2018 IEEE 34th International Conference on Data Engineering

RSTORE: A Distributed Multi-version Document Store

Souvik Bhattacherjee, Amol Deshpande

University of Maryland, College Park
bsouvik@cs.umd.edu, 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 off-line 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.**

I. 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 need to 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 [3], [4], [5], [6] 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, 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 [7] (e.g., Apache Cassandra, HBase, MongoDB) are appealing in many collaborative scenarios spanning geographically distributed teams, since they offer centralized hosting of the 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

2375-026X/18/$31.00 ©2018 IEEE
DOI 10.1109/ICDE.2018.00043

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 [8], [9]. 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 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 succintly 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 bipartitite 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 decisions.

Our key contributions are as follows: (1) We systematically explore the design space for supporting versioning as a first-class construct in distributed key-value stores; (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 the similarities between versions; (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, called RSTORE, on top of Apache Cassandra that we use to empirically 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.

II. System Design

A. 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 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 parallely modified by a team of users in a collaborative fashion, resulting in a set of **versions** over time; each version is identified uniquely by a **version-id** (either an auto-incremented value, or *hashes* as in *git*). We assume there is a single *root* version of the dataset (which may be empty), from which all versions are derived.

Thus, a new version is derived from an existing version through an update 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 version V_i to version V_j by $\Delta_{i,j}$, referred to 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 as a directed *version graph* (Fig. 1).

Composite Keys. Since a record may be unchanged from one version to the next, to be able to refer to a specific version of a specific record, we use a **composite key**: ⟨primarykey,

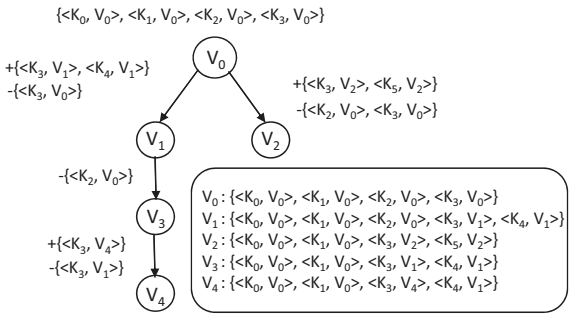


Fig. 1. An Example Version Graph with 5 Versions

version-id⟩; here 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 originating 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:* Analagous to a key-value store, a user/application may want to retrieve a record with a specific primary key K from a specific version V. Note that we cannot simply look for 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:* Retrieves 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 its current state; i.e., given a *primary key*, 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 $\langle K_2, V_0 \rangle$, and so on. 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).

B. Key Trade-Offs

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

- **Storage and compression.** There are two somewhat related issues here. First, ideally we only store a single copy of a record that appears in multiple versions; this however complicates the performance of version retrieval queries since the required records may be stored all over the place. Second, there may be only small differences between two different versions of a record, especially when records are large (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" 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 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.** The data structures used by the system should be easily updatable when new versions are added. This is, in general, difficult to achieve while guaranteeing good query performance. Ideally the cost of incorporating a new version should be 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 baselines 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

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

TABLE I

COMPARING DIFFERENT OPTIONS FOR STORING VERSIONED RECORDS UNDER SIMPLIFYING ASSUMPTIONS. n = # VERSIONS (ARRANGED IN A CHAIN); m_v = # RECORDS IN A VERSION (CONSTANT), d = % 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 QUERIES, THE TABLE SHOWS: AMOUNT OF DATA RETRIEVED, # QUERIES. WE ASSUME THE COST OF CONSULTING ANY INDEXES IS NEGLIGIBLE.

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.

- **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 I summarizes some of these trade-offs, by showing expressions for various different costs assuming a version with m_v records, with a sequence of changes each updating a fraction d of the records; thus the version graph is a "chain".

C. 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 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.

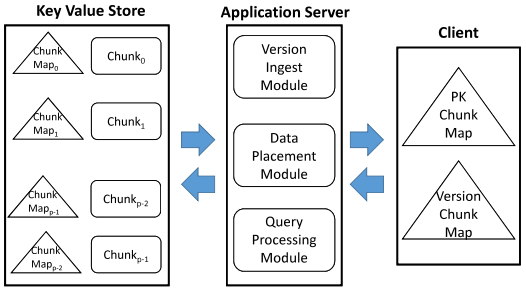


Fig. 2. System Architecture

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 records than required to recreate a version. However the overhead of retrieving the (fewer) chunks and scanning through them to extract the records is significantly less. This illustrates the 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

D. Architecture

Figure 2 shows the high-level architecture of our system. Next, we describe its primary components as well as the different design choices we made while building the system.

Backend Key-value Store: Our system is intended to act as a layer on an extant distributed key-value store (KVS), 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 KVS 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 divided into sub-chunks, each of which corresponds to records with the same primary key, 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 motivation behind chunking was to address the problem of too many queries.

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 AS and is returned to the user after the commit process is complete. Even if two versions committed are exactly the same, AS will generate different version-ids for the two different commits (to account for different users, times at which they are committed, etc.). AS only requires the "delta" from the previous (parent) version from the client; 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. Since updating the KVS and all the indexes for every new version would be impractical, the received deltas are kept in a separate storage area, that are processed as a batch by the data placement module.

Data Placement Module: This module is responsible for organizing the ingested data for efficient query processing, for placing them into appropriate chunks, and for constructing the required indexes. 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. As depicted in Fig. 2, we maintain two lossy projections or indexes for query answering, (i) version to chunk map ($\mathcal{I}_{VC}$): a mapping between versions and chunks that tells us which chunks contain records from a given version, (ii) primary key to chunk map ($\mathcal{I}_{PKC}$): a mapping between primary keys and chunks that tells us which chunks contain records for a given primary key. Query processing itself is straightforward given these indexes. For *version retrieval*, $\mathcal{I}_{VC}$ is consulted to identify which chunks need to be retrieved, and appropriate queries are issued in parallel to the KVS. The chunk maps are then used to extract the required records from the chunks. Record evolution queries proceed similarly, but use $\mathcal{I}_{PKC}$ instead. Range or single record retrieval utilizes both maps (analogous to "index-ANDing") to reduce the number of chunks that need to be brought in to AS. 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.

E. 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 II-B, both the amount of data retrieved and the number of chunks queried 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.

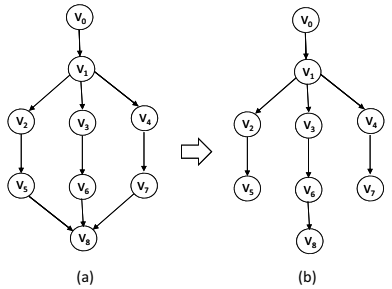


Fig. 3. Converting a version DAG to a version tree

Storage Cost. The storage cost is dominated by the sizes of the chunks; the different indexes required for query processing constitute a relatively small and largely fixed overhead. Due to the fixed chunk size assumption, we use the *number of chunks required* as a proxy for the total storage cost.

Retrieval Costs. For a query, let θ_i denote the total number of chunks that need to be queried (and 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) plus 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 2 different objectives here, analogously to [4], we can formalize optimization problems differently. However, the fixed chunk size assumption simplifies the problem somewhat if there is no compression.

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

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

Converting Version Graphs to Version Trees. Several of our proposed algorithms exploit the fact that versions that are close to each other in the version graph are more similar. Due to the inherent complexity of the problem, we assume that the version graphs have *no merges* (henceforth referred to as *version trees*). Figure 3 demonstrates how we deal 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 used to answer any queries afterwards.

F. 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. Any system that aims to solve the problem must contain four features: (1) exploit overlap across versions by not duplicating unchanged records, (2) support retrieving a specific record from a specific version through appropriate indexes, (3) solve too many queries problem, and (4) compress multiple versions of large records without compromising retrieval performance. Support for *range* queries does not obviate the need for any of these, because the list of chunks or sub-chunks that need to be retrieved for a query cannot be encapsulated as a range (see Figure 1). Efficient support for *large IN* queries reduces, but does not eliminate, the need for (3) – that problem instead shifts to the KVS since there will be too many queries between the server that is collecting the query answer and the backend servers that host the data. 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.

III. Partitioning Algorithms

A. Shingles-based Partitioning

To minimize the average number of chunks that a version is spread across, records that are common to a large number of versions should be placed together. This is equivalent to finding large *bi-cliques* in the version-record bipartite graph. This algorithm adapts a standard technique for finding bi-cliques based on *shingles* or *min-hashing*, which provide an estimate of the similarity between large sets [11]. Briefly, for each distinct record, we compute l min-hashes to summarize the set of versions that it belongs to, and use this sequence of min-hashes to sort the records in lexicographical fashion. This ordering places records whose version sets have high similarity (i.e., overlap) in close proximity to each other, and is then used to place the records into the chunks.

We also build the chunk maps 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 overall time complexity of this algorithm can be shown to be $O(nm' + ml \log ml)$, where n, m denote the number of versions and distinct records respectively, and m' denotes the average number of records per version.

B. Bottom-Up Traversal

In this approach, we partition the records in the versions by traversing the version tree bottom-up[1]. The key idea here is to identify and chunk records that do not belong to versions above as we move up through the versions in the version tree. For simplicity, we will first describe the approach for 1-ary version trees and then extend it to general trees. Let us consider a version V_i as depicted in Fig. 4 which needs to be processed. Since we follow a bottom-up approach, the versions below V_i in the version tree have already been processed. Let S_i denote the set of records in V_i. The collection of sets $\pi_{i+1} = \{S^1_{i+1}, S^2_{i+1}, \ldots, S^p_{i+1}\}$ contain the records that are returned by version V_{i+1} and denote the following:

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

Here p denotes the number of versions from the current version (in this case V_{i+1}) up to the leaf version. Similarly, V_i needs to return these sets to its parent V_{i-1}. In the present iteration, we compute the collection $\pi_i = \{S^1_i, S^2_i, \ldots, S^p_i\}$ as:

$S^1_i = S_i \setminus (S^2_i \cup S^3_i \ldots \cup S^p_i)$: in V_i but in no version below
$S^2_i = S^1_{i+1} \cap S_i$: in V_i, V_{i+1} but not in any version below.
$S^3_i = S^2_{i+1} \cap S_i$: in V_i, V_{i+1}, V_{i+1} but not in any version below.
:
S^k_i : records present in $V_i, V_{i+1}, \ldots, V_{i+k}$.

These sets can be directly computed from the deltas between versions. Specifically, a delta Δ between V_i and V_j can be split into two disjoint sets: Δ^+_{ij} denoting records that were added, and Δ^-_{ij} denoting records that were deleted (an update is treated as a delete followed by an insert). Assuming deltas are consistent [14], i.e., $\Delta^+_{ij} \cap \Delta^-_{ij} = \phi$, we have that:

$$S^1_i = \Delta^-_{i,i+1}, \qquad S^2_i = \Delta^-_{i+1,i+2} \setminus \Delta_{i,i+1}$$
$$S^p_i : V_n \setminus \bigcup_{j=0}^{p-1} \Delta_{i+j,i+(j+1)}$$

For general trees, computing π_i changes slightly only for versions which have more than one child, where S^1_i is the union of the Δ^- between version V_i and its children.

[1]The Bottom-Up algorithm is inspired by [12] that gives an algorithm for partitioning a graph into two equal-sized partitions. In general, partitioning even trees is NP-hard [13].

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 $\psi_i = \{\alpha^1_i, \alpha^2_i, \ldots, \alpha^p_i\}$. Thus we have,

$\alpha^1_i = S^1_{i+1} \setminus S^2_i$ (records present only in V_{i+1})
:
$\alpha^p_i = S^p_{i+1} \setminus S^p_i$ (records present in $V_{i+1}, V_{i+2}, \ldots, V_{i+p}$)

Lemma 1: Given a linear chain of versions, we have $\bigcap_{j=1}^{p} \alpha^j_i = \phi$, at any version i.

Since the records in α^1_i to α^p_i are not present in any version from V_i or above, we can chunk these records now. The records in set α^p_i must be chunked first, followed by those in α^{p-1}_i and so on. This is because records in α^p_i belong to p consecutive versions, followed by records in α^{p-1}_i 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 are merged at the end to reduce fragmentation.

Example 3: In Fig. 4, 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 box representing α^1_i, and blue box representing α^2_i and so on. Per our heuristic, records in red are chunked first, followed by the records in green box and so on.

For general trees, the primary difference lies in processing versions with more than one child. If V_i has λ children, then it may receive upto $\lambda \times p$ 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, we assign a count to every record based on the number of consecutive versions it belongs to, and use it to sort the records.

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. Due to space limitations, we refer the reader to [10] for a detailed explanation.

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 $O(n\beta m')$. Constructing chunks & chunk maps is $O(nm')$ as before.

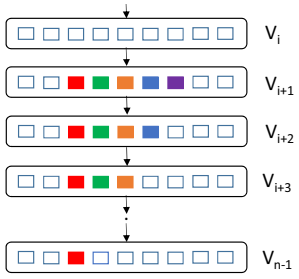


Fig. 4. Bottom-Up Partitioning for Linear Chains

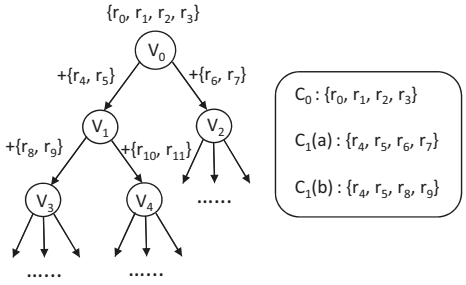


Fig. 5. Version Tree Partitioning, using BFS ($C_1(a)$) & DFS ($C_1(b)$)

C. Depth-First/Breadth-First Traversal

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

Example 4: Consider the version tree in Fig. 5, 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).

Complexity. The complexity of this algorithm, including chunk map construction, is $O(nm')$.

D. 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., the versions that they belong to form a connected subgraph of the version tree. This is done to increase potential compression (since records are likely to be more similar to their immediate ancestors/descedants) and to maintain the version tree semantics during transformation.

Due to lack of space, we sketch the algorithms here, and refer to [10] for details and examples.

Sub-chunk Creation. This algorithm traverses the original version tree bottom-up. At a given version, for a record r that was inserted or updated in that version, let a denote the total number of distinct records with the same primary key in the descendant versions that are *not already sub-chunked*. If $a+1 \geq k$, then we create one of more sub-chunks with those records and r while obeying the constraint above. If $a+1 < k$, then we create sub-chunk if the record r denotes an insert (i.e., not an update), otherwise we postpone the decision to an ancestor version.

Transformed Version Tree. Next, we construct the transformed version tree T_{VT} from the original tree O_{VT} by treating the sub-chunks as individual records. Each sub-chunk is assigned a representative composite key $\langle K_i, V_i \rangle$, where V_i is denotes the oldest version among the records in that sub-chunk, and the deltas are appropriately adjusted (see example below). These adjustments may lead to two versions becoming identical (either the delta from parent is empty, or it has the same delta from the parent as a sibling), and we remove such duplicate versions from the tree. Different values of k lead to different transformations of O_{VT}. The original partitioning algorithms can now be executed on this transformed dataset.

Complexity. The complexity of the sub-chunk construction algorithm, including creating a transformed version tree, can be shown to be is $O(nm' + m \log m)$ [10].

IV. 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.

V. Experiments

Next we present a comprehensive evaluation of RStore. We use a distributed installation of Cassandra across upto 16 dual-core 16GB memory nodes for storing the partitioned records and their associated indexes. The application server was hosted on a 2.2GHz Intel Xeon E5-2430 server with 64GB memory, running 64-bit RedHat Enterprise Linux 6.5.

A. 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 generate a set of modifications to it using the method outlined in [4], which closely follows real-life version graphs generated in a data science setting. 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 II for a detailed description of the datasets.

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

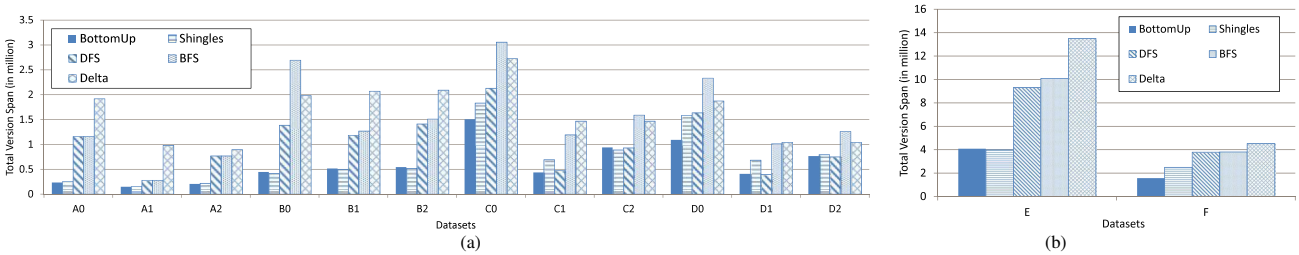
TABLE II
Description of datasets: 1) #V: #Versions, 2) AD: Average Depth, 3) RPV: ∼Records per Version, 4) %U: %updates, 5) UT: Update Type (R: Random, S: Skewed), 6) #UR: Unique Records (in Million), 7) URS: Size of Unique Records (in GB), 8) TS: Total size (in GB)

B. 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). We also omit the results of the Single Address Space technique due to the relatively high version retrieval times as a result of "too many queries" to the KVS.

In Fig. 6, 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 Sec. III-C) except for linear chains where they are identical.

Effect of Subtree size on performance. We vary the size of the subtree (β) for Bottom-Up and observe the total version span (Fig. 7). As the subtree size decreases, the total version span increases as expected (Sec. III-B). 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 $\beta < 20$ can be attributed to increased processing time for merging the nodes. As β decreases, the number of nodes needed to merge also increases.



(a) (b)

Fig. 6. Comparison of Total Version Span (without compression)

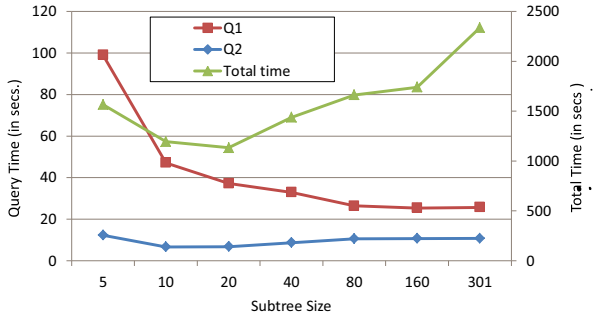


Fig. 7. Effect of sub-tree size on performance of BOTTOM-UP (Dataset B0)

C. Effect of Compression on Partitioning

We now attempt to understand the performance of the partitioning algorithms on the compressed representation (Table III). The degree of compression in the datasets is affected by two factors: (i) the number of records in the sub-chunk (i.e., its size), (ii) the amount of relative difference introduced between records due to updates. We simulate the second factor by generating 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 setting P_d to 10%, 5% and 1%. For each such dataset, we vary the sizes of the sub-chunks from $1 - 50$ and measure the total version span at each sub-chunk value. We also plot the compression ratio of the dataset at every value of sub-chunk size. There are two factors that affect the total version span: (1) **Sub-chunk size** (k): 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 against DELTA 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, the total version span for same sub-chunk values decreases across all partitioning techniques and across all datasets. For example consider dataset C0, 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 the values corresponding to $P_d = 10\%$ we observe an increase in total version span with k which can be attributed to Factor 1 which is dominant here. But as the degree of compression increases in $P_d = 5\%$ 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 $P_d = 1\%$, Factor 2 dominates Factor 1 as the total version span now decreases with an increase in k. This was observed for Dataset D0 and several other datasets [10]. However, for A0, which is a linear chain, Factor 2 has a higher influence due to better compression ratios.

D. Query Processing Performance

In the following experiments (Table IV), we evaluate the query processing performance of BOTTOM-UP, DEPTHFIRST, SHINGLE, DELTA and SUBCHUNK 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 k from 1 to 50 and measure the query execution time against a randomly generated workload. Since intra-record compression is a limitation for DELTA, we restrict the DELTA experiment only to when k = 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. 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 storage representations with increasing values of k results in better Q3 performance. For DELTA, we need to reconstruct all the versions that contain the queried primary key and then filter out the required records which renders execution of Q3 impractical. Note that the results for SUBCHUNK technique is independent of k as every version of a record (of a primary key) is stored in a sub-chunk. Although the full and partial version retrieval queries performs the worst for SUBCHUNK, it outperforms all other techniques for record evolution query.

E. 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

Dataset A0	$P_d = 10\%$ (total version span in million)						$P_d = 5\%$ (total version span in million)						$P_d = 1\%$ (total version span in million)					
	Max. sub-chunk size (k)						Max. sub-chunk size (k)						Max. sub-chunk size (k)					
	1	2	5	12	25	50	1	2	5	12	25	50	1	2	5	12	25	50
BOTTOM-UP	**0.23**	**0.25**	**0.31**	**0.39**	**0.46**	**0.46**	**0.23**	**0.24**	**0.27**	**0.31**	**0.34**	**0.34**	**0.23**	**0.24**	**0.24**	**0.24**	**0.25**	**0.25**
DEPTHFIRST	1.16	0.82	0.61	0.56	0.51	0.51	1.16	0.78	0.54	0.45	0.39	0.39	1.16	0.76	0.49	0.36	0.29	0.29
SHINGLE	0.26	0.64	0.58	0.55	0.50	0.50	0.26	0.60	0.52	0.44	0.38	0.38	0.26	0.58	0.47	0.35	0.29	0.29
COMP. RATIO	1.0	1.72	3.0	4.12	4.62	4.62	1.0	1.79	3.38	4.98	5.78	5.78	1.0	1.86	3.75	5.98	7.24	7.24
Dataset C0	1	2	5	12	25	50	1	2	5	12	25	50	1	2	5	12	25	50
BOTTOM-UP	**1.50**	**1.50**	**1.53**	**1.72**	**2.22**	**2.94**	**1.50**	**1.47**	**1.39**	**1.44**	**1.59**	**1.95**	**1.50**	**1.45**	**1.29**	**1.16**	**1.11**	**1.08**
DEPTHFIRST	2.13	1.84	1.73	2.07	2.74	3.70	2.13	1.79	1.54	1.64	1.93	2.36	2.13	1.73	1.39	1.28	1.28	1.27
SHINGLE	1.83	2.80	2.66	2.94	3.55	4.52	1.83	2.74	2.47	2.47	2.73	3.16	1.83	2.68	2.31	2.10	2.09	2.08
COMP. RATIO	1.0	1.43	2.18	3.0	3.68	4.21	1.0	1.47	2.33	3.39	4.32	5.12	1.0	1.50	2.47	3.77	5.03	6.19

TABLE III
PARTITIONING QUALITY AND COMPRESSION RATIOS AS SUB-CHUNK SIZE IS VARIED FOR DIFFERENT ALGORITHMS

Dataset A0	Q1 (query time in secs.)						Q2 (query time in secs.)						Q3 (query time in secs.)					
	Max. sub-chunk size (k)						Max. sub-chunk size (k)						Max. sub-chunk size (k)					
	1	2	5	12	25		1	2	5	12	25		1	2	5	12	25	
BOTTOM-UP	**35.5**	**45.06**	**57.99**	**68.43**	**78.28**		**21.26**	**25.62**	**28.03**	**32.16**	**35.86**		0.49	0.19	0.09	0.05	0.03	
DEPTHFIRST	141.2	120.32	96.64	88.78	84.14		57.12	47.16	41.18	39.78	37.94		0.53	0.26	0.16	0.08	0.05	
SHINGLE	40.53	108.67	92.56	85.89	83.12		26.87	45.29	40.45	38.98	37.12		0.59	0.22	0.15	0.07	0.05	
DELTA	207.51	-	-	-	-		216.68	-	-	-	-		-	-	-	-	-	
SUBCHUNK	4075.68						132.42						**0.0058**					
Dataset C0	1	2	5	12	25	50	1	2	5	12	25	50	1	2	5	12	25	50
BOTTOM-UP	5.2	**6.37**	**8.02**	**11.05**	**16.46**	**24.99**	**4.62**	**4.71**	**5.98**	**8.04**	**11.61**	**15.81**	8.30	4.48	3.04	2.00	1.81	1.22
DEPTHFIRST	7.26	7.63	8.89	13.13	20.8	32.59	5.29	5.10	6.89	8.94	12.73	16.37	8.83	4.67	2.93	2.31	1.96	1.40
SHINGLE	**4.93**	10.13	12.24	16.97	24.62	36.88	5.35	6.23	7.82	10.49	13.45	17.30	8.07	4.91	3.17	2.95	2.10	1.54
DELTA	7.87	-	-	-	-	-	8.07	-	-	-	-	-	-	-	-	-	-	-
SUBCHUNK	406.17						107.23						**0.03**					

TABLE IV
QUERY PROCESSING PERFORMANCE

Query Worload 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

Fig. 8. Scalability Experiments

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

(a) Dataset B1

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

(b) Dataset C1

Fig. 9. Online Partitioning Performance

by doubling the number of versions. We used two datasets specifically for this experiment, whose 16-node configurations were as follows: (a) **Dataset G**: size of the unique records = 255 GB, with 10K versions having $\approx$ 50K records each (version size: $\sim$275 GB, total size: 2.6 TB), (c) **Dataset H**: size of unique records = 280 GB, with 2K versions having $approx$ 100K records each (version size: $\sim$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. 8 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 (in the client) while constructing the query result and cannot benefit from the increased parallelism; support for parallel processing of retrieved chunks in the client will result in further improvements in query latencies.

F. Online Partitioning

In this experiment (Fig. 9), 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.

VI. RELATED WORK

Although there has been much work on NoSQL systems, to our knowledge, no existing system provides complete or systematic support for versioning. Recently, there have been several attempts at supporting *naive* forms of versioning using the existing APIs (e.g., [9], [8] 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 record

different versions. A downside of the approach as described is that records cannot be updated in batches and older versions are more expensive to retrieve. It is also not clear if they support compressing multiple versions of the same record.

There has been significant work on workload-aware partitioning in recent years [15], [16], [17], [18], 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. Our algorithms also exploit the inherent structure in the version graph.

There has been much prior work on versioning of XML, RDF and graph datasets [3], [19], [20], [21]. The focus of most of that work is on compactly representing (compressing) different versions of a document by merging them, and they can usually only support a linear (temporal) chain of versions. Further, that prior work has not looked at developing a distributed VCS that can support the range of retrieval queries that we consider here. Similarly, there is extensive work on temporal databases [22], [23], [24] that manages a linear version chain and supports "time-travel" queries. 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 [2]. An experimental evaluation in DEX [1] reveals that the techniques developed for linear chains [3] do not extend to branched version graphs.

Several version control systems geared towards handling different types of datasets have been recently developed, for unstructured files [4], relational databases [5], [25], arrays [26]. Our work explores a different design point in that space, with a focus on storing versions of a collection of semi-structured or unstructured records in a distributed setting and supporting efficient key-based access to them. Among these, OrpheusDB [25] also addresses somewhat similar partitioning issues; however, the trade-offs there are significantly different as they focus on a single-machine disk-based system.

There has also been much work on de-duplication in archival systems [27]; however, the focus of that work is on achiving high ingest rates and high compression, and those only support rudimentary retrieval queries.

Acknowledgement: This work was supported by NSF under grants IIS-1513972 and IIS-1650755. We thank the anonymous reviewers for their valuable feedback that helped us improve the paper.

VII. Conclusions

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 as 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.

References

[1] A. Chavan and A. Deshpande, "DEX: Query Execution in a Delta-based Storage System", *SIGMOD*, pp. 171–186, 2017.
[2] J. M. Hellerstein, E. Koutsoupias and C. H. Papadimitriou, "On the Analysis of Indexing Schemes", *PODS*, pp. 249–256, 1997.
[3] P. Buneman, S. Khanna, K. Tajima, and W. C. Tan, "Archiving scientific data," *ACM Trans. Database Syst.*, vol. 29, pp. 2–42, 2004.
[4] S. Bhattacherjee et al, "Principles of dataset versioning: Exploring the recreation/storage tradeoff," *PVLDB*, 2015.
[5] M. Maddox et al., "Decibel: The relational dataset branching system," *PVLDB*, 2016.
[6] J. Hellerstein et al. , "Ground: A data context service," in *CIDR*, 2017.
[7] R. Cattell, "Scalable SQL and NoSQL data stores," *SIGMOD Rec.*, vol. 39, no. 4, pp. 12–27, May 2011.
[8] "Vermongo: Simple Document Versioning with MongoDB," https://github.com/thiloplanz/v7files/wiki/Vermongo.
[9] "How to: Implement Document Versioning with Couchbase," https://blog.couchbase.com/how-implement-document-versioning-couchbase, accessed: February 12, 2017.
[10] S. Bhattacherjee, A. Deshpande, "RStore: A Distributed Multi-version Document Store," *CoRR*, abs/1802.07693, 2018.
[11] G. Buehrer and K. Chellapilla, "A scalable pattern mining approach to web graph compression with communities," in *WSDM*, 2008.
[12] K. Jansen et al., "Polynomial time approximation schemes for MAX-BISECTION on planar and geometric graphs," in *STACS*, 2001.
[13] A. E. Feldmann and L. Foschini, "Balanced partitions of trees and applications," *Algorithmica*, vol. 71, no. 2, pp. 354–376, 2015.
[14] S. Ghandeharizadeh et al., "Heraclitus: Elevating deltas to be first-class citizens in a database programming language," *TODS*, 1996.
[15] K. A. Kumar et al., "Data placement and replica selection for improving co-location in distributed environments," *CoRR*, abs/1302.4168, 2013.
[16] A. Pavlo, C. Curino, S. B. Zdonik, "Skew-aware automatic database partitioning in shared-nothing, parallel OLTP systems," *SIGMOD*, 2012.
[17] C. Curino et al., "Schism: a workload-driven approach to database replication and partitioning," *PVLDB*, 2010.
[18] K. A. Kumar et al., "SWORD: workload-aware data placement and replica selection for cloud data management systems," *VLDB J.*, 2014.
[19] S. Chien, V. J. Tsotras, C. Zaniolo, and D. Zhang, "Supporting complex queries on multiversion XML documents," *ACM TOIT*, 2006.
[20] C. Gutierrez, C. A. Hurtado, and A. A. Vaisman, "Introducing time into RDF," *IEEE Trans. Knowl. Data Eng.*, vol. 19, no. 2, pp. 207–218, 2007.
[21] U. Khurana and A. Deshpande, "Efficient snapshot retrieval over historical graph data," in *ICDE*, 2013.
[22] 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.
[23] R. Snodgrass and I. Ahn, "A Taxonomy of Time in Databases," in *SIGMOD*, 1985, pp. 236–246.
[24] B. Salzberg and V. J. Tsotras, "Comparison of access methods for time-evolving data," *ACM Computing Surveys (CSUR)*, 1999.
[25] S. Huang et al., "OrpheusDB: bolt-on versioning for relational databases," in *VLDB*, 2017.
[26] H. Miao, A. Li, L. S. Davis, A. Deshpande, "ModelHub: Towards unified data and lifecycle management for deep learning," in *ICDE*, 2017.
[27] J. Paulo and J. Pereira, "A survey and classification of storage deduplication systems," *ACM Comput. Surv.*, 2014.