

Managing and Querying Multi-versioned Documents using a Distributed Key-Value Store

Souvik Bhattacharjee, Amol Deshpande
University of Maryland, College Park
{bsouvik, amol}@cs.umd.edu

ABSTRACT

We address the problem of compactly storing a large number of versions (snapshots) of a collection of *keyed documents* or *records* in a distributed environment, while efficiently answering a variety of *retrieval* queries over those, including retrieving full or partial versions, and evolution histories for specific keys. We motivate the increasing need for such a system in a variety of application domains, carefully explore the design space for building such a system and the various storage-computation-retrieval trade-offs, and discuss how different storage layouts influence those trade-offs. We propose a novel system architecture that satisfies the key desiderata for such a system, and offers simple tuning knobs that allow adapting to a specific data and query workload. Our system is intended to act as a layer on top of a distributed key-value store that houses the raw data as well as any indexes. We design novel offline storage layout algorithms for efficiently partitioning the data to minimize the storage costs while keeping the retrieval costs low. We also present an online algorithm to handle new versions being added to system. Using extensive experiments on large datasets, we demonstrate that our system operates at the scale required in most practical scenarios and often outperforms standard baselines, including a delta-based storage engine, by orders-of-magnitude.

1. INTRODUCTION

The desire to derive valuable insights from large and diverse datasets produced in nearly all application domains today, has led to large collaborative efforts, often spanning multiple organizations. The iterative and exploratory nature of the data science process, combined with an increasing need to support debugging, historical queries, auditing, provenance, and reproducibility, means that a large number of *versions* of a dataset may need to be stored and queried. This realization has led to many efforts at building data management systems that support versioning as a first-class construct, both in academia [10, 4, 30, 23] and in industry (e.g., git, Datomic, noms). Unlike archival storage systems which also maintain large histories, these systems typically support rich versioning/branching functionality and, in some cases, complex queries over versioned information.

We motivate the design and development of our system using a concrete example from a real-life scenario.

EXAMPLE 1. 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. 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).

Similar requirements are beginning to arise in diverse application domains such as knowledge bases, content management systems, computation biology, and many others. Although there has been much work on version control systems in recent years, none of those prior systems are designed for hosting versions of a collection of keyed records or documents in a distributed environment or a cloud, while providing querying functionality similar to the wildly popular *key-value stores*. Key-value stores, a term loosely used here to describe any SQL/NoSQL system that supports key-based retrieval [11] (e.g., Apache Cassandra, HBase, MongoDB) are appealing in many collaborative scenarios spanning geographically distributed teams, since they offer centralized hosting of the

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to publish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

Copyright 20XX ACM X-XXXXX-XX-X/XX/XX ...\$15.00.

data, are resilient to failures, can easily scale out, and can handle a large number of queries efficiently. However, those do not offer rich versioning and branching functionality akin to hosted version control systems (VCS) like *GitHub*. The necessity of maintaining document versions have resulted in several quick and dirty extensions of systems like MongoDB and Couchbase, to satisfy immediate user needs [2, 1]. Unfortunately, the solutions presented there have several limitations and fail to provide any guarantees on the quality of the solution.

In this paper, our primary focus is to provide versioning and branching support for collections of records with unique identifiers that can act as *primary keys*. Like popular NoSQL systems, we aim to support a flexible data model; records with varying sizes, ranging from a few bytes to a few MBs; and a variety of retrieval queries to cover a wide range of use cases. Specifically, similar to NoSQL systems, we aim to support efficient retrieval of a specific record in a specific version (given a key and a version identifier), or the entire evolution history for a given key. Similar to VCS, we aim to support retrieving all the records belonging to a specific version to support use cases that require updating a large number of the records (e.g., by applying a data cleaning step). Finally, since retrieval of an entire version might be unnecessary and expensive, we also aim to support *partial* version retrieval given a range of keys and a version identifier. In addition, we aim to support efficient *ingest* (“commit”) of new versions from users, where the change from the previous version (“delta”) may be a small update to one record, or updates to a large subset of the records.

We begin with a careful exploration of the design space, outline the different trade-offs, and discuss the limitations of the baseline alternatives with respect to the desired requirements listed above. As observed in prior work (e.g., [4]), there is a natural trade-off between the storage requirements and the querying efficiency. However, the baseline approaches suffer from more fundamental limitations. (a) First, most of those approaches cannot directly support point queries targetting a specific record in a specific version (and by extension, full or partial version retrieval queries), without constructing and maintaining explicit indexes. (b) Second, all the viable baselines fundamentally require too many back-and-forths between the retrieval module and the backend key-value store; this is because the desired set of records cannot be succinctly described. (c) Third, ingest of new versions is difficult for most of the baseline approaches. (d) Finally, exploiting “record-level compression” is difficult or impossible in those approaches; this is crucial to be able to handle common use cases where large records (e.g., documents) are updated frequently with relatively small changes.

To address these problems, we investigate a new architecture that partitions the distinct records into approximately equal-sized “chunks”, with the goal to minimize the number of chunks that need to be retrieved for a given query workload. We show how the system can adapt to different data and workload requirements through a few simple tuning knobs. The key computational challenge boils down to deciding how to optimally partition the records into chunks; we draw connections to well-studied problems like compressing bipartite graphs and hypergraph partitioning to show that the problem is NP-Hard in general. We then present a novel algorithm, that exploits the structure of the version graph, to find an effective partitioning of the records. We have built a working prototype of our system, called RSTORE, on top of the Apache Cassandra key-value store. RSTORE can handle arbitrary types of records, including semi-structured (JSON/XML) documents, and text or binary files. We conduct an extensive experimental evaluation over a large number of synthetically constructed datasets to show the effectiveness of our system and to validate our design de-

isions.

Our key contributions can be summarized as follows: (1) We systematically explore the design space for supporting versioning as a first-class construct in distributed key-value stores; (2) We present a detailed analysis of the different trade-offs and how different baselines fare with respect to those; (3) We propose a flexible system architecture that supports the key desiderata through use of “chunking”; (4) We design novel partitioning algorithms that exploit how the versions relate to each other to identify good chunking strategies; (5) We present an online algorithm to keep the partitioning and the indexes up-to-date as new versions are committed. (6) We have built a working prototype 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.

2. SYSTEM DESIGN

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 provide a brief overview of the overall system architecture.

2.1 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 parallelly modified 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 $V = \{V_0, V_1, \dots, V_{n-1}\}$ 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**: $\langle \text{primarykey}, \text{version-id} \rangle$, 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 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.

- *Record Retrieval*: Analogous to a key-value store, a user/application

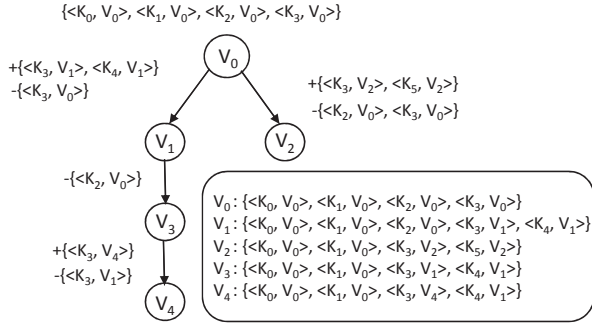


Figure 1: An Example Version Graph with 5 Versions

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 $\langle K, V \rangle$, 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 2. Fig. 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. 1), that must be consulted to identify the composite key to be retrieved ($\langle K_3, V_1 \rangle$ in this case).

2.2 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 on-line 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 1 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.

2.3 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.

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 extract the records is significantly less. This 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.

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

2.4 Architecture

Figure 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.

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 2, the basic unit of storage in the key-value store is 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 di-

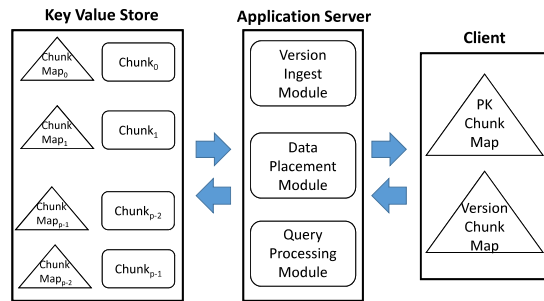


Figure 2: System Architecture

vided 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 3. Table below shows two different partitionings for the data from Example 2. 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.

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

Algorithms	Storage Space	Random Version (total data, #queries)	Point Query
Independent w/chunking	$nm_v s$	$m_v s, m_v s/s_c$	$s_c, 1$
DELTA	$m_v s + cd(n-1)m_v s$	$m_v s + cd(n-1)m_v s/2, n/2$	$m_v s + cd(n-1)m_v s/2, n/2$
SUBCHUNK	$m_v s + cd(n-1)m_v s$	$m_v(s + cd(n-1)s), m_v$	$s + cd(n-1)s, 1$
Single-address space	$m_v s + d(n-1)m_v s$	$m_v s, m_v s$	$s, 1$

Table 1: Comparing the different options for storing versioned records along different dimensions under some simplifying assumptions. 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.

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	$\{\langle K_3, V_4 \rangle\}$	$\{\langle K_3, V_4 \rangle\}$

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 3a 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 $\mathcal{M}(K_i, V_j, C_k) = V_i$ if a record with composite key $\langle K_i, V_i \rangle$ 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.

2.5 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 2.2, both the amount of data retrieved

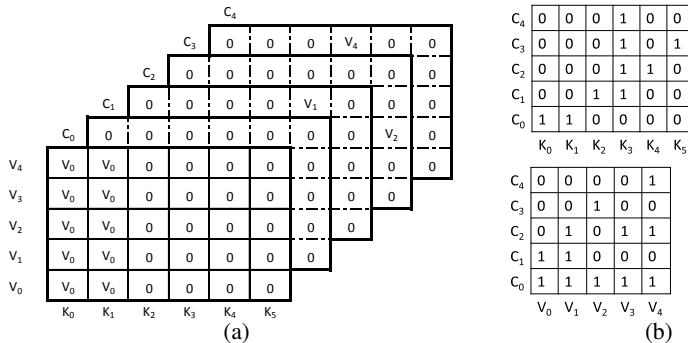


Figure 3: (a) Entire 3D mapping; (b) Lossy projections maintained as indexes

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 [4], we can formalize optimization problems in different manners. However, our fixed chunk size assumption simplifies the problem some-

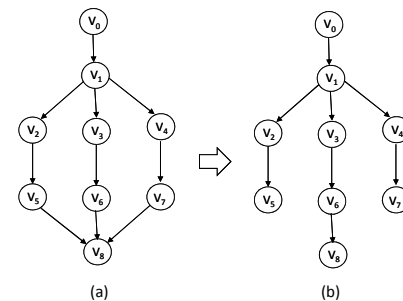


Figure 4: Converting a version DAG to a version tree

what 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 retrieval 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.

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 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 [18]. 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 [9]. 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 $\langle \text{chunk-id}, \text{record-key} \rangle$ 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 [5, 37], 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 [3, 6, 16].

2.6 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 2.2 and 2.3, 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.

3. PARTITIONING ALGORITHMS

In this section, we present three different algorithms to solve the partitioning problem formalized in Section 2.5. We begin with an adaptation of a standard *shingles*-based algorithm for finding bi-cliques, followed by two algorithms that exploit the inherent structure in the problem as embodied in the version graph.

3.1 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 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 [9]. 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, \dots, 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 1). 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 2).

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 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

Algorithm 1: 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

```
1 shingles[ $S_i$ ]  $\leftarrow$  {}  
2 for each  $h \in H$  do  
3 | shingles[ $S$ ]  $\leftarrow$  {shingles[ $S$ ],  $\min_{v \in V} h(v)$ }  
4 end  
5 return shingles
```

Algorithm 2: Shingle-based partitioning

Input : A set r of records, version graph G_t , chunk capacity C
Output : Set of chunks that partitions the records

```
1 // Traverse the versions, scan records, construct record to version map  
2 // Compute Shingles for each record  
3 for each  $r_i \in r$  do  
4 |  $\omega_i = \text{ComputeShingles}(r_i)$   
5 end  
6 // Sort the records based on shingle values( $\omega_i$ )  
7 Sort( $r$ )  
8 for each  $r_i \in r$  do  
9 | // Assign records to chunks  $C_j$  using the shingle-based sort-order  
10 | if  $C_j < C$  then  
11 | |  $C_j \leftarrow C_j \cup r_i$   
12 | end  
13 | else  
14 | | Create a new chunk and assign  $r_i$   
15 | end  
16 end
```

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 $O(m \log m + nm')$.

3.2 Bottom-Up Traversal

In this approach, we partition the records in the versions by traversing the version tree bottom-up¹. 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 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 $\pi_{i+1} = \{S_{i+1}^1, S_{i+1}^2, \dots, S_{i+1}^p\}$ contain the records that are returned by version V_{i+1} and denote the following:

S_{i+1}^1 : records present in V_{i+1} but not in any version below.
 S_{i+1}^2 : records present in V_{i+1}, V_{i+2} but not in any version below.
:
 S_{i+1}^p : records present in $V_{i+1}, V_{i+2}, \dots, V_{i+p}$.

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

S_i^1 : records present in V_i but not in any version below.
:
 S_i^p : records present in $V_i, V_{i+1}, \dots, V_{i+p}$.

¹The Bottom-Up algorithm is inspired by [25] that gives an algorithm for partitioning a graph into two equal-sized partitions. In general, partitioning even trees is NP-hard [19].

These sets are computed as follows:

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

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 [20], i.e., $\Delta_{ij}^+ \cap \Delta_{ij}^- = \phi$. The collection π_i can be expressed in terms of Δ as follows:

$$\begin{aligned} S_i^1 &= \Delta_{i,i+1}^-, & S_i^2 &= \Delta_{i+1,i+2}^- \setminus \Delta_{i,i+1} \\ &: & & \\ S_i^p &: V_n \setminus \bigcup_{j=0}^{p-1} \Delta_{i+j,i+(j+1)} \end{aligned}$$

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 $\psi_i = \{\alpha_i^1, \alpha_i^2, \dots, \alpha_i^p\}$. Thus we have,

$$\begin{aligned} \alpha_i^1 &= S_{i+1}^1 \setminus S_i^2 \text{ (records present only in } V_{i+1}) \\ &: \\ \alpha_i^p &= S_{i+1}^p \setminus S_i^p \text{ (records present in } V_{i+1}, V_{i+2}, \dots, V_{i+p}) \end{aligned}$$

LEMMA 1. *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 4. *Consider Fig. 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 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 1), 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 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.

3.2.1 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_v^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 3.

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 $O(nm')$.

3.3 Depth-First/Breadth-First Traversal

Algorithm 3: Bottom-Up Traversal for Partitioning

Input : Version graph G_t , root version V_r and deltas, sub-tree limit β , chunk capacity C

Output : Set of chunks that partitions the records

```

1 Bottom-Up ( $V_r$ )
2 return set of chunks
3 Bottom-Up ( $v$ ) {
4 if  $v$  is not null then
5   for each child  $\in v$  do
6     | Bottom-Up ( $child$ )
7   end
8   // process the sub-tree  $T_v$  rooted at  $v$ 
9   for each version  $V_j \in T_v$  do
10    | compute  $S_v^j$ 
11  end
12  // return set collection  $\pi_v$  to parent of  $v$ 
13  // compute the records exclusive to  $v$  and chunk them
14  // adjust sub-tree  $T_v$  if the size of sub-tree  $> \beta$ 
15 end
16 }
```

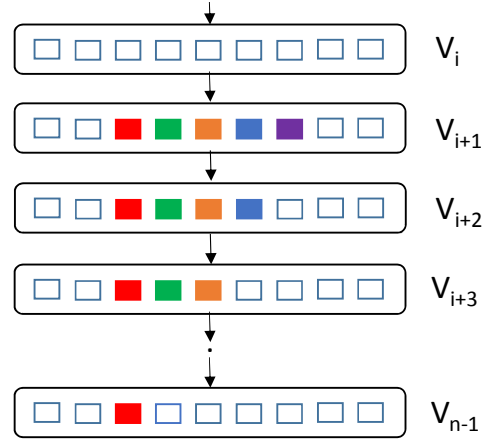


Figure 5: Bottom-Up Partitioning for Linear Chains

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

EXAMPLE 5. Consider the version tree in Fig. 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 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

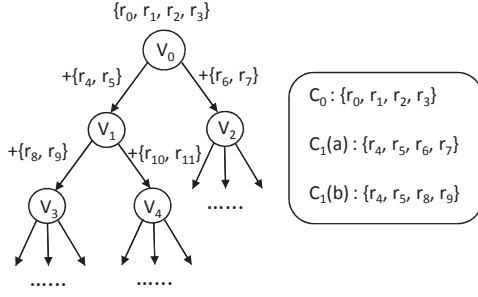


Figure 6: Version Tree Partitioning (using DFS)

Algorithm 4: 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

```

1 dfsStack  $\leftarrow$  {}
2 for each  $V_i \in G_t$  do
3   | visited[ $V_i$ ]  $\leftarrow$  false
4 end
5 push dfsStack,  $V_r$ 
6 while dfsStack is not empty do
7   |  $u \leftarrow$  peek dfsStack
8   | if  $u$  has child then
9     |  $v \leftarrow$  getNextChild( $u$ )
10    | if not visited[ $v$ ] then
11      | visited[ $v$ ]  $\leftarrow$  true
12      | push dfsStack,  $v$ 
13      | // read the delta and populate the chunk
14      | for each record  $r_i \in \Delta_{u,v}$  do
15        |  $C_i \leftarrow C_i \cup r_i$ 
16        | // if  $C_i$  is full then allocate a new chunk
17      end
18    end
19  end
20 else
21  | pop dfsStack
22 end
23 end
24 return set of chunks

```

turns out to be more beneficial than breadth-first approach. An outline of the depth-first partitioning algorithm is provided in Algorithm 4.

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 $O(nm')$.

3.4 Partitioning Compressed Records

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.,

Algorithm 5: 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

```

1 for each  $K_i \in \sigma(V_i)$  do
2   | if  $e(K_i) = 1$  then
3     | if  $s(K_i) = k - 1$  then
4       | construct sub-chunk.
5     end
6   else if  $s(K_i) \leq k - 2$  then
7     | construct an union of records; add to  $\Psi$ 
8   end
9   else
10    | construct sub-chunk out of the largest set. Repeat.
11  end
12 end
13 else
14   | if  $s(K_i) \leq k - 1$  then
15     | add the children with  $K_i$  to parent of  $V_i$ 
16   end
17   else
18     | construct sub-chunk out of the largest set. Repeat.
19   end
20 end
21 end

```

the versions that they belong to form a connected subgraph of the version tree. For example, in Figure 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 5). 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 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

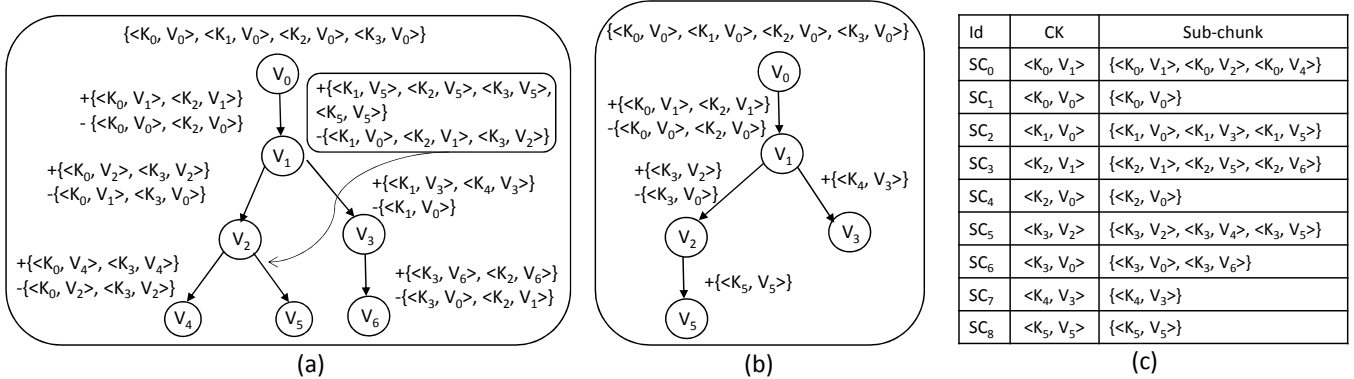


Figure 7: Partitioning compressed records: (a) Original Version Tree, (b) Transformed Version Tree, (c) Sub-chunk list with $k = 3$ – CK indicates composite keys of the sub-chunks

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 6. Fig. 7(a) represents the original version tree and Fig. 7(b) represents the transformed version tree. The sub-chunks corresponding to $k = 3$ are extracted from O_{VT} and are listed in Fig. 7(c) along with composite keys assigned to them. For deriving Fig. 7(b) from Fig. 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. 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 $O(nm')$.

4. 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. 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.

5.1 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 [4], 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 2 for a detailed description of the datasets.

5.2 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 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. 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, 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 3.3) 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. 9). As the size of the subtree decreases, the total version span increases as explained in Section 3.2.1. 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. 9 can be attributed to increased processing time for merging the nodes. As β decreases the number of nodes needed to merge also increases.

5.3 Effect of Compression on Partitioning

We now attempt to understand the performance of the partitioning algorithms on the compressed representation (Fig. 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. 10d, Fig. 10e and Fig. 10f; 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. 10d, 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. 10e, 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. 10f, 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.

5.4 Query Processing Performance

In the following experiments (Fig. 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 ver-

Dataset	#versions	Avg. depth	~#records/version	%update	Update Type	#unique records	Size of unique records (in GB)	Total size (in GB)
A0	300	300	100K	50	Random	12355366	11.9	31.67
A1	300	300	100K	5	Skewed	1510097	5.77	140.14
A2	300	300	100K	5	Random	1343434	5.14	141.26
B0	1001	293.5	100K	5	Skewed	4175023	8	192.24
B1	1001	293.5	100K	5	Random	4216366	8.07	193.77
B2	1001	293.5	100K	10	Random	8349864	8.02	195.69
C0	10001	143	20K	10	Random	16532342	15.95	196.46
C1	10001	143	20K	1	Random	1758517	1.69	193.01
C2	10001	143	20K	5	Skewed	8169026	7.87	193.05
D0	10002	94.4	20K	10	Random	16621314	16.03	196.48
D1	10002	94.4	20K	1	Random	1773281	1.71	193.07
D2	10002	94.4	20K	5	Skewed	8195193	7.90	193.09
E	10001	170	20K	10	Random	16524584	78.96	972.84
F	1001	56	100K	20	Random	16665072	79.64	981.11

Table 2: Description of the datasets used in experiments

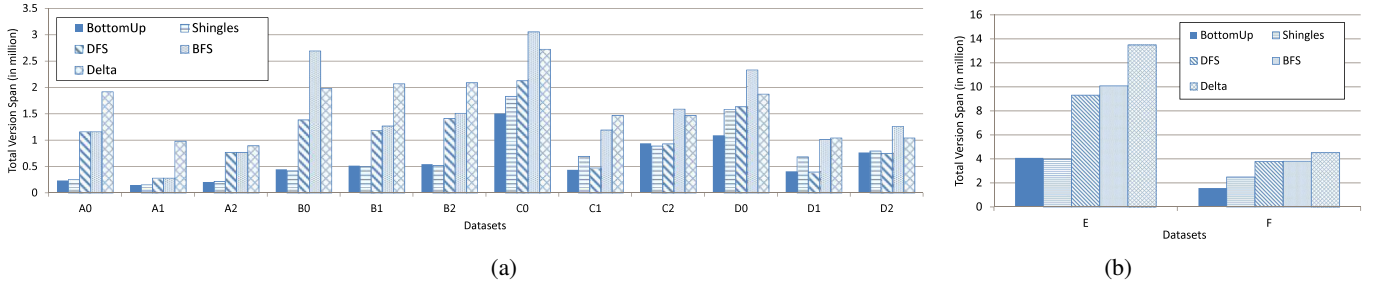


Figure 8: Comparison of Total Version Span (without compression)

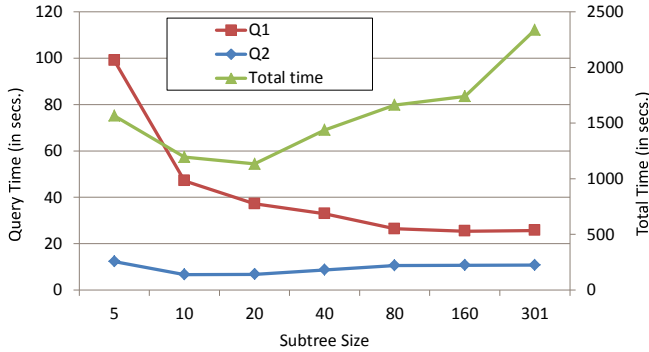


Figure 9: Effect of sub-tree size on performance of BOTTOM-UP (Dataset B0)

sions that and then filter out the required records which renders 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.

5.5 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: ~ 275 GB, total size: 2.6 TB), (c) **Dataset 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. 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.

5.6 Online Partitioning

In this experiment (Fig. 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.

6. RELATED WORK

Most cloud-based database 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 range queries [12, 33]; however it would

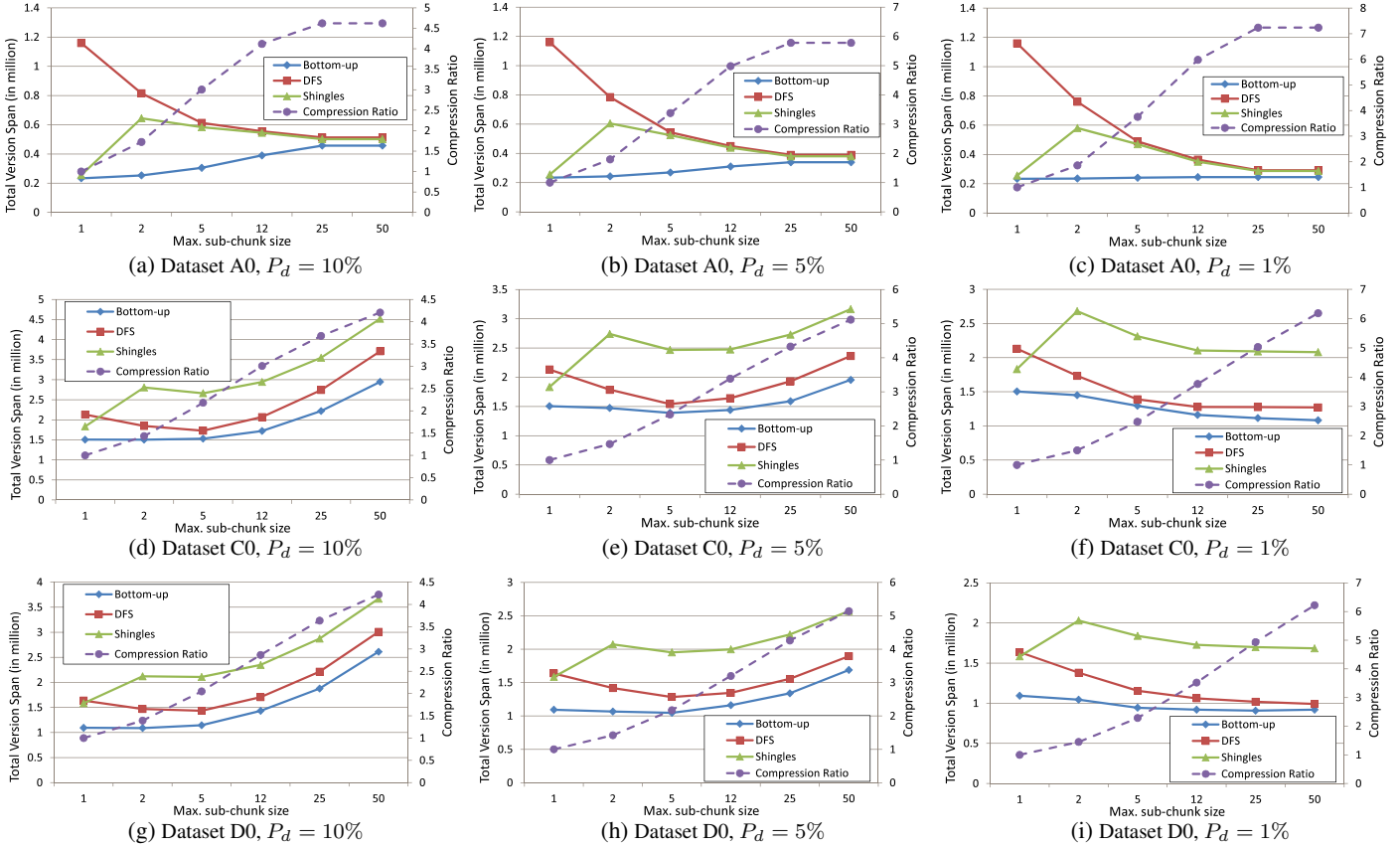


Figure 10: Partitioning quality and compression ratios as sub-chunk size is varied for different algorithms

be difficult for them to support range queries on versioned datasets in the absence of special indexes (Section 2.2). Although there is no full-fledged support for managing 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, [1, 2] describe how to implement versioning features in Couchbase and MongoDB. The techniques described are similar and advocate storing 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 different versions. A downside of this approach as described is that records cannot be updated in batches and older versions are more expensive to retrieve. Moreover it is not clear if they support compressing multiple versions of the same record together.

There has been significant work on workload-aware partitioning in recent years [27, 32, 17, 28], with several of those approaches mapping the problem to a hypergraph partitioning problem with data items (records) as vertices and queries as hyperedges. Conceptually, the problem we address is identical, with the query workload defined by the version retrieval queries. However, the sizes of the hyperedges for us are very large (since a version may contain millions of records) and those prior algorithms (which implicitly assume small hyperedges) cannot be used. Another key difference is that, our algorithms exploit the inherent structure in the version graph.

There has been prior work on providing versioning support and compactly storing graph and XML data. [10] proposed an archiv-

ing 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. The hierarchical data and the resulting archive is represented in XML format which enables use of 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. Moreover it does not provide support for range queries or record provenance queries that we support in our system. Finally their technique is not designed to work in a distributed setting that is essential for handling datasets that do not fit in a single machine.

There is extensive work on the temporal databases literature [7, 36, 35] that manages a linear chain of versions and support version retrieval at a specific point in time. There, a specific version of a record/tuple is associated with a time interval, whereas in versioned databases, it is associated with a set of version-ids. This seemingly small difference leads to fundamentally different challenges – e.g., whereas one could use an interval tree for indexing intervals optimally (e.g., to find all timestamps where a record is alive), doing the same for “sets” is considered nearly impossible [22]. An experimental evaluation in DEX [13] reveals that the techniques developed for linear chains [10] do not extend to branched version graphs. There also has been prior work on compressing XML data [29] and providing update and versioning schemes for XML through an edit-based schema [14]. [15] devises alternate storage schemes and provides support for complex queries on versioned XML documents. However the proposed techniques are single-

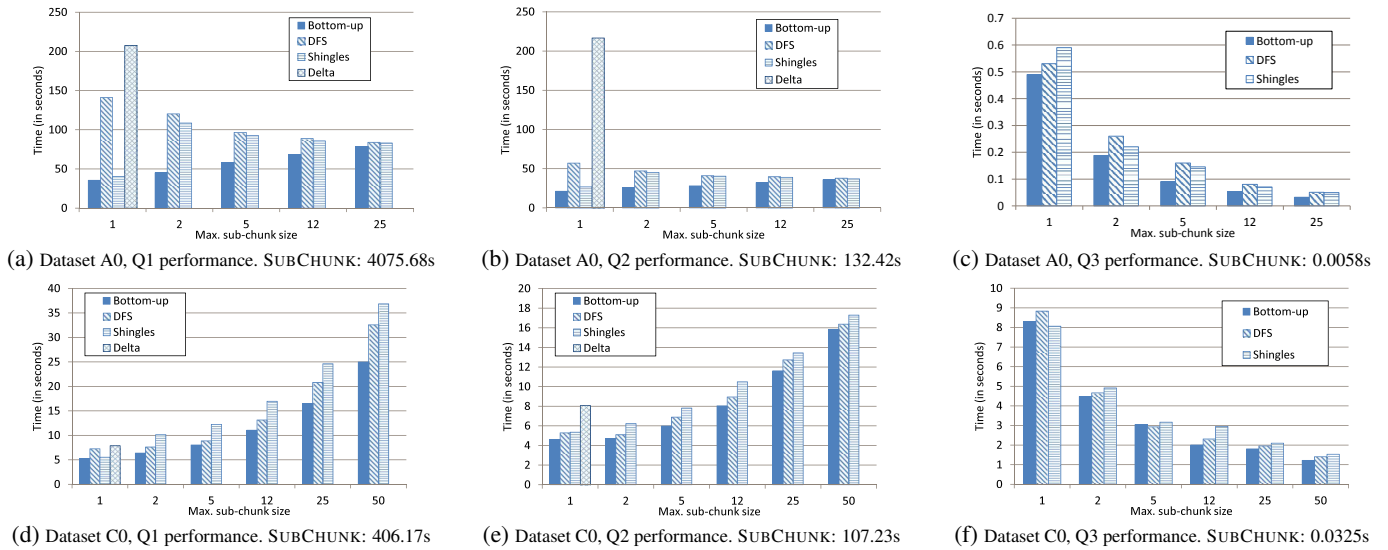


Figure 11: Query Processing Performance

Query Workload Avg. Version Span	Dataset	# nodes in cluster					
		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 12: Scalability Experiments

Batch Size	# of versions				Batch Size	# of versions			
	250	500	750	1001		2500	5000	7500	10001
125	1.13	1.36	1.52	1.63	1250	1.04	1.05	1.06	1.08
250	1.00	1.12	1.23	1.32	2500	1.00	1.004	1.001	1.018
500	-	1.00	-	1.10	5000	-	1.00	-	1.005

(a) Dataset B1

(b) Dataset C1

Figure 13: Online Partitioning Performance

node disk-based algorithms and are not designed to work in a distributed setting. There is work on developing a framework for incorporating temporal reasoning into RDF and studying the interplay between timestamp and snapshot semantics in temporal RDF graphs [21]. [26] present an approach for managing historical graph data for large information networks, and for executing snapshot retrieval queries on them.

Several version control systems geared towards handling different types of datasets have been recently developed, for unstructured files [4], relational databases [30, 24], arrays [31]. Our work can be seen as exploring a different design point in that space, with a focus on storing versions of a collection of semi-structured or unstructured records in the cloud and supporting efficient key-based access to them. [24] in particular proposes a partitioning scheme for minimizing version retrieval time by grouping versions into partitions and replicating records across them given a storage budget. Our approach of optimizing version retrieval cost does not consider replication of records across partitions. Further they do not support record compression grouped by keys to compress the data and then partitioning the compressed dataset.

Our approach is related to file content deduplication by indexing that employs hashes (or signatures) for identifying similar blocks of data [8, 34]. 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. However these techniques do not involve any sort of partitioning.

7. CONCLUSION

We designed and built a system 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. In future work, we plan to develop more sophisticated online algorithms that effectively re-partition the records as new records are committed into the system. We also aim to explore the effect of replication as it reduces the cost of version reconstruction but increases the cost of storing the versions.

8. REFERENCES

- [1] How to: Implement Document Versioning with Couchbase. <https://blog.couchbase.com/how-implement-document-versioning-couchbase>. Accessed: February 12, 2017.
- [2] Vermongo: Simple Document Versioning with MongoDB. <https://github.com/thiloplanz/v7files/wiki/Vermongo>. Accessed: February 12, 2017.
- [3] M. Adler and M. Mitzenmacher. Towards compressing web graphs. In *Data Compression Conference*, pages 203–212, 2001.

- [4] S. Bhattacharjee, A. Chavan, S. Huang, A. Deshpande, and A. G. Parameswaran. Principles of dataset versioning: Exploring the recreation/storage tradeoff. *PVLDB*, 8(12):1346–1357, 2015.
- [5] D. K. Blandford and G. E. Blelloch. Index compression through document reordering. In *Data Compression Conference*, 2002.
- [6] P. Boldi and S. Vigna. The webgraph framework I: compression techniques. In *WWW*, pages 595–602, 2004.
- [7] A. Bolour, T. L. Anderson, L. J. Dekeyser, and H. K. T. Wong. The role of time in information processing: a survey. *SIGMOD Rec.*, 1982.
- [8] A. Broder. On the resemblance and containment of documents. In *Proceedings of the Compression and Complexity of Sequences 1997*, 1997.
- [9] G. Buehrer and K. Chellapilla. A scalable pattern mining approach to web graph compression with communities. In *WSDM*, pages 95–106, 2008.
- [10] P. Buneman, S. Khanna, K. Tajima, and W. C. Tan. Archiving scientific data. *ACM Trans. Database Syst.*, 29:2–42, 2004.
- [11] R. Cattell. Scalable SQL and NoSQL data stores. *SIGMOD Rec.*, 39(4):12–27, May 2011.
- [12] 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 *OSDI*, 2006.
- [13] A. Chavan and A. Deshpande. DEX: query execution in a delta-based storage system. In *SIGMOD*, pages 171–186, 2017.
- [14] S. Chien, V. J. Tsotras, and C. Zaniolo. Efficient management of multiversion documents by object referencing. In *VLDB*, 2001.
- [15] S. Chien, V. J. Tsotras, C. Zaniolo, and D. Zhang. Supporting complex queries on multiversion XML documents. *ACM Trans. Internet Techn.*, 6(1):53–84, 2006.
- [16] F. Chierichetti, R. Kumar, S. Lattanzi, M. Mitzenmacher, A. Panconesi, and P. Raghavan. On compressing social networks. In *KDD*, 2009.
- [17] C. Curino, Y. Zhang, E. P. C. Jones, and S. Madden. Schism: a workload-driven approach to database replication and partitioning. *PVLDB*, 3(1):48–57, 2010.
- [18] T. Feder and R. Motwani. Clique partitions, graph compression, and speeding-up algorithms. In *STOC*, 1991.
- [19] A. E. Feldmann and L. Foschini. Balanced partitions of trees and applications. *Algorithmica*, 71(2):354–376, 2015.
- [20] S. Ghandeharizadeh, R. Hull, and D. Jacobs. Heraclitus: Elevating deltas to be first-class citizens in a database programming language. *TODS*, 21(3), 1996.
- [21] C. Gutierrez, C. A. Hurtado, and A. A. Vaisman. Introducing time into RDF. *IEEE Trans. Knowl. Data Eng.*, 19(2):207–218, 2007.
- [22] J. M. Hellerstein, E. Koutsoupias, and C. H. Papadimitriou. On the analysis of indexing schemes. In *PODS*, pages = 249–256, year = 1997,.
- [23] J. M. Hellerstein, V. Sreekanti, J. E. Gonzalez, J. Dalton, A. Dey, S. Nag, K. Ramachandran, S. Arora, A. Bhattacharyya, S. Das, M. Donsky, G. Fierro, C. She, C. Steinbach, V. Subramanian, and E. Sun. Ground: A data context service. In *CIDR*, 2017.
- [24] S. Huang, L. Xu, J. Liu, A. J. Elmore, and A. G. Parameswaran. OrpheusDB: Bolt-on versioning for relational databases. In *PVLDB*, 2017.
- [25] K. Jansen, M. Karpinski, A. Lingas, and E. Seidel. Polynomial time approximation schemes for MAX-BISECTION on planar and geometric graphs. In *STACS*, 2001.
- [26] U. Khurana and A. Deshpande. Efficient snapshot retrieval over historical graph data. In *ICDE*, 2013.
- [27] 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.
- [28] K. A. Kumar, A. Quamar, A. Deshpande, and S. Khuller. SWORD: workload-aware data placement and replica selection for cloud data management systems. *VLDB J.*, 23(6):845–870, 2014.
- [29] H. Liefke and D. Suciu. XMILL: an efficient compressor for XML data. In *SIGMOD*, 2000.
- [30] M. Maddox, D. Goehring, A. J. Elmore, S. Madden, A. G. Parameswaran, and A. Deshpande. Decibel: The relational dataset branching system. *PVLDB*, 2016.
- [31] H. Miao, A. Li, L. S. Davis, and A. Deshpande. Modelhub: Towards unified data and lifecycle management for deep learning. In *ICDE*, 2017.
- [32] A. Pavlo, C. Curino, and S. B. Zdonik. Skew-aware automatic database partitioning in shared-nothing, parallel OLTP systems. In *SIGMOD*, pages 61–72, 2012.
- [33] 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.
- [34] S. Quinlan and S. Dorward. Venti: A new approach to archival storage. In *FAST*, pages 89–101, 2002.
- [35] B. Salzberg and V. J. Tsotras. Comparison of access methods for time-evolving data. *ACM Computing Surveys (CSUR)*, 31(2):158–221, 1999.
- [36] R. Snodgrass and I. Ahn. A Taxonomy of Time in Databases. In *SIGMOD*, pages 236–246, 1985.
- [37] H. Yan, S. Ding, and T. Suel. Inverted index compression and query processing with optimized document ordering. In *WWW*, pages 401–410, 2009.