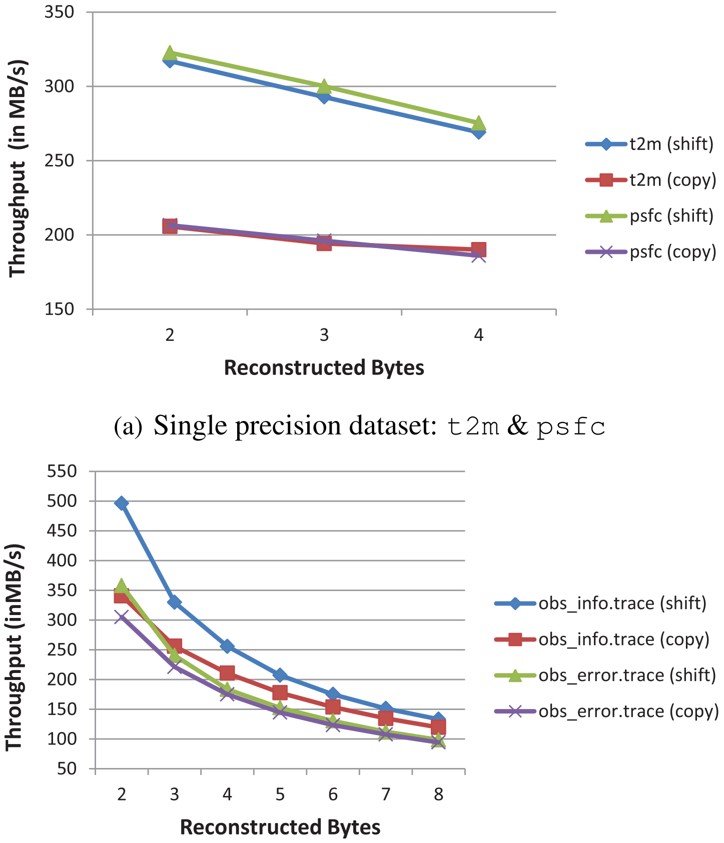
4D dataset: This dataset differs from the 3D dataset in that it has an added *height* dimension, which makes each snapshot a 138×195×35 matrix. We used two variables from this 4D dataset T and P which denotes perturbation temperature and perturbation pressure, respectively.

The total size of the dataset is around 140 GB and each measurement is stored as a single-precision floating point variable. The dataset is represented in the netCDF format [16].

We performed our experiments on an Intel Xeon E7450 (2.4 GHz). This machine has 4 sockets each having 6 cores with a total of 24 threads and a 48 GB main memory.

4.5.1 Effect of Bytewise Partitioning on Compression

We study the effect of bytewise partitioning on the compression ratio. While reading partial or reduced precision data from the disk, we always read the higher order bytes which contain the sign, exponent and a subset of the mantissa bits. For single precision data, we have two possibilities for reduced precision data, either 16 bits (upper 2 bytes) or 24 bits (upper 3 bytes). From Figure 4.2 we



(a) Single precision dataset: `t2m` & `psfc`

(b) Double precision dataset: `obs_info.trace` & `obs_error.trace`

Figure 4.4: Throughput of `shift` and `copy` reconstruction technique for (a) single and (b) double precision datasets

observe that for every compression scheme, the compression ratio of the higher order 2 bytes is best. This is because there is a high degree of correlation between the adjacent variables in this dataset and this correlation is captured best by the sign, exponent and the higher order mantissa bits, which causes these values to be very similar or equal for much of the data. We also observe that the compression ratio for 16-bit precision is almost 5× better than for 24-bit precision. The large difference can be explained by the fact that lower order mantissa bits are highly entropic, so are responsible for the decrease in compression ratio. Therefore, an application that can tolerate 0.26% relative error in precision (see Table 4.2) can achieve a large savings in disk space and faster query response times due to the high compressibility of the data at reduced precision.

Figure 4.3 shows query retrieval times for full and partial precision data for different partitioning parameters with different compression schemes. Figure 3(a) shows the retrieval times when the upper 2 bytes of data are compressed with *bwc*+`zlib` and *bwc*+`lzo`, and Figure 3(b) shows the retrieval times when XOR-ing had been applied during compression and unrolling must be done during query retrieval. The lower 2 bytes are not compressed because they are highly entropic. We observe that the partial query retrieval for *bwc*+`zlib` takes around 70% of the full query retrieval time across all partition configurations whereas the partial query retrieval time is 50% of the full query retrieval time for *bwc*+lzo, since the decompression time for *bwc*+zlib is higher than for *bwc*+lzo. From Figure 4.2 we see that the compression ratio for zlib is higher than for lzo. Moreover lzo fails to compress the data when $k = 20$, i.e., when sub-chunks are smaller, and therefore we have no data points at that value of k. Figure 3(b) shows that using XOR and unrolling introduces significant overhead in the retrieval process, which causes the partial query retrieval time to increase to around 80% and 70% of the full query retrieval time for *bwcXOR*+zlib and *bwcXOR*+lzo schemes, respectively.

However, bytewise partitioning introduces some overhead due to the reconstruction required to build a floating point number from its individual bytes. We study the overhead for two different types of reconstruction techniques, measuring overall throughput. The first technique is to reconstruct the floating-point number by byte shifting while the second technique copies the individual bytes to their respective offsets in memory to reconstruct the original number. Figure 4.4 presents the throughput obtained while reconstructing the bytes using both the shift and the copy method. We observe that the shift method outperforms the copy method in the throughput obtained. This is because the copy method involves moving many single or small groups of bytes (if subsequent bytes in a variable are kept together in a partition) to the target location requiring multiple byte copy operations with each copy operation associated with some fixed overhead. In case of byte shift operation the entire data element (4 or 8 bytes) is constructed in-place by byte shifting (which is a cheap operation) and then assigned to the target variable requiring a single move operation. As previously noted, we reconstruct at least the most significant two bytes for both single and double precision data. The throughput gain is at least 54% and 17% when reconstructing the two most significant bytes, for single and double precision data respectively. We also note that, not surprisingly, the throughput gain decreases with an increase in the number of bytes reconstructed.

4.5.2 Two-level Chunking

We demonstrate experimentally the variation in query response time as the number of sub-chunks and chunks for a given variable in the dataset is varied. We also show that our proposed approach to two-level chunking with compression outperforms single-level chunking with compression. The experiments are performed on both 3D and 4D datasets.

We compare single-level (1L) chunking to two-level (2L) chunking in the context of compression. We measure the performance of slicing queries for 1000

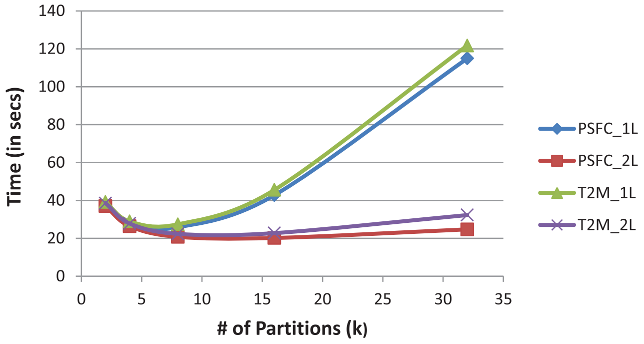
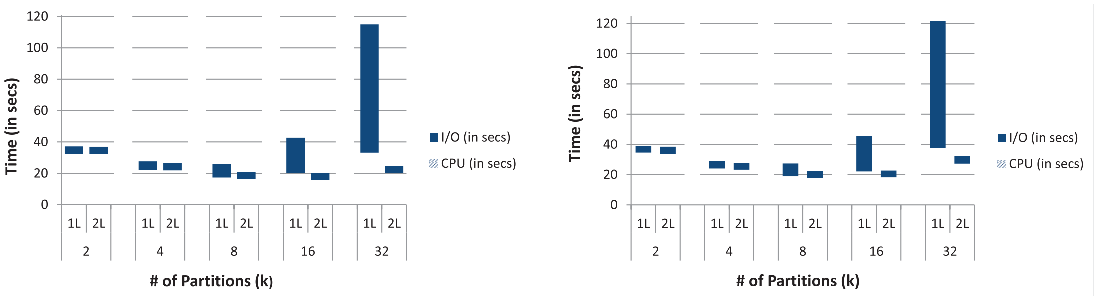


Figure 4.5: Performance of array slicing queries for single-level and two-level chunking.

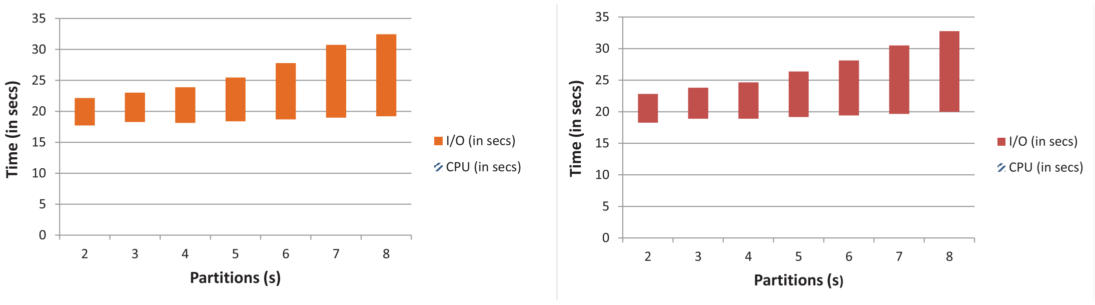


(a) 3D dataset `t2m`

(b) 3D dataset `psfc`

Figure 4.6: Total execution time (CPU + I/O) for array slicing queries for single-level (1L) and two-level (2L) chunking with different partition numbers on two different datasets

queries generated uniformly at random for a 3D dataset. For experiments involving 4D datasets, we restrict the number of such queries to 100, as this dataset is bigger than the 3D dataset by around an order of magnitude. The 3D dataset has two spatial dimensions while the 4D dataset has three of them, with each dataset having a temporal dimension denoting the timestamps in which the simulations were run. In the experiments, each spatial dimension is k-way partitioned, i.e.



(a) Dataset t2m: $k = 8$, $t = 16$

(b) Dataset t2m: $k = 16$, $t = 16$

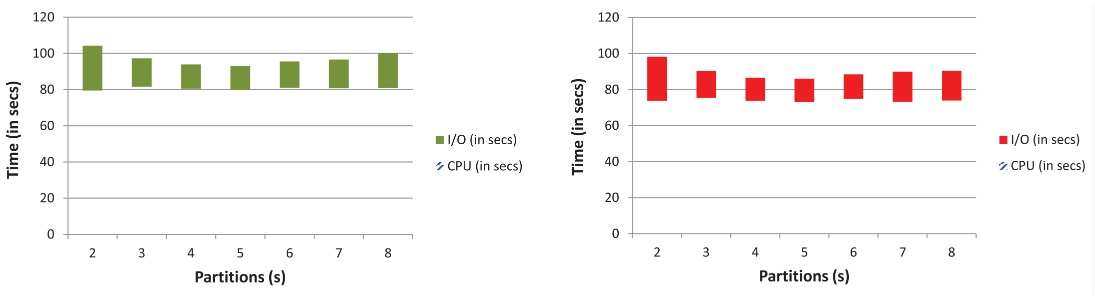
Figure 4.7: Query performance (CPU + I/O time) varying chunk size, for the 3D dataset t2m

each dimension is divided into k partitions. Therefore each snapshot has k^d sub-chunks, where d is the number of spatial dimensions. For example, when $k = 4$ and $d = 2$, 16×16 snapshot is partitioned into 16 (4^2) sub-chunks, where each sub-chunk is stored in a separate file. The value t gives the number of temporal dimensions or snapshots that will be included in each file. If $t = 16$, then each file has $4 \times 4 \times 16$ elements, which we refer to as single-level chunking. This configuration can also be denoted by the tuple (k, t). For two-level chunking, we specify an additional parameter s, which denotes the number of chunks to partition the data into. In other words, it also indicates the number of sub-chunks that would be allowed in a chunk. The 16 sub-chunks that were created by 4-way partitioning before, can be viewed as a 4×4 array of sub-chunks. In a similar way, each dimension in this array is now s-way partitioned. For $s = 2$, the 4×4 array is partitioned into 4 (2×2) chunks and each chunk is stored in a separate file.

We denote a two-level partitioning by the tuple (k, t, s). Although each spatial dimension is partitioned equally into k or s parts, our framework supports unequal partitioning across different dimensions as well.

Figure 4.6 shows a breakdown into CPU time and I/O time for the 3D datasets `t2m` and `psfc`. For 3D dataset, the chunk is always 2-way partitioned ($s = 2$) and the number of snapshots in a chunk is 16. From Figure 4.5, we observe that the performance of the 1L and 2L chunking strategies is similar for smaller number of partitions. Since the number of partitions is small, the number of chunks to be written to disk is small, resulting in fewer disk seeks. However the performance of the 1L strategy degrades with an increase in the number of partitions. This is because of the increase in disk seeks for the 1L strategy, whereas the number of disk seeks remains almost constant for all the configurations for the 2L strategy. This is due to the use of full chunks as the unit of disk access for the 2L strategy. The best performance for the set of partitions selected is achieved for the (8, 16, 2) and (16, 16, 2) configurations for the 3D dataset, as can be observed from Figure 4.5. Increasing the number of sub-chunks beyond 16 decreases the performance of the 2L strategy. The decreased performance comes from an increase in CPU processing time, due to the overhead of decompressing a large number of relatively small sub-chunks.

We study the effect of variation in chunk size in Figure 4.7 and Figure 4.8 by fixing k and t. We observe that CPU time remains almost unaffected by the change in the chunking parameters, confirming that using chunk as the unit of disk I/O does not affect CPU time significantly. For the 3D dataset, $s = 2$ turns out to be the optimal choice of chunk partition.

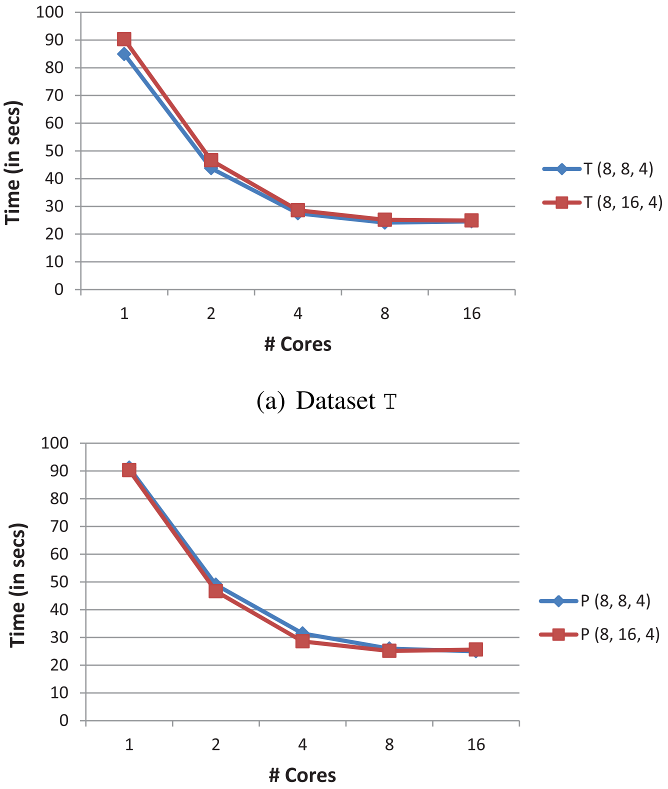


(a) Dataset T: $k = 8, t = 16$

(b) Dataset T: $k = 16, t = 16$

Figure 4.8: Query performance (CPU + I/O time) varying chunk size for the 4D dataset T

This value of s is best for different values of k and t, as can be seen from Figure 7(a) and 7(b). Figure 8(a) and 8(b) show the variation of s for the 4D dataset. In this case, $s = 5$ is the optimal choice for chunk partition. In general, the optimal value of s for a given dataset can be found by analyzing a sample of the dataset in the pre-processing step.



(a) Dataset T

(b) Dataset P

Figure 4.9: Strong scalability of the query retrieval framework for 4D datasets for different partition configurations

4.5.3 Parallelizing Query Retrieval

In a two-level chunking scheme, the sub-chunks can be processed in parallel once the chunks intersecting with the query region are retrieved from disk. However, these two processes are performed in a pipelined fashion using double buffering by default. To see this, we executed the query framework with partition parameters (8, 8, 4) and (8, 16, 4), since those configurations were the best parameters for the 4D dataset for the given query workload, determined empirically. We observe from Figure 4.9 that the framework does not scale beyond 8 cores/threads for these queries. As the number of core increases, the compute time decreases and becomes equal to the I/O time, which remains fixed irrespective of the increase in the number of cores. The application cannot perform better once the I/O time becomes greater than or equal to the compute time for the queries. For the parallelization process, we assign each compute thread a sub-chunk to perform the decompression process in parallel, to remove the performance bottleneck of the expensive decompression operations. However, we note that there are still load balance issues that limit performance even if we assign n threads evenly to the k^d sub-chunks ($n \leq k^d, d = 3$ in this case). It is likely that at some times in executing the queries, the number of sub-chunks to be processed will be less than n. This is because a query region might intersect with too few sub-chunks in some cases, or that different sub-chunks take different amounts of time to process. Load imbalance is therefore another reason why the application may not scale linearly with increasing number of threads, if overall performance is limited by CPU computations rather than I/O time.

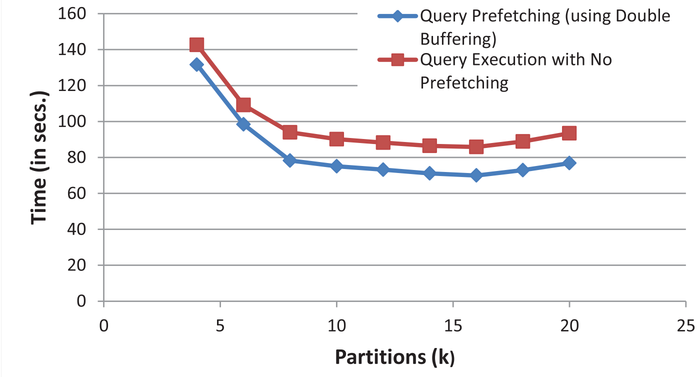


Figure 4.10: Effect of prefetching and double buffering on query performance with different partition configuration (k) on the 4D dataset P

4.5.4 Chunk Prefetching

Figure 4.10 shows the reduction in query execution time due to chunk prefetching. Chunk prefetching is achieved by using double-buffering in memory with the chunk to be processed next prefetched and stored into memory while processing for the current chunk takes place from a different buffer. Additional buffers could be employed if more than one disk is available. To perform the prefetching, we use two separate types of threads; several dedicated solely to CPU processing (one per available CPU/core) while the other is dedicated to disk I/O. This straightforward optimization hides the disk access time behind the processing time, as long as the processing time dominates the I/O time, as we have seen is true in most cases we have experimented with. The current experiment is performed on the 4D dataset where the partition parameter k is varied from 4 to 20 in steps of 4, fixing t and s at 16 and 4, respectively. From Figure 4.10 we observe that it is possible to completely overlap the disk access time with the CPU processing time via chunk prefetching.

4.6 Conclusion

In this chapter, we presented the design, implementation, and evaluation of PSTORE, a no-overwrite storage manager for managing array data generated during scientific simulations. The data ingestion module in PSTORE contains a suite of compression techniques designed to handle diverse types of floating point datasets generated during scientific simulations, thereby permitting applications the flexibility to choose the most appropriate compression technique. PSTORE also supports approximate query processing by retrieving partial precision data if that is sufficient for the application needs, and contains several other optimizations for efficient query execution. Our extensive experimental evaluation illustrates that different compression techniques work better for different datasets, and further that using bytewise partitioning and two-level chunking can lead to significantly higher compression ratios and lower query execution times respectively.

Chapter 5: RSTORE: A System for Managing Datasets with Keyed Records

As demonstrated in the challenges in Section 1.2, the techniques meant for unstructured or array datasets are not suitable for datasets with keyed records. In this chapter, we describe RSTORE, a system aimed at managing datasets with keyed records and executing queries efficiently over the data. We start with a brief description of the underlying data model, followed by a description of the retrieval queries we aim to support. Thereafter, we describe briefly the individual system components to provide an overview of the overall system architecture along with a set of baselines and discuss the trade-offs under different settings. Next, we define the problem formally and discuss the algorithms for solving the problem and demonstrate the efficiency of the algorithms through extensive experiments.

5.1 Data and Query Model

Data Model. The primary unit of storage and retrieval in our system is a **record**, which may refer to a tuple/row in a tabular dataset, a JSON document in a document collection, or a time series. A record is considered to be *immutable*, and any change to it results in a new *version* of the record. We make no assumptions about the structure, type or the size of a record, except for assuming the existence of a **primary key**, denoted K_i, that can be used to uniquely identify a specific record within a collection of records. For simplicity, we assume there is a single such collection (also called a *dataset*) that the system needs to manage, that is being modified simultaneously by a team of users over time in a collaborative fashion, resulting in a set of **versions** over time. We assume there is a single *root* version of the dataset, from which all other versions are derived (an empty root version can be added to handle the scenario where there are multiple starting collections).

Let $V = \{V_0, V_1, \ldots, V_{n-1}\}$ denote the set of *versions* stored in the system; each version is identified uniquely by a **version-id** (either an auto-incremented value, or *hashes* as in *git*). A new version is derived from an existing version through an update operation or a transformation, that essentially boils down to modifying/deleting existing records and/or adding a new set of records. We denote the set of changes from V_i to V_j by $\Delta_{i,j}$ and refer to it as the *delta* from V_i to V_j. Note that in this case, $\Delta_{i,j}$ is symmetric, i.e., $\Delta_{i,j}$ may be used to derive V_i from V_j as well, thus making $\Delta_{i,j} = \Delta_{j,i}$. These derivations are encoded in the form of a directed *version graph*.

Composite Keys. Since a record may be unchanged from one version to the next, to be able to refer to a specific record within a specific version, we use a **composite key**: ⟨primarykey, version-id⟩, where the second part refers to the **version-id** of the version where the record was created. This allows us to uniquely reference records within a global address space. We chose to use **version-id** of the appropriate version instead of an auto-incremented value as the latter introduces additional synchronization overhead in a decentralized setting with no obvious benefits.

Query Model. In a collaborative setting with large datasets, the query workload may consist of a variety of queries, with differing characteristics, as shown in Section 1.1.

- *Record Retrieval:* Analogous to a key-value store, a user/application may want to retrieve a record with a specific primary key K from a specific version V. Note that we cannot simply retrieve the record with the composite key $\langle K, V \rangle$,

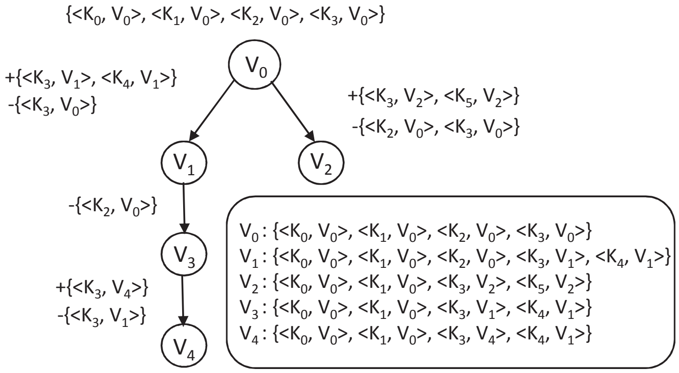

{<K0, V0>, <K1, V0>, <K2, V0>, <K3, V0>}
V0
+{<K3, V1>, <K4, V1>}
-{<K3, V0>}
+{<K3, V2>, <K5, V2>}
-{<K2, V0>, <K3, V0>}
V1
V2
-{<K2, V0>}
V3
+{<K3, V4>}
-{<K3, V1>}
V4
V0 : {<K0, V0>, <K1, V0>, <K2, V0>, <K3, V0>}
V1 : {<K0, V0>, <K1, V0>, <K2, V0>, <K3, V1>, <K4, V1>}
V2 : {<K0, V0>, <K1, V0>, <K3, V2>, <K5, V2>}
V3 : {<K0, V0>, <K1, V0>, <K3, V1>, <K4, V1>}
V4 : {<K0, V0>, <K1, V0>, <K3, V4>, <K4, V1>}

Figure 5.1: An Example Version Graph with 5 Versions

since the record may have originated in one of the predecessor versions to V. This, in fact, forms a major challenge in this setting.

- *Version Retrieval:* Analogous to typical VCS, here the goal is to retrieve the entire version given a version-id, i.e., all the records that belong to the version.
- *Range Retrieval:* This query enables retrieving a version partially, by specifying a range of primary keys and a version-id.
- *Record Evolution:* Finally, we may want to analyze the evolution of a record from its point of origin to the current state of the system. In other words, given a *primary key*, we want to find all the different records with that primary key across all versions.

Example 7. *Fig. 5.1 displays a version graph with five versions* V_0 $(root), V_1, V_2, V_3, V_4,$

with a total of nine distinct *records.*

We create composite keys for the records in V_0 by adding V_0 as the second component to the keys. V_1 is derived by modifying K_3 of V_0 and adding a new record $\langle K_4, V_1\rangle$. In this case $\Delta_{0,1} = \{+\langle K_3, V_1\rangle, +\langle K_4, V_1\rangle, -\langle K_3, V_0\rangle\}$. V_2 is derived from V_0 (and after V_1) by modifying K_3 as well, adding a new record $\langle K_5, V_2\rangle$, and deleting record $\langle K_2, V_0\rangle$. V_3 is derived next by deleting record $\langle K_2, V_0\rangle$ from V_1. Finally, V_4 is derived by modifying $\langle K_3, V_1\rangle$ of V_2. Note that the derived version forms the version identifier component in the composite key, which is also the version in which the particular record appears for the first time. To retrieve a specific record, say K_3 from version V_3, it is not sufficient to look for composite key $\langle K_3, V_3\rangle$ (which does not exist), rather, we need to maintain a version-to-record mapping (Fig. 5.1), that must be consulted to identify the composite key to be retrieved ($\langle K_3, V_1\rangle$ in this case).

5.2 Key Trade-Offs

We begin with a brief discussion of the key trade-offs in storing such versioned datasets in the cloud, and then evaluate 3 baseline options with respect to those trade-offs.

- **Storage and compression.** There are two considerations here. First, we would prefer to only store a single copy of a record that appears in multiple versions. This however complicates performance of full or partial version retrieval queries since the requisite records may be stored all over the place. Second, we would like RSTORE to handle records of varying sizes, from a few bytes to a few MBs. In the latter case, there may be small differences between two different versions of a record (e.g., only a single attribute may be updated in a large JSON document). One way to exploit this overlap is to store the two versions of the record together in a "compressed" fashion, with specific compression technique chosen according to the data properties (e.g., one may store "deltas" (differences) between the two records, or use an off-the-shelf compression tool that in effect does the same thing). Such compression, however, negatively impacts the query performance by restricting the data placement opportunities.

- **Query performance.** Different partitioning and layout schemes are appropriate for the different classes of queries listed above. Record evolution queries are best served by grouping together all the different records with the same primary key, whereas full version retrieval queries prefer grouping together all records that belong to the same version. A general-purpose system must offer knobs that allow adapting to a specific query workload.

- **Online updates.** Another important consideration is the ability to handle updates, i.e., new versions being added. Ideally the cost of incorporating a new version is proportional to the size of the update itself, i.e., the difference between the new version and the version it derives from.

Next, we discuss a few baseline approaches that serve as layers on top of a key-value store, and how they fare w.r.t. these trade-offs.

- **Single address space:** Perhaps the simplest option is to store the records directly, using the composite key as the key for the underlying key-value store. Although simple to implement and offering best performance for updates (ingest), this approach has several disadvantages. First, there is no way to use compression to reduce storage requirements, since different records with the same primary key are stored separately. Second, given a specific version V and a specific primary key K, retrieving the record with that primary key from that version (if present) requires an additional index. This is because of the way composite keys are generated – we first need to identify the predecessor version to V where that primary key was last modified. This complicates the execution of all the queries listed above. Not only does the index have to be repeatedly consulted, we may need to issue many queries against the backend key-value store.

- **"Sub-chunk" approach:** Here, we group together all the records with the same primary key K, and store it in compressed fashion using K as the key; we call such a group of records with the same primary key a **sub-chunk**. This approach has the best storage cost and best performance for record evolution queries (and possibly single record queries, if the average number of different records per primary key is small). However, full or partial version retrieval queries require retrieving significant amounts of irrelevant data, especially if the data is not highly compressible (i.e., different records with the same primary key are more different than similar). Further, ingest is expensive since each of the relevant sub-chunks must be retrieved, de-compressed, and compressed after adding the appropriate record.

  One alternative here is to create multiple sub-chunks per primary key, which results in retrieving less data and also speeds up online updates (the "single address space" approach is a special case). However, this negates much of the simplicity of the approach, since additional indexes need to be used to identify the specific sub-chunks that contains the data for a given version.

- **Delta approach:** Here, analogous to how version control systems like *git* work, for each version, we store the difference from its predecessor version, i.e., the "delta" that allows us to get to the version from the predecessor version. The predecessor version itself may be stored as a delta from its predecessor and so on, forming *delta chains*. The main advantage of this approach is that updates are easy to handle, especially since we assume that a new version is presented as a delta from its predecessor version. Assuming that the delta is computed by exploiting similarities at the level of records, this approach naturally accrues the benefits of compression. However, performance of key-centric queries, i.e., specific record queries and record evaluation queries, is very poor for this approach. Even partial retrieval queries are difficult to do with this approach.

Table 5.2 summarizes some of these trade-offs, by showing expressions for various different costs under some simplifying assumptions. Specifically, we assume a version with m_v records, with a sequence of changes each updating a fraction d of the records; thus the version graph here is a "chain". Note that this is a worst-case scenario for the delta approach; however, the main problem with the delta approach is partial or single record retrieval queries, where it has abysmal performance.

Too Many Queries Problem. None of the baseline approaches are thus appropriate for storing and querying a large number of versions of keyed records. Further, all of these approaches require *making a large number of queries* to the underlying key-value store for full or partial version retrieval. This is because the records belonging to a specific version V cannot be easily described. For example, in the first approach (and the partial sub-chunk approach), we need to use separate indexes to identify the "keys" that must be retrieved, and all of those must be retrieved separately from each other (efficient support for large IN queries from the key-value store may help, but only shifts the problem to the key-value store). Similarly, in the Delta approach, all the requisite deltas must be retrieved one-by-one.

Chunk size	1	10	100	1000	10000
Time (in secs.)	65.42	14.18	3.10	1.07	0.56

Table 5.1: Benefits of "chunking"

To validate our claim, we performed a simple experiment using Apache Cassandra. Each version in the dataset has about 100K 100-byte records, with a total of 1 million unique records stored in the KVS. The query here is to reconstruct a version, i.e., we need to retrieve around 100K records for every version reconstruction query from the KVS. In the naive setting, we maintain a chunk of unit size and issue around 100K requests to the KVS. In comparison, if we create larger sized chunks using a random assignment of records to chunks, we need to retrieve more number of chunks than exactly required to recreate a version. However the overhead of retrieving additional chunks and scanning through them to

Algorithms	Storage Space	Random Version (total data, #queries)	Point Query (total data, #queries)
IND	$nm_v s$	$m_v s$, $m_v s/s_c$	s_c, 1
DELTA	$m_v s(1 + cd(n-1))$	$m_v s\left(1 + \dfrac{cd(n-1)}{2}\right)$, $n/2$	$m_v s\left(1 + \dfrac{cd(n-1)}{2}\right)$, $n/2$
SUBCHUNK	$m_v s(1 + cd(n-1))$	$m_v s(1 + cd(n-1))$, m_v	$s + cd(n-1)s$, 1
SA	$m_v s(1 + d(n-1))$	$m_v s$, $m_v s$	s, 1

Table 5.2: Comparing the different options for storing versioned records along different dimensions under some simplifying assumptions (IND: Independent w/chunking, SA: Single Address Space). n = number of versions (arranged in a chain); m_v = number of records in a version (constant), d = fraction of records that are updated in every version update, c = compression ratio (typically $c, d \ll 1$), s = size of a record, s_c = size of a chunk. For the queries, the table shows: amount of data retrieved, number of queries. This analysis assumes the cost of consulting any indexes is negligible.

extract the records is significantly less. Table 5.1 illustrates the significant benefits of reducing the number of queries made to the key-value store. Unfortunately, because of the aforementioned problem, this problem must be solved by explicitly creating "chunks" of records, where records belonging to the same set of versions are grouped together.

5.3 Architecture

Figure 5.2 shows the high-level architecture of our system. In what follows, we describe the primary components that constitute our system, namely (i) Data Ingest Module, (ii) Data Placement Module, and (iii) Query Processing Module, as well as the different design choices that were made while building this system.

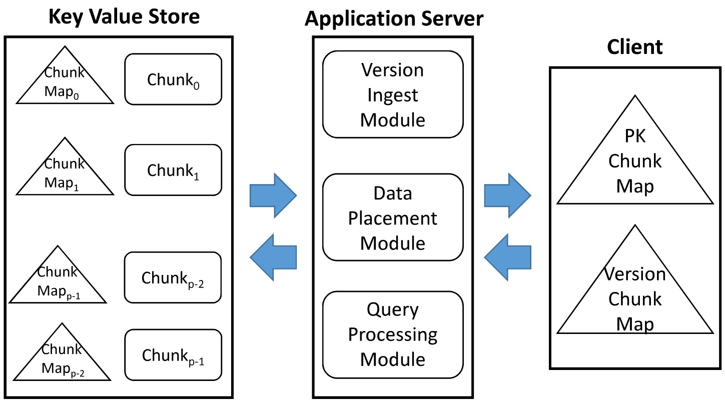


Figure 5.2: System Architecture

Backend Key-value Store Our system is intended to act as a layer on an extant distributed key-value store, in order to leverage the significant research and implementation that has gone into designing scalable, fault-tolerant systems. Our implementation specifically builds on top of Apache Cassandra, but we only assume basic `get/put` functionality from it. As shown in Figure 5.2, the basic unit of storage in the key-value store a *chunk* of records, with the keys called *chunk-ids*; chunk-ids are generated internally and are not intended to be semantically meaningful. Each chunk is divided into sub-chunks, each of which corresponds to records with the same primary key and are stored in a compressed fashion; sub-chunks often may contain only one record. In addition, a chunk also contains a *mapping* that indicates, for each record, which versions it belongs to (as a list of version-ids). Such a mapping is essential since a record may belong to multiple versions, and as discussed above, there is no easy way to identify which records belong to which versions.

This design was motivated by the desire to address the shortcomings of the baseline approaches discussed above, by having several tuning knobs that could be used to adapt to different data and query workloads. The main reason behind chunking was to address the problem of too many queries. By keeping records that need to be retrieved together in a single chunk, we minimize the number of queries that need to be made to the backend store. At the same time, through appropriately setting the parameters, our system can easily emulate the behavior of the different baselines discussed above. For example, the "sub-chunk" approach can be easily emulated by forcing the partitioner to put all records with the same primary key in a single chunk, and keep different primary keys in separate chunks. However, in general, for mixed workloads, a hybrid solution ends up being ideal, where each chunk contains a few sub-chunks, each containing a subset of the records with the same primary key; such a partitioning not only reduces storage requirements by exploiting compression, but also reduces the number of queries that need to be made to the back-end.

Example 8. *Table 5.3 shows two different partitionings for the data from Example 7. To reconstruct version* V_1 *which contains* $\langle K_0, V_0 \rangle$, $\langle K_1, V_0 \rangle$, $\langle K_2, V_0 \rangle$, $\langle K_3, V_1 \rangle$, $\langle K_4, V_1 \rangle$, *we must retrieve chunks* C_0, C_1, C_2, C_3 *for* $\mathcal{P}_0$, *and chunks*

C_0, C_1, C_2 for $\mathcal{P}_1$ (using the indexes discussed below). Overall, $\mathcal{P}_1$ reduces the average number of chunks to be retrieved per version by 0.6, and is thus a better option.

Partition	$\mathcal{P}_0$	$\mathcal{P}_1$
C_0	$\{\langle K_0, V_0\rangle, \langle K_1, V_0\rangle\}$	$\{\langle K_0, V_0\rangle, \langle K_1, V_0\rangle\}$
C_1	$\{\langle K_2, V_0\rangle, \langle K_3, V_0\rangle\}$	$\{\langle K_2, V_0\rangle, \langle K_3, V_0\rangle\}$
C_2	$\{\langle K_3, V_1\rangle, \langle K_3, V_2\rangle\}$	$\{\langle K_3, V_1\rangle, \langle K_4, V_1\rangle\}$
C_3	$\{\langle K_4, V_1\rangle, \langle K_5, V_2\rangle\}$	$\{\langle K_3, V_2\rangle, \langle K_5, V_2\rangle\}$
C_4	$\{\langle K_3, V_4\rangle\}$	$\{\langle K_3, V_4\rangle\}$

Table 5.3: Different Partitionings for Data

Application Server (AS) The application server serves as the interface between the clients and the backend KVS, and comprises of three main modules described next. It uses the KVS for persisting any of its data structures. Multiple copies of AS could co-exist, with the standard caveat that any data structures must be kept consistent across them (not currently supported in RSTORE).

AS currently provides a basic set of VCS commands. A user can *pull* any specific version by specifying its ID, or may *pull* the latest version in a branch (including the main *master* branch). Unlike a typical VCS, AS also provides the ability to retrieve partial versions or evolution history of a specific key as discussed in Section II(A). Any changes made by the user can be committed as a new version as discussed below.

Data Ingest Module Whenever a user commits a version, a *version-id* is generated by the system and is returned to the user after the commit process is complete. Even if two versions committed are exactly the same, the system will generate different version-ids for the two different commits (to account for different users, times at which they are committed, etc.). Due to the relatively large sizes of the datasets, the system requests only those records from the client that have changed, which in essence is the delta from the predecessor version. Thus the delta includes those records which have changed w.r.t. the previous version, records that are newly added and records that are deleted. If the client is unable to provide the delta, then the server needs to retrieve the prior version and perform a *diff* operation to check which records have been modified. However, in most settings, it is reasonable to assume that the client can do this.

Since updating the key-value store, and all the indexes, for every new version would be impractical, the received deltas are kept in a separate storage area, that are processed in a batch fashion by the data placement module.

Data Placement Module This module is responsible for organizing the ingested data for efficient query processing. Once all the tuples ingested have been assigned a composite key, the data storage module scans through the records and places them into appropriate chunks using the underlying partitioning algorithm.

In addition to placing the records, it is also responsible for constructing the version-record index for every chunk constructed and the version-chunk index that resides with the client for retrieving the versions. The chunks and associated indexes are stored in the KVS separately, in two distinct tables.

Indexes and Query Processing Module After the partitioning is completed, the system needs to know which chunks must be retrieved to extract the records belonging to a version. As discussed above, such an index is required even in the simplest approach, to be able to store any specific record only once even if it appears in multiple versions. Figure 3(a) depicts the entire mapping, denoted $\mathcal{M}_{|K|\times|V|\times|C|}$, between primary keys, version-ids, and chunks, that captures where each record is stored, and which versions contain it. The cells of this 3-dimensional matrix are annotated with version-ids that are required to construct the appropriate composite keys. Specifically, the entry $\mathcal{M}(K_i, V_j, C_k) = V_l$ if a *record* with composite key $\langle K_i, V_l \rangle$ is placed in *chunk* C_k and belongs to *version* V_j; otherwise the entry is set to 0. This matrix is expected to be very large and highly sparse, but the information it depicts must be somehow maintained, either implicitly or explicitly, in the system.

We maintain this information as follows. First, with each chunk in the backend key-value store, we maintain the slice of the matrix corresponding to that chunk,

$\mathcal{M}^{C_i}$. This allows us to extract the records that belong to any specific version after the chunk has been retrieved from the backend key-value store. In aggregate, all of these "chunk maps" contain exactly the same information as $\mathcal{M}_{|K|\times|V|\times|C|}$. Note that, the chunk maps will exploit the sparsity of the 2D matrix by using a *value list* representation of the matrix.

Second, in order to be able to decide what chunks to retrieve for a given query, we maintain two *lossy* projections of the matrix: (1) a mapping between primary keys and chunks that tells us which chunks contain records for a given primary key, and (2) a mapping between versions and chunks that tells us which chunks contain records from a given version. We use in-memory hashmaps to store these mappings.

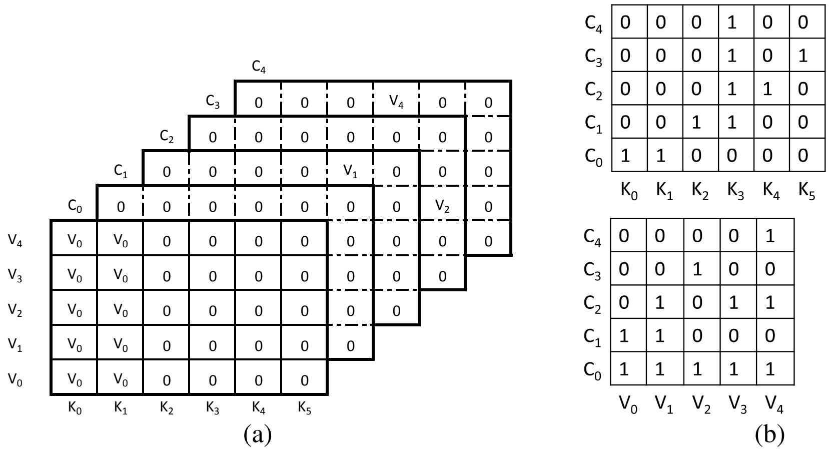
Query processing itself is straightforward given these indexes. We briefly summarize it below.

Version Retrieval: The second projection is consulted to identify which chunks need to be retrieved, and those chunks are retrieved by issuing queries in parallel to the backend store. After the chunks are retrieved, the chunk maps are used to extract the records that belong to that version.

Record Evolution: Similar to the above but the first projection is used instead.

Range Retrieval/Record Retrieval: Similar to “index-ANDing”, both the projections are used here to obtain two lists of chunks, and all chunks in the intersection are retrieved from the backend store. Note that, it is possible for us to retrieve a chunk and, after analyzing the chunk map, discover that it contains no records of interest – this is an artifact of these being lossy projections.

The size of the *version-to-chunk mapping* is essentially the sum total version span across all versions, assuming the mappings are stored as adjacency lists. For dataset C0 in Table II (one of our bigger datasets), this results in a total index size of 11.25MB, compared to a total dataset size of 16GB after deduplicating. The size of the *primary key-to-chunk mapping* is governed by the number of primary keys and the number of different chunks they belong to, which in turn is depends on the size of the chunk and the degree of compression. The size of the map for dataset C0 ranges from 25MB to 75MB. Thus even with significantly larger datasets and numbers of versions, these indexes can easily fit in the large main memory machines that are available today. In fact, with larger datasets, we would typically use larger chunk sizes and sub-chunk sizes, both of which directly lead to lower index sizes. We further note that these sizes are before compressing the indexes themselves – standard techniques from *inverted indexes literature* can be used to compress the adjacency lists without compromising performance.



	K_0	K_1	K_2	K_3	K_4	K_5
C_4	0	0	0	1	0	0
C_3	0	0	0	1	0	1
C_2	0	0	0	1	1	0
C_1	0	0	1	1	0	0
C_0	1	1	0	0	0	0

	V_0	V_1	V_2	V_3	V_4
C_4	0	0	0	0	1
C_3	0	0	1	0	0
C_2	0	1	0	1	1
C_1	1	1	0	0	0
C_0	1	1	1	1	1

Figure 5.3: (a) Entire 3D mapping; (b) Lossy projections maintained as indexes

5.4 Formalizing the Optimization Problem

The key computational challenge here is deciding how to partition the records into chunks to minimize the storage cost and maximize the query performance (or minimize the *retrieval costs*). As we discussed in Section 5.2, both the amount of data retrieved and the number of chunks retrieved are crucial performance factors from the perspective of querying, whereas compressing records by putting different records with the same primary key in the same chunk is crucial for minimizing storage costs. To achieve predictable performance, we made the following design decision.

(Fixed chunk size assumption). All chunks are assumed to be approximately the same size, denoted C, with variations of upto 25% allowed.

This variation in the chunk size gives us flexibility while assigning variable-sized records to chunks, and ensures that we are not forced to do frequent reorganization when adding new versions. We recommend that the specific percentage be chosen based on the ratio of the average record size and the chunk size, so that a small number of records could be added to an already full chunk while staying within the limit; for our datasets, 25% ends up being a somewhat conservative number, and in our experimental evaluation, the chunks were rarely more than 5-10% overfull.

This allows us to focus on the number of chunks retrieved for queries as the key performance metric. Formally, we define the **span of a query** to be the number of chunks that must be retrieved to answer that query.

Let n denote the total number of versions, m denote the total number of distinct records in them, and G denote the version graph depicting the relationships between the versions. For a given partitioning (i.e., chunking), the *storage cost* and the *retrieval costs* can be calculated as follows.

Storage Cost. The total storage required is dominated by the chunks; the different indexes constitute a relatively small and largely fixed overhead. However, because of compression, it is hard to express the total storage required by the chunks analytically. Instead we use the *number of chunks required* as a proxy for the total storage cost. Since all chunks are about the same size, this faithfully captures the relative storage costs of different partitionings.

Retrieval Costs. For a query, let θ_i denote the total number of chunks that need to be accessed for answering it. The total retrieval cost is comprised of the *communication cost*, which in turn depends on the number of queries made to the backend (θ_i) as well as the total number of bytes transferred, and the *CPU cost* of extracting the relevant records from the chunks. Once again, it is difficult to express this cost analytically; however, given the fixed chunk size assumption, the overall cost is largely proportional to θ_i, and we use that as our retrieval cost metric.

Since there are two different objectives here, analogously to [23], we can formalize optimization problems in different manners. However, our fixed chunk size assumption simplifies the problem somewhat if there is no compression.

Case 1: No Record-Level Compression. In this case, the total number of chunks is approximately equal to the total number of bytes across all the records divided by the size of a chunk (C). Thus the optimization problem can simply be stated as minimizing the retreival cost for a query workload by appropriately assigning records to the chunks.

Case 2: Record-Level Compression Allowed. In this case, the number of chunks required depends on how much compression can be obtained by grouping together the records with the same primary key. In this paper, we do not attempt to solve the problem in its full generality. Instead, we simplify the problem by assuming that a parameter, denoted k, is provided that controls how many records with the same primary key may be compressed together. ($k = 1$ corresponds to No Record-level Compression case). We use this parameter to partition the records with the same primary key into *sub-chunks* that are compressed together in a first phase. Then, the problem of assigning sub-chunks to chunks reduces to Case 1, since the total number of chunks required is once again fixed.

Converting Version Graphs to Version Trees. The following observation leads to the importance of version graphs in partitioning the records: A record (or a group of records) that appears in a version in a given branch of a tree can only be present in versions which are its descendants thereby allowing records present in different branches to be placed in different chunks, resulting in better partitioning decisions. In the next section, we discuss three different algorithms that partition the records into respective chunks. Except the shingles-based algorithm, the other algorithms use the version graph as a guide while creating the partitions. Those algorithms traverse the nodes of this graph in some particular order, read the records in the deltas and place them in the chunks.

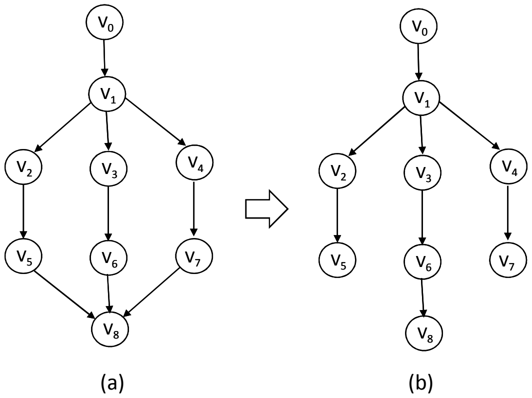


Figure 5.4: Converting a version DAG to a version tree

Due to the inherent complexity of the problem of partitioning, we use version graphs with no merges (henceforth referred to as *version trees*), in the subsequent partitioning techniques that use it. We use Figure 5.4 to demonstrate the process of dealing with merges in version graph. Versions V_5, V_6 and V_7 form the list of parents of V_8. To convert the DAG to a tree, we choose a parent of V_8 arbitrarily (in this case V_6) retaining the edge between them while deleting the other two edges. In this process, there are records in V_8 that arrived exclusively from V_5 and V_7 which are renamed to make them appear as newly inserted records. This conversion is solely used during the partitioning phase and the original version graph is still available to any queries afterwards.

Connection to other problems. The problem of partitioning records into chunks is closely related to the problem of identifying bicliques in a bipartite graph [40].

In the current problem setting, the relationships between records and versions can be represented as a bipartite graph where there is an edge between a version and a record if that record appears in that version. We want to identify records that are present across a large number of versions from the bipartite graph. This is essentially finding the maximal biclique in the graph. In this case, we are interested in enumerating all maximal bicliques in the graph and then selecting bicliques that have disjoint set of records. However the problem of enumerating all maximal bicliques turns out to be NP-Hard [29]. Once we have the set of bicliques, we need to chunk them into bins of fixed size such that the number of bins required is minimized. This is the classical bin-packing problem and is NP-Hard.

The indexes used for recreating the versions have significant redundancy. Recall that an index for a version stores the keys of the records present in the version. In the current setting, the $\langle$chunk-id, record-key$\rangle$ pair can be used as a substitute of record keys. For two versions differing only by a few keys, the amount of redundancy is huge and therefore necessitates development of techniques for compressing the index. This problem is exactly equivalent to the problem of compression of posting lists [24, 86], where the version-id and the record keys correspond to the *term* and the *document-id's* in which the terms appear, respectively. This problem is also related to compressing graphs where the version-id and the record keys correspond to a graph node and its neighbors [20, 25, 35].

5.4.1 Discussion

In our discussion so far and in our prototype implementation, we assume that the backend KVS supports only a basic *get/put* interface. This raises the question of whether KV stores with richer functionality like *range* queries or *stored procedure* may negate the need for our approach. Although the trade-offs would be somewhat different, the key aspects of our approach are fundamental to the problem setting of maintaining versioned collections of records. Briefly, there are four key issues here: (1) exploiting overlap across versions, which we handle by not duplicating records, (2) retrieving a specific record from a specific version, which requires maintaining several large indexes efficiently, (3) too many queries problem, which we mitigate through careful assignment of records to chunks, and (4) compressing multiple versions of large records without compromising retrieval performance, which we handle through "sub-chunking".

As we discuss in Sections 5.2, not addressing any of these will lead to substantial performance issues. Hence, all four of the above proposed solutions must be present in any system that solves this problem effectively. In our prototype implementation that we presented in the paper, we assumed a simple key-value store (for maximum portability), and thus all four of those techniques had to be part of the RSTORE layer on top. Conceivably, a key-value store could handle one of those issues internally (i.e., implement one or more of our proposed techniques inside the key-value store), eliminating the need for them in RSTORE. However, we are not aware of prior work along these lines, and we consider the development of these approaches to be a lasting contribution of our work.

Having support for range queries does not unfortunately remove the need for any of the four techniques as described above. The "index" that tells us which records constitute a version still needs to be maintained in RSTORE; and the queries that will be posed against the backend key-value store will not be "range" queries. For example, in the example in Figure 1, the list of records that constitute any of the versions cannot be captured as a range query on the composite key. Need for compression further complicates this because we need to retrieve "sub-chunks" that contain the required records, and the sub-chunk IDs are effectively random.

Support for efficient large IN queries may help to some extent, depending on how they are implemented (in Cassandra, they are implemented by broadcasting to all data servers which leads to worse performance). In particular, that support will reduce the benefits of chunking, but not eliminate it. We still have the "too many queries" problem, but **internally to the key-value store**, i.e., there will be too many queries between the server that is collecting the query answer and the backend servers that host the data. So a chunking approach may lead to improved performance. Unfortunately none of the key-value stores we investigated support large IN queries to investigate this properly.

Finally, "stored procedures" cannot help here unless a large amount of the logic in RSTORE, including indexes, compression/decompression modules, and query module, is duplicated there. Even then, the "too many queries" problem is still present between the query and the data servers.

5.5 Partitioning Algorithms

In this section, we present three different algorithms to solve the partitioning problem formalized in Section 5.4. We begin with an adaptation of a standard *shingles*-based algorithm for finding bicliques, followed by two algorithms that exploit the inherent structure in the problem as embodied in the version graph.

5.5.1 Shingles-based partitioning

To minimize query spans, we want to place records together that are common to a large number of versions. This requires determining the similarity between

Algorithm 4: Computing shingles for a set of versions

```
Input  : Set of versions $V \in S_i$, a family of $l$ pairwise-independent hash
         functions $H$
Output : Shingle array of size $l$
shingles$[S_i] \leftarrow \{\}$
for each $h \in H$ do
    shingles$[S] \leftarrow \{$shingles$[S], \min_{v \in V} h(v)\}$
end
return shingles
```

records based on the versions they belong to. Here we adapt a standard technique for finding bi-cliques based on *shingles* or *min-hashing*, which provide an estimate of the similarity between large sets [29]. Briefly, for each set (here the set of versions that a specific record belongs to), we compute l min-hashes, using hash functions $h_1, ..., h_l$; for each h_i, we apply the hash function to all the elements in the set and take the minimum of those as the i^{th} min-hash. This gives us a list of l-shingles (min-hash values) for each record (Algorithm 4). To compute the shingle ordering, we sort and order the records based on this list of shingle values in a lexicographical fashion. This ordering places records whose version sets have high similarity (i.e., overlap) in close proximity to each other. This shingle-based order is then used to place the records into the chunks (Algorithm 5).

We also build the chunk maps, $\mathcal{M}^{C_i}$ after all records have been assigned to their chunks. For every record in version V_i, we determine the chunk C_i that it belongs to and add it to set of composite keys for that chunk. After scanning the

Algorithm 5: Shingle-based partitioning

```
Input  : A set $r$ of records, version graph $G_t$, chunk capacity $C$
Output : Set of chunks that partitions the records
// Traverse the versions, scan records, construct record to version map
// Compute Shingles for each record
for each $r_i \in r$ do
    $\omega_i =$ ComputeShingles($r_i$)
end
// Sort the records based on shingle values($\omega_i$)
Sort($r$)
for each $r_i \in r$ do
    // Assign records to chunks $C_j$ using the shingle-based sort-order
    if $C_j < C$ then
        $C_j \leftarrow C_j \cup r_i$
    end
    else
        Create a new chunk and assign $r_i$
    end
end
```

full version, we visit every chunk that contained records from V_i and write the version to composite key list to the corresponding chunk map file on disk. After this process is repeated for every version, we have the complete chunk map file for every chunk. The adjacency list in each chunk map file is then converted to a bitmap, compressed and stored in the KVS. Note that we use this algorithm for constructing the chunk maps for the subsequent partitioning algorithms as well.

Complexity. The complexity of the shingle-based technique for partitioning the records may be broken down as follows:

1) Constructing the record to version map takes $O(nm')$ time which requires visiting every version and scanning every record in it, where $O(m')$ is the average number of records in a version.

2) Next we compute the shingles for every record. If each record belongs to $O(n_V)$ versions, then the time taken is $O(mn_V)$. Note that the quantity mn_V is $O(nm')$ as both of them denote the total number of records in the dataset.

3) Sorting the records based on l shingle values takes $O(m \log m)$. Here the value of l is a small constant.

4) Assigning the records to chunks can be done in $O(m)$ time.

5) Building the chunk maps takes $O(n(m' + \rho_C))$ time. Here ρ_C denotes the average number of chunks that records of any given version belongs to. Thus we have $\rho_C = O(m')$ and the time complexity of constructing the chunks is $O(nm')$.

Therefore the overall time complexity of this algorithm is $O(m \log m + nm')$.

5.5.2 Bottom-Up Traversal

In this approach, we partition the records in the versions by traversing the version tree bottom-up[1]. The key idea here is to identify and chunk records that do not belong to versions above as we move up through the versions in the version tree. For simplicity, we will first describe the approach for 1-ary version trees and then extend it to general trees. Let us consider a version V_i as depicted in Fig. 5.5 which needs to be processed. Since we follow a bottom-up approach, the versions below V_i in the version tree have already been processed. Let S_i denote the set of records in V_i. The collection of sets $\pi_{i+1} = \{S^1_{i+1}, S^2_{i+1}, \ldots, S^p_{i+1}\}$ contain the records that are returned by version V_{i+1} and denote the following:

S^1_{i+1} : records present in V_{i+1} but not in any version below.

S^2_{i+1} : records present in V_{i+1}, V_{i+2} but not in any version below.

$\vdots$

S^p_{i+1} : records present in $V_{i+1}, V_{i+2}, \ldots, V_{i+p}$.

Here p denotes the number of versions from the current version (in this case V_{i+1}) up to the leaf version. Similarly, V_i needs to return these sets to its parent V_{i-1}. In

[1] The Bottom-Up algorithm is inspired by [47] that gives an algorithm for partitioning a graph into two equal-sized partitions. In general, partitioning even trees is NP-hard [41].

the present iteration, we compute the collection $\pi_i = \{S_i^1, S_i^2, \ldots, S_i^p\}$, where

$$\begin{aligned}
S_i^1 &: \text{records present in } V_i \text{ but not in any version below.}\\
&\vdots\\
S_i^p &: \text{records present in } V_i, V_{i+1}, \ldots, V_{i+p}.
\end{aligned}$$

These sets are computed as follows:

$$\begin{aligned}
S_i^2 &= S_{i+1}^1 \cap S_i, \qquad S_i^3 = S_{i+1}^2 \cap S_i\\
&\vdots\\
S_i^1 &= S_i \setminus (S_i^2 \cup S_i^3 \ldots \cup S_i^p)
\end{aligned}$$

It is also possible to express the sets in π_i in terms of deltas. First, we will define deltas, discuss some of their algebraic properties and then describe the expressions. A delta Δ between two versions V_i and V_j is a set of records that may be split into two disjoint sets, a positive delta set, Δ^+ and a negative delta set, Δ^-. Δ_{ij}^- denotes the set of records that are present in V_i but not in V_j, whereas Δ_{ij}^+ denotes the set of records that are present in V_j but not in V_i. It is easy to see that $\Delta_{ij}^+ = \Delta_{ji}^-$ and $\Delta_{ij}^- = \Delta_{ji}^+$. For the following expression to hold, we require the deltas to be consistent [43], i.e., $\Delta_{ij}^+ \cap \Delta_{ij}^- = \phi$. The collection π_i can expressed in terms of Δ as follows:

$$S_i^1 = \Delta^-_{i,i+1}, \qquad S_i^2 = \Delta^-_{i+1,i+2} \setminus \Delta_{i,i+1}$$

$$\vdots$$

$$S_i^p : V_n \setminus \bigcup_{j=0}^{p-1} \Delta_{i+j,i+(j+1)}$$

Note that for the last term we have the whole version V_n instead of a Δ^- since the last version does not have a Δ to some other version that captures the records that are exclusively present in version V_n. For general trees, computing π_i changes slightly only for versions which have more than one child. In those cases S_i^1 is the union of the Δ^- between version V_i and its children.

Given the collection of sets obtained from V_{i+1} and the sets computed at V_i, it is now possible to determine the records that exclusively belong to certain versions, denoted by $\psi_i = \{\alpha_i^1, \alpha_i^2, \ldots, \alpha_i^p\}$. Thus we have,

$$\alpha_i^1 = S_{i+1}^1 \setminus S_i^2 \text{ (records present only in } V_{i+1}\text{)}$$

$$\vdots$$

$$\alpha_i^p = S_{i+1}^p \setminus S_i^p \text{ (records present in } V_{i+1}, V_{i+2}, \ldots, V_{i+p}\text{)}$$

Lemma 4. *Given a linear chain of versions, we have $\bigcap_{j=1}^p \alpha_i^j = \phi$, at any version*

i.

Note that the records present in the sets from α_i^1 to α_i^p are not present in any version from V_i or above; so we can chunk these records. The records in set α_i^p must be chunked first, followed by those in α_i^{p-1} and so on. This is because records in α_i^p belong to p consecutive versions, followed by records in α_i^{p-1} which belong to $p-1$ consecutive versions and so on, the chunking process at any given version starts filling a new chunk (or bin). This is to ensure that access to highly common records during version reconstruction is not split across multiple chunks, which in turn results in increasing the version span. The partial chunks that may get created at the end of every chunking step are merged at the end to reduce fragmentation. We demonstrate the chunking process through an example as follows.

Example 9. *Consider Fig. 5.5 where we have a linear chain of versions. Boxes represent records within versions and the colored boxes are the records which appear in V_{i+1} and not in any prior version. Therefore the colored boxes represent the records in ψ_i with the purple record representing α_i^1, since they appear only in version V_{i+1}. Similarly, we have the blue record in α_i^2 and so on. It is easy to see that the record in red must be chunked first, followed by the records in green and orange, then blue and finally purple.*

For general trees, the primary difference with the existing approach lies in

Algorithm 6: Bottom-Up Traversal for Partitioning

```
Input  : Version graph $G_t$, root version $V_r$ and deltas, sub-tree limit $\beta$, chunk
         capacity $C$
Output : Set of chunks that partitions the records
Bottom-Up $(V_r)$
return set of chunks
Bottom-Up $(v)$ {
if v is not null then
    for each child $\in$ v do
        Bottom-Up $(child)$
    end
    // process the sub-tree $T_v$ rooted at $v$
    for each version $V_j \in T_v$ do
        compute $S_v^j$
    end
    // return set collection $\pi_v$ to parent of $v$
    // compute the records exclusive to $v$ and chunk them
    // adjust sub-tree $T_v$ if the size of sub-tree $> \beta$
end
}
```

processing versions with more than one child. Recall that at every version V_i, the child of V_i returns p different sets to its parent. If V_i has λ children, then it receives $\lambda \times p$ sets from its children. Unlike in linear chains (Lemma 4), a given record may be present in more than one set (and no more than λ sets, one from each child) for general trees. In the presence of multiple sets obtained from multiple children, ordering the records may be performed as follows:

1) For every record, assign a count based on the number of consecutive versions it belongs to. The count is added for records that appear in multiple

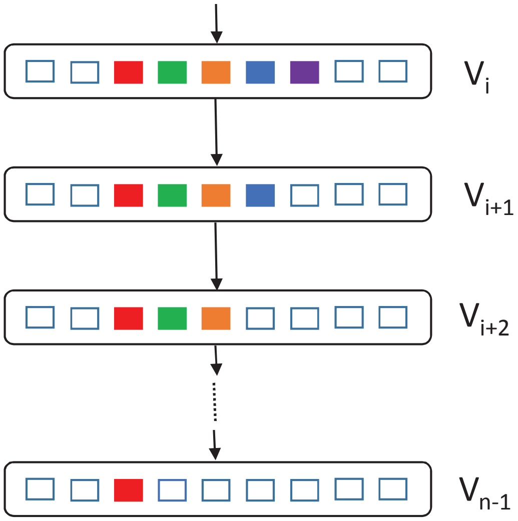


Figure 5.5: Bottom-Up Partitioning for Linear Chains

sets.

2) Sort the records.

A close approximation to the above technique may be obtained by considering sets of records that belong to similar number of consecutive versions. Therefore sets from different children that correspond to same number of consecutive versions, are chunked together. To deal with duplicate records, a hash-table is maintained to identify records that have already been chunked.

Controlling the subtree of a version. The size of the subtree corresponding to a version in the tree dictates the amount of processing that needs to be done per version. For general trees, the size of subtrees is significantly larger compared to linear chains due to the presence of multiple branches per version on an average. In order to bound the amount of processing, we may choose to have at most β nodes (or sets) in the subtree; the subtree can be reduced by merging nodes within it. Recall that each version in subtree corresponds to a set of records S_{i+1}^{j} that V_{i+1} returns to V_i. The merging involves the following steps:

1) Sort the leaves of the subtree by the sizes of the corresponding sets and store it in L_s.

2) For every version V_x in the sorted set:

    a) Merge the contents with its parent V_p. Remove V_x from L_s.

    b) If every child of V_p have been merged, then include V_p in L_s.

3) Repeat until the number of nodes in subtree is equal to β.

It is easy to see that a reduction in the size of the subtree reduces the total execution time of the BOTTOM-UP algorithm as the amount of processing per version is proportional to β. This may be true upto a certain β as the overhead of merging the nodes may dominate for smaller values of β. However, with a decrease in a number of sets, the partitioning quality may also degrade, explained as follows. Consider that there are 10 sets below a version forming a linear chain and we want to determine the records in ψ_i. We find that record $\langle K_i, V_i \rangle$ belong to 10 consecutive versions whereas record $\langle K_j, V_j \rangle$ belong to 5 consecutive versions, among other records. Therefore record $\langle K_i, V_i \rangle$ has a higher ordering than record $\langle K_j, V_j \rangle$ during the chunking process. Now, consider that $\beta = 5$. In this case, both the records may be placed together; record $\langle K_j, V_j \rangle$ may be chunked with other records that have higher depth instead of $\langle K_i, V_i \rangle$, which leads to degradation of the partitioning strategy.

An outline of the bottom-up partitioning algorithm is provided in Algorithm 6.

Complexity. At every version, the number of set operations we perform is proportional to the the number of versions below it. Each set operation can be bounded by $O(m')$ although in practice this is significantly less as this is proportional to the size of a delta. Thus the total complexity of set operations for all versions is $O(n\beta m')$. The complexity of constructing the chunks and chunk maps is $O(nm')$.

5.5.3 Depth-First/Breadth-First Traversal

To see if the benefits of the Bottom-up approach could be obtained using a simpler algorithm, we designed two algorithms which also use the version tree but make the partitioning choices greedily. These approaches traverse the version tree starting from the root in a depth-first or a breadth-first fashion, and chunk the

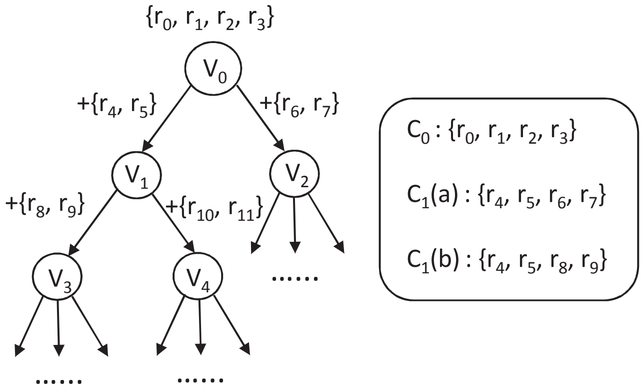


Figure 5.6: Version Tree Partitioning (using DFS)

records as they are encountered. We illustrate this with an example.

Example 10. *Consider the version tree in Fig. 5.6, and assume the chunk size is 4 records. As the the root version V_0 is visited, all the records are placed in the first chunk C_0. Next, we visit one of the descendants of V_0, say V_1 and place the 2 records in the next available chunk C_1. Now, we have two options here, (a) visit version V_2 (breadth-first traversal) and place the two records in the remaining space in chunk C_1, (b) visit version V_3 (depth-first traversal) and place the two records in the remaining space in the chunk C_1. Note that going with option (a) implies that any descendant of V_1 will not access any of the records from V_2. Similarly, none of the descendants of V_2 will access any of the records added to chunk C_1(a) from V_2 resulting in the possibility of increasing the span of the versions. In contrast, option (b) admits all the descendants of V_3 to acces all the*

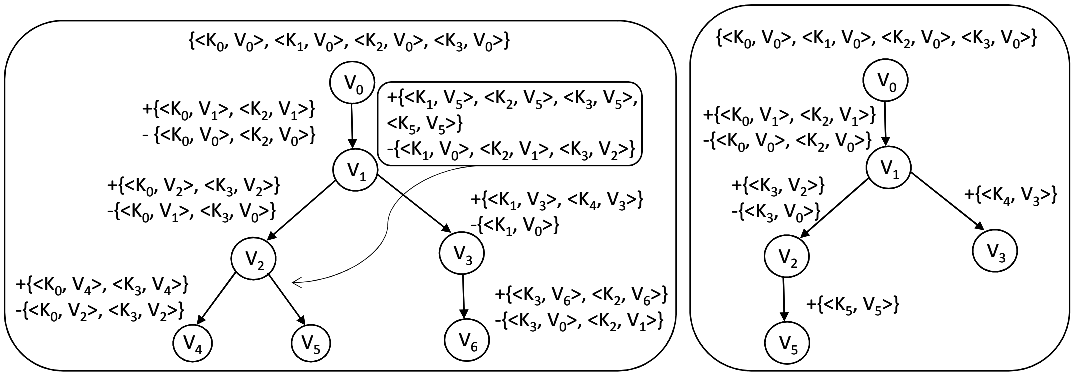
Algorithm 7: Depth-First Traversal for Partitioning

```
Input  : Version graph $G_t$, root version $V_r$ and deltas, chunk capacity $C$
Output : Set of chunks that partitions the records
dfsStack ← {}
for each $V_i \in G_t$ do
    visited[$V_i$] ← false
end
push dfsStack, $V_r$
while dfsStack is not empty do
    $u$ ← peek dfsStack
    if $u$ has child then
        $v$ ← getNextChild($u$)
        if not visited[v] then
            visited[v] ← true
            push dfsStack, v
            // read the delta and populate the chunk
            for each record $r_i \in \Delta_{u,v}$ do
                $C_i \leftarrow C_i \cup r_i$
                // if $C_i$ is full then allocate a new chunk
            end
        end
    end
    else
        pop dfsStack
    end
end
return set of chunks
```

records in chunk C_1(b).

Assuming that most of the versions do not differ significantly from their parent version, traversing the version tree depth-first turns out to be more beneficial than breadth-first approach. An outline of the depth-first partitioning algorithm is provided in Algorithm 7.

Complexity. The complexity of this algorithm is $O(nm')$, where $O(nm')$ is for traversing the all the records in each version. The complexity of chunk map construction is $O(nm')$.



(a)

(b)

Id	CK	Sub-chunk
SC_0	$<K_0, V_1>$	$\{<K_0, V_1>, <K_0, V_2>, <K_0, V_4>\}$
SC_1	$<K_0, V_0>$	$\{<K_0, V_0>\}$
SC_2	$<K_1, V_0>$	$\{<K_1, V_0>, <K_1, V_3>, <K_1, V_5>\}$
SC_3	$<K_2, V_1>$	$\{<K_2, V_1>, <K_2, V_5>, <K_2, V_6>\}$
SC_4	$<K_2, V_0>$	$\{<K_2, V_0>\}$
SC_5	$<K_3, V_2>$	$\{<K_3, V_2>, <K_3, V_4>, <K_3, V_5>\}$
SC_6	$<K_3, V_0>$	$\{<K_3, V_0>, <K_3, V_6>\}$
SC_7	$<K_4, V_3>$	$\{<K_4, V_3>\}$
SC_8	$<K_5, V_5>$	$\{<K_5, V_5>\}$

(c)

Figure 5.7: Partitioning compressed records: (a) Original Version Tree, (b) Transformed Version Tree, (c) Sub-chunk list with $k = 3$ – CK indicates composite keys of the sub-chunks

Next, we show how we handle the case where $k > 1$, i.e., we wish to exploit compression by putting together records with the same primary key in the same chunk. As discussed in Section 2.5, we use a two-phase approach, where we first create the sub-chunks by grouping together records with the same primary key (with at most k per sub-chunk), and then choose one of the partitioning algorithms discussed so far for the chunking itself by treating the sub-chunks as records. Similar to records, we assign composite keys to these sub-chunks. One issue here is that, the original version tree may not be valid any more, and must be transformed (as discussed below) before the partitioning algorithms are invoked.

We impose the following constraint on any sub-chunk: the records that are grouped together are "connected" in the version tree, i.e., the versions that they belong to form a connected subgraph of the version tree. For example, in Figure 5.7, we would never group together $\langle K_1, V_3 \rangle$ and $\langle K_1, V_5 \rangle$, without $\langle K_1, V_0 \rangle$ (their common ancestor). This is being done in order to boost compression as records are more likely to be similar to their parents than their siblings. Delta-encoding may be used to compress records within chunks; thus all the sibling records would be delta-ed against their common parent.

The sub-chunk creation algorithm proceeds by traversing the version tree bottom-up; at every version (excluding the leaf versions) we inspect its children and consider the records that originated in those child versions via inserts or updates.

Every version can be assumed to have a collection of sets Ψ, each set in the collection Ψ corresponds to a primary key that originated in that version. At any given version V_i, we either construct sub-chunks (for every primary key present in V_i or any of its children) or delay the process until the next ancestor of V_i. Let $e(K_i)$ be a binary variable associated with a primary key K_i, which is 1 if K_i is present in V_i, otherwise 0. Let $s(K_i)$ denote the count of the number of records for primary key K_i across every child of V_i. If $e(K_i) = 1$ and $s(K_i) \leq k - 2$, an union of the records is constructed and the set is added to Ψ of V_i. However if $e(K_i) = 0$, instead of an union, the versions containing the records having primary key K_i are added to the child list of the parent of V_i. In the situation, when $s(K_i) + e(K_i) \geq k$, we construct subchunks out of the largest set in Ψ even though the set size may be less than k and then recurse on the conditions mentioned above. We present an algorithm for sub-chunk construction at a given version V_i (Algorithm 8). At every version V_i, we aggregate a list of primary keys that appears in V_i or any of its children and denote it by $\sigma(V_i)$. For each K_i in $\sigma(V_i)$, we execute the steps described earlier.

Transformed Version Tree. The next step is to construct the transformed version tree T_{VT} from the actual tree O_{VT} by treating the sub-chunks as individual records. Each sub-chunk is assigned a representative composite key $\langle K_i, V_i \rangle$ which may

Algorithm 8: Sub-chunk Construction Algorithm at Version V_i

```
Input  : Version graph $G_t$, version $V_i$ and deltas, sub-chunk size $k$
Output : Set of sub-chunks
for each $K_i \in \sigma(V_i)$ do
    if $e(K_i) \ = 1$ then
        if $s(K_i) = k - 1$ then
            construct sub-chunk.
        end
        else if $s(K_i) \leq k - 2$ then
            construct an union of records; add to $\Psi$
        end
        else
            construct sub-chunk out of the largest set. Repeat.
        end
    end
    else
        if $s(K_i) \leq k - 1$ then
            add the children with $K_i$ to parent of $V_i$
        end
        else
            construct sub-chunk out of the largest set. Repeat.
        end
    end
end
```

lead to duplicate versions. Given the sub-chunks, the example below demonstrates the assignment of sub-chunks to records and the construction of the transformed version tree. Different values of k will lead to different transformations of O_{VT} where each transformed version can be treated as a new dataset. The original partitioning algorithms can now be executed on these transformed datasets while taking into account the duplicate versions.

Example 11. *Fig. 5.7(a) represents the original version tree and Fig. 5.7(b) represents the transformed version tree. The sub-chunks corresponding to $k = 3$ are extracted from O_{VT} and are listed in Fig. 5.7(c) along with composite keys assigned to them. For deriving Fig. 5.7(b) from Fig. 5.7(a), we make a breadth-first traversal of O_{VT} and at each version visit all the records that originated in that version. For every record, we pull up the corresponding sub-chunk that it belongs to and check whether it has already been used or not. For the root version in V_0, none of the sub-chunks corresponding to the records would have been assigned already. Therefore, the sub-chunks SC_1, SC_2, SC_4 and SC_6 are assigned the following representative composite keys: $\langle K_0, V_0\rangle$, $\langle K_1, V_0\rangle$, $\langle K_2, V_0\rangle$ and $\langle K_3, V_0\rangle$, respectively. Next, we move on to the records in V_1. We observe that $\langle K_0, V_1\rangle$ and $\langle K_2, V_1\rangle$ does not belong to the sub-chunks that were assigned composite keys in the previous step. So we assign SC_0 and SC_3 to $\langle K_0, V_1\rangle$ and $\langle K_2, V_1\rangle$, respectively. At V_2, we see that $\langle K_0, V_2\rangle$ is already in SC_0 whereas $\langle K_3, V_2\rangle$ isn't part of any sub-chunk that has been assigned already. Thus $\langle K_3, V_2\rangle$ is the representative composite key of SC_5. Similarly, $\langle K_4, V_3\rangle$ is assigned to SC_7. Next we visit V_4 and observe records that were new to it have already been a part of sub-chunk that have been assigned to its ancestors. In other words, V_4 has the same records as that of V_2 and hence V_4 is a duplicate of V_2 and hence V_4 is deleted. As we move* *on to V_5 we note that $\langle K_5, V_5\rangle$ has not assigned; thus SC_8 is assigned to $\langle K_5, V_5\rangle$. Finally, we observe that V_6 is a duplicate of V_3 and hence deleted. These steps result in the transformed version tree in Fig. 5.7(b).*

Creating the sub-chunks is expensive since the algorithm has to extract the sub-chunks by visiting all the different versions. For creating a single sub-chunk consisting of k records, we have to visit k different versions. To speed up this process, we first create the sub-chunks where we just have the composite keys of the records that form the sub-chunk. Thereafter, we concatenate the records from the versions and sort them by their primary keys on disk. Next, we scan the sorted record list and read all the records belonging a given primary key into memory. Since we maintain a record to sub-chunk map, we now create all the sub-chunks corresponding to the primary key, compress them and store them into a disk-based key-value store. Thus the sub-chunk creation is completed in a single pass over this sorted list of records.

Complexity. The complexity of the sub-chunk construction algorithm is $O(nm' + m \log m)$, where $O(nm')$ is for traversing the all the records in each version and the second component is for sorting the unique records for sub-chunk extraction. The complexity of chunk map construction is $O(nm')$.

5.6 Online Partitioning

The main challenge with keeping the partitioning up-to-date with every new version is that, even if a version V_c differs from its parent version V_p by just a few records, all the chunks that contain V_p's records need to be updated (if only to update the chunk maps). As discussed earlier, we instead incorporate new versions in a batched fashion, by maintaining the deltas corresponding to the new versions in a separate write store, called a *delta store*, and by using an adapted version of a partitioning algorithm when the number of versions reaches a certain size (called the **batch size**, a user-configurable parameter).

To exploit the possibly high overlap across versions in the current batch, we compute a union of the chunk maps that need to be updated and then update every chunk map only once per batch. In order for a chunk map to be updated if it already exists, it has to be fetched from the KVS, updated and then written back again. Instead, every time a chunk map needs to be updated per batch, we recreate the chunk index from scratch and then write it back to KVS, saving the cost of fetching the chunk indexes from the KVS. This is possible by maintaining the required indexes around due to its small memory footprint. The complexity of the background process is determined by the size of the batch and the choice of the partitioning algorithm. In general, a smaller batch size would result in faster partitioning, however the quality of partitioning degrades with respect to a larger batch as more versions in a batch is beneficial for making better record placement decisions. Note that we do not re-partition records once they have been partitioned, however record re-partitioning, although expensive, may result in improving the overall version span. We leave this problem for future work.

5.7 Experiments

In this section, we present a comprehensive evaluation of the RSTORE system. We use a distributed installation of Apache Cassandra across upto 16 nodes for storing the partitioned records and their associated indexes. Each node has 16 GB of main-memory. We ran our experiments on a 2.2 GHz Intel Xeon CPU E5-2430 server with 64GB of memory, running 64-bit RedHat Enterprise Linux 6.5.

5.7.1 Datasets

We use a collection of synthetically generated datasets for the experiments. For each dataset, we first generate a corresponding version graph by starting with a single version, and then generating a set of modifications to it using the method outlined in [23], which closely follows real-life version graphs. Thereafter, we create a set of records for the base (root) version where each record is created as a JSON document. Every record in the base version is assigned an auto-incremented primary key and a randomly generated value of the requisite size. Each of the other versions is generated by updating or deleting a set of records in its parent, or inserting new records. The selection of records for updating and deleting either follows a random or a skewed (Zipf) distribution.

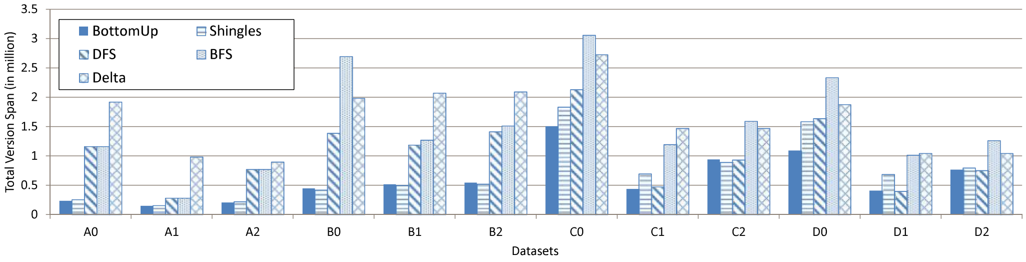
We have generated a wide spectrum of version graphs and corresponding datasets that mimics real-world use cases. They differ primarily along five factors: 1) *branching factor* (linear to highly branched), 2) *average version graph depth* (56 to 300), 3) *nature and percentage of updates* (random vs skewed updates with $1 - 50\%$ change), 4) *number of records in a version* (from a few thousand to hundreds of thousands of records), and 5) *number of versions* (from a few hundred to several thousand). The size of the records in the dataset also vary widely from a few bytes to several kilobytes. The number of unique records in the dataset varies from a little more than 1M records to around 17M records and total size of a dataset varies from ≈ 30 GB to close to 1 TB. We refer to Table 5.4 for a detailed description of the datasets.

Dataset	#V	AD	RPV	%U	UT	#UR (M)	URS (GB)	TS (GB)
A0	300	300	100K	50	R	12.3	11.9	31.67
A1	300	300	100K	5	S	1.51	5.77	140.14
A2	300	300	100K	5	R	1.34	5.14	141.26
B0	1001	293.5	100K	5	S	4.17	8	192.24
B1	1001	293.5	100K	5	R	4.22	8.07	193.77
B2	1001	293.5	100K	10	R	8.35	8.02	195.69
C0	10001	143	20K	10	R	16.53	15.95	196.46
C1	10001	143	20K	1	R	1.75	1.69	193.01
C2	10001	143	20K	5	S	8.17	7.87	193.05
D0	10002	94.4	20K	10	R	16.62	16.03	196.48
D1	10002	94.4	20K	1	R	1.77	1.71	193.07
D2	10002	94.4	20K	5	S	8.20	7.90	193.09
E	10001	170	20K	10	R	16.52	78.96	972.84
F	1001	56	100K	20	R	16.67	79.64	981.11

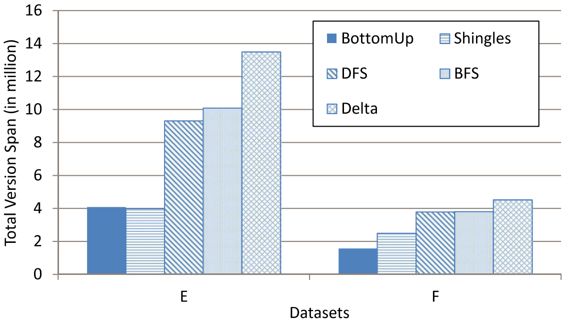
Table 5.4: Description of datasets: 1) #V: #Versions, 2) AD: Average Depth, 3) RPV: ∼Records per Version, 4) %U: %updates, 5) UT: Update Type (R: Random, S: Skewed), 6) #UR: Unique Records (in Million), 7) URS: Size of Unique Records (in GB), 8) TS: Total size (in GB)

5.7.2 Evaluation of Partitioning Algorithms

Comparison based on Total Version Span. We begin with comparing the performance of the partitioning algorithms: BOTTOM-UP, SHINGLE, DEPTHFIRST, and BREADTHFIRST. Here, we use the total version span (i.e., the total number of chunks retrieved for reconstructing all versions) for comparing the algorithms while fixing the chunk size to 1MB (we chose this chunk size since it provides a good balance between the number of queries and amount of data retrieved). In addition to algorithms that partition the record space for minimizing the version



(a)



(b)

Figure 5.8: Comparison of Total Version Span (without compression)

span, we also show performance of the DELTA baseline. We omit the SUBCHUNK baseline since the total version span for that approach is very high (all chunks must be retrieved for any version query).

In Fig. 5.8, we observe that BOTTOM-UP, SHINGLE and DEPTHFIRST outperform DELTA across all datasets, thus establishing that DELTA is inferior as a technique for handling keyed datasets (BOTTOM-UP outperforms DELTA by upto $8.21\times$ and on an average by about $3.56\times$ across all datasets). The performance of SHINGLE degrades with a decrease in the average depth of the version trees,

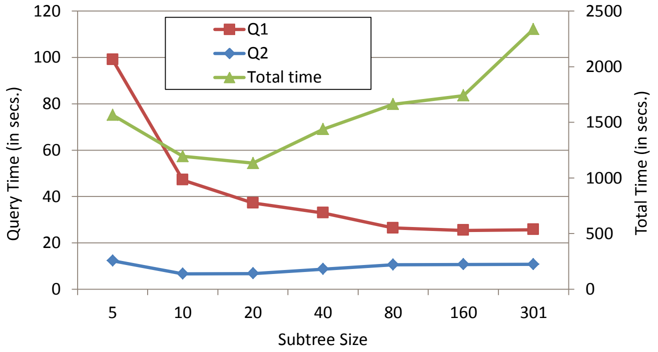
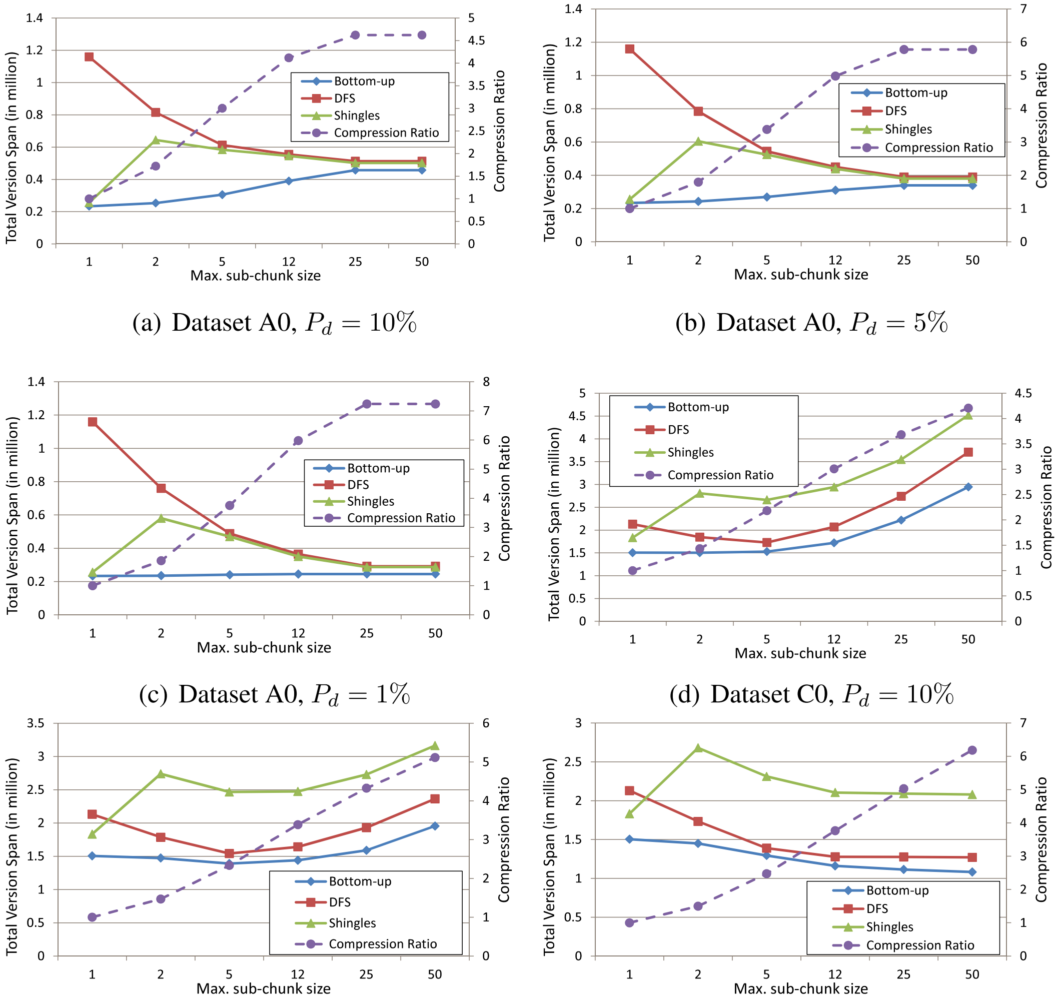


Figure 5.9: Effect of sub-tree size on performance of BOTTOM-UP (Dataset B0)

while that of DEPTHFIRST improves. **However unlike BOTTOM-UP, none of these techniques perform uniformly well across all datasets**. BREADTHFIRST is always worse than DEPTHFIRST (for reasons described in Section 5.5.3) except for linear chains when they reduce to the same technique.

Effect of Subtree size on performance. We vary the size of the subtree (β) BOTTOM-UP and observe the total version span (Fig. 5.9). As the size of the subtree decreases, the total version span increases as explained in Section 5.5.2. The total time taken by the algorithm first decreases with decrease in subtree size (due to decrease in processing per node) and then increases. The increase in total time for $\beta < 20$ in Fig. 5.9 can be attributed to increased processing time for merging the nodes. As β decreases the number of nodes needed to merge also increases.



(a) Dataset A0, $P_d = 10\%$

(b) Dataset A0, $P_d = 5\%$

(c) Dataset A0, $P_d = 1\%$

(d) Dataset C0, $P_d = 10\%$

(e) Dataset C0, $P_d = 5\%$

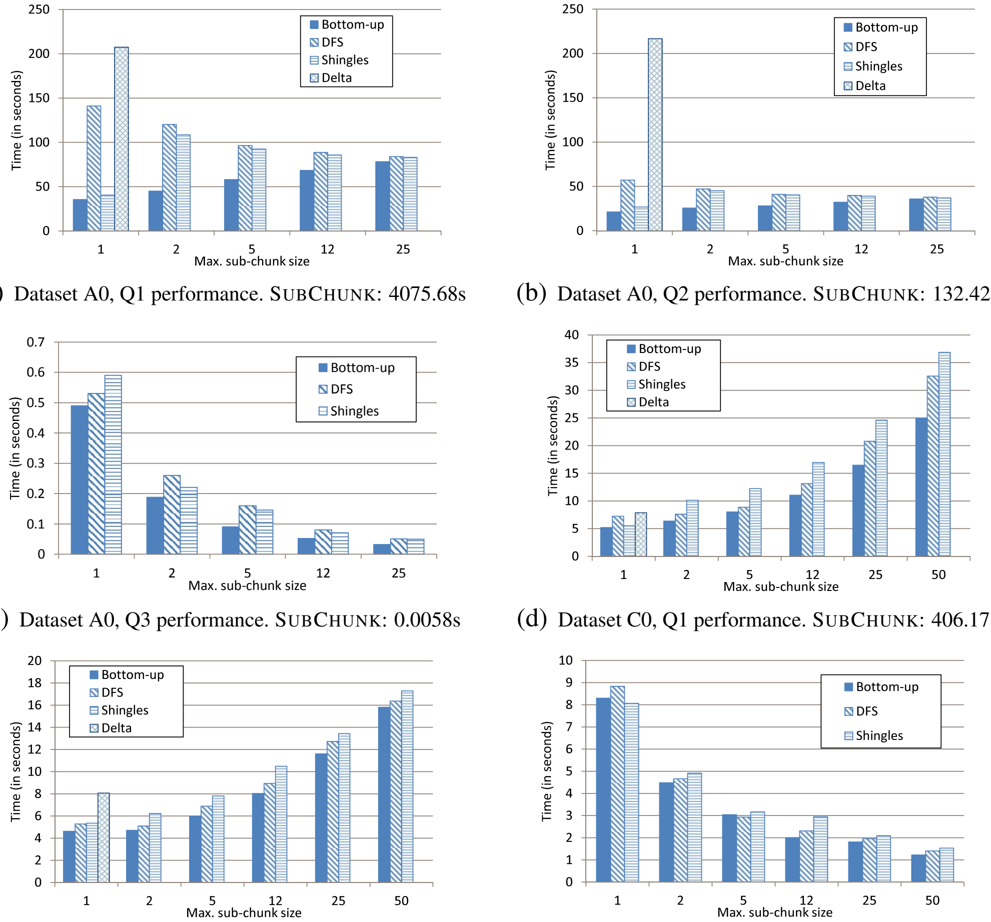
(f) Dataset C0, $P_d = 1\%$

Figure 5.10: Partitioning quality and compression ratios as sub-chunk size is varied for different algorithms

5.7.3 Effect of Compression on Partitioning

We now attempt to understand the performance of the partitioning algorithms on the compressed representation (Fig. 5.10). The degree of compression in the datasets is affected by two factors: (i) the number of records or the size of the sub-chunk, (ii) the amount of relative difference introduced between records due to updates. We simulate the second factor by generating the datasets such that when a record is updated, the amount of change w.r.t to the parent record is limited by a certain percentage, denoted by P_d. For a given version tree, we generate three datasets by limiting the change to 10%, 5% and 1%. For each such dataset, we vary the sizes of the sub-chunks (X-axis) and measure the total version span (Y-axis) at each sub-chunk value. We also plot the compression ratio (parallel Y-axis) of the dataset at every value of sub-chunk size. There are two factors that affect the total version span: (1) **Sub-chunk size**: As the number of records in each sub-chunk increases, the total version span increases due to a decrease in the number of records fetched per chunk. (2) **Compression Ratio**: Compressing the sub-chunks brings down the total number of chunks required to store the records. As a result, with increasing compression ratio the total version span is also expected to decrease. Note that we do not compare DELTA against these techniques as it is not possible to perform compression of records across multiple versions.

We observe that across all datasets, BOTTOM-UP has the best performance in terms of total version span. As P_d decreases, we note that the total version span for same sub-chunk values decreases across all partitioning techniques and across all datasets. For example consider dataset C0, Fig. 10(d), Fig. 10(e) and Fig. 10(f); the total version span at max sub-chunk size 50 decreases steadily with P_d across all the partitioning techniques. This is because Factor 2 outperforms Factor 1 stated above and results in an overall decrease in total version span. However if we just consider Fig. 10(d), we observe an increase in total version span with max sub-chunk size which can be attributed to Factor 1 which is dominant here. But as we increase the degree of compression in Fig. 10(e), the effect of Factor 2 helps in reducing the effect of Factor 1, resulting in an overall reduction in total version span compared to the previous figure. Finally in Fig. 10(f), Factor 2 dominates over Factor 1 as the total version span now decreases with an increase in max sub-chunk size. This behavior was observed for Dataset D0 and other datasets as well (not plotted). However this is not true for Dataset A which is a linear chain as opposed to a branched tree like in the previous case. This is because Factor 2 has a more dominant role over Factor 1 due to the compression ratios which is better for dataset A compared to the other datasets.



(a) Dataset A0, Q1 performance. SUBCHUNK: 4075.68s

(b) Dataset A0, Q2 performance. SUBCHUNK: 132.42s

(c) Dataset A0, Q3 performance. SUBCHUNK: 0.0058s

(d) Dataset C0, Q1 performance. SUBCHUNK: 406.17s

(e) Dataset C0, Q2 performance. SUBCHUNK: 107.23s

(f) Dataset C0, Q3 performance. SUBCHUNK: 0.0325s

Figure 5.11: Query Processing Performance

5.7.4 Query Processing Performance

In the following experiments (Fig. 5.11), we evaluate the query processing performance of BOTTOM-UP, DEPTHFIRST, SHINGLE and DELTA for three types of queries, namely, 1) Full Version Retrieval (Q1), 2) Partial Version Retrieval (Q2) and, 3) Record Evolution (Q3) on two different datasets. In all of these experiments we vary the max sub-chunk size from 1 to 50 and measure the total time taken to execute each of these queries against a randomly generated workload. Since intra-record compression is a limitation for DELTA, we restrict the DELTA experiment only to when the sub-chunk size is 1. We observe that BOTTOM-UP outperforms DEPTHFIRST, SHINGLE and DELTA in terms of the query performance for Q1 and Q2; the performance curve of Q2 is similar to that of Q1 as partial version span is loosely proportional to full version span. Also note that time taken by DELTA for Q2 is greater than Q1. This is because in the worst-case the full version is first reconstructed and then the required records are filtered.

Recall that we fetch all the records corresponding to a primary key for Q3. Therefore we observe that storage representations with increasing sub-chunk sizes lead to better query processing performances for Q3. For DELTA, we need to reconstruct all the versions that and then filter out the required records which ren-

Query Worload	Dataset	# nodes in cluster					
Avg. Version Span		1	2	4	8	12	16
Q1 (in secs.)	G	7.35	7.95	8.99	10.49	10.97	11.39
Avg. version span		507.99	559.49	622.88	702.92	710.24	702.21
Q3 (in secs.)	G	0.35	0.48	0.49	0.46	0.63	0.48
Avg. key span		21	32	34	33	46	34
Q1 (in secs.)	H	61.83	63.24	64.38	73.71	74.30	78.86
Avg. version span		400.24	436.48	451.20	554.92	561.60	594.92
Q3 (in secs.)	H	0.98	1.33	2.29	2.38	2.69	3.05
Avg. key span		6	9	16	18	21	24

Figure 5.12: Scalability Experiments

ders execution of Q3 impractical. We also report the query performance of SUBCHUNK technique against the caption of each query for each dataset. Although the full and partial version retrieval queries performs the worst for SUBCHUNK, it outperforms all other techniques for record evolution query.

5.7.5 Scalability of RSTORE

To demonstrate scalability of RSTORE, we ran a series of experiments where we doubled the cluster size starting at 1 up to 16, and then approximately double the amount of data by doubling the number of versions. We used two datasets specifically for this experiment, whose 16-node configurations were as follows: (a) **Dataset G**: total size of the unique records = 255 GB, with 10k versions having $\approx$ 50K records each (version size: $\sim$275 GB, total size: 2.6 TB), (c) **Dataset**

Batch	# of versions			
Size	250	500	750	1001
125	1.13	1.36	1.52	1.63
250	1.00	1.12	1.23	1.32
500	-	1.00	-	1.10

(a) Dataset B1

Batch	# of versions			
Size	2500	5000	7500	10001
1250	1.04	1.05	1.06	1.08
2500	1.00	1.004	1.001	1.018
5000	-	1.00	-	1.005

(b) Dataset C1

Figure 5.13: Online Partitioning Performance

H: size of the unique records = 280 GB, with 2k versions having *approx* 100K records each (version size: ~2.86 GB, total size: 5.76 TB). We partition the records using Bottom-Up approach. At each cluster configuration, we measure the full version retrieval times (partial version retrieval times showed similar behavior) and the record evolution times. As Fig. 5.12 shows, RStore exhibits good *weak* scalability, and is able to handle appropriate larger datasets with larger clusters; the increased query times are largely attributable to increased version or key spans. We also note that RStore currently processes the retrieved chunks sequentially while constructing the query result and cannot benefit from the increased parallelism; we are working on parallelizing the entire end-to-end process, which will result in further improvements in the query latencies.

5.7.6 Online Partitioning

In this experiment (Fig. 5.13), we measure the performance of the online partitioning algorithm under different batch sizes for two datasets using the BOTTOM-UP partitioning technique. To measure the partitioning quality at a given point, we compute the ratio of the total version span obtained by online partitioning using that batch size, to that obtained by running an offline version of BOTTOM-UP for the same number of versions. Overall, even with small batch sizes, we observe reasonable penalties, with the partitioning quality improving with an increase in batch size. Thus, online partitioning without repartitioning, combined with a full repartitioning periodically, presents a pragmatic approach to handling updates.

5.8 Conclusion

In this chapter, we designed and built a system, RSTORE, for managing a large number of versions and branches of a collection of keyed records in a distributed hosted environment, and systematically analyzed the different trade-offs therein. Our work is motivated by the popularity of key-value stores for storing large collections of keyed records or documents, the increasing trend towards maintaining histories of all changes that have been made to the data at a fine granularity, and the desire to collaboratively analyze and simultaneously modify or transform datasets. We showed that simple baseline approaches to adapting a key-value store to add versioning functionality suffer from serious limitations, and proposed a flexible and tunable framework intended to be used a layer on top of any key-value store. We also designed several novel algorithms for solving the key optimization problem of partitioning records into chunks. Through an extensive set of experiments, we validated our claims, design decisions, and our partitioning algorithms.

Chapter 6: Conclusion

Dataset version control system is an important data management tool that enable collaborations across multiple teams working with large amounts of data. In this dissertation, we have explored various storage layout designs for a wide spectrum of versioned datasets, used across a wide range of applications, that constitute an important problem in the design of a DVCS. We observe that each dataset present a unique storage layout challenge that prevents the application of techniques developed for other datasets.

For highly unstructured versions of datasets, the predominant form of data generated from various sources, we used delta compression to store the datasets in a compact manner that exploits the high overlap and duplication among datasets. However such compression leads to higher query latencies when retrieving specific datasets. We studied the trade-off between the storage and reconstruction cost in a principled manner, by formulating several optimization problems that trade-off the storage and reconstruction cost in different ways. We also proposed several algorithms for designing storage layouts that effectively explore the trade-off and present an extensive experimental evaluation that demonstrates the effectiveness of the algorithms. We also demonstrate that our algorithms outperform existing version control systems like Git and SVN thereby showing that they are not the best tools for managing large datasets.

Next, we considered array datasets generated mostly in scientific simulations, that predominantly consist of floating point data. We have developed a system PSTORE, that partitions the dataset along three dimensions – spatial, temporal and bytewise, to alleviate the dimension dependency while processing range queries and to enable compression of high entropy floating point data. Thereafter, we chose the best compression technique from a suite of compression schemes by analyzing a sample from the dataset. PSTORE also supports approximate query processing by retrieving partial precision data if that is sufficient for the application needs, and contains several other optimizations for efficient query execution. Our extensive experimental evaluation illustrates that different compression techniques work better for different datasets, and further that using bytewise partitioning and two-level chunking can lead to significantly higher compression ratios and lower query execution times respectively.

Finally, we focussed on dataset versions with keyed records and have designed and built a system, RSTORE, for managing such datasets in a distributed environment. Our work is motivated by the popularity of key-value stores for storing large collections of keyed records or documents, the increasing trend towards maintaining histories of all changes that have been made to the data at a fine granularity, and the desire to collaboratively analyze and simultaneously modify or transform datasets. We showed that simple baseline approaches to adapting a key-value store to add versioning functionality suffer from serious limitations, and proposed a flexible and tunable framework intended to be used a layer on top of any key-value store. We also designed several novel algorithms for solving the key optimization problem of partitioning records into chunks. We have also designed an online algorithm for partitioning the records as they enter the system. Through an extensive set of experiments, we validated our claims, design decisions, and our partitioning algorithms.

Bibliography

[1] Apache Cassandra. http://cassandra.apache.org.

[2] Couchbase. http://couchbase.com, .

[3] How to: Implement Document Versioning with Couchbase. https://blog.couchbase.com/how-implement-document-versioning-couchbase, . Accessed: February 12, 2017.

[4] Climate-Weather Research and Forecasting model. http://cwrf.umd.edu/.

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

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

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

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

[9] A Matter of Time: Temporal data management in DB2 10. https://www.ibm.com/developerworks/data/library/techarticle/dm-1204db2temporaldata/. Accessed: February 8, 2017.

[10] Large Synoptic Survey Telescope. http://www.lsst.org/.

[11] Lzo compression library. http://www.oberhumer.com/opensource/lzo/.

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

[13] MongoDB. http://mongodb.com/, .

[14] Vermongo: Simple Document Versioning with MongoDB. https://github.com/thiloplanz/v7files/wiki/Vermongo, . Accessed: February 12, 2017.

[15] Temporal Tables. https://msdn.microsoft.com/en-us/library/dn935015.aspx. Accessed: February 8, 2017.

[16] NetCDF: Network Common Data Form. http://www.unidata.ucar.edu/software/netcdf/.

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

[18] Zlib compression library. http://www.zlib.net.

[19] IEEE standard for floating-point arithmetic. *IEEE Std 754-2008*, pages 1–70, 2008.

[20] M. Adler and M. Mitzenmacher. Towards compressing web graphs. In *Data Compression Conference, (DCC)*, pages 203–212, 2001.

[21] P. Baumann, A. Dehmel, P. Furtado, R. Ritsch, and N. Widmann. The multidimensional database system rasdaman. In *ACM International Conference on Management of Data, (SIGMOD)*, pages 575–577, 1998.

[22] A. P. Bhardwaj, S. Bhattacherjee, A. Chavan, A. Deshpande, A. Elmore, S. Madden, and A. Parameswaran. DataHub: Collaborative Data Science & Dataset Version Management at Scale. *Conference on Innovative Database Research (CIDR)*, 2015.

[23] S. Bhattacherjee, A. Chavan, S. Huang, A. Deshpande, and A. G. Parameswaran. Principles of dataset versioning: Exploring the recreation/storage tradeoff. *Proceedings of the VLDB Endowment, (PVLDB)*, 8 (12):1346–1357, 2015.

[24] D. K. Blandford and G. E. Blelloch. Index compression through document reordering. In *Data Compression Conference (DCC)*, pages 342–351, 2002.

[25] P. Boldi and S. Vigna. The webgraph framework I: compression techniques. In *International conference on World Wide Web, (WWW)*, pages 595–602, 2004.

[26] P. Bourhis, J. L. Reutter, F. Suárez, and D. Vrgoč. JSON: Data model, query languages and schema specification. In *ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems, (PODS)*, pages 123–135, 2017.

[27] A. Broder. On the resemblance and containment of documents. In *Proceedings of the Compression and Complexity of Sequences SEQUENCES*, 1997.

[28] P. G. Brown. Overview of scidb: Large scale array storage, processing and analysis. In *ACM International Conference on Management of Data, (SIGMOD)*, pages 963–968, 2010.

[29] G. Buehrer and K. Chellapilla. A scalable pattern mining approach to web graph compression with communities. In *International Conference on Web Search and Web Data Mining, (WSDM)*, pages 95–106, 2008.

[30] P. Buneman, S. Khanna, K. Tajima, and W. C. Tan. Archiving scientific data. *ACM Trans. Database Syst.*, 29:2–42, 2004.

[31] R. C. Burns and D. D. Long. In-place reconstruction of delta compressed files. In *ACM Symposium on Principles of Distributed Computing, (PODC)*, pages 267–275, 1998.

[32] M. Burtscher and P. Ratanaworabhan. FPC: A high-speed compressor for double-precision floating-point data. *IEEE Trans. Computers*, 58(1):18–31, 2009.

[33] F. Chang, J. Dean, S. Ghemawat, W. C. Hsieh, D. A. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. Gruber. Bigtable: A distributed storage system for structured data. In *Symposium on Operating Systems Design and Implementation (OSDI)*, pages 205–218, 2006.

[34] S. Chien, V. J. Tsotras, C. Zaniolo, and D. Zhang. Supporting complex queries on multiversion XML documents. *ACM Transactions on Internet Technology, (TOIT)*, 2006.

[35] F. Chierichetti, R. Kumar, S. Lattanzi, M. Mitzenmacher, A. Panconesi, and P. Raghavan. On compressing social networks. In *ACM International Conference on Knowledge Discovery and Data Mining, (KDD)*, pages 219–228, 2009.

[36] J. Clifford, C. Dyreson, T. Isakowitz, C. S. Jensen, and R. T. Snodgrass. On the semantics of “now” in databases. *ACM Transactions on Database Systems (TODS)*, 22(2):171–214, 1997.

[37] G. Cobena, S. Abiteboul, and A. Marian. Detecting changes in XML documents. In *IEEE International Conference on Data Engineering, (ICDE)*, pages 41–52, 2002.

[38] V. Dhar. Data science and prediction. *Commun. ACM*, 56(12):64–73, 2013.

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

[40] T. Feder and R. Motwani. Clique partitions, graph compression, and speeding-up algorithms. In *ACM Symposium on Theory of Computing, (STOC)*, pages 123–133, 1991.

[41] A. E. Feldmann and L. Foschini. Balanced partitions of trees and applications. *Algorithmica*, 71(2):354–376, 2015.

[42] J. Finis, R. Brunel, A. Kemper, T. Neumann, F. Färber, and N. May. DeltaNI: an efficient labeling scheme for versioned hierarchical data. In *ACM International Conference on Management of Data,(SIGMOD)*, pages 905–916, 2013.

[43] S. Ghandeharizadeh, R. Hull, and D. Jacobs. Heraclitus: Elevating deltas to be first-class citizens in a database programming language. *ACM Trans. Database Syst. (TODS)*, 21(3):370–426, 1996.

[44] B. Goeman, H. Vandierendonck, and K. de Bosschere. Differential FCM: Increasing value prediction accuracy by improving table usage efficiency. In *International Symposium on High-Performance Computer Architecture (HPCA)*, pages 207–216, 2001.

[45] A. Y. Halevy, F. Korn, N. F. Noy, C. Olston, N. Polyzotis, S. Roy, and S. E. Whang. Goods: Organizing google's datasets. In *ACM International Conference on Management of Data, (SIGMOD)*, pages 795–806, 2016.

[46] Y. He, R. Lee, Y. Huai, Z. Shao, N. Jain, X. Zhang, and Z. Xu. RCFile: a fast and space-efficient data placement structure in MapReduce-based warehouse systems. In *IEEE International Conference on Data Engineering (ICDE)*, pages 1199–1208, 2011.

[47] K. Jansen, M. Karpinski, A. Lingas, and E. Seidel. Polynomial time approximation schemes for MAX-BISECTION on planar and geometric graphs. In *Annual Symposium on Theoretical Aspects of Computer Science (STACS)*, pages 365–375, 2001.

[48] J. Jenkins, E. R. Schendel, S. Lakshminarasimhan, D. A. B. II, T. Rogers, S. Ethier, R. B. Ross, S. Klasky, and N. F. Samatova. Byte-precision level of detail processing for variable precision analytics. In *Supercomputing Conference (SC)*, pages 48–58, 2012.

[49] G. Karypis and V. Kumar. hMETIS: A hypergraph partitioning package version 1.5.3.

[50] S. Khuller, B. Raghavachari, and N. Young. Balancing minimum spanning trees and shortest-path trees. *Algorithmica*, 14(4):305–321, 1995.

[51] U. Khurana and A. Deshpande. Efficient snapshot retrieval over historical graph data. In *IEEE International Conference on Data Engineering, (ICDE)*, pages 997–1008, 2013.

[52] P. Kulkarni, F. Douglis, J. D. LaVoie, and J. M. Tracey. Redundancy elimination within large collections of files. In *USENIX Annual Technical Conference, (ATC)*, pages 59–72, 2004.

[53] K. A. Kumar, A. Deshpande, and S. Khuller. Data placement and replica selection for improving co-location in distributed environments. *CoRR*, abs/1302.4168, 2013. URL `http://arxiv.org/abs/1302.4168`.

[54] X.-Z. Liang, M. Xu, X. Yuan, T. Ling, H. Choi, F. Zhang, L. Chen, S. Liu, S. Su, F. Qiao, J. Wang, K. Kunkel, E. J. W. Gao, V. Morris, T.-W. Yu, J. Dudhia, and J. Michalakes. Regional climate-weather research and forecasting model (CWRF). *Bull. Amer. Meteor. Soc.*, 2012.

[55] H. Liefke and D. Suciu. XMILL: an efficient compressor for XML data. In *ACM International Conference of Management of Data, (SIGMOD)*, pages 153–164, 2000.

[56] P. Lindstrom and M. Isenburg. Fast and efficient compression of floating-point data. *IEEE Trans. Vis. Comput. Graph.*, 12(5):1245–1250, 2006.

[57] D. Lomet, R. Barga, M. F. Mokbel, G. Shegalov, R. Wang, and Y. Zhu. Immortal DB: transaction time support for SQL server. In *ACM International Conference on Management of Data, (SIGMOD)*, pages 939–941, 2005.

[58] D. Lomet, M. Hong, R. Nehme, and R. Zhang. Transaction time indexing with version compression. *Proceedings of the VLDB Endowment (PVLDB)*, 1(1):870–881, 2008.

[59] D. B. Lomet, A. Fekete, R. Wang, and P. Ward. Multi-version concurrency via timestamp range conflict management. In *IEEE International Conference on Data Engineering, (ICDE)*, pages 714–725, 2012.

[60] P. Macko, V. J. Marathe, D. W. Margo, and M. I. Seltzer. LLAMA: efficient graph analytics using large multiversioned arrays. In *IEEE International Conference on Data Engineering, (ICDE)*, pages 363–374, 2015.

[61] M. Maddox, D. Goehring, A. J. Elmore, S. Madden, A. G. Parameswaran, and A. Deshpande. Decibel: The relational dataset branching system. *Proceedings of the VLDB Endowment, (PVLDB)*, 9(9):624–635, 2016.

[62] Y. Miao, W. Han, K. Li, M. Wu, F. Yang, L. Zhou, V. Prabhakaran, E. Chen, and W. Chen. Immortalgraph: A system for storage and analysis of temporal graphs. *ACM Transactions on Storage, (TOS)*, 11(3):14:1–14:34, 2015.

[63] E. J. Otoo, D. Rotem, and S. Seshadri. Optimal chunking of large multidimensional arrays for data warehousing. In *International Workshop On Data Warehousing and OLAP (DOLAP)*, pages 25–32, 2007.

[64] Z. Ouyang, N. Memon, T. Suel, and D. Trendafilov. Cluster-based delta compression of a collection of files. In *International Conference on Web Information Systems, (WISE)*, 2002.

[65] G. Özsoyoğlu and R. T. Snodgrass. Temporal and real-time databases: A survey. *IEEE Transactions on Knowledge and Data Engineering, (TKDE)*, 7(4):513–532, 1995.

[66] J. a. Paulo and J. Pereira. A survey and classification of storage deduplication systems. *ACM Comput. Surv.*, 47(1):11:1–11:30, June 2014.

[67] P. Pirzadeh, J. Tatemura, O. Po, and H. Hacigümüs. Performance evaluation of range queries in key value stores. *J. Grid Comput.*, 10(1):109–132, 2012.

[68] F. Provost and T. Fawcett. Data science and its relationship to big data and data-driven decision making. *Big Data*, 1(1):51–59, 2013.

[69] S. Quinlan and S. Dorward. Venti: A new approach to archival storage. In *USENIX Conference on File and Storage Technologies, (FAST)*, pages 89–101, 2002.

[70] B. Reiner, K. Hahn, G. Hofling, and P. Baumann. Hierarchical storage support and management for largescale multidimensional array database management systems. In *International Conference on Database and Expert Systems Applications (DEXA)*. Springer-Verlag, 2002.

[71] S. Sarawagi and M. Stonebraker. Efficient organization of large multidimensional arrays. In *IEEE International Conference on Data Engineering (ICDE)*, pages 328–336, 1994.

[72] Y. Sazeides and J. E. Smith. The predictability of data values. In *International Symposium on Microarchitecture (MICRO)*, pages 248–258, 1997.

[73] E. R. Schendel, Y. Jin, N. Shah, J. Chen, C.-S. Chang, S.-H. Ku, S. Ethier, S. Klasky, R. Latham, R. B. Ross, and N. F. Samatova. Isobar preconditioner for effective and high-throughput lossless data compression. In *IEEE International Conference on Data Engineering (ICDE)*, pages 138–149, 2012.

[74] A. Seering, P. Cudré-Mauroux, S. Madden, and M. Stonebraker. Efficient versioning for scientific array databases. In *IEEE International Conference on Data Engineering, (ICDE)*, pages 1013–1024, 2012.

[75] T. Shimada, T. Tsuji, and K. Higuchi. A storage scheme for multidimensional data alleviating dimension dependency. In *International Conference on Digital Information Management (ICDIM)*, pages 662–668, 2008.

[76] R. Snodgrass. The temporal query language TQuel. *ACM Trans. Database Syst. (TODS)*, 12(2):247–298, 1987.

[77] R. T. Snodgrass, S. Gomez, and L. E. McKenzie Jr. Aggregates in the temporal query language TQuel. *IEEE Transactions on Knowledge and Data Engineering, (TKDE)*, 5(5):826–842, 1993.

[78] E. Soroush and M. Balazinska. Time travel in a scientific array database. In *IEEE International Conference on Data Engineering, (ICDE)*, pages 98–109, 2013.

[79] E. Soroush, M. Balazinska, and D. L. Wang. Arraystore: a storage manager for complex parallel array processing. In *ACM International Conference of Management of Data, (SIGMOD)*, pages 253–264, 2011.

[80] M. Stonebraker, L. A. Rowe, and M. Hirohama. The implementation of POSTGRES. *IEEE Transactions on Knowledge & Data Engineering, (TKDE)*, (1):125–142, 1990.

[81] M. Stonebraker, C. Bear, U. Çetintemel, M. Cherniack, T. Ge, N. Hachem, S. Harizopoulos, J. Lifter, J. Rogers, and S. B. Zdonik. One size fits all? part 2: Benchmarking studies. In *Biennial Conference on Innovative Data Systems Research, (CIDR)*, pages 173–184, 2007.

[82] A. U. Tansel, J. Clifford, S. Gadia, S. Jajodia, A. Segev, and R. Snodgrass. *Temporal databases: theory, design, and implementation*. Benjamin-Cummings Publishing Co., Inc., 1993.

[83] R. E. Tarjan. Finding optimum branchings. *Networks*, 7(1):25–35, 1977.

[84] Y. Wang, D. J. DeWitt, and J. Cai. X-diff: An effective change detection algorithm for XML documents. In *IEEE International Conference on Data Engineering, (ICDE)*, pages 519–530, 2003.

[85] T. A. Welch. A technique for high-performance data compression. *Computer*, 17(6):8–19, 1984.

[86] H. Yan, S. Ding, and T. Suel. Inverted index compression and query processing with optimized document ordering. In *International Conference on World Wide Web, (WWW)*, pages 401–410, 2009.