Decibel: The Relational Dataset Branching System

Michael Maddox
MIT CSAIL
maddox@mit.edu

David Goehring
MIT CSAIL
dggoeh1@mit.edu

Aaron J. Elmore
University of Chicago
aelmore@cs.uchicago.edu

Samuel Madden
MIT CSAIL
madden@csail.mit.edu

Aditya Parameswaran
University of Illinois (UIUC)
adityagp@illinois.edu

Amol Deshpande
University of Maryland (UMD)
amol@cs.umd.edu

ABSTRACT

As scientific endeavors and data analysis become increasingly collaborative, there is a need for data management systems that natively support the *versioning* or *branching* of datasets to enable concurrent analysis, cleaning, integration, manipulation, or curation of data across teams of individuals. Common practice for sharing and collaborating on datasets involves creating or storing multiple copies of the dataset, one for each stage of analysis, with no provenance information tracking the relationships between these datasets. This results not only in wasted storage, but also makes it challenging to track and integrate modifications made by different users to the same dataset. In this paper, we introduce the Relational Dataset Branching System, Decibel, a new relational storage system with built-in version control designed to address these shortcomings. We present our initial design for Decibel and provide a thorough evaluation of three versioned storage engine designs that focus on efficient query processing with minimal storage overhead. We also develop an exhaustive benchmark to enable the rigorous testing of these and future versioned storage engine designs.

1. INTRODUCTION

With the rise of "data science", individuals increasingly find themselves working collaboratively to construct, curate, and manage shared datasets. Consider, for example, researchers in a social media company, such as Facebook or Twitter, working with a historical snapshot of the social graph. Different researchers may have different goals: one may be developing a textual analysis to annotate each user in the graph with ad keywords based on recent posts; another may be annotating edges in the graph with weights that estimate the strength of the relationship between pairs of users; a third may be cleaning the way that location names are attached to users because a particular version of the social media client inserted place names with improper capitalization. These operations may happen concurrently, and often analysts want to perform them on multiple snapshots of the database to measure the effectiveness of some algorithm or analysis. Ultimately, the results of some operations may need to be visible to all users, while others need not be shared with other users or merged back into the main database.

Existing mechanisms to coordinate these kinds of operations on shared databases are often ad hoc. For example, several computational biology groups we interviewed at MIT to motivate our work reported that the way they manage such shared repositories is to simply make a new copy of a dataset for each new project or group member. Conversations with colleagues in large companies suggest that practices there are not much better. This ad hoc coordination leads to a number of problems, including:

- Redundant copies of data, which wastes storage.
- No easy way for users to share updates to datasets with others or merge them into the "canonical" version of the dataset.
- No systematic way to record which version of a dataset was used for an experiment. Often, ad hoc directory structures or loosely-followed filename conventions are used instead.
- No easy way to share and track use of a particular dataset.

One potential solution to this problem is to use an existing distributed version control system such as git or mercurial. These tools, however, are not well-suited to versioning large datasets for several reasons. First, they generally require each user to "check-out" a separate, complete copy of a dataset, which is impractical within large, multi-gigabyte or terabyte-scale databases. Second, because they are designed to store unstructured data (text and arbitrary binary objects), they have to use general-purpose differencing tools (like Unix diff) to encode deltas and compare versions. Moreover, version control systems like these do not provide any of the high-level data management features (e.g., SQL, transactions) typically found in database systems, relational or otherwise.

In this paper, we address the above limitations via Decibel, a system for managing large collections of relational dataset versions. Decibel allows users to create working copies (*branches*) of a dataset based either off of the present state of a dataset or from prior versions. As in existing version control systems such as git, many such branches or working copies can co-exist, and branches may be *merged* periodically by users. Decibel also allows modifications across different branches, or within the same branch.

We describe our versioning API and the logical data model we adopt for versioned datasets, and then describe several alternative approaches for physically encoding the branching structure. *Choosing the right physical data layout is critical for achieving good performance and storage efficiency from a versioned data store.* Consider a naive physical design that stores each version in its entirety: if versions substantially overlap (which they generally will), such a scheme will be hugely wasteful of space. Moreover, data duplication could prove costly when performing cross-version operations like *diff* as it sacrifices the potential for shared computation.

In contrast, consider a *version-first* storage scheme which stores modifications made to each branch in a separate table fragment (which we physically store as a file) along with pointers to the table

fragments comprising the branch's direct ancestors. A linear chain of such fragments thus comprises the state of a branch. Since modifications to a branch are co-located within single files, it is easier to read the contents of a single branch or version by traversing its lineage. However, this structure makes it difficult to perform queries that compare versions, e.g., that ask *which versions* satisfy a certain property or contain a particular tuple [7].

As an alternative, we also consider a *tuple-first* scheme where every tuple that has ever existed in any version is stored in a single table, along with a bitmap to indicate the versions each tuple exists in. This approach is very efficient for queries that compare the contents of versions (because such queries can be supported through bitmap intersections), but can be inefficient for queries that read a single version since data from many versions is interleaved.

Finally, we propose a *hybrid* scheme that stores records in segmented files like in the version-first scheme, but also leverages a collection of bitmaps like those in the tuple-first scheme to track the version membership of records. For the operations we consider, this system performs as well or better than both schemes above, and also affords a natural parallelism across most query types.

For each of these schemes, we describe the algorithms required to implement key versioning operations, including version scans, version differencing, and version merging. *The key contribution of this paper is a thorough exploration of the trade-offs between these storage schemes across a variety of operations and workloads.* Besides describing the functionality of these schemes, this paper makes a number of other contributions:

- We provide the first full-fledged integration of modern version control ideas with relational databases. We describe our versioning API, our interpretation of versioning semantics within relational systems, and several implementations of a versioned relational storage engine.
- We describe a new versioning benchmark we have developed, modeled after several workloads we believe are representative of the use cases we envision. These workloads have different branching and merging structures, designed to stress different aspects of the storage managers.
- We provide an evaluation of our storage engines, showing that our proposed hybrid scheme outperforms the tuple-first and version-first schemes on our benchmark. We also compare against an implementation of database versioning in `git`.

Decibel is a key component of DataHub [4], a collaborative data analytics platform that we're building. DataHub includes the version control features provided by Decibel along with other features such as access control, account management, and built-in data science functionalities such as visualization, data cleaning, and integration. Our vision paper on DataHub [4] briefly alluded to the idea of version and tuple-first storage, but did not describe any details, implementation, or evaluation, and also did not describe the hybrid approach presented here (which, as we show, outperforms the other approaches significantly, sometimes by an order-of-magnitude or more.) Also in recent work [5], we presented algorithms to minimize the storage and recreation costs of a collection of unstructured datasets, as opposed to building and evaluating an end-to-end structured dataset version management system like Decibel.

We begin by presenting motivating examples, showing how end users could benefit from Decibel. We then provide an overview of our versioning API and data model in Section 2. A detailed overview of the aforementioned physical storage schemes is presented in Section 3. We then describe our versioned benchmarking strategy in Section 4 and the experimental evaluation of our storage models on a range of versioned query types in Section 5.

1.1 Versioning Patterns & Examples

We now describe two typical dataset versioning patterns that we have observed across a wide variety of scenarios in practice. These patterns are synthesized from our discussions with domain experts, and inspire the workloads we use to evaluate Decibel.

Science Pattern: This pattern is used by data scientist teams. These data scientists typically begin by taking the latest copy of an evolving dataset, then may perform normalization and cleaning, add features, and train models, all iteratively. At the same time, the underlying dataset that the data scientists started with may typically evolve, but often analysts will prefer to limit themselves to the subset of data available when analysis began. Using Decibel, such scientists and teams can create a private branch in which their analysis can be run without having to make a complete copy of the data. They can return to this branch when running a subsequent analysis, or create further branches to test and compare different cleaning or normalization strategies, or different models or features, while retaining the ability to return to previous versions of their work. This pattern applies to a variety of data science teams including *a*) The *ads team* of a startup, analyzing the impact of the ad campaigns on website visitors. *b*) A *physical scientist* team, building and testing models and physical theories on snapshots of large-scale simulation data. *c*) A *medical data analysis* team, analyzing patient care and medical inefficiencies and are only allowed to access records of patients who have explicitly agreed to such a study.

Curation Pattern: This pattern is used by teams collectively curating a structured dataset. While the canonical version of the dataset evolves in a linear chain, curators may work on editing, enhancing, or pruning portions of this dataset via branches, and then apply these fixes back to the canonical version. While this is cumbersome to do via current tools, Decibel can easily support multiple users simultaneously contributing changes to their branches, and then merging these changes back to the canonical version. This way, curators can "install and test" changes on branches without exposing partial changes to other curators or production teams using the canonical version until updates have been tested and validated. This is similar to feature branches in software version control systems. This pattern applies to a variety of data curation teams including *a*) The team managing the *product catalog* of a business with individuals who manage different product segments, applying updates to their portion of the catalog in tandem. *b*) A volunteer team of community users contributing changes to a *collaboratively managed map*, e.g. OpenStreetMaps, where individual users may focus on local regions, adding points of interest or fixing detailed geometry or metadata (e.g., one way information) of roads. *c*) A team of *botanists* collaboratively contributing to a dataset containing the canonical properties of plants found in a tropical rainforest.

2. Decibel API AND ARCHITECTURE

We begin with a brief overview of the Decibel architecture before describing the version control model and API that Decibel provides to enable branched multi-versioning of structured files or tables.

2.1 Architecture

Decibel is implemented in Java, on top of the MIT SimpleDB database. In this paper, we focus on the design of the Decibel storage engine, a new version-optimized data storage system supporting the core operations to scan, filter, difference, and merge branching datasets. Note, however, that Decibel does support general SQL query plans, but most of our query evaluation (joins, aggregates) is done in the (unmodified) SimpleDB query planning layer. The changes we made for Decibel were localized to the storage layer.
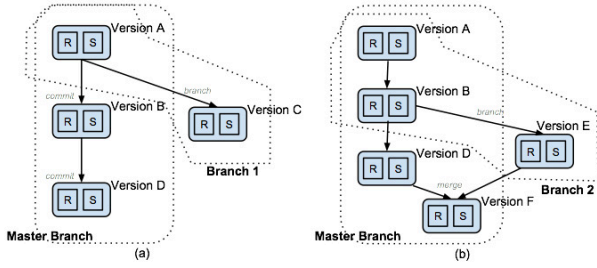
Figure 1: Two Example Workflows

Table 1: Sample Queries

Query Type	SQL Equivalent
1: Single version scan: find all tuples in relation R in version v01	`SELECT * FROM R` `WHERE R.Version = 'v01'`

2: Multi-version pos. diff: positive diff relation R between versions v01, v02	`SELECT * FROM R`
	`WHERE R.Version = 'v01' AND R.id`
	`NOT IN (SELECT id from R`
	`WHERE R.Version = 'v02')`

3: Multi-version join: join tuples in R in versions v01 and v02 satisfying Name = Sam	`SELECT * FROM R as R1, R as R2 WHERE`
	`R1.Version = 'v01' AND R1.Name = 'Sam'`
	`AND R1.id = R2.id AND R2.Version = 'v02'`

4: Several version scan: find all head versions of relation R	`SELECT * FROM R WHERE` `HEAD(R.Version) = true`

The storage layer reads in data from one of the storage schemes, storing pages in a fairly conventional buffer pool architecture (with 4 MB pages), exposing iterators over different single versions of data sets. The buffer pool also encompasses a lock manager used for concurrency control. In addition to this buffer pool we store an additional version graph on disk and in memory. We focus in this paper on the versioned storage manager and versioning data structures, with support for versioning operations in several different storage schemes, not the design of the query executor.

By implementing Decibel inside of a relational DBMS, we inherit many of their benefits. For example, fault tolerance and recovery can be done by employing standard write-ahead logging techniques on writes, and role-based access control primitives can be applied to different versions of the same table. We leave a complete exploration of these aspects of Decibel to future work.

2.2 Decibel Model and API

We first describe the logical data model that we use, and then describe the version control API, all in the context of Figure 1, where (a) and (b) depict two evolution patterns of a dataset.

2.2.1 Data Model.

Decibel uses a very flexible logical data model, where the main unit of storage is the *dataset*. A dataset is a collection of *relations*, each of which consists of a collection of *records*. Each relation in each dataset must have a well-defined primary key; the primary key is used to track records across different versions or branches, and thus is expected to be immutable (a change to the primary key attribute, in effect, creates a new record). For the same reason, primary keys should not be reused across semantically distinct records; however, we note that Decibel does not attempt to enforce either of these two properties.

2.2.2 Version Control Model

Decibel uses a version control model that is identical to that of software version control systems like `git`. As some readers may not be familiar with these systems, we now describe the model in the context of Decibel. In Decibel, a *version* consists of a point-in-time snapshot of one or more relations that are semantically grouped together into a dataset (in some sense, it is equivalent to the notion of a `commit` in git/svn). For instance, Versions A—D in Figure 1(a) all denote versions of a dataset that contain two relations, R and S. A version, identified by an ID, is immutable and any update to a version conceptually results in a new version with a different version ID (as we discuss later in depth, the physical data structures are not necessarily immutable and we would typically not want to copy all the data over, but rather maintain differences). New versions can also be created by merging two or more versions (e.g., Version F in Figure 1(b)), or through the application of transformation programs to one or more existing versions (e.g., Version B from Version A in Figure 1(a)). The version-level provenance that

captures these processes is maintained as a directed acyclic graph, called a *version graph*. For instance, the entire set of nodes and edges in Figure 1(a) or (b) comprises the version graph.

In Decibel, a *branch* denotes a working copy of a dataset. There is an active branch corresponding to every leaf node or version in the version graph. Logically, a branch is comprised of the history of versions that occur in the path from the branch leaf to the root of the version graph. For instance, in Figure 1(a) there are two branches, one corresponding to Version D and one corresponding to C. Similarly, in Figure 1(b) there is one branch corresponding to version F, and another branch corresponding to version E. The initial branch created is designated the *master* branch, which serves as the authoritative branch of record for the evolving dataset. Thus, a version can be seen as capturing a series of modifications to a branch, creating a point-in-time snapshot of a branch's content. The leaf version, i.e., the (chronologically) latest version in a branch is called its *head*; it is expected that most operations will occur on the heads of the branches. Although our current implementation does not support access control, we envision that each branch could have different access privileges for different users.

2.2.3 Decibel Operational Semantics

We now describe the semantics of the core operations of the version control workflow described above as implemented in Decibel. Although the core operations Decibel supports are superficially similar to operations supported by systems like `git`, they differ in several ways, including: i) Decibel supports centralized modifications to the data and needs to support both version control commands as well as data definition and manipulation commands; ii) unlike `git`-like systems that adopt an ordered, line-by-line semantics, Decibel treats a dataset as an unordered collection of records, where records are identified by primary keys; as a result, many operations take on different semantics (as described below); iii) unlike `git`-like systems that support a restricted, finite set of multi-version operations (specifically, those that have hard-coded implementations within these systems, e.g., *blame, status, diff, grep*), Decibel can support arbitrary declarative queries comparing multiple versions, enabling a class of operations that are very difficult in systems like `git`. We describe operations in the context of Figure 1(a).

Users interact with Decibel by opening a connection to the Decibel server, which creates a *session*. A session captures the user's state, i.e., the commit (or the branch) that the operations the user issues will read or modify. Concurrent transactions by multiple users on the same version (but different sessions) are isolated from each other through two-phase locking.

Init: The repository is initialized, i.e., the first version (Version A in the figure) is created, using a special *init* transaction that creates the two tables as well as populates them with initial data (if needed). At this point, there is only a single Master branch with a single version in it (which is also its head).

Commit and Checkout: *Commits* create new versions of datasets, adding an extra node to one of the existing branches in the version graph. Suppose a user increments the values of the second column by one for each record in relation R, then commits the change as Version B on the Master branch. This commit in Decibel creates a new logical snapshot of the table, and the second version in the master branch. Version B then becomes the new head of the Master branch. Any version (commit) on any branch may be *checked out*, which simply modifies the user's current session state to point to that version. Different users may read versions concurrently without interference. For example, after making a commit corresponding to Version B, any other user could check out Version A and thereby revert the state of the dataset back to that state within their own session. Versions also serve as logical checkpoints for branching operations as described below.

In Decibel, every modification conceptually results in a new version. In update-heavy environments, this could result in a large number of versions, most of which are unlikely to be of interest to the users as logical snapshots. Hence, rather than creating a new version that the user can check out after every update (which would add overhead as Decibel needs to maintain some metadata for each version that can be checked out), we allow users to designate some of these versions as being interesting, by explicitly issuing commits. This is standard practice in source code version control systems like git and svn. Only such committed versions can be checked out. Updates made as a part of a commit are issued as a part of a single transaction, such that they become atomically visible at the time the commit is made, and are rolled back if the client crashes or disconnects before committing. Commits are not allowed to non-head versions of branches, but a new branch can be made from any commit. Concurrent commits to a branch are prevented via the use of two-phase locking.

Branch: A new branch can be created based off of any version within any existing branch in the version graph using the *branch* command. Consider the two versions A and B in Figure 1(a); a user can create a new branch, *Branch 1* (giving it a name of their choice) based off of Version A of the master branch. After the branch, suppose a new record is added to relation S and the change is committed as Version C on Branch 1. Version C is now the head of Branch 1, and Branch 1's lineage or ancestry consists of Version C and Version A. Modifications made to Branch 1 are not visible to any ancestor or sibling branches, but will be visible to any later descendant branches. The new branch therefore starts a new line of development starting from Version C.

Merge: At certain points, we may *merge* two branches into a single branch, e.g., master and Branch 2 in Figure 1(b). The head commits in both branches (i.e., Versions D and E) are merged to create a new version (F). The merge operation needs to specify whether the merged version should be made the head of either or both of the branches, or whether a new branch should be created as a result of the operation (in the example, Version F is made the new head of the master branch). Decibel supports any user specified *conflict resolution* policy to merge changes when the same record or records have a field that changed across the branches that are being merged; by default in our initial implementation, non-overlapping field updates are auto-merged and for conflicting field updates, one branch is given precedence and is the authoritative version for each conflicting field (this is specified as part of the merge command).

The semantics of conflicts are different than those of a software version control system, where conflicts are at the text-line level within a file, which is similar to detecting tuple level conflicts. Decibel tracks conflicts at the field level. Specifically, two records

in Decibel are said to conflict if they (a) have the same primary key and (b) different field values. Additionally, a record that was deleted in one version and modified in the other will generate a conflict. Exploring conflict models and UI's for conflict resolution are rich research areas we plan to explore.

Difference: The operation *diff* is used to compare two dataset versions. Given versions A and B, diff will materialize two temporary tables: one representing the "positive difference" from A to B — the set of records in B but not in A, and one representing the "negative difference" — the records in B but not in A.

2.3 Versioned Query Language Overview

Decibel supports a versioning query language, termed VQuel, which was formally defined in our previous TaPP paper [7]. Some of the queries supported by Decibel and evaluated as part of our benchmark, described subsequently, are listed in Table 1. We omit the VQuel syntax and only provide the equivalent SQL queries. (Nothing we describe in the paper is tied to the choice of language.) These queries span a range of functionality, ranging from Query #2, which does a positive diff between versions v01 and v02 of relation R, i.e., finding all tuples in v01 but not in v02, to Query #4, which finds all versions at the head of branches, using a special function HEAD. For the corresponding VQuel queries see [16].

3. PHYSICAL REPRESENTATIONS

In this section, we explore several alternative physical representations of our versioned data store. Note that we depend on a version graph recording the relationships between the versions being available in memory in all approaches (this graph is updated and persisted on disk as a part of each branch or commit operation).

3.1 Overview

Our first approach, called *tuple-first*, stores tuples from all branches together in a single shared heap file. Although it might seem that the branch a tuple is active in could be encoded into a single value stored with the tuple, since tuples can be active in multiple branches, a single value insufficient. Instead, we employ a bitmap as our indexing structure to track which branch(es) each tuple belongs to. Bitmaps are space-efficient and can be quickly intersected for multi-branch operations.

There are two ways to implement tuple-first bitmaps, *tuple-oriented* or *branch-oriented*. In a tuple-oriented bitmap, we store T bitmaps, one per tuple, where the ith bit of bitmap T_j indicates whether tuple j is active in branch i. Since we assume that the number of records in a branch will greatly outnumber the number of branches, all rows (one for each tuple) in a tuple-oriented bitmap are stored together in a single block of memory. In branch-oriented bitmaps, we store B bitmaps, one per branch, where the ith bit of bitmap B_j indicates whether tuple i is active in branch j. Unlike in the tuple-oriented bitmap, since we expect comparatively few branches, each branch's bitmap is stored separately in its own block of memory in order to avoid the issue of needing to expand the entire bitmap when a single branch's bitmap overflows. Throughout this section, we describe any considerable implementation differences between the two where appropriate.

Figure 2 demonstrates the tuple-first approach with a set of tuples in a single heap file accompanied by a bitmap index indicating which tuples belong to one or more branches $A - E$. While tuple-first gives good performance for queries that scan multiple branches or that ask which branches some set of tuples are active in (for either tuple-oriented or branch-oriented variations), the performance of single branch scans can be poor as tuples in any branch may be fragmented across the shared heap file.
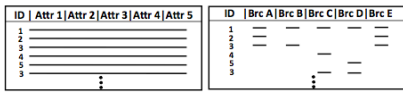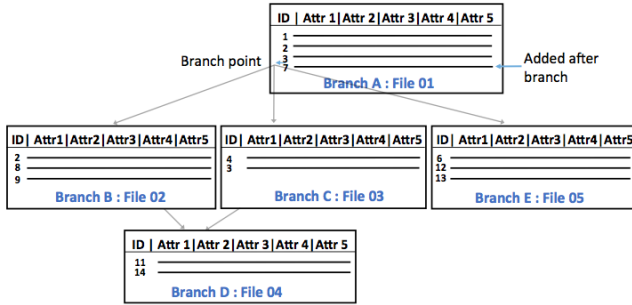
Figure 2: Example of Tuple-First

Figure 3: Example of Version-First (depicts branches resulting from merges, but could be commits)

Figure 4: Example of Hybrid

An alternative representation is the *version-first* approach. This approach stores modifications to each branch in a separate *segment file* for that branch. Each new child branch creates a new file with a pointer to the branch point in the ancestor's segment file; a collection of such segment files constitutes the full lineage for a branch. Any modifications to the new child branch are made in its own segment file. Modifications made to the ancestor branch will appear after the branch point in the ancestor's segment file to ensure this modification is not visible to any child branch. Ancestor files store tuples that may or may not be live in a child branch, depending on whether they been overwritten by a descendent branch. Figure 3 shows how each segment file stores tuples for its branch. This representation works well for single branch scans as data from a single branch is clustered within a lineage chain without interleaving data across multiple branches, but is inefficient when comparing several branches (e.g., diff), as complete scans of branches must be performed (as opposed to tuple-first, which can perform such operations efficiently using bitmaps.)

The third representation we consider is a *hybrid* of version- and tuple-first that leverages the improved data locality of version-first while inheriting the multi-branch scan performance of tuple-first. In hybrid, data is stored in fragmented files as in version-first. Unlike version-first, however, hybrid applies a bitmap index onto the versioned structure as a whole by maintaining local bitmap indexes for each of the fragmented heap files as well as a single, global bitmap index which maps versions to the segment files which contain data live in that version. The local bitmap index of a segment tracks the versions whose bits are set for that segment in the global bitmap index, indicating the segment contains records live in that version. This is distinct from tuple-first which must encode membership for every branch and every tuple in a single bitmap index. Figure 4 shows how each segment has an associated bitmap index indicating the descendent branches for which a tuple is active. We omit the index for single version segments for clarity.

3.2 Tuple-First Storage

Tuple-first stores tuples from different branches within a single shared heap file. Recall that this approach relies on a bitmap index with one bit per branch per tuple to annotate the branches a tuple is active in.

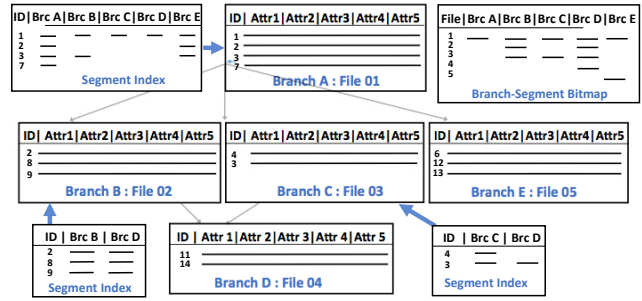**Branch:** A branch operation clones the state of the parent branch's bitmap and adds it to the index as the initial state of the child branch. A simple memory copy of the parent branch's bitmap can be performed. With a branch-oriented bitmap, this memory copy is straightforward; in tuple-oriented, however, the entire bitmap may need to be expanded (and copied) once a certain threshold of branches has been passed. This can be done via simple growth doubling, amortizing the branching cost.

Commit: A commit on a branch in tuple-first stores a copy of the bits representing the state of that branch at commit time. Since we assume that operations on historical commits will be less frequent that those on the head of a branch, we keep historical commit data out of the bitmap index, instead storing this information in separate, compressed *commit history* files for each branch. This file is encoded using a combination of delta and run length encoding (RLE) compression. When a commit is made, the delta from the prior commit (computed by doing an XOR of the two bitmaps) is RLE compressed and written to the end of the file. To checkout a commit (version), we deserialize all commit deltas linearly up to the commit of interest, performing an XOR on each of them in sequence to recreate the commit. To speed retrieval, we aggregate runs of deltas together into a higher "layer" of composite deltas so that the total number of chained deltas is reduced, at the cost of some extra space. There could potentially be several such layers, but our implementation uses only two as checkout performance was found to be adequate (taking a few hundred ms).

Data Modification: When a new record is added to a branch, a set bit is added to the bitmap indicating the presence of the new record. When a record is updated in a branch, the index bit of the previous version of the record is unset in that branch's bitmap to show that the record is no longer active; as with inserts, we also set the index bit for the new, updated copy of the record inserted at the end of the heap file. Similarly, deletes are performed by updating the bitmap index to indicate that this record is not active in the branch. Because commits result in snapshots of bitmaps being taken, deleted and updated records will still be visible when reading historical commits; as such, old records cannot be removed entirely from the system. To support efficient updates and deletes, we store a primary-key index indicating the most recent version of each primary key in each branch.

Tuple-oriented requires only that the new "row" in the bitmap for the inserted tuple be appended to the bitmap. However, in a branch-oriented bitmap, the backing array of the bitmap may occasionally need to be expanded via a doubling technique. Since each logical column of the bitmap is stored independently, overwriting the bounds of an existing branch's bitmap effectively requires only that logical column be expanded, not the bitmap as a whole.

Single-branch Scan: Often queries will only involve data for a single branch. To read all tuples in a branch in tuple-first, Decibel emits all records whose bit is set in that branch's bitmap. When the bitmap is branch-oriented, these bits are co-located in a single

bitmap; in tuple-oriented bitmaps, the bits for a given branch are spread across the bitmaps for each tuple. As such, resolving which tuples are live in a branch is much faster with a branch-oriented bitmap than with a tuple-oriented bitmap because in the latter case the entire bitmap must be scanned.

Multi-branch Scan: Queries that operate on multiple branches (e.g., select records in branch A and B, or in A but not B) first perform some logical operation on the bitmap index to extract a result set of records relevant to the query. Tuple-first enables shared computation in this situation as a multi-branch query can quickly emit which branches contain any tuple without needing to resolve deltas; this is naturally most efficient with a tuple-oriented bitmap. For example, if a query is calculating an average of some value per branch, the query executor makes a single pass on the heap file, emitting each tuple annotated with the branches it is active in.

Diff: Recall that diff(A,B) emits two iterators, indicating the modified records in A and B, respectively. Diff is straightforward to compute in tuple-first: we simply XOR bitmaps together and emit records on the appropriate output iterator.

Merge: To merge two branches in tuple-first, records that are in conflict between the merged branches are identified. If a tuple is active in both branches, then the new child branch will inherit this tuple. The same is true if the tuple is inactive in both of these branches. Otherwise, if a tuple is active in at least one, but not all of the parent branches, then we must check to see if this is a new record (i.e., no conflict), whether it was updated in one branch but not the other (again, no conflict), the fields updated in the branches do not overlap (no conflict), or if overlapping fields are modified in multiple branches (in which case there is a conflict).

At the start of the merge process, the *lca* commit is restored and a comparison of that bitmap column with the columns of the branches being merged can identify the tuples differing in each branch from the *lca* and the *lca* records that were updated in both branches. If a row in the bitmap is encountered where the *lca* commit is a 1 but both branches have a 0 in the same location, then it is known that the record has been updated in both branches, a conflict. Similarly, if the *lca* bit is 0 and either of other branch's bits are 1 then this reflects a record added after the *lca*. Using the bitmap this way reduces the amount of data that needs to be scanned from the *lca* when detecting and resolving conflicts. To find conflicts, we create two hash tables, one for each branch being merged. These tables contain the keys of records that occur in one branch but not the other; we join them as we scan, performing a pipelined hash join to identify keys modified in both branches. Specifically, we perform a diff to find modified records in each branch. For each record, we check to see if its key exists in the other branch's table. If it does, the record with this key has been modified in both branches and must be checked for conflict. To do so, we find the common ancestor tuple and do a three-way merge to identify if overlapping fields have been updated through field level comparisons. If the record is not in the other branch's table, we add it to the hash table for its branch. Conflicts can be sent to the user for resolution, or the user may specify that a given branch should take precedence (e.g., keep conflicting field values from A.) In this paper, we don't investigate conflict resolution policies in detail, and instead use precedence.

3.3 Version-First Storage

In version-first, each branch is represented by a head segment file storing local modifications to that branch along with a chain of parent head segment files from which it inherits records.

Branch: When a branch is created from an existing branch, we locate the current end of the parent segment file (via a byte offset) and create a *branch point*. A new child segment file is created that notes the parent file and the offset of this branch point. By recording offsets in this way, any tuples that appear in the parent segment after the branch point are isolated and not a part of the child branch. Any new tuples, or tuple modifications made in the child segment and are also isolated from the parent segment.

Commit: Version-first supports commits by mapping a commit ID to the byte offset of the latest record that is active in the committing branch's segment file. The mapping from commit IDs to offsets are stored in an external structure.

Data Modification: Tuple inserts and updates are appended to the end of the segment file for the updated branch. Updates are performed by inserting a new copy of the tuple with the same primary key and updated fields; branch scans will ignore the earlier copy of the tuple. Since there is no an explicit index structure to indicate branch containment for a record and since a branch cannot delete a record for historical reasons, deletes require a tombstone. Specifically, when a tuple is deleted, we insert a special record with a deleted header bit to indicate the key of the record that was deleted and when it was deleted.

Single-branch Scan: To perform branch scans, Decibel must report the records that are active in the branch being scanned, ignoring inserts, updates, and deletes in ancestor branches after the branch points in each ancestor. Note that the scanner cannot blindly emit records from ancestor segment files, as records that are modified in a child branch will result in two copies of the tuple: an old record from the ancestor segment (that is still active in the ancestor branch and any prior commit) and the updated record in the child segment. Therefore, the version-first scanner must be efficient in how it reads records as it traverses the ancestor files.

The presence of merges complicates how we perform a branch scan, so we first explain a scan with no merges in the version graph. Here, a branch has a simple linear ancestry of segment files back to the root of the segment tree. Thus, we can scan the segments in reverse order, ignoring records already seen, as those records have been overwritten or deleted by ancestor branch. Decibel uses an in-memory set to track emitted tuples. For example, in Figure 3 to scan branch D request that the segment for D be scanned first, followed by C, and lastly A up to the branch point. Each time we scan a record, that record is output (unless it is a delete) and added to the emitted tuple list (note that deleted records also need to be added to this emitted list). Merges require that the segments are scanned in a manner that resolves according to some conflict resolution policy, which is likely user driven. For example, on D the scan order could be $D - B - A - C$ or $D - B - C - A$.

Merges result in a segment files with multiple parent files. As a result, a given segment file can appear in multiple ancestor paths (e.g., if both parents branched off the same root). So that we do not scan the same file multiple times, version-first scans the version tree to determine the order in which it should read segment files.

Multi-branch Scan: The single branch scanner is efficient in that it scans every heap file in the lineage of the branch being scanned only once. The multi-branch case is more complex because each branch may have an ancestry unique to the branch or it may share some common ancestry with other branches being scanned. The unique part will only ever be scanned once. For the common part, a naive version-first multi-branch scanner would simply run the single branch scanner once per branch, but this could involve scanning the common ancestry multiple times.

A simple scheme that works in the absence of merges in the version graph is to topologically sort segment files in reverse order, such that segments are visited only when all of their children

have been scanned. The system then scans segments in this order, maintaining the same data for each branch being scanned as in single-version. This ensures that tuples that were overwritten in any child branch will have been seen when the parent is scanned.

Unfortunately, with merges the situation is not as simple, because two branches being scanned may need to traverse the same parents in different orders (e.g., branch C with parents A and B where B takes precedence over A, and D with parents A and B where A takes precedence over B). In this case, we do two passes over the segment files in the pertinent branches. In the first pass, we build in-memory hash tables that contain primary keys and segment file/offset pairs for each record in any of the branches being scanned. To construct these final hash tables, multiple intermediate hash tables are created, one for each portion of each segment file contained with any of the branches that is scanned. Each hash table is built by scanning the segment from the branch point backwards to the start of the next relevant branch point (so if two branches, A and B both are taken from a segment S, with A happening before B, there will be two such hash tables for S, one for the data from B's branch point to A's branch point, and one from A to the start of the file.) Then, for each branch, these in-memory tables can be scanned from leaf-to-root to determine the records that need to be output on each branch, just as in the single-branch scan. These output records are added to an output priority queue (sorted in record-id order), where each key has a bitmap indicating the branches it belongs to. Finally, the second pass over the segment files emits these records on the appropriate branch iterators.

Diff: Diff in version-first is straightforward, as the records that are different are exactly those that appear in the segment files after the lowest common ancestor version. Suppose two branches B_1 and B_2 branched from some commit C in segment file F_C; creating two segment files F_1 and F_2. Their difference is all of the records that appear F_1 and F_2. If B_1 branched from some commit C_1 and B_2 branched from a later commit C_2, then the difference is the contents of F_1 and F_2, plus the records in F_C between C_1 and C_2.

Merge: By default, merging involves creating a new branch, a new child segment, and branch points within each parent. In a simple precedence based model, where all the conflicting records from exactly one parent are taken and the conflicting records from the other are discarded, all that is required is to record the priority of parent branches so that future scans can visit the segments in the appropriate order, with no explicit scan required to identify conflicts. To allow the user to manually resolve conflicts or to use field level conflicts, we need to identify records modified in both branches from their lowest common ancestor. The approach uses the general multi-branch scanner (that can also scan specific commits) to collectively scan the *head* commits of the branches being merged and the lowest common ancestor commit. A scan of the lowest common ancestor commit is required for a field level merge of records that were updated in both branches.

We materialize the primary keys and segment file/offset pairs of the records in all three commits into in-memory hash tables, inserting every key. We perform a comparison on these hash tables to determine where the conflicts are. As in tuple-first, a value-based three-way diff is used to detect which fields have been updated. The resultant record is inserted into the new *head* segment, which must be scanned before either of its parents.

3.4 Hybrid Storage

Hybrid combines the two storage models presented above to obtain the benefits of both. It operates by managing a collection of *segments*, each consisting of a single heap file (as in version-first) accompanied by a bitmap-based *segment index* (as in tuple-first).

As described in Section 3.1, hybrid uses a collection of smaller bitmaps, one local to each segment. Each local bitmap index tracks only the set of branches which inherit records contained in that segment; this contrasts with the tuple-first model which stores liveness information for all records and all branches within a single bitmap. Additionally, a single *branch-segment bitmap*, external to all segments, relates a branch to the segments that contain at least one record alive in the branch. Bit-wise operations on this bitmap yield the set of segments containing records in any logical aggregate of branches. For example, to find the set of records represented in either of two branches, one need only consult the segments identified by the logical OR of the rows for those branches within this bitmap. This enables a scanner to skip segments with no active records and allows for parallelization of segment scanning.

As in the version-first scheme, this structure naturally co-locates records with common ancestry, but with the advantage that the bitmaps make it possible to efficiently perform operations across multiple branches (such as differences and unions) efficiently, as in the tuple-first scheme.

In hybrid, there exist two classes of segments: *head segments* and *internal segments*. Head segments track the evolution of the "working copy" of a single branch; fresh modifications to a branch are placed into that branch's head segment. Head segments become internal segments following a branch operation, at which point the contents of the segment become frozen, and after which only the segment's bitmap may change.

We now describe the details of how specific operations are performed in hybrid. Most operations are similar to tuple-first, but first involve locating the relevant segments for an operation on multiple branches and then applying the tuple-first procedure on each of the corresponding bitmaps for those segments. For brevity we discuss a few of the hybrid operations to give a sense of what is required for segment identification and modifying segment bitmaps, the other operations logically follow and can be found in [16].

Branch: The branch operation creates two new head segments that point to the prior parent head segment: one for the parent and one for the new child branch. The old head of the parent becomes an internal segment that contains records in both branches (note that its bitmap is expanded). These two new head segments are added as columns to the branch-segment bitmap, initially marked as present for only a single branch, while a new row is created for the new child branch (creation of the new head segments could, in principle, be delayed until a record is inserted or modified.) As in tuple-first, the creation of a new branch requires that all records live in the direct ancestor branch be marked as live in a new bitmap column for the branch being created. Unlike tuple-first, however, a branch in hybrid instead requires a bitmap scan be performed only for those records in the direct ancestry instead of on the entire bitmap.

Single-branch Scan: Single branch scans check the branch-segment index to identify the segments that need to be read for a branch. Thus, as in tuple-first, a segment read filters tuples based on the segment index to only include tuples that are active for the given branch. Due to the branch-segment index, the segments do not need to be scanned in a particular order.

Merge: As in tuple-first, the segment bitmaps can be leveraged (also requiring the lowest common ancestor commit) to determine where the conflicts are within the segment, the only difference now is identifying the new segments from the second parent that must track records for the branch it is being merged into. A conflict is output for records which have overlapping fields modified in at least one of the branches being merged. The original copies of these records in the common ancestry are then scanned to ob-
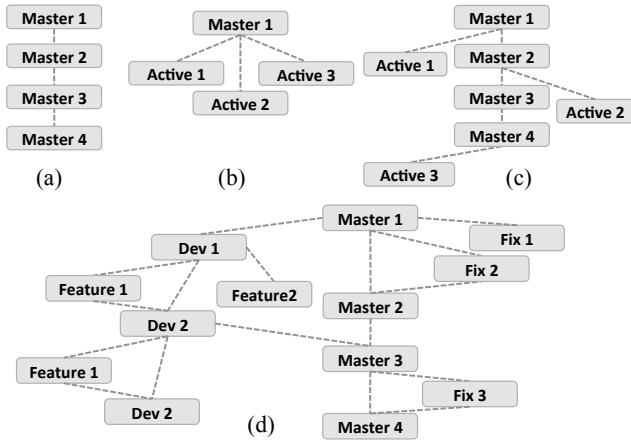
Figure 5: The various branching strategies in the versioning benchmark: a) Deep b) Flat c) Science (Sci.) d) Curation (Cur.)

tain their primary keys and thus the record identifiers of the updated copies within each branch being merged. A three-way diff determines if conflicting fields have been updated. Subsequently, any join operation may be used once the identifiers of conflicting records have been obtained. Once conflicts have been resolved, the records added into the child of the merge operation are marked as live in the child's bitmaps within its containing segments, creating new bitmaps for the child within a segment if necessary.

4. VERSIONING BENCHMARK

To evaluate Decibel, we developed a new versioning benchmark to measure the performance of our versioned storage systems on the key operations described above. The benchmark consists of four types of queries run on a synthetic versioned dataset, generated using one of four branching strategies. The benchmark is designed as a single-threaded client that loads and updates data according to branching strategy, and measures query latency.

4.1 Branching Strategies

Branches in these datasets are generated according to a branching strategy. The first two patterns, deep and flat, are not meant to be representative of real workloads, but instead serve as extreme cases to stress different characteristics of the storage engines. The remaining two patterns are modeled on typical branching strategies encountered in practice as described in Section 1.1. Figure 5 depicts these strategies with nodes as commits or branches.

Deep: This is a single, linear branch chain. Each branch is created from the end of the previous branch, and each branch has the same number of records. Here, once a branch is created, no further records are inserted to the parent branch. Thus, inserts and updates always occur in the branch that was created last. Single-version scans are performed on the tail, while multi-branch operations select the tail in addition to its parent or the head of the structure.

Flat: Flat is the opposite of deep. It creates many child branches from a single initial parent. Again, each branch has the same number of records. For single-version scans, we always select the newest branch, though this choice is arbitrary as all children are equivalent. For multi-branch operations, we use the single common ancestor branch plus one or more randomly-selected children.

Science: As in the data science pattern in Section 1.1, each new branch either starts from some commit of the master branch ("mainline"), or from the head of some existing active working branch.

This is meant to model a canonical (evolving) data set that different teams work off of. There are no merges. Each branch lives for a fixed lifetime, after which it stops being updated and is no longer considered active. All single-branch modifications go to either the end of an active branch or the end of mainline. Inserts may be optionally skewed in favor of mainline. Unless specified otherwise, single and multi-version scans select either the mainline, oldest active branch, or youngest active branch with equal probability.

Curation: As in the data curation pattern described in Section 1.1, there is one master data set (e.g., the current road network in OpenStreetMaps), that is on a mainline branch. Periodically "development" branches are created from the mainline branch. These development branches persist for a number of operations before being merged back into the mainline branch. Moreover, short-lived "feature" or "fix" branches may be created off the mainline or a development branch, and are eventually being merged back into their parents. Data modifications are done randomly across the heads of the mainline branch or any of the active development, fix, or feature branches (if they exist). Unless specified otherwise, single or multi-version scans randomly select amongst the mainline branch and the active development, fix, and feature branches (if they exist).

4.2 Data Generation and Loading

In our evaluation, generated data is first loaded and then queried. The datasets we generate consist of a configurable number of randomly generated integer columns, with a single integer primary key. We fix the record size (1KB), number of columns (250), page size (4 MB), and create commits at regular intervals (every 10,000 insert/update operations per branch). Our evaluation in Section 5 uses 4-byte columns; experiments were also run using 8-byte columns, but no differences were noticed. The benchmark uses a fixed mix of updates to existing records and inserts of new records in each branch (20% updates and 80% inserts by default in our experiments). For each branching strategy described earlier, we vary the dataset size and number of branches. Each experimental run uses the same number of branches. A parameter controls how to intersperse operations and branch creation. The benchmark also supports two loading modes, *clustered* and *interleaved*. In clustered mode, inserts into a particular branch are batched together before being flushed to disk. In our evaluation, we only consider the interleaved mode as we believe it more accurately represents the case of users making concurrent modifications to different branches. In interleaved mode, each insert is performed to a randomly selected branch in line with the selected branching strategy: for deep, only the tail branch accepts inserts; for flat, all child branches are selected uniformly at random; for the data science and data curation strategies, any active branch is selected uniformly at random (recall that those strategies may "retire" branches after a certain point). The benchmark additionally supports insert skew for non-uniform insertion patterns; our evaluation of the scientific strategy favors the mainline branch with a 2-to-1 skew, for example.

4.3 Evaluated Queries

The queries targeted in our benchmark are similar to those in Table 1; we summarize them briefly here.

Query 1: Scan and emit the active records in a single branch.

Query 2: Compute the difference between two branches, $B1$ and $B2$. Emit the records in $B1$ that do not appear in $B2$.

Query 3: Scan and emit the active records in a primary-key join of two branches, $B1$ and $B2$, that satisfy some predicate.

Query 4: A full dataset scan that emits all records in the head of any branch that satisfy a predicate. The output is a list of records annotated with their active branches.

Our benchmarking software, including a data generator and benchmark driver (based on YCSB [8]), is available at `http://datahub.csail.mit.edu/www/decibel`.

5. EVALUATION

In this section, we evaluate Decibel on the versioning benchmark. The goals of our evaluation are to compare the relative performance of the version-first, tuple-first, and hybrid storage schemes for the operations described in Section 4. We first examine how each of the models scales with the number of branches introduced to the system. Next, we examine relative performance across the query types described in Section 4.3 for a fixed number of branches. We then examine the performance of each model's commit and snapshot operations, as well as load times. Finally, we provide a brief comparison with a `git`-based implementation.

For tuple-first and hybrid, we use a branch-oriented bitmap due to its suitability for our commit procedure. We flush disk caches prior to each operation to eliminate the effects of OS page caching.

5.1 Scaling Branches

Here we examine how each storage model scales with the number of branches introduced into the version graph. We focus on deep and flat branching strategies as these patterns represent logical extremes to designed to highlight differences between the three designs. Moreover, we examine only Query 1 (scan one branch) and Query 4 (scan all branches) as these queries also represent two fundamental extremes of versioning operations.

Figure 6a shows how the storage models scale across structures with 10, 50, and 100 branches for Query 1 on the flat branching strategy. As tuple-first stores records from all versions into a single heap file, ordered by time of insertion, we see single-branch scan times for tuple-first greatly underperform both version-first and hybrid. Note that the latencies for version-first and hybrid decline here since the total data set size is fixed at 100GB, so each branch in the flat strategy contains less data as the number of branches is increased. On the other hand, tuple-first's performance deteriorates as the bitmap index gets larger. In contrast, Query 1 on the deep structure (not shown for space reasons) results in uniform latencies as expected (250 seconds ±10%) for each storage model and across 10, 50, and 100 branches as all branches must be scanned.

Unlike Query 1, Query 4 (which finds all records that satisfy a non-selective predicate across versions) shows where version-first performs poorly. The results are shown in Figure 6b. This figure shows the performance issue inherent to the version-first model for Query 4. Performing this query in version-first requires a full scan of the entire structure to resolve all differences across every branch. The tuple-first and hybrid schemes, on the other hand, are able to use their bitmap indexes to efficiently answer this query.

The intuition in Section 3 is validated for the version- and tuple-first models: the tuple-first scheme performs poorly in situations with many sibling branches which are updated concurrently, while the version-first model performs poorly on deep multi-version scans. Also, in both cases hybrid is comparable with the best scheme, and exhibits good scalability with the number of branches.

5.2 Query Results

Next, we evaluate all three storage schemes on the queries and branching strategies described in Section 4. All experiments are with 50 branches. Note that the deep and flat strategies were loaded with a fixed 100 GB dataset, but the scientific and curation strategies were loaded with a fixed number of branches to result in a dataset as close to 100 GB as possible, but achieving this exactly was not possible; consult Table 5 for the specific dataset sizes.
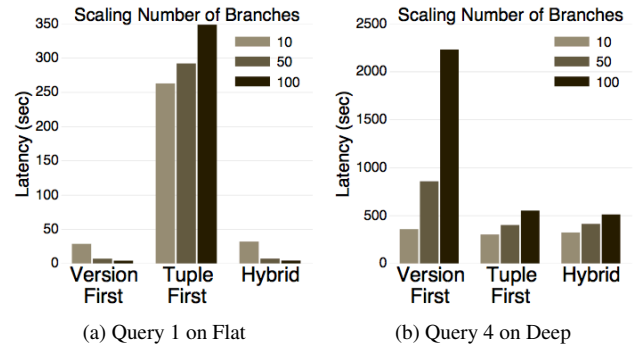

(a) Query 1 on Flat   (b) Query 4 on Deep

Figure 6: The Impact of Scaling Branches

Query 1 (Q1): Figure 7 depicts the results of Query 1 across each storage model. We also include data for tuple-first where records from each version are clustered together into page-sized (4MB) blocks. Here, we scan a single branch and vary the branching strategy and active branch scanned. The bars are labelled with the branching strategy and the branch being scanned. For the deep strategy, we scan the latest active branch, the tail. Since each successive branch is derived from all previous branches, this requires all data to be scanned. Note that we are scanning 100 GB of data in about 250s, for a throughput of around 400 MB/sec; this is close to raw disk throughput that we measured to be 470 MB/sec using a standard disk diagnostic tool (hdparm). For flat, we select a random child. For tuple-first, this results in many unnecessary records being scanned as data is interleaved; data clustering improves performance most in this case. The use of large pages increases this penalty, as an entire page is fetched for potentially a few valid records. Something similar happens in scientific (sci). Both the youngest and oldest active branch and branches have interleaved data that results in decreased performance for tuple-first. When reading a young active branch, more data is included from many mainline branches, which results in a higher latency for version-first and hybrid in comparison to reading the oldest active branch. Tuple-first has to read all data in both cases. For curation (cur.), we read either a random active development branch, a random feature branch, or the most recent mainline branch. Here, tuple-first exhibits similar performance across use cases, as it has to scan the whole data set. Version-first and hybrid exhibit increasing latencies largely due to increasingly complicated scans in the presence of merges. As the level of merges for a particular branch increases (random feature to current feature to mainline), so does the latency. As expected version-first has increasingly worse performance due to its need to identify the active records that are overwritten by a complicated lineage, whereas hybrid leverages the segment-indexes to identify active records while also leveraging clustered storage to avoid reading too many unnecessary records. Thus, in this case, hybrid outperforms both version and tuple-first.

Query 2 (Q2): Figure 8 shows the results for Q2. Recall that Q2 does a diff between two branches. In the figure we show four cases, one for each branching strategy: 1) diffing a deep tail and it's parent; 2) diffing a flat child and parent; 3) diffing the oldest science active branch and the mainline; and 4) diffing curation mainline with active development branch. Here, version-first uniformly has worse performance due to the need to make multiple passes over the dataset to identify the active records in both versions. This is in part due to the implementation of diff in version-first not incrementally tracking differences between versions from a common ancestor. Tuple-first and hybrid are able to leverage the index to quickly identify the records that are different between versions. As the amount of interleaving increases (dev to flat), we see that hybrid
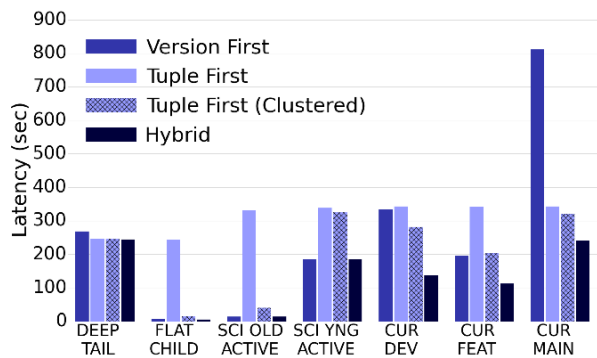
Figure 7: Query 1

Table 2: Bitmap Commit Data (50 Branches)

		Agg. Pack File Size (MB)	Avg. Commit Time (ms)	Avg. Checkout Time (ms)
DEEP	TF	234	15	501
	HY	198	13	25
FLAT	TF	532	86	193
	HY	155	10	66
SCI	TF	601	35	544
	HY	277	9	836
CUR	TF	510	10	570
	HY	280	6	43

is able to scan and compare fewer records than tuple-first, resulting in a lower average query latency.

Query 3 (Q3): Figure 9 depicts the results for Q3 which scans two versions, but finds the common records that satisfy some predicate. This is effectively a join between two versions. The trends between Q2 and Q3 are similar, however for version-first in Q2 we must effectively scan both versions in their entirety as we cannot rely on metadata regarding precedence in merges to identify the differences between versions. In Q3, we perform a hash join for version-first and report the intersection incrementally; in the absence of merges, the latencies are better (comparable with hybrid), but in curation with a complex ancestry we need two passes to compute the records in each branch and then another pass to actually join them.

Query 4 (Q4): Figure 10 depicts the results for Q4 with full data scans to emit the active records for each branch that match some predicate. We use a very non-selective predicate such that sequential scans are the preferred approach. As expected tuple-first and hybrid offer the best (and comparable) performance due to their ability to scan each record once to determine if which branch's the tuple should be emitted to. Version-first however, must sometimes make multiple passes to identify and emit the records that are active for each branch; in particular this is true in the curation workload, where there are merges. In addition, version-first has a higher overhead for tracking active records (as a result of its need to actively materialize hash tables containing satisfying records). The deeper and more complicated the branching structure, the worse the performance for version-first is. Also note in flat, hybrid outperforms tuple-first with near max throughput. This largely due to working with smaller segment indexes instead of a massive bitmap.

5.3 Bitmap Commit Performance

We now evaluate the commit performance of the different strategies. Our benchmark performed commits at fixed intervals of 10000 updates per branch. Table 2 reports the aggregate on-disk size of the compressed bitmaps for the tuple-first and hybrid schemes as well as averages of commit creation and checkout times. The aggregate size reported includes the full commit histories for all branches in

Table 3: Overview of Merge Performance

	Two-Way Avg MB/sec	Three-Way Avg MB/sec
VF	14.2	9.6
TF	15.8	15.1
HY	26.5	33.2

Table 4: Storage Impact of Table-Wise Updates from Figure 11, in GB.

	Pre-Size	Post-Size
DEEP	100	180
FLAT	100	108
SCI	83	146
CUR	91	130

the system. Recall from Section 3.2 that in tuple-first the commit history for each branch is stored within its own file; in hybrid, each (branch, segment) has its own file. This results in a larger number of smaller commit history files in the hybrid scheme.

Commit time and checkout time was evaluated by averaging the time create/checkout a random set of 1000 commits agnostic to any branch or location. Checkout times for hybrid are better since the total logical bitmap size is smaller (as bitmaps are split up) and the fragmentation of inserts in tuple-first increases dispersion of bits in bitmaps, enabling less compression. Note that the overall storage overheads are less than 1% of the total storage cost in all cases, and commit and checkout times are less than 1 second in all cases.

5.4 Merge Performance

Table 3 shows the performance of Decibel's merge operation for each storage model, in terms of throughput (MB/sec). We evaluated merge on the curation branching strategy with 50 branches. Results represent the throughput of the operation relative to the size of the `diff` between each pair of branches being merged. Numbers are in aggregate across the (approx. 30) merge operations performed during the build phase. The `diff` sizes of the merge operations varied from about 200 MB to 3 GB. We report results for both two-way and three-way merge strategies, that is, with both tuple and field-level conflicts. Version-First underperforms more in the three-way merge model because the *lca* commit must still be scanned in its entirety to determine conflicts, whereas the other models can leverage the bitmap indexes to reduce the component of the *lca* that is scanned. Note that field-level merges for the other strategies are as fast or better than tuple-level merges.

5.5 Table-Wise Updates

We also investigated Decibel's performance on table-wise update operations that touch every record in a table. Since Decibel copies complete records on each update, a table-wise update to a branch will tend increase the data set size by the current size of that branch, and also effectively cluster records into a new heap file. Figure 11 presents performance numbers for a single-version scan (Query 1) for each branching strategy before and after a table-wise update. We ran these experiments across each branching strategy and across 10 branches instead of 50 to more clearly display the effects of each table-wise update as each branch will have more data on average compared to the 50 branch case. Version-first degrades proportionally to the amount of new data that is inserted. The bitmap-based methods do not suffer from this issue. In addition, tuple-first in particular sees considerable performance gains as a result of the data clustering. Other operations and larger branch sizes displayed the same relative performance presented in previous experiments, in proportion with the larger data set sizes that arise as a result of table-wise updates. However, table-wise updates do increase data set sizes considerably. Table 4 displays the data set size increases corresponding to the operations represented in Figure 11. Such space overhead also arises in other no-overwrite systems, e.g., Postgres and Vertica. This can be mitigated with judicious use of
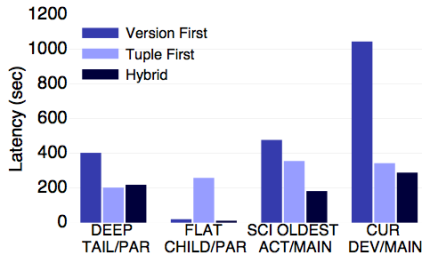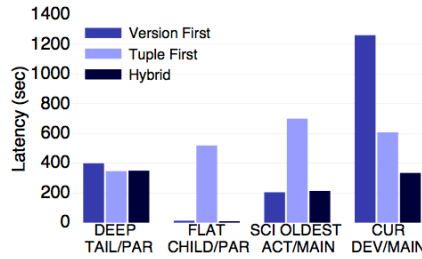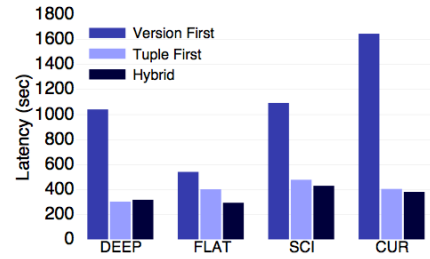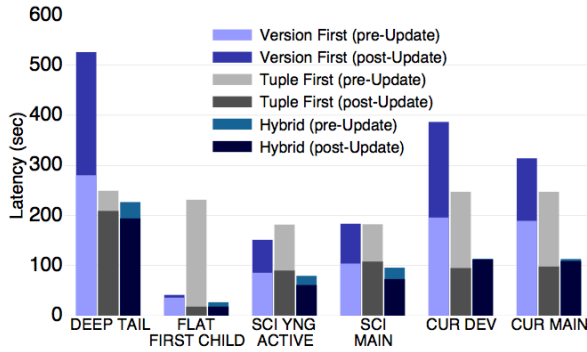
Figure 8: Query 2

Figure 9: Query 3

Figure 10: Query 4

Figure 11: Table-Wise Updates: Query 1 (10 Branches)

Table 5: Build Times (seconds)

Branches		10	50		10	50
		100 GB	**100 GB**		**83 GB**	**148 GB**
VF	DEEP	810	789	SCI	665	1540
TF		918	1020		914	2155
HY		908	941		805	1630
		100 GB	**100 GB**		**91 GB**	**139 GB**
VF	FLAT	888	1083	CUR	992	6238
TF		1138	1946		1229	4532
HY		993	1197		1030	3079

tuple and field-level compression, at some materialization cost. We postpone investigation of this to future work.

5.6 Load Time

Table 5 shows the total load time for our evaluations. This time includes inserting records, creating branches, updating records, merging branches, and creating commits. Here, we deterministically seed the random number generator to ensure each scheme performs the same set of operations in the same order. In general, version-first is fast for loading and branching as no bitmap index maintenance is required. However, with curation the performance for version-first suffers due to the complex branching structure. The load time for hybrid is lower than tuple-first due to smaller indexes, and is comparable with version-first. As noted earlier, due to random generation process science and curation data sizes vary. In general load times are reasonable (on the order of 100 MB/sec).

5.7 Comparison with git

A natural question is whether it would be possible to build Decibel on top of an existing version control system like git. To answer this question, we implemented the Decibel API using git as a storage manager. We created a local git repository, and call git commands (e.g. branch) in place of Decibel API calls. We implemented this in two ways: *git 1 file*, which uses a single heap file for all records versioned by git, and *git file/tup*, which creates a file for each tuple in the database. Other implementations, such as grouping tuples from the same commit into the same file, are also possible, but quickly begin to approximate what we built in Decibel. We also implemented CSV-based and binary-based storage

Table 6: git performance vs. Decibel (Hybrid) on the deep structure with 100% inserts to 10 branches over 10000 commits

	Data Size (MB)	Repo Size (MB)	repack Time (sec)	Commit Time $\mu \pm \sigma$ (ms)	Checkout Time $\mu \pm \sigma$ (ms)
git 1 file (bin)	10	10.95	1231	137 ± 50	134 ± 45
git 1 file (csv)	10	15.77	364	313 ± 120	266 ± 71
git file/tup (bin)	10	15.11	62	80 ± 44	694 ± 379
git file/tup (csv)	10	17.3	56	74 ± 31	686 ± 348
Decibel	10	10.63	N/A	3 ± 2	6 ± 4
git 1 file (bin)	100	39.37	1597	781 ± 393	2522 ± 2040
git 1 file (csv)	100	79.5	3286	1749 ± 937	1652 ± 586
git file/tup (bin)	100	53.17	509	694 ± 371	7300 ± 3859
git file/tup (csv)	100	96.4	478	693 ± 367	7481 ± 4425
Decibel	100	101.12	N/A	3 ± 2	10 ± 5
git file/tup (bin)	1000	366.83	42502	6686 ± 3692	56415 ± 37464
git file/tup (csv)	1000	453.91	48234	6848 ± 3819	66489 ± 74004
Decibel	1000	1003.56	N/A	3 ± 10	26 ± 46

formats to compare differencing operations. Note that CSV results in a larger raw size due to string encoding. We use the benchmark from Section 4.2 and configured git with default settings.

Table 6 shows performance of the deep branching strategy with 10 separate branches and 10000 commits (evenly spaced over the dataset). Commit times include the git add and git commit commands. As the commit and checkout times show, git suffers with even modest data set sizes and commit frequencies. We had to manually run repack operations to force git to compact its repositories (we did this once after loading in our benchmarks); this took substantial time (more than 13 hours for the 1GB benchmark). Not repacking resulted in 20x or more space overheads. It is also worth noting that a 10 GB benchmark was not able to complete within several days for any of the git-based systems; the 1 GB benchmark on 1-file CSV and binary did not complete after 4 days. Decibel's storage overhead is higher because it does full record-copies on writes, but its performance is much better, with up to 3 orders of magnitude lower latency for commit and checkout operations, while using less than 1% space overhead for commit metadata.

Part of git's poor performance is from storing each version as a separate object, with periodic creation of "packfiles" to contain several objects, either in their entirety or using a delta encoding. As shown, computing diffs can be slow and restoring binary objects is inefficient, as git exhaustively compares objects to find the best delta encoding to use, compute SHA-1 hashes for each commit (proportional to data set size), and compresses blobs.

The results in Table 6 highlight key differences between git and Decibel. In particular git's use of delta chains minimizes storage overhead, but takes a long time to both commit and checkout. In contrast, Decibel's simple appends for commits and new files for branches, improves performance. Table 7 shows similar performance numbers for a more update-intensive workload.

6. RELATED WORK

There has been plenty of work on linear dataset versioning (i.e., for a linear, temporal chain of versions without any branching.) For instance, temporal databases [2, 26, 23, 21] support "time-travel", i.e., the ability to query point-in-time snapshots of a linear chain of

Table 7: git performance vs. Decibel (Hybrid) on the deep structure with 50% updates to 10 branches over 10000 commits

	Data Size (MB)	Repo Size (MB)	repack Time (sec)	Commit Time $\mu \pm \sigma$ (ms)	Checkout Time $\mu \pm \sigma$ (ms)
git 1 file (csv)	100	80.7	4262	1788 ± 1037	2014 ± 1270
git file/tup (csv)	100	73.31	337	441 ± 508	9326 ± 6433
Decibel	100	101.32	N/A	4 ± 2	15 ± 8

database versions. Lomet et al. introduced *ImmortalDB*, a temporal versioning system built directly into SQLServer [14]. ImmortalDB also leverages an append-only, copy-on-write strategy for updates, but embeds backpointers into tuples to record the (linear) provenance information. Later work investigated compression strategies across tuple versions [15]. Recent work has looked into linear versioning for specialized databases, e.g., graph and scientific array databases. Khurana et al. [11] develop snapshot retrieval schemes for graph databases, while Seering et al. [22] develop compression schemes for array databases. Soroush et al. [24] develop a storage manager for approximate timetravel in array databases through the use of skip-lists (or links), tailored to a linear version chain. Since a branched system like Decibel lacks a total ordering of branches, the temporal methods explored in this body of work do not apply.

There is also prior work on temporal RDF data and temporal XML Data. Motik [18] presents a logic-based approach to representing valid time in RDF and OWL. Several papers (e.g., [1, 27]) have considered the problems of subgraph pattern matching or SPARQL query evaluation over temporally annotated RDF data. There is also much work on version management in XML data stores and scientific datasets [6, 12, 17]. These approaches are largely specific to XML or RDF data, and cannot be directly used for relational data; for example, many of these papers assume unique node identifiers to merge deltas or snapshots.

Multi-versioning is extensively used in databases to provide snapshot isolation [3, 19]. However, these methods only store enough history to preserve transactional semantics, whereas Decibel preserves historical records to maintain branched lineage.

Existing software version control systems like git and mercurial inspired this work. While these systems work well for modest collections of small text or binary files, they are not well-suited for large sets of structured data. Moreover, they do not provide features of databases, such as transactions or high-level query interfaces.

There exists a considerable body of work on "fully-persistent" data structures, B+Trees in particular [9, 13, 20]. Some of this work considers branched branches, but is largely focused on B+Tree-style indexes that point into underlying data consisting of individual records, rather than accessing the entirety or majority of large datasets from disk. Jiang et al. [10] present the *BT-Tree* which is designed as an access method for "branched and temporal" data. Each update to a record at a particular timestamp constitutes a new "version" within a branch. Unfortunately, their versioning model is limited and only supports trees of versions with no merges; furthermore, they do not consider or develop algorithms for the common setting of scanning or differencing multiple versions.

A recent distributed main-memory B-Tree [25] considers branchable clones which leverage existing copy-on-write algorithms for creating cloneable B-Trees [9]. However, like the BT-Tree, these methods heavily trade off space for point query efficiency and therefore make snapshot creation and updating very heavyweight operations. In addition, the paper does not evaluate any operations upon snapshots but only the snapshot creation process itself. Merging and differencing of data sets are also not considered.

7. CONCLUSION

We presented Decibel, our database storage engine for managing and querying large numbers of relational dataset versions. To the best of our knowledge, Decibel is the first implemented and evaluated database storage engine that supports arbitrary (i.e., non-linear) versioning of data. We evaluated three physical representations for Decibel, and compared and contrasted the relative benefits of each, and identified hybrid as the representation that meets or exceeds the performance of the other two representations; we also evaluated column and row-oriented layouts for the bitmap index associated with each of these representations. In the process, we also developed a versioning benchmark to allow us to compare these representations as well as representations developed in future work.

8. REFERENCES

[1] A. Pugliese et al. Scaling RDF with time. In *WWW*, 2008.

[2] I. Ahn and R. Snodgrass. Performance evaluation of a temporal database management system. In *SIGMOD*, pages 96–107, 1986.

[3] H. Berenson, P. Bernstein, J. Gray, J. Melton, E. O'Neil, and P. O'Neil. A critique of ANSI SQL isolation levels. In *SIGMOD*, pages 1–10, 1995.

[4] A. P. Bhardwaj, S. Bhattacherjee, A. Chavan, A. Deshpande, A. J. Elmore, S. Madden, and A. G. Parameswaran. DataHub: collaborative data science & dataset version management at scale. In *CIDR*, 2015.

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

[6] P. Buneman, S. Khanna, K. Tajima, and W. Tan. Archiving scientific data. *ACM TODS*, 29(1):2–42, 2004.

[7] A. Chavan, S. Huang, A. Deshpande, A. J. Elmore, S. Madden, and A. G. Parameswaran. Towards a unified query language for provenance and versioning. In *TaPP*, 2015.

[8] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears. Benchmarking cloud serving systems with ycsb. In *SoCC*, 2010.

[9] J. R. Driscoll, N. Sarnak, D. D. Sleator, and R. E. Tarjan. Making data structures persistent . In *STOC*, pages 109–121, 1986.

[10] L. Jiang, B. Salzberg, D. Lomet, and M. Barrena. The BT-Tree: A branched and temporal access method. In *PVLDB*, pages 451–460, 2000.

[11] U. Khurana and A. Deshpande. Efficient snapshot retrieval over historical graph data. *ICDE*, 2013.

[12] N. Lam and R. Wong. A fast index for XML document version management. In *APWeb*, 2003.

[13] S. Lanka and E. Mays. Fully persistent B+-Trees . In *SIGMOD*, 1991.

[14] D. Lomet, R. Barga, M. F. Mokbel, G. Shegalov, R. Wang, and Y. Zhu. Transaction time support inside a database engine. In *ICDE*, 2006.

[15] D. Lomet, M. Hong, R. Nehme, and R. Zhang. Transaction time indexing with version compression. In *VLDB*, 2008.

[16] M. Maddox, D. Goehring, A. Elmore, S. Madden, A. Parameswaran, and A. Deshpande. Decibel: The relational dataset branching system. Technical Report, available at: http://web.engr.illinois.edu/~adityagp/decibel-tr.pdf, 2016.

[17] A. Marian, S. Abiteboul, G. Cobena, and L. Mignet. Change-centric management of versions in an XML warehouse. In *VLDB*, 2001.

[18] B. Motik. Representing and querying validity time in RDF and OWL: A logic-based approach. In *ISWC*, 2010.

[19] D. R. K. Ports and K. Grittner. Serializable snapshot isolation in PostgreSQL. In *PVLDB*, pages 1850–1861, 2012.

[20] O. Rodeh. B-trees, shadowing, and clones. *ACM Transactions on Storage*, 2008.

[21] B. Salzberg and V. J. Tsotras. Comparison of access methods for time-evolving data. *ACM Computing Surveys*, pages 158–221, 1999.

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

[23] R. T. Snodgrass, editor. *The TSQL2 Temporal Query Language*. Kluwer, 1995.

[24] E. Soroush and M. Balazinska. Time travel in a scientific array database. In *ICDE*, 2013.

[25] B. Sowell, W. Golab, and M. A. Shah. Minuet: A scalable distributed multi-version B-Tree. In *PVLDB*, pages 884–895, 2012.

[26] A. Tansel, J. Clifford, S. Gadia, S. Jajodia, A. Segev, and R. S. (editors). *Temporal Databases: Theory, Design, and Implementation*. 1993.

[27] J. Tappolet and A. Bernstein. Applied temporal RDF: Efficient temporal querying of RDF data with SPARQL. In *ESWC*, pages 308–322, 2009.