Decibel: Transactional Branched Versioning for Relational Data Systems

by

David Goehring

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

Masters of Engineering in Computer Science

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

June 2016

Author . . . .
Department of Electrical Engineering and Computer Science
May 20, 2016

Certified by . . . .
Samuel Madden
Professor
Thesis Supervisor

Accepted by . . . .
Christopher Terman
Chairman, Masters of Engineering Thesis Committee

Decibel: Transactional Branched Versioning for Relational Data Systems

by

David Goehring

Submitted to the Department of Electrical Engineering and Computer Science
on May 20, 2016, in partial fulfillment of the
requirements for the degree of
Masters of Engineering in Computer Science

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. Transaction management (ACID) for such systems requires additional tools to efficiently handle concurrent changes and ensure *transactional consistency* of the version graph (concurrent versioned commits, branches, and merges as well as changes to records). Furthermore, a new conflict model is required to describe how versioned operations can *interfere* with each other while still remaining serializable. Decibel is a new relational storage system with built-in version control and transaction management designed to address these shortcomings. Decibel's natural versioning primitives can also be leveraged to implement *versioned transactions*. Thorough evaluation of three versioned storage engine designs that focus on efficient query processing with minimal storage overhead via the development of an exhaustive benchmark suggest that Decibel is vastly superior to and enables more cross version analysis and functionality than existing techniques and DVCS software like *git*. Read only and historical cross-version query transactions are non-blocking and proceed all in parallel with minimal overhead. The benchmark also supports analyzing performance of versioned databases with transactional support. It also enables rigorous testing and evaluation of future versioned storage engine designs.

Thesis Supervisor: Samuel Madden
Title: Professor

Acknowledgments

This research was funded in part by Intel Science and Technology Center for Big Data.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Decibel

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 data with others or to keep track of who is using a particular dataset. , besides using file system permissions (which are not practical to use when sharing with users in other organizations who don't have accounts).

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 "checkout" 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 thesis, we address how we have solved these problems in 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 [20].

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.

1.2 Indexing

Traditional databases make extensive use of indexing to enable efficient lookup of records by a key, be it a primary key or another attribute in a record. Such queries include either a point look up (access a single record with a particular key) or a range query (given with all the records where the corresponding keys are in a particular range). To enable point and range queries across branches and commits, efficient indexing of records across versions is required. The proposed scheme builds on traditional indexing (B-Tree or Hash [71]) to provide a minimal storage overhead solution that is able to answer cross version index queries.

1.3 Transactions

Dataset versioning systems not only require the ability to version efficiently but also need to support transactions with traditional database semantics with the addition of versioning operations. Specifically, a group of `branch`, `merge`, `commit`, and data operations should be able to be done with traditional ACID guarantees so that resulting execution of the transactions is *Serializable*, the traditional correctness criterion for concurrent transactions [71, 10]. Concurrency is crucial in a system like this to not only ensure that modifications to the dataset remain transactional, but to support high throughput read only queries that are non-blocking and ensure that those read only queries have a *transactionally consistent* view of the dataset. Any user should be able to read a transactionally consistent snapshot of the database at any time and should not wait for concurrent writers. Decibel also allows multiple transactions to query different commits simultaneously.

Much research has been done in the areas of transaction management and concurrency control protocols [70, 48, 58, 11, 28, 74], but Decibel uses the underlying data structures for version management to provide transaction isolation. Furthermore, a new conflict model is required to describe how versioned operations can *interfere* with each other while still remaining serializable. In addition to presenting Decibel, this thesis presents a transaction management scheme for Decibel that leverages its natural versioning properties to make *versioned transactions* serializable and support high throughput, non-blocking read only queries.

Decibel is a key component of DataHub [13], 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.

The key contributions of this thesis are:

- 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. The benchmark also supports evaluating transaction performance of versioned databases.
- 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`. We thoroughly analyze the trade-offs between these storage schemes across a variety of operations and workloads.
- We show how to leverage Decibel's natural versioning power to implement transactions and discuss a conflict model for versioned databases.

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 2.2. A detailed overview of the aforementioned physical storage schemes is presented in Section 2.3. We then describe our versioned benchmarking strategy in Section 2.5 and the experimental evaluation of our storage models on a range of versioned query types in Section 2.6. After this description of Decibel fundamentals, we discuss the challenges for concurrency control and a conflict model for versioned databases in 3.2. Then in 3.3 we discuss how Decibel's protocol for handling concurrent changes. There we discuss how the natural versioning primtives provided by Decibel can be leveraged to implement transactions. We then revisit the versioned benchmark in 3.4 to discuss how it was augmented to analyze transaction performance of versioned databases and the type of versioned transactions evaluated. We then evaluate Decibel's transaction implementation in 3.5. We then provide implementation details of the versioned transaction manager in 4.2.

Chapter 2

Decibel: The Dataset Branching System

2.1 Versioning Patterns & Examples

We now describe two typical dataset versioning patterns that we have observed across a wide variety of scenarios in practice. We describe how they motivate the need for Decibel, and capture the variety of ways in which datasets are versioned and shared across individuals and teams. 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. (e.g., remove or merge columns, deal with NULL values or outliers), annotate the data with additional derived features, separate into test and training subsets, and run models as part of an iterative process. 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.

2.2 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.

2.2.1 Architecture

Decibel (Figure 2-1) 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. 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.

2.2.2 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 2-2, where (a) and (b) depict two evolution

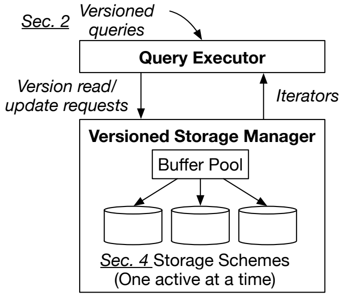


Figure 2-1: Decibel Architecture

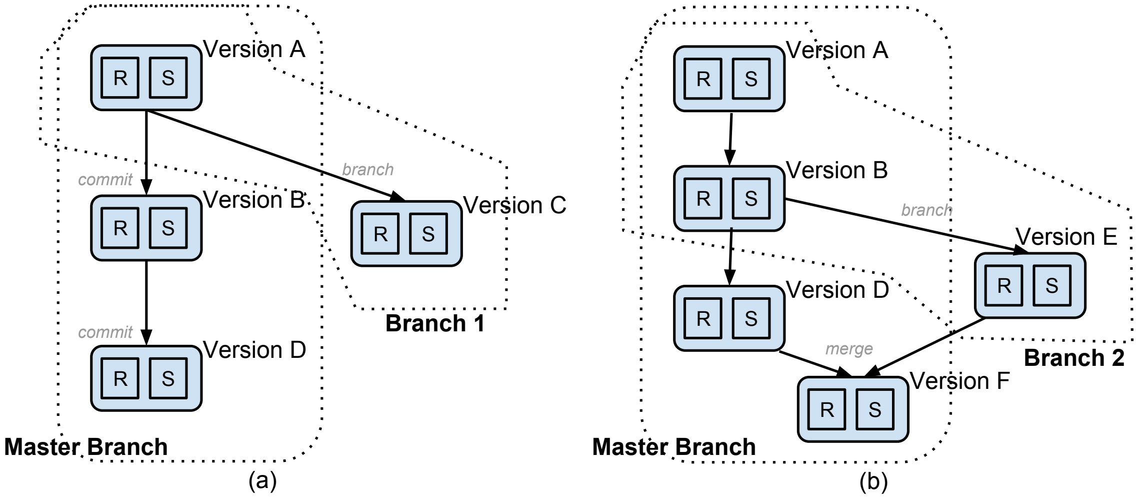


Figure 2-2: Two Example Workflows

patterns of a dataset.

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.

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-intime 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 2-2(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 2-2(b)), or through the application of transformation programs to one or more existing versions (e.g., Version B from Version A in Figure 2-2(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 2-2(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 2-2(a) there are two branches, one corresponding to Version D and one corresponding to C. Similarly, in Figure 2-2(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

Table 2.1: 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`

access control, we envision that each branch could have different access privileges for different users.

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 2-2(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. Decibel also naturally supports optimistic concurrency control through branching and merging mechanisms (as discussed below). In future work, we further plan to understand the interplay between the two different mechanisms for optimistic concurrency control.

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 2-2(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 2-2(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.

Complete Version Graph Description: Now that the core operations of Decibel have been described, the version graph is formally defined as follows:

- *Branch* operations result in a create commit and there is a directed edge from the *head* commit of the parent branch at the time of a branch to the creation (first commit) on the newly created branch.

- *Commit* operations move the *head* commit on a branch. There is a directed edge from the old *head* commit of the branch to the new *head* commit.

- *Merge* operations involve two branches, where the contents of the second branch (the *secondary*) are merged into the first branch (the *primary*). Merging results in a merge commit (new *head* commit on the primary) on the primary branch (from which another branch can be created). There is a directed edge from the *head* commit of the secondary to the merge commit in the primary. This edge is needed to properly track previous merges and move the *lowest common ancestor commit* between the two branches so that subsequent merges between the two branches do not merge previously merged data.

Merging and Conflict Model Discussion: Decibel supports both *two-way* and *three-way* merges. *Two-way* merges involve comparing the two branches in their entirety, ignoring common ancestry. This will generate a superfluous number of conflicts, especially between the commonly inherited data, but a user may wish to employ such a strategy because conflict resolution can boil down record level acceptance of conflicts from one branch (the primary) and simply incorporating the newly inserted data from the other branch (the secondary). This merge procedure is also easy to implement and does not require explicit version graph tracking. A *three-way* merge procedure is more sophisticated and enables field level conflict resolution. This involves finding the *lowest common ancestor (lca)* commit of the current *head* commits of the two branches being merged. Then by looking at the *lca* and two *head* commits Decibel can determine for each conflicting record which fields have been modified since the two branches diverged based on the contents of the (lca) record and then merge the 3 record appropriately. Decibel adopts a precedence model whereby if there is a conflicting field change the *primary*'s version of the field is taken in the merged record. This is probably the more desirable merge procedure, but is significantly more complex to implement.

Decibel currently adopts the following conflict semantics based on the changes in each branch. Some conflicts are considered *auto-resolvable (non-conflicts)*. These include the set of records that were update/deleted in one parent but were not modified in the other branch from the point at which the branches diverged, the *lca commit*. These are resolved by the simply incorporating the changes into the *primary* branch, there is no explicit need to explicitly invoke a conflict resolution strategy. The other class of conflicts are *merge procedure resolvable (true conflicts)* where the record was changed in a particular way in both branches. They are called *merge procedure resolvable* because the conflict could be handed to a user program (custom merge procedure/conflict handler) for resolution, but Decibel attempts to auto-resolve these conflicts using a precedence based merge model whereby the *primary*'s changes are taken over the *secondary*'s. There are several types of these conflicts:

- *Update/Delete*: A record was updated in the *primary* and deleted in the *secondary*. Decibel handles this by taking the update from the *primary*.

- *Delete/Update*: A record was deleted in the *primary* and updated in the *secondary*. Decibel handles this by taking the delete from the *primary*.

- *Update/Update*: A record was updated in both the *primary* and *secondary*. Decibel handles this by performing the three-way merge of the *lca commit* version and the *head commit* versions of the *primary* and *secondary* as follows:

For every field j if the *primary*'s j-th field matches the *lca*'s j-th field then the *secondary*'s j-th field is taken in the merge otherwise the *primary*'s j-th field is taken (since it has precedence).

- *Insert/Insert*: Two different records with the same primary key that was not in the common ancestry were inserted into the two branches being merged. Decibel handles this by taking the *primary*'s record.

Performing a three-way merge requires 3 different versions of a record for each conflict that are most likely not located on the same page and so to prevent an exorbitant amount of random I/O another approach is required. In our experiments, most conflict sizes are only a few GB so it is reasonable to just buffer the conflicts in memory as they are detected (e.g. scan the *lca* commit and two *head* commits and just buffer the conflicting records, note that only the conflicting records in each commit will be scanned). For large conflict sizes, however, it may be necessary to write out the conflicts from each commit into separate files (e.g. the *lca* commit's versions of the conflicting record are written into file 1 and the *primary*'s *head* commit's versions of the conflicting records are written into file 2). Then each of these files are sorted by primary key and the conflicts are read from these files.
Also, note that in some cases a version graph can contain more than one *lca commit* for the two branches being merged, and it still remains to be investigated how to handle this. From our research *git* would just use one. However, this particular scenario did not occur in our experiments.

2.2.3 Versioned Query Language Overview

Decibel supports a versioning query language, termed VQuel, which was formally defined in our previous TaPP paper [20]. Some of the queries supported by Decibel and evaluated as part of our benchmark, described subsequently, are listed in Table 2.1. 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 [57].

VQuel draws from classical query languages Quel [81] and GEM [?], none of which had a versioning/temporal component, and adds versioning-specific features. Examples of queries in VQuel and their corresponding queries in a variant of SQL are provided in Table 2.1, while a complete specification can be found in [20]. Note that we are not advocating the use of VQuel as the only possible versioning query language; as our table itself indicates, a variant of SQL itself could be a an alternative.

VQuel supports a range of functionality aimed at manipulating and querying across a range of versions. Consider the rows of Table 2.1. VQuel includes basic branching and merging operations (as described in the previous section). To "iterate" over objects (versions, relations, branches), VQuel introduces the `range` command; the `retrieve` command specifies the attributes that are part of the output (akin to a `SELECT` in SQL), and the `where` command plays an identical role to the `WHERE` in a SQL statement. To illustrate, consider Query #2, aimed at performing a positive diff between versions v01 and v02 of relation R, i.e., finding all tuples in v01 not present in v02. The VQuel query sets up an iterator E1 to loop over all the tuples of relation R of version v01, and outputs all the attributes of these tuples, while verifying that E1.id is not present in the set of ids output by a nested VQuel statement that captures the ids present in the tuples of relation R of version v02. The corresponding SQL is self-explanatory, where the version is treated as "just another attribute" of the relation R. As yet another example, consider Query #4, which retrieves all the tuples in all the "head" versions that satisfy some predicate, i.e., the versions that are the latest commits across all the branches. Each record reported is annotated with the set of versions that contain that record. Verifying if a version is a head version or not is done using the HEAD() function. The queries listed in Table 2.1 capture a range of functionality from scanning a single version, to comparing or joining a couple of versions, to scanning all the versions that satisfy certain conditions.

2.3 Physical Representations

In this section, we explore several alternative physical representations of our versioned data store. We begin by presenting two intuitive representations, the *tuple-first* and *version-first* models. The tuple-first model stores all records together, and uses an index to identify the branches a tuple is active in, whereas the version-first model stores all modifications made to each branch in a separate heap file, affording efficient examination of records in a single version. The tuple-first model outperforms on queries that compare across versions, while the version-first model underperforms for such queries; conversely, version-first outperforms for queries targeting a single version, while tuple-first underperforms. Finally, we present a *hybrid* storage model which bridges these approaches to offer the best of both. We now describe each of these implementations and how the core versioning functionality is implemented in each. 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). As discussed earlier, we also assume that each record has a unique primary key.

2.3.1 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

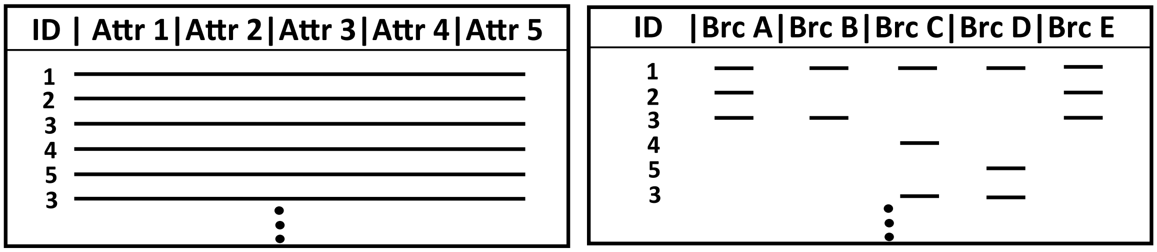

ID | Attr 1|Attr 2|Attr 3|Attr 4|Attr 5
1
2
3
4
5
3
ID |Brc A|Brc B|Brc C|Brc D|Brc E
1
2
3
4
5
3

Figure 2-3: Example of Tuple-First

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 2-3 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.

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 2-4 shows how each segment

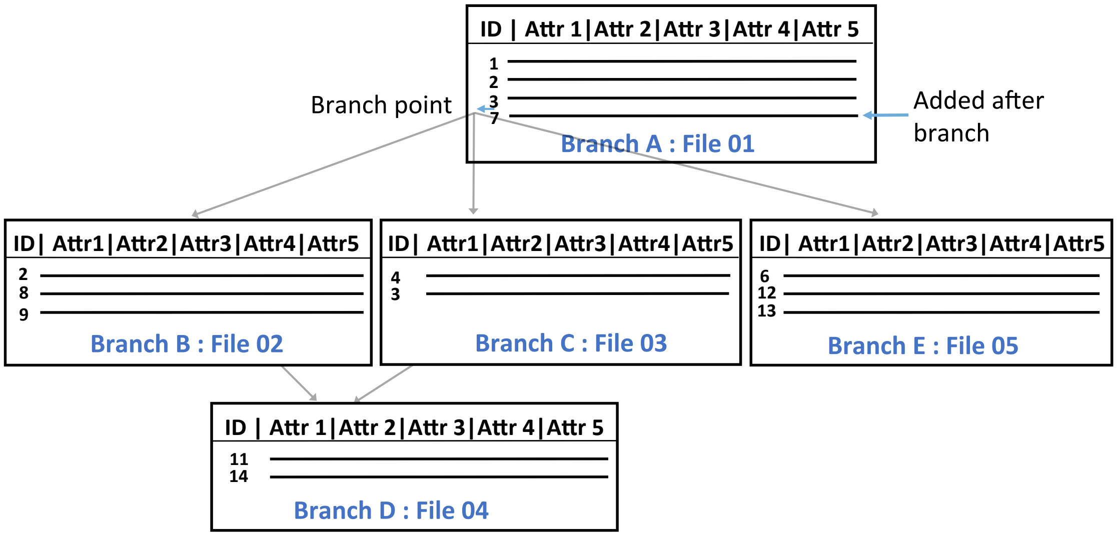

ID | Attr 1|Attr 2|Attr 3|Attr 4|Attr 5
1
2
3
7
Branch A : File 01
Branch point
Added after branch
ID| Attr1|Attr2|Attr3|Attr4|Attr5
2
8
9
Branch B : File 02
ID| Attr1|Attr2|Attr3|Attr4|Attr5
4
3
Branch C : File 03
ID| Attr1|Attr2|Attr3|Attr4|Attr5
6
12
13
Branch E : File 05
ID | Attr 1|Attr 2|Attr 3|Attr 4|Attr 5
11
14
Branch D : File 04

Figure 2-4: Example of Version-First (depicts branches resulting from merges, but could be commits)

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 2-5 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.

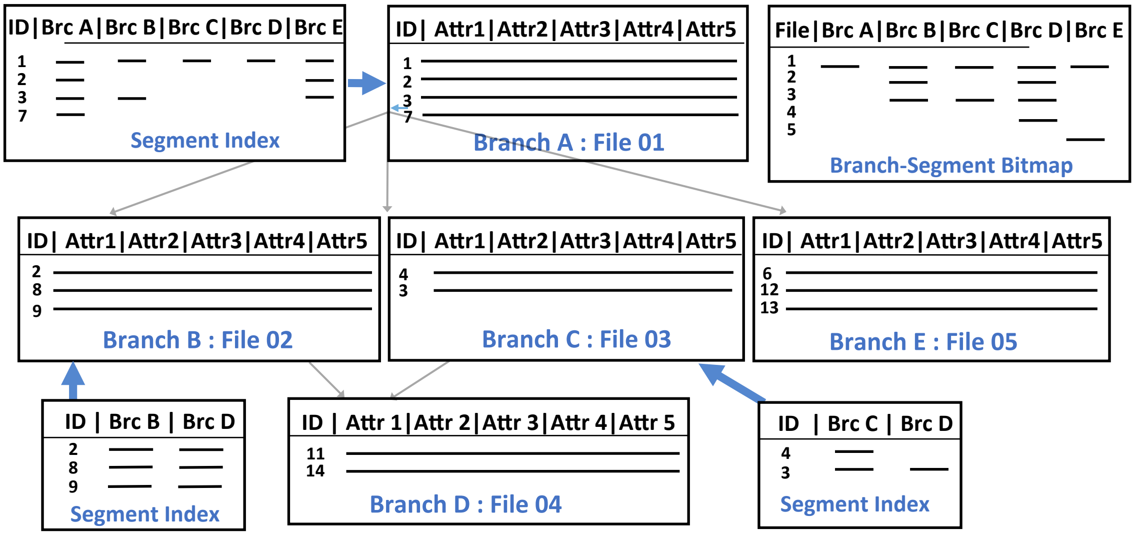

ID|Brc A|Brc B|Brc C|Brc D|Brc E
1
2
3
7
Segment Index
ID| Attr1|Attr2|Attr3|Attr4|Attr5
1
2
3
7
Branch A : File 01
File|Brc A|Brc B|Brc C|Brc D|Brc E
1
2
3
4
5
Branch-Segment Bitmap
ID| Attr1|Attr2|Attr3|Attr4|Attr5
2
8
9
Branch B : File 02
ID| Attr1|Attr2|Attr3|Attr4|Attr5
4
3
Branch C : File 03
ID| Attr1|Attr2|Attr3|Attr4|Attr5
6
12
13
Branch E : File 05
ID | Brc B | Brc D
2
8
9
Segment Index
ID | Attr 1|Attr 2|Attr 3|Attr 4|Attr 5
11
14
Branch D : File 04
ID | Brc C | Brc D
4
3
Segment Index

Figure 2-5: Example of Hybrid

Our discussion focuses on how we minimized the number of repeated accesses to data in our implementation of these schemes.

2.3.2 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 âĂIJlayerâĂİ 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 (their *head* commits) 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. However, the diff is modified such that records that were inherited from the *lca* by one branch and updated in the other are excluded, specifically, rows where the *lca was 1* and only one branch had a 1 in the same location (e.g. 1,1,0). These types of differences would normally be reported by the diff operator but they represent conflicts that Decibel can automatically resolve using the bitmap since the inherited record was only updated in one branch. For the purposes of conflict resolution we are only interested in rows where the *lca* bit is 0, but one of the branches being merged have the corresponding bit set (e.g. 0,0,1 or 0,1,0). These represent possible places where conflicting inserts/updates may have occurred (e.g. an update to a *lca*, a previously detected (1,0,0). Such rows need to be calculated per parent and can be done by XORing the *lca* commit column with one parent's *head* commit followed by an AND with the same parent's *head commit* (e.g. (*lca* XOR *primary*) AND *primary.* 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.

More Conflict Scenarios Detected Using Bitmaps

Without loss of generality assume branch C is being merged into branch B and the following notation is used to identify scenarios when comparing the *head* commit bitmap columns of B and C to the *lca* commit bitmap column for a particular row (that corresponds to a record): (*lca* bit, B's *head* bit, C's *head* bit). We have the following scenarios:

- **(1,1,1)**: This record was inherited from the *lca* and is still present in the *head*s of both parents. No bits need to be flipped.

- **(1,0,1)**: This is corresponds auto-resolvable (non-conflict) update/delete from B, the *lca* record is still present in the *head* of C. If it is an update it is followed by a record with a (0,1,0) row somewhere later in the bitmap. Since we are merging C in B no bits need to be flipped in this row either since B's 0 is already there.

- **(1,1,0)**: This is an auto-resolvable (non-conflict) update/delete from C, the *lca* record is still present in the *head* of B. If it is an update it is followed by a record with a (0,0,1) row somewhere later in the bitmap. We need to flip B's 1 to a 0 since this inherited record was either deleted or updated by C.

- Note: (0,1,1) is impossible since two two *head*s are being compared to the *lca* commit (the point where they were last synced).

2.3.3 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 2-4 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). While this approach is simple, it does result in a higher memory usage to manage the in-memory set. Although memory usage is not prohibitive, were it to become an issue, it is possible to write these sets for each segment file to disk, and the use external sort and merge to compute record/segment-file pairs that should appear in the output. 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 $D - B - C - A$.

Scanning a commit (rather than the head of a branch) works similarly, but instead of reading to the end of a segment file, the scanner starts at the commit point.

Decibel scans backwards to ensure more recently updated tuples will not be overwritten by a tuple with the same primary key from earlier in the ancestry. By doing so, we allow pipelining of this iterator as we know an emitted record will never be overwritten. However, reading segment files in reverse order leads to performance penalties as the OS cannot leverage sequential scans and pre-fetching. Our implementation seeks to lay out files in reverse order to offset this effect, but we omit details due to space reasons.

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. The complexities are described next, but it generally requires multiple passes over the data relevant to the branch to reconstruct the current state of that branch even for single branch scans.

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

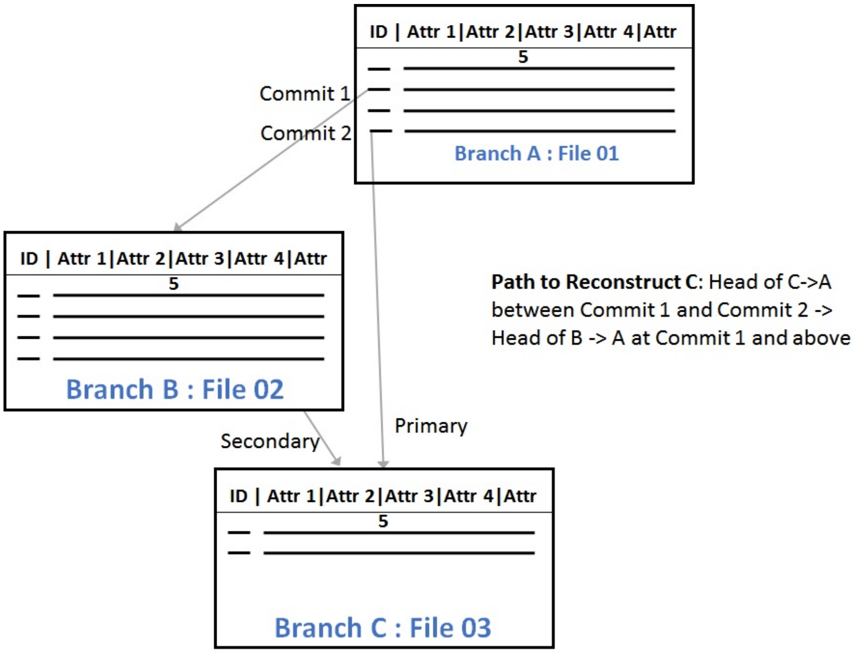

ID | Attr 1|Attr 2|Attr 3|Attr 4|Attr 5
Commit 1
Commit 2
Branch A : File 01
ID | Attr 1|Attr 2|Attr 3|Attr 4|Attr 5
Branch B : File 02
Path to Reconstruct C: Head of C->A between Commit 1 and Commit 2 -> Head of B -> A at Commit 1 and above
Secondary
Primary
ID | Attr 1|Attr 2|Attr 3|Attr 4|Attr 5
Branch C : File 03

Figure 2-6: Example of merge scenario that requires Version-First to make 2 passes

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). Difficulties also arise even in the single branch scan case since merges may have occurred in a branch's ancestry. See Figure 2-6, to reconstruct the *head* of C, VF would have to revisit File 01 and so would perform more random I/O which may become prohibitive as the complexity and number of merges up the ancestry increases. 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.) This can be thought of as building *commit change logs.* Also, note that the order in which the segments are scanned does not matter since the hash tables will be combined in-memory, but it is imperative that every relevant segment (e.g. in the union of the ancestries of all the branches to be scanned) be scanned starting at the latest relevant commit in that segment so as to construct all of the commit hash tables for that segment in one pass. Then, for each branch, these in-memory tables can be scanned from leaf-to-root (called the *linearized ancestry*) and takes into account merge precedence for combining data from two merged branches to determine the records that need to be output on each branch, just as in the single-branch scan. Alternatively, the final per branch hash tables can be built by reversing the *linearized ancestry* and incorporating the changes from the intermediate hash tables (*commit change logs*) in order.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. Since commits are branch points without the added segment file, the multi-branch scanner also supports scanning arbitrary commits.

A crucial facet to understand is that the ancestry linearization is non-trivial. Initially, it requires creating a new graph called the *precedence scan ordering (PSO) graph* from the *version graph.* The *PSO graph* is formed by first reversing all of the edges in the *version graph* (to generate a scan path from leaf to root) and then adding additional directed edges based on the conflict model. For instance, to support the three-way merge semantics with precedence based conflict resolution,, for every commit that is reachable from the leaf commit of the branch to be reconstructed (which can be found using a simple breadth-first search) and has two parents (is a merge) a *precedence edge* needs to be added from the first commit in the divergence of the *primary* branch from the *lca* commit to the *head* commit in the *secondary*. Adding this edge requires that the *primary* parent's commits its divergence from the *lca* are are processed before the *secondary*'s (thus handling the delete/update and update/delete conflicts). Now to linearize the ancestry of the leaf commit to reconstruct involves a topological sort of the *PSO graph* .

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.

However, if an update or deleted occurred only in one branch after diverging from the lowest common ancestor, it is necessary to scan part of the common ancestry to retrieve the old version of the record present in the other branch, which can be done in a hash join fashion, for brevity we omit the details.

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. For instance, if a key exists both the *lca*'s table and the *primary* parent's table, but the offsets are different then we know the record was updated in the *primary*. 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. This handles the *update/update* conflicts and resolving the other types of conflicts are resolved by the multi-branch scanner when it reconstructs the branch by using the *linearized ancestry* and incorporating the changes from the *primary* parent's segments such that they take precedence over the *secondary* parent's by combining the intermediate hash tables in the correct order.

2.3.4 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 2.3.1, 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 [57].

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: Merging is again similar to the tuple-first model except that the operation is localized to a particular set of segments containing records in the branches involved in the merge operation. 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 obtain 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. Though Hybrid can leverage bitmap operations to conduct the merge more efficiently, including records from new segments breaks the natural branch locality and so after many merges it is possible that every segment will need to be read when reconstructing a branch.

Commit: The hybrid commit process is analogous to that of the tuple-first model except that the bitmap column of the target branch of the commit must be snapshotted within each segment containing records in that branch, as well as the branch's entry in the branch-segment bitmap.

Data Modification: The basic process for inserts, deletes, and updates is as in tuple-first. Updates require that a new copy of the tuple is added to the branch's head segment, and that the segment with the previous copy of the record have the corresponding segment index entry updated to reflect that the tuple in prior segment is no longer active in this branch. If the prior record was the last active record in the segment for the branch being modified, then the branch-segment bitmap is updated so that the segment will not be considered in future queries on that branch.

Multi-branch Scan: As in tuple-first, multi-branch scans require less work than in version-first as we can pass over each tuple once, using the segment-index to determine how to apply the tuple for scan. However, compared with tuple-first, hybrid benefits from scanning fewer records as only the segments that correspond to the scanned branches need to be read.

Diff: The differencing operation is again performed similarly to the tuple-first model except that only the set of segments containing records in the branches being differenced are consulted. The storage manager first determines which segments contain live records in either branch, then each segment is queried to return record offsets comprising the positive and negative difference between those branches within that segment. The overall result is an iterator over the union of the result sets across all pertinent segments.

2.3.5 Discussion

The previous sections discussed the details of the three schemes. We now briefly summarize the expected differences between their performance to frame the evaluation, shown in Table 2.2. Tuple-first's use of bitmaps allows it to be more efficient at multi-branch scans, but its single heap file does poorly when records from many versions are interleaved. Bitmap management can also be expensive. Version-first, in contrast, co-locates tuples from a single version/branch, so does well on single-branch scans, but because it lacks an index performs poorly on multi-version operations like diff and multi-version scan. Hybrid essentially adds a bitmap to version-first to allow it to get the best of both worlds.

2.4 Indexing

Indexing is a crucial function in a standard databases to speed up access of single records or records that have an attribute in a particular range. Furthermore, indexing is crucial for benchmarking (e.g. benchmarks such as YCSB [21]). Thus, it is crucial for a versioned database to support the same functionality, except across versions,

Type	Pro	Con
Tuple-First	Efficient Multi-Branch Scans. Simple diffs.	Declustered storage. Large bitmap growth.
Version-First	Efficient single branch scan. Cheap branch creation. Minimal Indexing. Cheap updates.	Expensive scans with merges. Expensive multi-branch scans.
Hybrid	Good single and multi branch scans. Smaller indexes. Clustering.	None

Table 2.2: Summary of the Approaches

which can either *commits* and *branches*. We begin exploration of indexing with a few sample indexed queries and then introduce the indexing algorithm and show how it can answer these queries and queries like them.

2.4.1 Sample Range Queries

```
select * from R where R.version="c1" AND R.field > value
```

Figure 2-7: Example Single Version Range Query

Figure 2-7 is an example of a single *version (commit)* range query that selects all records with *field* > *value* from a particular commit. This commit can be any commit that exists in the system and is useful for finding relevant values from some historical version of the dataset when examining changes of sets of records across the lineage.

```
select * from R where R.version="c1" OR R.version="c2" AND R.field > value
```

Figure 2-8: Example Cross Version Range Query

Figure 2-8 is an example of the type of cross version range query that a user may request to the system.This query requests all records that have *field* > *value* and exists in either version *c1* or *c2*. This can be generalized to any logical combination of versions.

2.4.2 Indexing Scheme

The goal with these indexing algorithm is to find records that satisfy a predicate (e.g. $age > 5$) but only report the matching records that belong to a particular group of commits. A naive solution would be to build a separate index on the desired field (e.g. *age*) per commit and then visit each index to answer the range query on each desired commit. First, the storage overhead alone soon makes this solution prohibitive even with a modest number of commits on a moderate size data set. Furthermore, a lot of the data between commits is shared and so this storage overhead is somewhat artificial. Second, performing a linear number of separate range queries is inefficient since for this incurs numerous random page reads when conducting the index search for disk bound indexes.

Before discussing the indexing scheme in detail, we make a few clarifications. The proposed solution works with an index on any attribute of a record. We make the distinction between a record's *key* and its *RecordId*. The *key* is the field on which the index is built (e.g. age). The *RecordId* is the record's physical location in some segment file (i.e. combination of segment file, offset within file, e.g. page number, tuple number).

The solution works as follows: have a single traditional index (e.g. B-tree) store a set of RecordIds for a particular key. These RecordIds represent where records with that key are physically located across segments. This with the version bitmaps for every commit is enough to answer any cross version range query that involves any logical combination of the versions.

The query in Figure 2-7 can be answered as follows:

1) Use the index to find the sets of *RecordId*s with $field > value$. Take the union of these sets.

2) Group these *RecordId*s by segment file and create an in-memory bitmap that records the record numbers in that segment that correspond to answering the range query. Call these sets of bitmaps *RB*.

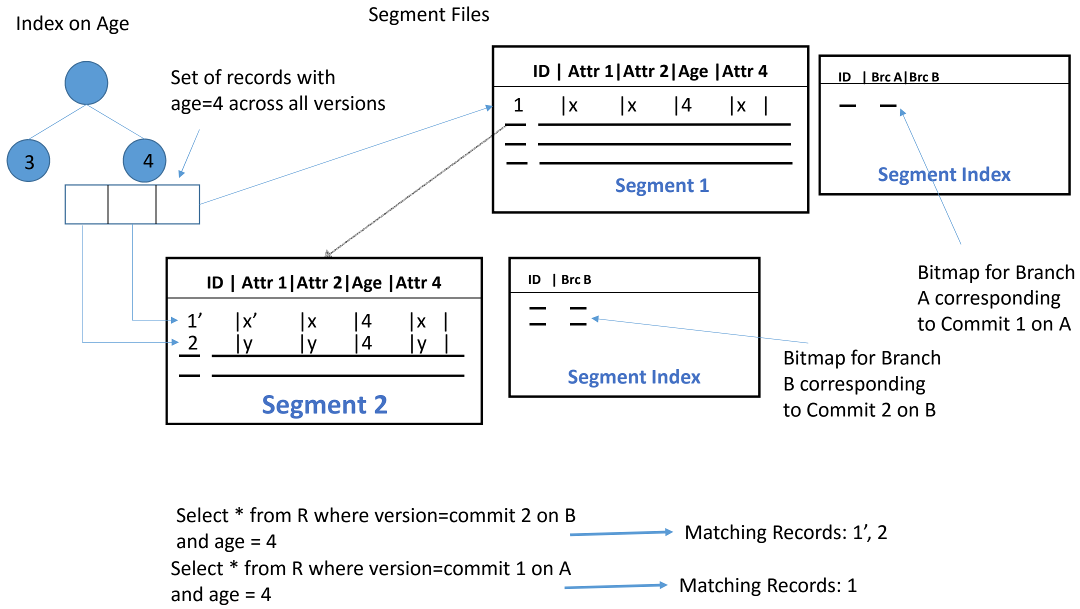


Figure 2-9: Indexing Example

3) Use the pack files to retrieve the per segment bitmaps to get the records in that commit. Call these sets of bitmaps *VB*.

4) For each bitmap in $RB \cap VB$ (there are bitmaps for the same segment) for the same segment perform the logical AND between the bitmap in *RB* and *VB*. Ignore the bitmaps not in the intersection since they contain records that are not in the specified commit or do not satisfy the range query. The remaining bitmaps contain exactly the physical locations of all the records that are in the specified version and satisfy the range query.

5) Sort the *RecordIds* such that each segment is visited once and sequentially and report out the satisfying records.

What makes this work is that the index just stores the places where any such record with the desired key could exist (the RecordIds) and then we use the bitmaps from the desired commits to pick out the relevant records that also match the range query. This is illustrated in Figure 2-9 where to get all the records with $age = 4$ from commit 2 on branch B requires looking at the index on age to find the records in the system where the age is 4 and the consulting the appropriate bitmaps to find the records satisfying the range query and belong to the desired version.

The query in Figure 2-8 can be answered in a similar way except VB is constructed by using the version bitmaps for both commits to construct the bitmaps per segment that contains records in any logical combination of the versions (e.g. AND, OR, etc.). This works for both primary key and secondary indexes.

Decibel's indexing scheme leverages the version bitmaps and involves only a single traditional index. Each update to a record involves an insert into the index; nothing is ever removed from the index so the index grows in linearly with the number of updates. Also, because the index tracks records with a particular key across multiple commits and branches, obtaining records in a particular branch becomes more expensive. Thus, it is beneficial to maintain a traditional index for each branch that only tracks the most recent versions of records in that branch, excluding older records from prior commits. This incurs O(branch size) overhead per branch but can dramatically speed up the typical data operations on a single branch for users only concerned with the current state of that branch, e.g. a user updating a record with primary key = "david".

2.5 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. Although highly concurrent use of versioned systems is possible, we believe that in most cases these systems will be used by collaborative

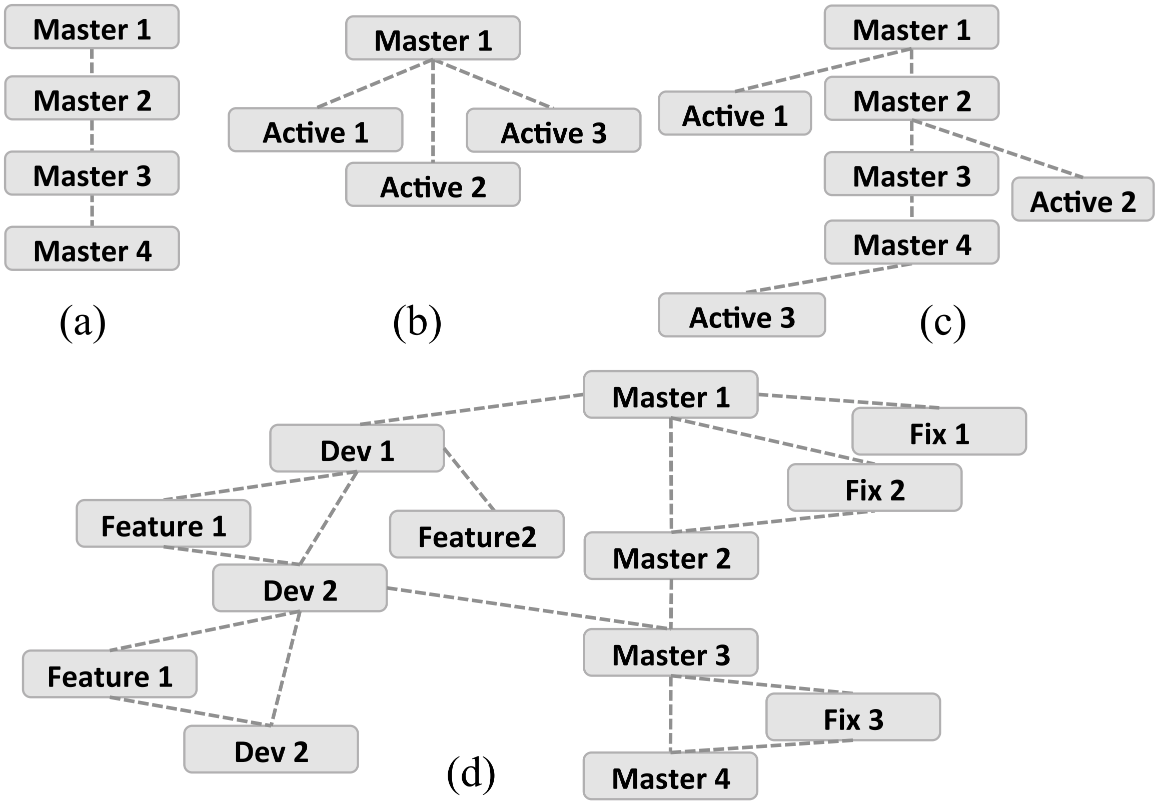


Figure 2-10: The various branching strategies in the versioning benchmark: a) Deep b) Flat c) Science (Sci.) d) Curation (Cur.)

data analytics and curation teams where high levels of concurrency in a single branch is not the norm.

2.5.1 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 2.1. Figure 2-10 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 2.1, 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 2.1, 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).

2.5.2 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 2.6 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.

2.5.3 Evaluated Queries

The queries targeted in our benchmark are similar to those in Table 2.1; 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.

A benchmark run consists of the time to evaluate each of the four queries on the four branching strategies, at a particular dataset size, and with a particular set of test storage engines (tuple-first, version-first, and hybrid, in our case.)

Our benchmarking software, including a data generator and benchmark driver (based on YCSB [21]), is available at `http://datahub.csail.mit.edu/www/decibel`.

2.6 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 2.5. 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 2.5.3 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.

2.6.1 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.

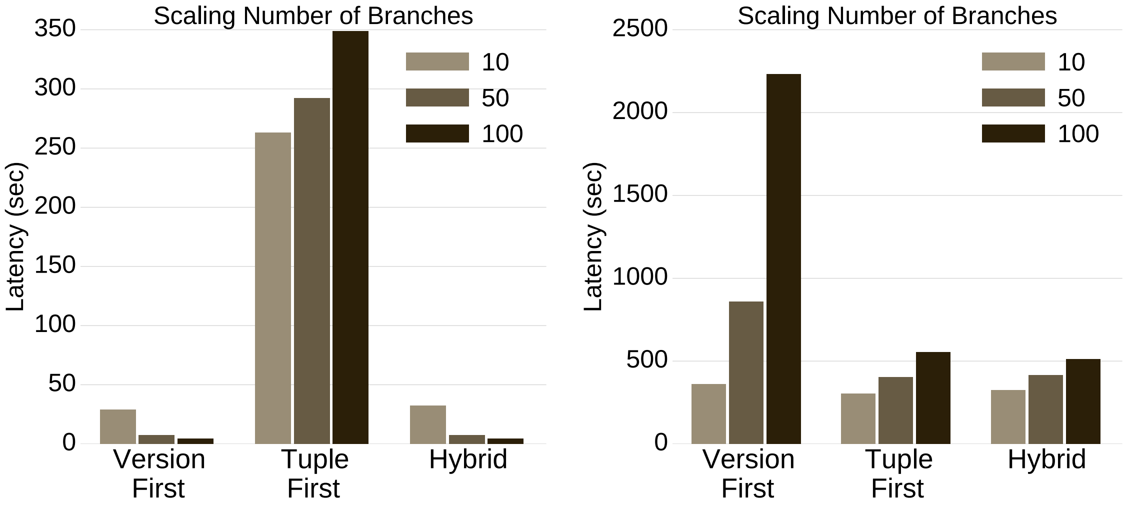
Figure 2-11a shows how the storage models scale across structures with 10, 50, and 100 branches for Query 1 on the flat branching strategy. As tuple-first stores records from all versions into a single heap file, ordered by time of insertion, we see single-branch scan times for tuple-first greatly underperform 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 2-11b. 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 2.3 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.

2.6.2 Query Results

Next, we evaluate all three storage schemes on the queries and branching strategies described in Section 2.5. All experiments are with 50 branches. Note that the deep



(a) Query 1 on Flat

(b) Query 4 on Deep

Figure 2-11: The Impact of Scaling Branches

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 2.6 for the specific dataset sizes.

Query 1 (Q1): Figure 2-12 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

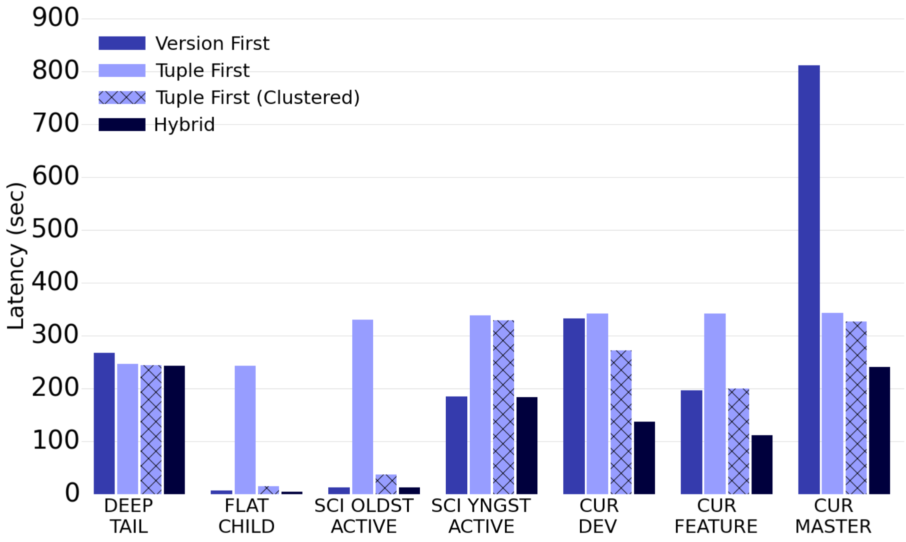


Figure 2-12: Query 1

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 2-13 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

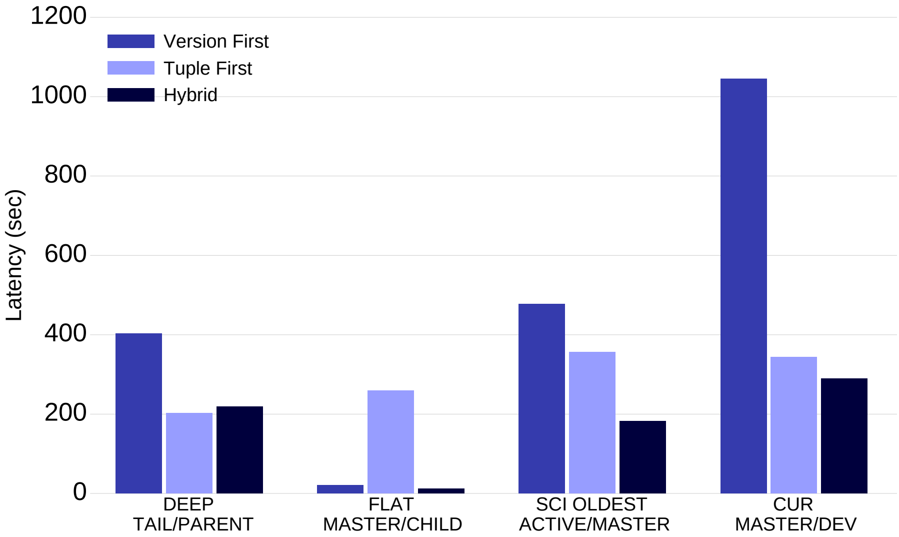


Figure 2-13: Query 2

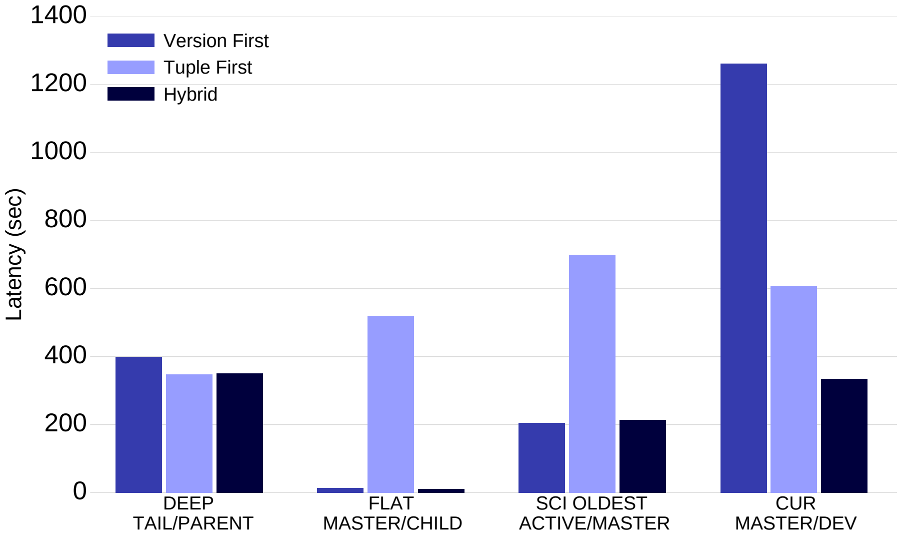


Figure 2-14: Query 3

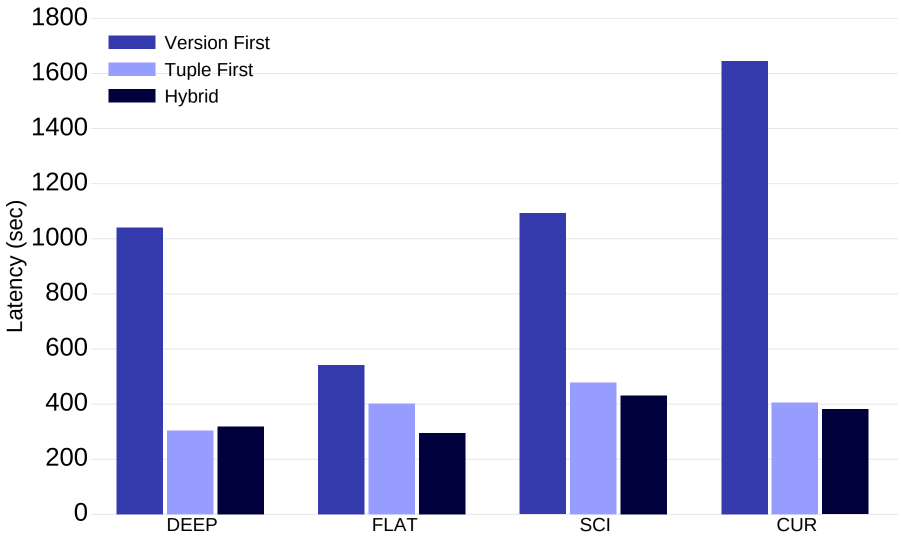


Figure 2-15: Query 4

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

Query 3 (Q3): Figure 2-14 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 2-15 depicts the results for Q4 with full data scans to emit

Table 2.3: 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

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.

2.6.3 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 2.3 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 the system. Recall from Section 2.3.2 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.

Table 2.4: 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 2.5: Storage Impact of Table-Wise Updates from Figure 2-16, in GB.

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

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. We flushed disk caches prior to each commit/checkout.

2.6.4 Merge Performance

Table 2.4 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.

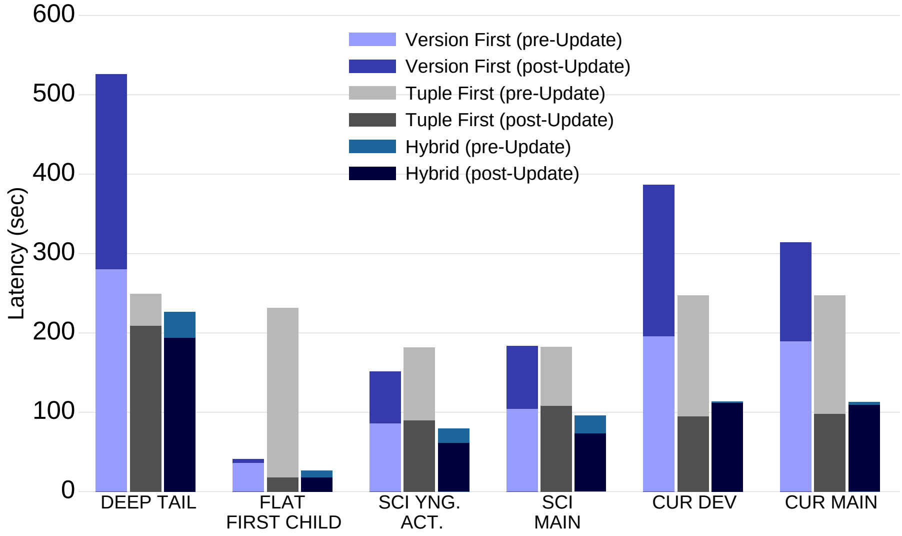


Figure 2-16: Table-Wise Updates: Query 1 (10 Branches)

2.6.5 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 2-16 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 proportionately 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 2.5 displays the data set size increases corresponding to the operations represented in Figure 2-16. (Note that the size increases are agnostic to the storage model used.) Such space overhead also arises in other no-overwrite systems, e.g., Postgres and Vertica. This can be mitigated with

Table 2.6: 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

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

2.6.6 Load Time

Table 2.6 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).

2.6.7 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 formats to compare differencing operations. Note that CSV results in a larger raw size due to string encoding. We use the benchmark from Section 2.5.2 and configured `git` with default settings.

Table 2.7 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. Despite use of compression, binary files have a large storage overhead as a result of git being unable to diff and delta chain binary files, so the entire binary object is stored again in the git object database even for a small changes. 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. We ran `git repack -d`, `git prune-packed`, and `git prune -expire now` to compress data before profiling version checkouts and reduce the repo size. Memory consumption used by `git` for each commit was equal to the total data set size at that point (`git` uses a differencing algorithm which compares entire versions against one another in memory). 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 2.7 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 2.8 shows similar performance numbers for a more update-intensive workload. For structured, multi-versioned data a hybrid of indexes and delta chains can improve performance while supporting lightweight commits. Additionally, concurrency with `git` relies on cloning an entire repo to isolate changes. Embedding versioning into a database enables existing concurrency control schemes without materialized snapshots and allows operations to efficiently query multiple versions.

Given that Decibel and Git offer very similar functionality, in some sense, this paper "is" showing how to do an adequate implementation of Git on top of a DBMS, and what the different tradeoffs are. As a specific example, a key design question for Git is the format of the "pack-file", i.e., exactly how a large number of objects (across many versions) should be stored compactly together so that objects can be retrieved efficiently and storage space is minimized. Git's solution to that problem is through use of delta chains. However, as we show in this paper, using deltas is not a good idea when we are dealing with relational and structured data, and instead, a hybrid of bitmaps and deltas may be preferred. Thus, our paper shows how the pack-file structure should be redesigned to handle structured datasets. Our paper also shows how the different operators should be reimplemented in light of this new format of a pack-file.

Furthermore, `git`'s concurrency control model is to clone the entire repo locally, operate on this local workspace, and then merge with other copies of the repo to incorporate the changes from other users. From a transactional standpoint this is infeasible since most datasets are several hundred GB (if not hundreds of TB), so forcing each transaction to clone a repo is not practical. In turn, as we propose a purely remote solution is required where the DBMS managers have the infrastructure to support large datasets (note that this remote infrastructure could be replicated itself to ensure fault tolerance, the key difference is every user does not need a copy).

Table 2.7: git performance vs. Decibel (Hybrid) on the deep structure with 100% inserts to 10 branches over 10000 commits

	Data Size	**Repo Size**	**repack Time**	**Commit Time**	**Checkout Time**
	(MB)	**(MB)**	**(sec)**	$\mu \pm \sigma$ **(ms)**	$\mu \pm \sigma$ **(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

Table 2.8: git performance vs. Decibel (Hybrid) on the deep structure with 50% updates to 10 branches over 10000 commits

	Data Size	**Repo Size**	**repack Time**	**Commit Time**	**Checkout Time**
	(MB)	**(MB)**	**(sec)**	$\mu \pm \sigma$ **(ms)**	$\mu \pm \sigma$ **(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

Furthermore, `git` doesn't support high performance concurrent operations on the same repo on the same machine and only allows one branch to be checked out a time. Thus, despite `git`'s ability to store the dataset more compactly, it lacks support required for high levels of concurrent transactions. Decibel can provide random access to every commit in the system without blocking new commits from being creating by obtaining the relevant bitmap columns from the packfiles, which can be cached in memory. All historical reads can go on in parallel. Furthermore, Decibel can support transactions using the branching model and though a lock may be acquired when creating the branch (capturing a snapshot of the global version graph), all subsequent operations to that branch can go on independent (lock-free) of other transactions operating on different branches. Overall, `git` is a general purpose DVCS, but Decibel shows a better approach can be taken to suit the needs of a versioned DBMS.

Chapter 3

Transactions

In this chapter we discuss the concept of a *versioned transaction*, a transaction that can include versioned operations and how the natural versioning capabilities of Decibel can be leveraged to support transactions and isolation between transactions. We also introduce a conflict model for versioned operations and present Decibel's concurrency control protocol and implementation, followed by an evaluation under different workloads and analyze the benefits and issues of using Decibel versioning primitives for transaction isolation.

3.1 Preliminaries

A *versioned transaction* is a standard database transaction that includes versioned operations (`commit`, `branch`, `merge`). Decibel handles concurrent versioned transactions, making them both durable and atomic (ACID guarantees). Each concurrent transaction's changes must be *isolated* from each other until transaction commit time, when those changes become *visible* to subsequent transactions and the system guarantees *durability.* To avoid confusion, we differentiate the *versioned commit* operation from *transaction commit*, by using *vc* to indicate a *versioned commit* and *tc* to specify *transaction commit.*

As per the traditional database transaction isolation definition, all updates must be isolated (hidden) from concurrent transactions until commit time when the transactions changes are made globally visible to all subsequent transactions. Furthermore, transactions that are still running should not see a new view of the database state that conflicts with what they have already read/written. We extend this definition of isolation to include version graph changes, any branches, commits, or merges edges produced by a transaction should be isolated from other transactions in the same way as data changes and transactions should not observe conflicting views as new transactions *tc* their changes.

The correctness criterion for these concurrent *versioned transactions* is *serializability* [71], i.e., that all of the transactions should appear to have run in some serial ordering despite the actual operations within the transaction being interleaved to achieve higher performance.

We adopt the following model for concurrent changes to the versioned data set:

- Each record has a *primary key* that not only is used to track the record across versions, but also to identify serialization conflicts.

- Each *branch* has a *workspace* that reflects the current state of that branch that contains the cumulative effects of all changes. Specifically, a branch workspace is the entry point for all changes to that branch and a *vc* of a workspace creates a new named snapshot in the version graph that can be retrieved at a later time. Thus, the workspace contains the cumulative effects of all prior *vc* and any dirty changes since the last *vc*. A workspace with changes since the last *vc* on that branch is called *dirty*. See Figure 3-1. Note: consider this separately from transactions, just think of it with respect to a single user operating on a branch.

- Updates and new changes to individual records in a branch are directed to the *workspace*.

- New *head vc*s are created from the branch workspace.

- `branch` creates a new branch from the current *head* commit of the parent branch, not the workspace. To branch the workspace, the parent branch must be *vc*ed.

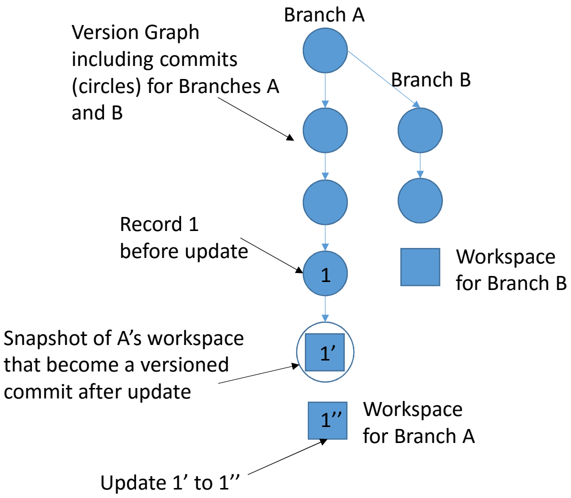

Branch A
Version Graph including commits (circles) for Branches A and B
Branch B
Record 1 before update
1
Workspace for Branch B
Snapshot of A's workspace that become a versioned commit after update
1'
1''
Workspace for Branch A
Update 1' to 1''

Figure 3-1: Branch Workspaces

This prevents gaps in the version graph from forming. For instance, let's say record 5 is updated in branch A with no *vc*, then branch B is created from A, if B inherits the change to 5 then since 5 was not part of any *vc* on branch A, the lineage of this record is not well defined in B. The lineage is not well defined because a subsequent change to record 5 in A will overwrite this earlier change and without a *vc* on A, there is no way to track where B's initial record 5 came from. Thus, a *vc* on the parent branch is required before branching to ensure the parent's workspace is properly snapshotted.

- Modifying non-head *vc*s can be achieved by branching the desired *vc*, making the modification, and then merging back into the parent branch.

- `merge` must be done from the *head vc* of the secondary parent to the *head vc* of the primary parent. This creates a new *head vc* on the primary parent's branch. If there is any state that has not been *vc*ed on either branch (dirty branch workspaces), a `merge` creates new *vc*s on these branches before merging. This is particularly required on the primary branch (merge target) to prevent smashing updates in the workspace and maintain proper provenance tracking required for subsequent merges.

These rules together yield a consistent versioned database where *vc*s are appended to the version graph and once appended (via transaction commit) become immutable. This last feature is particularly crucial for parallelism and high throughput in answering historical queries.

3.2 Conflict Model

There are numerous challenges in handling concurrent changes to a versioned database which together necessitate the development of a new conflict model on which traditional concurrency control schemes [48] can be applied and augmented. The complexities stem from not only maintaining transactional consistency of the collection of data items in the system, but also maintaining consistency of the *version graph* itself. In the following discussion we assume that both transactions start with a consistent view of the version graph and the underlying data. A *transactionally consistent* view is a snapshot of the database that reflects only the effects of the committed transactions in some serial order (serializable).

We now introduce and discuss certain types of conflicts that can occur between transactions. To begin, for the case where each concurrent transaction each reads and modifies a different branch alone then serializability is trivially achieved because operations on different branches generally commute with each other (unless a prior versioned operation on a branch introduced a constraint), so any ordering of those transactions produces the same final state on each branch. Also, clearly it is the case that reading the data in *vc*s that existed at the time the transaction started requires no conflict tracking/avoidance as those *vc*s have their positions solidified in the version graph and the data in those commits is immutable. Conflicts between transactions are not created from accessing historical data, they are created from changes to the branch workspaces and the addition of new *vc*s. We now delve into the conflicts produced when transactions try to read and modify the same sets of branches.

The discussion below describes record level serialization conflicts in terms of Optimistic Concurrency Control [48]. In OCC a transaction's execution is broken up into a *read phase*, *validation phase*, and *write phase*. In the read phase, the transaction records the identifiers of the records it read and wrote, when it writes a record it copies it from the database and makes a copy in a private workspace to provide isolation. Then when the transaction *tc*s it is validated for serialization conflicts, and if it passes validation then the transaction can copy its changes over to the database in the write phase. Validation checks for serialization conflicts as follows. A record (or set of records) serialization conflict occurs for transaction $T1$:

1) If the write set of any transaction that committed since $T1$ started intersects $T1$'s read set (in the serialization order $T1$ was required to see the other transaction's changes and it may not have been the case depending on when $T1$ read the record, e.g. before the write phase of the earlier transaction).

2) If the write set of any transaction that is running concurrently with $T1$ intersects $T1$'s read set or write set (there is a conflicting change and both cannot be serialized, one needed to see the change of the other).

The discussion tries to remain neutral to how a versioned system is implemented and talks more generally about the types of conflicts that arise in any versioned system. However, specifics are used when appropriate to provide clarity and concrete examples.

3.2.1 General Issues

There are two main points of contention: version graph appends (e.g. adding commits, branches, and merge nodes to the version graph) and branch workspace updates. Not only do branch workspace updates have to be atomic across multiple branches, but also the version graph changes have to be atomic across all branches. For instance, it cannot be the case that transaction $T1$ *vc*s on Branch A and B and $T2$ does the same and $T1$'s commit appears before $T2$'s in A but after $T2$'s in B.

3.2.2 Single Branch Concurrent Update to Branch Workspace

Here we describe the case where a transaction just makes an update to a branch and no *vc*, the transaction just updates the workspace of a branch. This case is most analogous to modifications in a standard database since it involves no modification of the version graph, just a modification to the branch's workspace. Reads/writes need to be tracked here to serialize changes to a branch's workspace. Furthermore, since creating new *vc*s requires first modifying the branch's workspace, changes to just the workspace conflict with concurrent *vc*s and *non-historical* branches, described next.

3.2.3 Single Branch Concurrent *vc*

In this section we describe serialization problems with respect to concurrent *vc*s to the same branch. Consider two users that start transactions $T1$ and $T2$, update disjoint records on branch A, both *vc*, and both *tc*. Without loss of generality, $T1$ *tc*s first. $T1$'s *vc* node now appears in the version graph as the new *head* for A. At this point $T2$'s view of the version graph has diverged. This appears to be a conflict causing $T2$ not be serializable, but in fact it is serializable. The key insight stems from the observation that even though $T2$ created a new *vc* which is a logically complete snapshot of the dataset, it never read or wrote $T1$'s record so from $T2$'s point of view it does not matter whether it has the new record in its snapshot, it never observed or modified the state of it. Thus, the serial ordering becomes $T1$ followed by $T2$ when $T2$ *tc*s and appends its node to the version graph. However, $T1$'s changes need to be incorporated into $T2$'s *vc*. This process is called *rebasing*; $T2$'s *vc* is rebased on $T1$'s *vc* to include $T1$'s changes. This is similar to the `git rebase` command [98] and involves re-writing a transaction local version graph history to create a new global history that incorporates a transaction's changes. See Figure 3-2 for an example of this process. This establishes a transactionally consistent lineage and will be discussed more in 3.3.7. Note that rebasing a *vc* incorporates that *vc*'s changes into the branch workspace, because the *vc*'s changes need to be present in the current state of the branch. If the operations performed by both transactions did conflict, e.g. $T1$ and

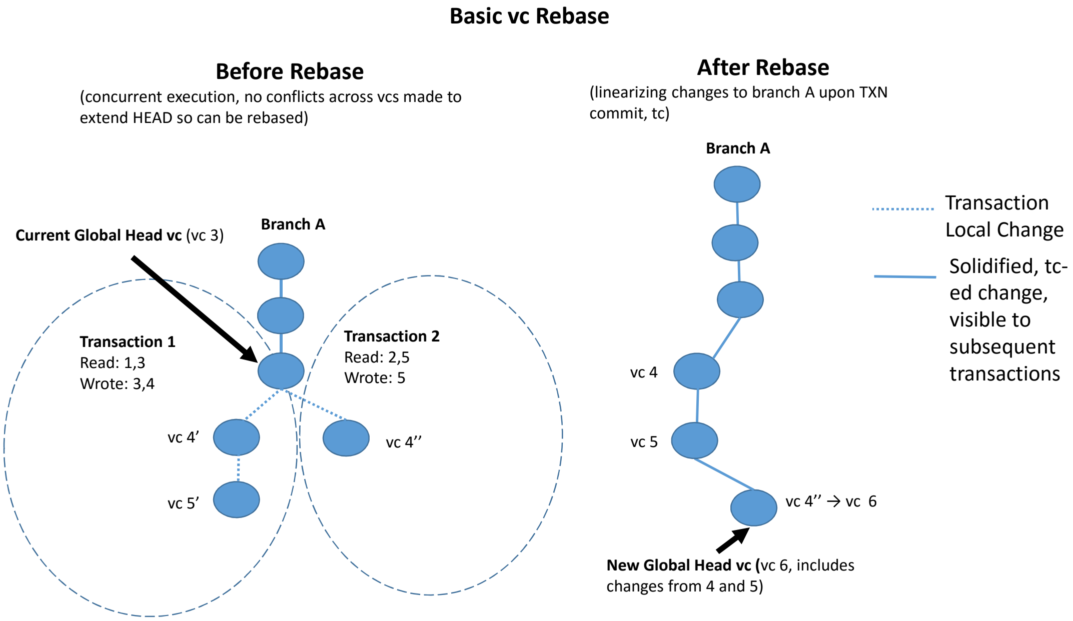

Basic vc Rebase
Before Rebase
(concurrent execution, no conflicts across vcs made to extend HEAD so can be rebased)
After Rebase
(linearizing changes to branch A upon TXN commit, tc)
Branch A
Current Global Head vc (vc 3)
Branch A
Transaction 1
Read: 1,3
Wrote: 3,4
Transaction 2
Read: 2,5
Wrote: 5
vc 4'
vc 5'
vc 4''
vc 4
vc 5
vc 4'' → vc 6
New Global Head vc (vc 6, includes changes from 4 and 5)
Transaction Local Change
Solidified, tc-ed change, visible to subsequent transactions

Figure 3-2: Rebasing vcs

$T2$ both read and wrote record $R1$ and then vced, one of the resulting vcs is incorrect since one must have observed the results of the other. It is worth noting that if instead transaction $T1$ just performed a workspace update and did not create a vc, that $T2$ vc would still have to be rebased to include the changes from $T1$s update, otherwise the new vc is not consistent with what the transaction that tced before it did. See Figure 3-3.

3.2.4 Concurrent Branching

In this section we describe serialization problems with respect to creating new branches within a transaction. There are two types of branching, create a new branch from a *historical* vc and create a new branch by creating a new vc on the parent branch and branching that vc. This latter branch type is called a *non-historical* branch operation. *Non-Historical* branches present a serialization problem since they involve transaction local modifications to the parent branch that may need to incorporate the changes of a concurrent transaction's vc that tced before this transaction tcs. Figure 3-4 defines and demonstrates the difference. Consider two transactions $T1$ and $T2$, $T1$ and $T2$ both vc on branch A. $T2$ creates branch B from the vc it created on Branch A. Branch

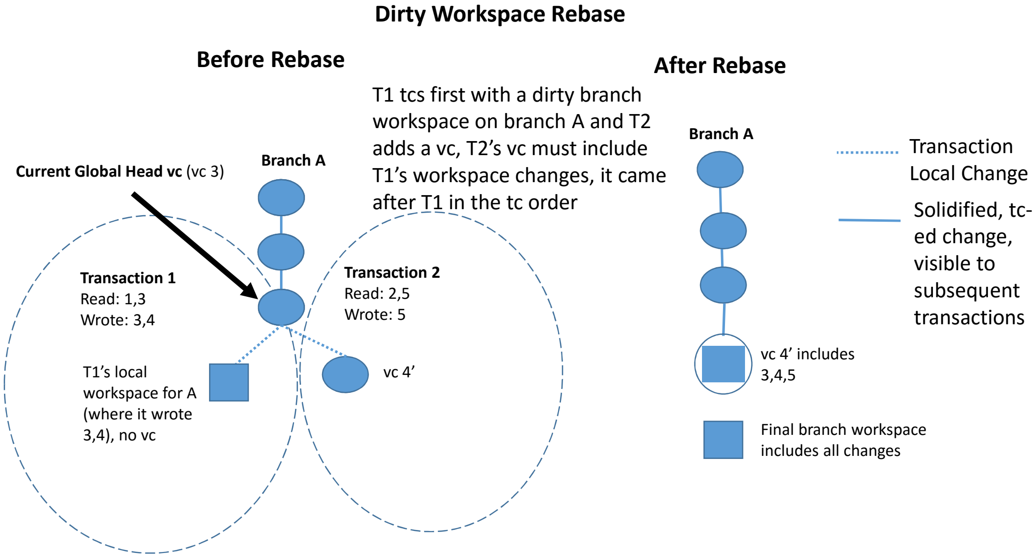

Dirty Workspace Rebase
Before Rebase
After Rebase
T1 tcs first with a dirty branch workspace on branch A and T2 adds a vc, T2's vc must include T1's workspace changes, it came after T1 in the tc order
Current Global Head vc (vc 3)
Branch A
Transaction 1
Read: 1,3
Wrote: 3,4
T1's local workspace for A (where it wrote 3,4), no vc
Transaction 2
Read: 2,5
Wrote: 5
vc 4'
Branch A
vc 4' includes 3,4,5
Final branch workspace includes all changes
Transaction Local Change
Solidified, tc-ed change, visible to subsequent transactions

Figure 3-3: Rebasing vcs from a Dirty Workspace

B is a *non-historical* branch. $T1$ *tc*s. $T2$'s *vc* to branch A must now be rebased to include $T1$'s changes. This also affects $T2$'s branch B because its creation *vc* depends on its *vc* on A which needs to be rebased when $T2$ *tc*s. Generally, in such a scenario if the transaction with the *Non-Historical* branch *tc*s, its *vc* and in turn the branch created from that *vc* must incorporate the changes from the updates or *vc*s it did not see by transactions that *tc*ed earlier, otherwise the branch is not consistent with the lineage of and modifications to the parent branch. Note this applies recursively to subsequent *vc*s on a non-historical branch and subsequent branches derived from *non-historical* branches created by the same transaction, dubbed *recursive non-historical branches*. Furthermore, if the transaction with the *non-historical* branch reads and updates a record in that *non-historical* branch that is also updated by a concurrent transaction in the parent branch, then the *non-historical* transaction can no longer be serialized. The problem stems from the fact that once a *vc* or workspace change is exposed to subsequent transactions it cannot be re-written or modified without producing conflicting views of the versioned database's snapshots. Thus, any transaction that adds *vc*s to a branch must include in its new *vc*s changes from the *vc*s created since the time the transaction started (and the most recent workspace changes) to preserve the semantics of the version graph. It might seem that this can be fixed by transforming the *vc* on the parent branch that caused these problems (the *vc* on the parent branch

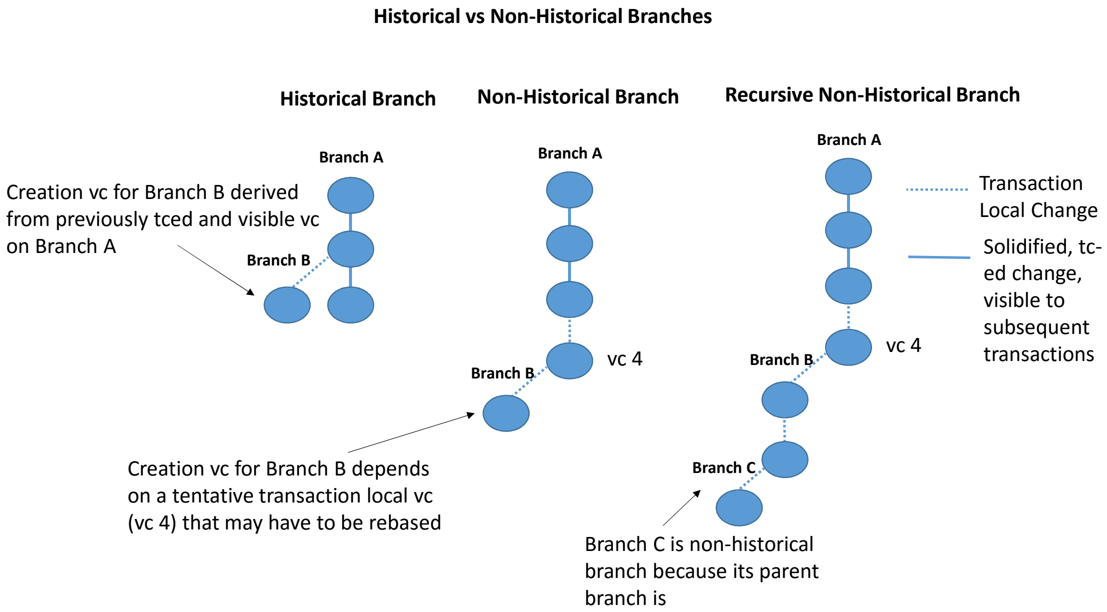

Historical vs Non-Historical Branches
Historical Branch
Non-Historical Branch
Recursive Non-Historical Branch
Branch A
Branch B
Creation vc for Branch B derived from previously tced and visible vc on Branch A
Branch A
vc 4
Branch B
Creation vc for Branch B depends on a tentative transaction local vc (vc 4) that may have to be rebased
Branch A
vc 4
Branch B
Branch C
Branch C is non-historical branch because its parent branch is
Transaction Local Change
Solidified, tc-ed change, visible to subsequent transactions

Figure 3-4: Difference between Historical Branches and Non-Historical Branches

from which the non-historical branch was derived) into a new branch itself, but this violates the notion of what it means to create a *vc* on the parent branch, i.e., the *vc* is a part of the parent branch where it was added.

Historical branches are always serializable because they are created from a transactionally solidified and consistent (*historical*) part of the ancestry. Thus, if transaction $T1$ adds a new *vc* to the parent branch, another transaction $T2$ that creates only a *historical* branch commutes in the serial ordering with respect to $T1$. This branch by $T2$ could have occurred either before or after $T1$'s *vc*, and from the semantics of `branch` in a branched versioned database, the ordering does not matter. The same argument applies if $T1$ just updated the branch workspace.

3.2.5 Extended Conflict Tracking: Changes Across Multiple Commits and Branches

Up until now we have only considered the problems when a transaction makes a single version graph change in a specific way, we now discuss how multiple *vc*s and created branches affect serialization. The previous analysis showed that concurrent modifications to the version graph are serializable as long as the transactions did not read/write any conflicting records in a concurrent *vc* or workspace change. Now we extend this to any (across all) of version operations (e.g. *vc*, `branch`). For instance, if two transactions add several *vc*s to the same branch, as long as they do not read/write any of the same records then the group of *vc*s can be placed one after the other in the branch lineage when the transactions *tc*. Records read/written to over the course of all these *vc*s must be tracked (or locked) to determine if there is a serialization conflict.

Another case where record level conflicts need to be tracked across multiple versions and that introduces complexity is the case of recursive *non-historical* branches, branches derived from *non-historical* branches. Because the *vc* a *non-historical* branch was derived from (its creation *vc*) may need to be rebased (is in contention so to speak), any *vc* or branch derived from it is also in contention (may need to be rebased) and must be removed if there is a serialization conflict. A transaction with a *non-historical* branch must be aborted if any update to the *non-historical* branch or any of its descendants conflict with any update made to the originating ancestor branch (branch where the conflict was first introduced, branch from which the first *non-historical* branch was created). The reason for this is that the update to the originating ancestor branch would cause the creation *vc* of the *non-historical* branch to be rebased. If this introduces a new version of a record that the transaction previously updated in the *non-historical* branch then its update is incoherent with respect to what the *non-historical* branch should have started with and the result is not serializable.

Additional complexity stems from the observation that the conflict checking/avoidance should only be done with respect to the originating parent branch where the conflict was first introduced. Concurrent changes to different non-historical branches should not conflict with each other. This is true simply because the updates are to disjoint branches that are not derived from one another. For instance, transaction $T1$ creates a *vc*, lets call it $C1$, on Branch A and then creates a *non-historical* Branch B from $C1$. Transaction $T2$ does the same and creates a new Branch C, its *vc* on Branch A is designated $C2$. As long as 1) none of the changes in $C1$ conflict with the changes in both $C2$ and Branch C, and 2),symmetrically, none of the changes in $C2$ conflict with the changes in $C1$ and Branch B then both can be serialized after some rebasing, depending on which transaction *tc*ed first. Note that the same record can be updated in Branch B and Branch C as long as these updates did not occur in either $C1$ or $C2$, which would threaten the creation *vc* of one of the *non-historical* branches. Once branch creation (the creation *vc*s of the non-historical branches) can be serialized, changes to these two different branches should not conflict with each by the virtue of being different branches. Thus, changes in B and C should only conflict with changes in A and not each other. However, what this means is that subsequent reads/writes of branches B and C (to the worksapce and any commits on those branches) must be tracked so as to detect/avoid conflicts with A. Again, this also applies to any subsequent branches derived from B or C within the same transaction that created them, the recursive *non-historical* branches.

In contrast, there are cases where conflicts do not need to be tracked, i.e., when a transaction accesses a *historical* (different from the *non-historical* case above) since these changes are independent of the flow of the rest of the version graph and so are always serializable.

3.2.6 Branch Level Conflicts

So far we have only considered how *point* record modifications across the versioned operations made by a transaction can cause serialization issues. There are types of branch level operations that do conflict with concurrent version graph changes even though the data contained in those new *vc*s or branches may have never been read/written by a transaction. For instance, a transaction that lists all or number of *vc*s back from the current *head vc* of branch (e.g. `git log`), we will call this the `listCommits(branch)` command. A `listCommits(branch)` conflicts with any concurrent append of a *vc* to that branch if a transaction that previously observed the set of *vc*s, calls `listCommits(branch)` again and observes the change. This is analogous to a phantom read [71, 10] for the set of *vc*s on a branch. The `listBranches()` (e.g. `git branch`) command that encounters the same problem on the set of branches that exist. Operations such as `diff` on the workspace of a branch or a new *vc* created by the transaction are branch level data read operations that conflict with any concurrent update or *vc* to the branch.

3.2.7 Merge

In this section we discuss the complications that arise when merging concurrently with other transactions performing versioned operations to the branches being merged. To summarize, merge is essentially a table level update and so not only do concurrent changes to the primary parent's branch conflict, but also changes to the secondary parent's branch have to be considered as well to ensure serializability.

Merge involves creating new *vc*s on both the parent branches prior to the merge and then results in an additional *merge vc* on the primary parent's branch. Thus, merge conflicts with workspace changes, new *vc*s, and non-historical branches on the primary branch. To make matters worse, merge generally involves reading/writing a large amount of data (average diff sizes as reported in Chapter 2 ranged from a couple hundred megabytes to a few gigabytes), making the probability of a merge transaction producing a record conflict with any transaction updating the primary branch very high.

Counter intuitively, any *vc* added to the secondary parent's branch also presents an issue because the definition of merge only allows *head*s of branches to be merged. If a merge results in a *vc* on the secondary branch (because its workspace is dirty), this *vc* may have to incorporate data from the new *head vc* (or just workspace update) added by a concurrent transaction. Thus, the merge as it occurred based on the *vc*s it had when it ran is incomplete and must include the data added in the *vc*s and workspace updates it did not see. This could introduce new merge conflicts, causing parts of the merge to be redone (e.g. consider field level merge, it involves creating new records). To make this concrete consider the following example:

1) Transaction $T1$ and $T2$ start at the same time and see the same initial version graph.

2) Transaction $T1$ starts a merge from Branch B (secondary parent) to Branch A (primary parent). $T1$ created a new *vc* on Branch B because it has a dirty workspace.

3) Transaction $T2$ adds a *vc* to Branch B and *tc*s before $T1$. The problem here is that $T1$ *tc*s physically in time after $T2$. $T2$ has already moved the *head* on B so $T1$'s new *vc* will have to be rebased on $T2$'s *vc* when $T1$ *tc*s, which in turn affects the merge $T1$ conducted since the data being merged from the secondary parent (B) might be different. For instance, let's say without $T2$'s changes $T1$ performed a field level merge between record R and R' in branches A and B, respectively. $T2$ updated R' in B to R'' and so the resulting record that $T1$ wrote to resolve the merge conflict is incorrect since it does not include the changes from $T2$ ($T1$ merged R and R' instead of R and R'').

These issues are mitigated to some extent by choosing a more relaxed merge (e.g. 2 way merge) procedure that involves just taking one record over another instead of producing a field level merged record. However, the problem still remains that the merge *vc* produced as of when the transaction executed the merge is incorrect and must be patched to include additional changes if a *vc* or update was made to either parent branches by a transaction that *tc*ed before the transaction conducting the merge.

Despite these challenges, all hope is not lost, because in principle a merge operation in a transaction by itself can be made serializable. Given the complete set of transactions, just performing the merge at the end is enough for serializability. This is correct because the merge operation does not care about the specific records taken during the merge, just that the merge makes sense with respect to the data in the *vc*s in the version graph and the version graph itself. The lesson here is that merge must somehow be delayed or view a transactionally consistent version graph and concurrent changes must be prevented.

One final issue must be addressed and that is how merges conflict between branches created by a transaction and the branches that were visible prior to transaction start.

- A merge between two *historical* branches created by a transaction incurs no conflicts with any other transaction because these branches are local to the transaction and can incur no concurrent modifications.

- A merge from a *historical* branch (secondary parent) created by a transaction to a currently publicly visible branch (existed prior to transaction start, was added by another transaction that successfully committed) incurs all the conflicts and conflict tracking on the primary branch, which already seems prohibitive.

- A merge from/to a *non-historical* branch must handle conflicts on the ancestor branch it was created from because any concurrent change to that ancestor branch may again cause new changes to be incorporated into the new branch and for the same reason as discussed with adding new *vc*s this causes problems for merge. Again, this also applies to recursive *non-historical* branches.

3.2.8 Relaxed Conflicts

It is worth noting that if a versioned system can guarantee that only transactionally consistent views of the version graph and branch workspace are ever observed (that is repeated reads to these structures return the same results with only the current transaction's modifications) that the issues described above only present serialization problems if *the transaction performs a read of a branch and then proceeds to update the same branch.* If a transaction wrote to/updated a separate branch from every other transaction then because operations between branches commute and the data the transaction read was transactionally consistent then the combination of branch read one branch and write another branch can be serialized before any additional update to the branch that was read.

To make this concrete consider the following example:

1) Transaction $T1$ gets a transactionally consistent snapshot of the database.

2) Transaction $T2$ does the same.

3) $T1$ reads branch A (does not matter if it is the version graph, point level data read, or branch level data read, e.g. diff) and then updates branch C.

4) $T2$ reads and updates branch A and *tc*s first.

5) Under normal OCC validation $T1$ would have to abort since $T2$'s write set overlapped its read set. However, since $T1$ never updated A and the snapshot it had is transactionally consistent, it does not conflict with $T1$'s changes. The serialization order is $T1$ and then $T2$ even though $T2$ committed first.

This can be generalized to sets of branches. What this suggests is that if a versioned system can guarantee transactionally consistent snapshot reads of the version graph and branch workspaces that some spurious conflicts can be removed.

3.3 Decibel's Concurrency Control Protocol

In this section we describe Decibel's concurrency control protocol for a versioned database at a high level and save the implementation details for 4.2. First, we provide an overview of the how the system manages isolation and discuss the invariants that Decibel maintains. Then we discuss how Decibel handles each of the problems specified in 3.2. In doing so we begin to outline what transaction local state is maintained and how the local non-conflicting changes of a transaction are incorporated into the global database state. Finally, we end on a discussion of how concurrent index updates are handled and the potential benefits of using bitmaps for transaction isolation.

3.3.1 Naive Solution: Branching and Merging for Every Transaction

One might envision a system where transactions branch each branch they intend to modify prior to accessing it and then direct all their writes to the new *transaction local* branch. Then at the end of the transaction merge their *transaction local* branch back into the originating branch. First, doing this in an atomic sense is a challenge in and of itself and using branching in this manner for isolation pollutes the version graph with temporary branches, which already contradicts the standard notion of a branch being long lived. Second, `branch` must be a relatively cheap operation and incur very little overhead for this to be effective and scale to thousands or tens of thousands of transactions. Unfortunately, since new branches in Decibel result in not only a new segment file but several new pack files (bitmaps) for every ancestor of the branch being modified, using this branch-per-*tc* strategy is very inefficient. Instead Decibel must do something much lighter weight, as described in the next section.

3.3.2 Overview and Invariants

Now we delve into the concurrency control protocol employed by Decibel. The primary goal here is to leverage the natural versioning of Decibel to implement transaction isolation (e.g. use bitmaps) as opposed to MVCC protocols that use timestamps, such as PostgreSQL [69]. The system should support high throughput concurrent reads that do not block, a read-only transaction should never have to wait and its answer should always be serializable. Since the system relies on the bitmap for transaction isolation and Decibel is append only, exposing the results of transactions boils down to atomically flipping bits in the appropriate bitmaps and atomically exposing version graph changes. This can simplify reasoning about changes to the state of the database and in turn concurrent transaction management. At a high level, this works because the bitmaps provide a lightweight snapshot of the database state and if updates to the bitmap are made in a controlled way then each transaction can get a set of bitmaps that contain exactly the records each transaction is allowed to view and none of the records added/changed by a concurrent transaction. However, since there are many bitmaps (one per branch per segment) that are disk bound, using the bitmaps for recovery purposes is an issue since forcing bitmaps to disk results in random I/O that makes *tc* slow even if batching is employed. Thus recovery is provided by traditional Write Ahead Logging.

Decibel ensures that each transaction gets a transactionally consistent and immutable *global snapshot* of the entire database state when it starts. The global snapshot just contains the meta data needed for a transaction to interpret the database state in a transactionally consistent way. When a transaction starts to update the database state if makes a local copy (copy on write) of the relevant *global snapshot* state it started with and modifies that, isolating its changes from other transactions. Creating a new global snapshot that incorporates the changes for a set of *tc*ing transactions exposes those updates to subsequent transactions and not transactions that have already accessed previous global snapshots. To preserve the non-blocking property for read only transactions, these transactions can just read the current global snapshot (since it is immutable) and the system employs a combination of OCC and locking to ensure transactions that update or read and update the database state do so serializably and have their changes exposed by creating a new global snapshot.

3.3.3 Global Snapshots

The global snapshot is essentially a Hybird storage model object with the appropriate branch workspaces and version graph so as to only expose a transactionally consistent version of the database state. A branch workspace in the context of Decibel corresponds to the current state of the all the bitmaps for every segment belonging to a branch in the current global snapshot. These are immutable and changes by non-conflicting transactions are only exposed by creating new global snapshots. At a high level, a global snapshot contains:

- The branch workspace for every branch in the system.
- Transactionally consistent prefix of the version graph.
- Copies of per segment meta data (this is much smaller than the bitmaps themselves).

3.3.4 Transaction Local Workspace Snapshot

Transactions get their global snapshots at transaction start and all operations go to the transaction's local workspace that uses the global snapshot as a starting point. The *transaction workspace* includes the relevant immutable *branch workspaces* from the global snapshot it acquired (if it is know that a transaction will not be operating on a particular subset of branches, then those bitmaps can be excluded). The *branch workspaces* are copy on write and updates are directed to a transaction's local copy. Note that a change to any of the branch workspaces within a transaction workspace does not modify the global snapshot from which it is derived, the branch workspace is modified locally to the transaction within the transaction workspace and subsequent reads of any of the branch workspaces within the transaction workspace will reflect the changes made by the transaction. A transaction can see its own updates on the original global snapshot, but other transactions cannot (its changes are isolated). The bitmaps are used to control visibility of concurrent updates even though the underlying segment files are constantly being mutated to incorporate new data that transactions are adding. A transaction's record level changes are installed directly in the segments a transaction changed, but the transaction gets copies of the relevant bitmaps from its initial global snapshot and modifies those directly in the transaction's *transaction workspace.*

The transaction also gets a *workspace version graph* which is a local version graph initialized with the head *vc*s in the version graph of the global snapshot. A transaction's version graph changes go to this *workspace version graph.* Whenever, a complete view of the version graph the transactions looks at the union of the workspace version graph and the global snapshot's (the one it started with) version graph.

When a transaction is ready to *tc* a new global snapshot is built that incorporates its transaction local workspace changes and this global snapshot replaces the old one, exposing the transaction's changes atomically to subsequent transactions. This process is called *global snapshot reconciliation.*

3.3.5 Conflict Tracking and Serialization

Optimistic Concurrency Control

Decibel adopts Optimistic Concurrency Control so that most operations are non-blocking and this is more conducive to a versioning system since OCC is naturally a versioned concurrency control protocol, but does not have the ability to create explicit versions.

Validation and Commit

Changes to a transaction's workspace are tracked as previously described and tracks conflicts as specified in Section 3.2 to determine serialization problems. A transaction is validated at *tc* time and if no conflicts are detected then the changes are used to build the next global snapshot for subsequent transactions.

Read Only Transactions

Since each transaction gets a valid serializable and immutable snapshot of all the transactions that mutated the database state, a read-only transaction does not need to be validated. It is serializable in the following way. Read only transaction T_p obtains a snapshot A that has the effects of transactions $T_1 ... T_k$ in some serial ordering. T_p's serialization number is exactly at T_k ($p = k$), seeing T_k's changes and before the next set of mutation operations that are used to create the new snapshot. Read only transactions that get the same snapshot can share the same serialization point. The OCC validation on the transactions that do modify the state ensure these snapshots are serializable and so read only transactions do not need to be validated and the read only snapshot isolation issued reported in [30] does not occur.

3.3.6 Overview of Global Snapshot Reconciliation

Maintaining the *global snapshot* is a challenge. The system not only needs to incorporate the changes from concurrent transactions that may have started based on different global snapshots in an isolated and atomic way, but also this reconciliation/build procedure needs to be fast and the structure needs to be lightweight so as to not become a bottleneck. More of this process is discussed 4.2, but here is an overview of the reconciliation process.

Reconciliation Procedure: Building the Next Global Snapshot

Database state changes are exposed to subsequent transactions by taking the current global snapshot as a starting point and then incorporated the changes in the transaction workspaces of the *tc*ing transactions to build a new global snapshot that exposes those changes. To get a sense of how this is done we explore integrating a single transaction's changes into the current global snapshot, Section 4.2 describes how Decibel handles incorporating changes from multiple transactions.

The algorithm below adds all of the changes made by a transaction in version graph dependency order (e.g. incorporating and rebasing ancestors before descendants). The algorithm must also ensure that if a transaction has multiple *vc*s across multiple branches then all the those transaction's *vc*s are rebased and new *vc*s are created before moving onto a new transactions changes. This is so each transaction's changes creates a new *front of the version graph*, all the transaction's appends need to be in the same order across all branches changed so that the transactions changes appear to happen atomically to the version graph. To be precise, if two transactions $T1$ and $T2$ both make *vc*s on branches A and B it cannot be the case that $T1$'s *vc* happens before $T2$'s on A and after it on B, there is no serial ordering equivalent.

The algorithm works as follows. The transaction workspace reports the historical branches the transaction created in a set called *histCreatedBranches* and the non-historical branches in *notHistCreatedBranches*. It also reports the *head vc*s of every branch that existed at the time the transaction started (derived from the global snapshot) in a set called *headsAtStart*. It maintains a map from *vc* identifier local to the transaction to the new global *vc* identifier that the *vc* was translated to after rebasing. This map is called *localToGlobalCommitIdTransalationMap*. As *vc*s are rebased or directly taken this map is filled. This map tracks processed *vc*s and another transaction local *vc* cannot be processed until its parent *vc*s have been processed.

- Initialize *localToGlobalCommitIdTransalationMap* to *headsAtStart* pointing to themselves since they existed at transaction start and are already global. It also places the creation *vc* of each historical branch in the map point to itself, because it is global by default since its creation *vc* does not need to be rebased.

- Topologically sort the *workspace version graph*. Remember the version graph has directed edges from parents to children and only the *workspace version graph* was topologically sorted so all the relevant *vc*s that may need to be rebased are processed in the order in which they need to be rebased.

- In topologically sorted order (ignoring *vc*s in *localToGlobalCommitIdTransalationMap*):

  – If the *vc* to process only has one parent then check to see if this is the creation *vc* of a non-historical branch (can be determined by probing *notHistCreatedBranches* using the branch identifier that the *vc* was on). If so, then the parent *vc* on which this branch was predicated has been processed and the branch is re-initialized based on the finalized parent *vc*. This includes initializing per segment state (in the new global snapshot), such as bitmap columns for the new branch on segments belonging to the parent branch. The new global snapshot's version graph is also updated to reflect the branch creation edge. *otherToMeCommitIdTransalationMap* is updated since this a creation *vc*, the *vc* is mapped to itself. If the *vc* is the creation *vc* of a historical branch then the *vc*s can be taken directly (since they do not need to be rebased), however, because these bitmaps are local to the transaction workspace, per segment state has to be iniitialized as well (workspace bitmaps have to be moved over into new objects that permit subsequent transactions to access the branch in a transactionally consistent way, this is discussed more in 4.2).

    If the *vc* is not the creation *vc* of a branch then the branch already exists or its creation *vc* was previously re-initialized so the algorithm becomes greedy and finds the maximum number of children *vc*s that belong to this branch in the topologically sorted order and rebases them all at once. The new global snapshot's version graph is also updated to reflect the rebased *vc*s (adding a linear chain). *otherToMeCommitIdTransalationMap* is updated and maps the transaction local *vc* identifier to the newly assigned global *vc* identifier.

  - If the *vc* to process has two parents then this was a merge *vc*. Decibel uses a locking protocol for merges discussed in 3.3.9 and it ensures the merge *vc* reflects the exact current state of the affected branches and so those *vc*s do not need to be rebased. Also, the per segment data relevent to those branches can be incorporated (taken) directly from the transaction's workspace. *otherToMeCommitIdTransalationMap* is updated.

- This continues until all new nodes are processed (end of list containing the topologically sorted nodes).

- Finally, after all the *vc*s have been rebased and their changes incorporated into the relevant branch workspces, the branch workspace changes for each branch modified from the transaction workspace are rebased on top of the new global snapshot's branch workspaces. This incorporates the transaction's local dirty branch workspace changes into the corresponding branch workspaces of the new global snapshot, making them dirty in the new global snapshot so that subsequent *vc*s that are rebased will include these changes in their first rebased *vc*.

This gives a sense of how changes per transaction are integrated to produce a new global snapshot and we leave the details of this process to the 4.2. Specifically, the system collects changes from many transactions and then a batch committer thread incorporates all of those changes into a new global snapshot, see: 4.2.7.

3.3.7 Rebasing Bitmaps: Handling Concurrent Commits to the Same Branch

Now given that we understand how new global snapshots are built at a high level, we delve into the details of the individual functions used to incorporate changes, such as rebasing.

In Section 3.2 we introduced the notion that *vc*s by concurrent transactions that do not conflict should all be allowed to be appended to the version graph. Decibel handles this by *rebasing* the bitmaps that the *vc*s affected. This is possible without conflict because of the OCC protocol that ensures the transactions updated different records and so independent bitmap locations. Thus, all that is required is for the system to decide on how to order the commits by the different transactions (since the transactions did not read/write any of the same records, any order is fine). Thus, the system selects a transaction to incorporate changes from. Starting from the *workspace bitmaps* of the current global snapshot (which may be different than the snapshots each transaction started with), for each *vc* the transaction made, take the bitmap deltas and apply those deltas (XOR the deltas with the bitmap to produce a bitmap containing the changes made by that transaction) to each bitmap affected by the *vc*. Then create a new *vc* that represents the new *global vc* that the transaction's local *vc* was translated to. Thus, the transaction's *vc* has been *rebased* on the current global snapshot and this is done for each transaction ready to *tc* and so each subsequent transaction's *vc*s are rebased on each other, building a new version graph and global snapshot that reflects all of the changes of non-conflicting concurrent transactions. If a transaction also made a workspace change (without a *vc*), the workspace for that bitmap is *dirty*. After the transaction's *vc*s are rebased the workspace changes must be XORed into the corresponding workspace bitmaps for the current snapshot being built so that when subsequent transaction's *vc*s are rebased they will include the workspace changes by the previous transaction (since it is as if all of the previous transaction's changes took effect before the subsequent transaction's, so the subsequent *vc*s that will be rebased need to include the previous workspace changes). Remember a transaction's record level changes are installed directly in the segments a transaction changed, but the transaction gets copies of the relevant bitmaps from its initial global snapshot and modifies those directly in the transaction's *transaction workspace*. Thus, concurrent appends get disjoint locations in a segment and so they get disjoint locations in their local copies of their bitmaps across all transactions. Thus, when XORing bitmap changes for two non-conflict updates the bits corresponding to the new records holding the result of the update (since the system is append only, new records are always generated from updates) do not cancel each other out.

To speed up this rebase procedure the transaction not only modifies it local branch workspaces directly, but it also stores the bitmap changes in a delta buffer. When a *vc* is made, the delta buffer is snapshotted and reset. The delta buffer snapshot is stored in a list local to the transaction so that a transaction can checkout its own *vc*s and so that the *vc* deltas are not available to other transactions (how this is done physically is discussed in 4.2). This means that rebasing just involves XORing in the supplied delta and not actually re-computing the delta using an XOR operation per pair of matching bitmaps in the global snapshot being built and the transacation's local snapshot (the reason to avoid this is that such an XOR on large bitmaps is expensive). Storing deltas directly also means that creating a new *vc* after rebase involves passing the delta directly to the underlying storage manager (note that if a bitmap workspace is dirty then the first *vc* delta needs to include the dirty workspace changes since the last *vc* as well).

Validation Revisited

We stated the system uses OCC for validation, a stricter form of validation is actually required. Transactions cannot write records that have previously been updated by transactions that *tc*ed since they started (and grabbed their global snapshots and corresponding bitmap copies). The reason for this is as follows. Let's say two transactions $T1$ and $T2$ start with the same bitmap column for Branch A with a single record $R1$ and its location in the initial bitmap is 0. Transaction $T1$ flips the bit at 0 and add record $R1'$ with bitmap location 1. It *tc*s and now the next global snapshot has $R1'$ at bitmap location 1 and the old bitmap location 0 holds a 0. $T2$ then updates $R1$ to $R1''$ unsets bitmap location 0 in its local workspace bitmap and then adds $R1''$ with bitmap location 2. Under normal OCC this would not be a conflict because $T2$'s update would simply overwrite the first update to $R1$. However, both transactions touched the same bitmap location for $R1$ which leads to an incorrect result during rebasing, concurrent overwrites are more complex. The rebasing approach based on XORing in a transaction's delta for the branch workspace would re-introduce both $R1$ and $R1''$ and leave $R1'$ in place. In turn, the system must prevent concurrent transactions from touching the same bitmap locations at all. Despite this, we discuss how to fix these issues and remove all blind writes to further improve concurrency in 3.3.11.

3.3.8 Non-Historical Branches

Since *non-historical* branches may have to have their creation *vc* rebased, if a rebase is required all the subsequent *vc*s (and descendant branches) to that branch are rebased on top of the new creation *vc*. Historical branches do not have to be rebased and neither do their *vc*s.

3.3.9 Handling Merge with Locking

As discussed in 3.2, merges represent an operation with a high high possibility of conflict and if they fail that means a tremendous amount of work was wasted conducting them. `merge` is virtually equivalent to a table-wise update. This necessitates a pessimistic (locking) protocol that will guarantee eventual success for a merge operation while in the common case of no merge allow for other operations to proceed optimistically.

Locking Protocol

Each branch has a long duration lock that supports both shared and exclusive modes. When a transaction first accesses a branch, if it is for any operation other than a merge then it acquires the branch lock in shared (S) mode. This allows for concurrent *vc*s and branches to go on in parallel. A transaction that merges two branches must acquire the lock in exclusive (X) mode on both of the branches being merged (for reasons specified in 3.2). This locks out other transactions that may mutate the branch in any way and ensures that the `merge` operation has stable *head vc*s to operate on to conduct the merge. This also prevent merges into or from any of the branches being currently merged by any other transaction, concurrent merges conflict themselves. Once acquired all locks are held until the end of the transaction after the new global snapshot has been created. If a transaction that previously accessed a branch for a purpose other than merge now accesses the branch for a merge, the a lock upgrade request is issued. However, especially in this case, the current global snapshot could have changed between the time the transaction started and the time the exclusive locks on the target branches are actually acquired, a *vc* or branch workspace change could have occurred and for merge to be successful, not conflict, and not miss any data that should have been in the merge, the transaction conducting the merge needs to observe the most recent branch workspaces and intermediate changes to the version graph that are not in it the global snapshot that it got when it started. A simple solution would be to abort and retry if a single change is detected, but this is unnecessary, instead if the actual changes made do not conflict with what the transaction has already observed then pieces of the current global snapshot can be added to the global snapshot it started with to create a transactionally consistent global snapshot that has stabilized (no more concurrent *vc*s/updates allowed) and will allow the merge to proceed and succeed. The transaction does not miss any updates because for it to have acquired the lock all the other transactions had to have released the lock and this only occurs after the new global snapshot has been created that incorporates the changes from the transactions that previously held the lock. This process is called a `snapshot reload` (reloading the local snapshot from the current global one). For a transaction with a merge by itself, it is serialized after all of the transactions that modified the two branches being merged. If the transaction also modified other branches then this imposes a serialization constraint and the reload procedure needs to check (using the constraints already tracked for this transaction, e.g. read/write sets) if given what the transaction has already done and could do if reloading and subsequently merging would cause a serialization conflict in which case the transaction that is merging needs to abort. Transactions that were waiting for the shared branch lock and come after the merge also have to reload for similar reasoning, so that they see the merge *vc* and base their changes off of it, otherwise the transaction could create a conflict with something the merge did (e.g. update the old version of a record that was three-way merged). Similar conflict detection must be done. Nonetheless, this is all primarily an optimization to reduce the number of aborts, the simple solution of aborting the transaction if the relevant parts of the global snapshot changed and trying again holding exclusive locks on the relevant branches at transaction start works. However, if this approach is taken, it is best if merges are placed in their own transactions, grabbing exclusive locks to begin with. To summarize with reloading fewer transactions abort and allows merging to be conducted in an online fashion (e.g. a transaction touches other branches and then merges, if a transaction just did a merge it could grabs the exclusive locks and then obtain the current global snapshot and this would work, but reduces the capability of a transaction).

Finally and most importantly, since the merge *vc* and all subsequent *vc*s are exactly in the correct state upon *tc* (there were no concurrent mutations after the exclusive lock was acquired), these *vc*s do not need to be rebased and so can be carried over directly into the next global snapshot (again the workspace bitmaps per segment of the affected branches are carried over directly to the new global snapshot).

There is also a subtlety with *non-historical* branches. First, let's consider a transaction merging two *historical* branches it created. This incurs no overhead and does not require a reload because these branches are local to the transaction and their creation *vc* is not in contention (does not have to be rebased). For a *non-historical vc* (as described in 3.2) any change here conflicts with any changes to the originating ancestor branch. Thus, any change into/from a *non-historical* branch must forward all its lock requests to the originating ancestor branches. This applies to branches created from *non-historical* branches within the transaction. As an example, if non-historical branch B is created from A within transaction $T1$ and B is branches to create C all lock requests to C should be redirected to lock requests for A because this entire lineage has to be rebased based on the concurrent updates to A. Merging to (to corresponds to acting as the primary) B or C from another branch could conflict with additions to A created by the rebase (meaning either abort or the merge would have to be redone). Merging from (corresponds to acting as the secondary) B or C to another branch could result in the merge missing data that should have been included in the merge because of a rebase that re-wrote *vc*s (at the time the merge was conducted it did not consider the data from A that needed to be included in B's *head vc* because of a concurrent *vc* to A that caused B's creation *vc* to be rebased). Since to create B, A needed to be locked in S mode, no merge can occur into A so there is no concern that a new merge *vc* will become the creation *vc* of the original *non-historical* branch which could create additional complications during rebase. Also, note that not only do the lock requests have to be forwarded but also conflict checking must be forwarded appropriately once the lock has been acquired.

3.3.10 Concurrent Indexing

Indexing in Decibel is described in Chapter 2.4, extending it to support concurrency is as simple as ensuring that the underlying B-Tree index or Hash Index is thread safe since the mechanism for isolating records from concurrent transactions are the bitmaps in each immutable global snapshot so it does not matter if the set contains *RecordIds* from ongoing transactions, they will be ignored until the next global snapshot when the new bitmaps are exposed. Updates of aborted transactions will always be ignored because of the same reason, but could lead to index bloat if they are left in place and slow down subsequent lookup so the recovery mechanism must remove them as in a traditional database.

3.3.11 Advantages of Bitmaps for Isolation

Transaction Internal Consistency

The OCC protocol guarantees *external consistency*, the execution of all the transactions so far to an outside observer appears serializable. However, the standard OCC protocol does not guarantee *internal consistency* for a transaction, that is the data the transactions reads is always consistent with some serial execution of the previously *tc*ed transactions, specifically ensuring that invariants preserved by transactions had they run serially are not violated when run in parallel. A more strict form of this is that the transaction does not see any new changes at all since the transaction started. OCC does not ensure this because consider two transactions $T1$ and $T2$. $T2$ reads records $R1$ and $R2$. Then $T1$ writes $R1'$, $R2'$, and *tc*s. Then $T2$ reads $R2'$. From an external point of view, $T2$ will abort because $T1$'s write set intersected $T2$'s read set. However, $T2$ read the old $R1$ and the new $R2'$ while it was running, which could cause it operate incorrectly (e.g. divide by 0 when if the transactions had run in any serial order this would have never happened). The advantage of having a bitmap/snapshot of the records in the global snapshot the transaction is using is that these bitmaps are isolated from each other and $T2$ will not observe any new updates it does not make itself, it will only see the state of the database as of the time it started, a transactionally consistent global snapshot. Though MVCC timestamp protocols used in systems like PostgeSQL provide similar guarantees, Decibel does as well and augments the OCC protocol.

Elimination of Write/Write Conflicts

Besides using the bitmap to efficiently index records across *vc*s, there are advantages to using bitmaps for isolation, one involves eliminating write/write (blind writes) conflicts for transactions that are executing concurrently (not previously *tc*ed) which under the OCC protocol would cause the transaction to abort (this can also be used to solve the problem presented in 3.3.7). First, with write/write conflicts it does not matter which write takes effects in the end state of the database so long as all of the transaction's writes across all records and branches appear to take effect atomically. The main concern with write/write conflicts not being allowed to proceed in parallel is that all of each transaction's updates may not be done atomically so the final state of the database is some mixed state. For instance, let's say we have two transactions $T1$ and $T2$ that both write records $R1$ and $R2$. If the writes to these records go on in parallel it could be the case because of some scheduler interleaving that the final effect of the database is $T1$'s $R1$ and $T2$'s $R2$ or vice versa when the final state in any serial ordering should either be all of $T1$'s changes or all of $T2$'s. Thus, the OCC protocol disallows concurrent writes to the same records. However, this is not the case with Decibel because snapshots are built atomically and when building the snapshot all of transaction's writes take effect (become visible) via bitmap flips on the appropriate segments, the data is already in place.

This works as follows when building the next global snapshot. Let us say an update to the same key in the same branch is detected (can be detected by keeping track of the keys from read/writes of transactions already processed) when incorporating all the changes for a transaction $T2$, so a concurrent transaction $T1$ with an update to the same key had its changes already incorporated into the global snapshot being built. When incorporating $T2$'s write, then the primary key index and current state of the bitmaps after the incorporating $T1$'s changes can be leveraged (see Chapter 2.4) to find $T1$'s update's bitmap location and so unset it in favor of $T2$'s update. That is unset $T1$'s bit for that update and then take $T2$'s bit for that record. By flipping bits $T2$ has overwritten part of $T1$'s changes instead of aborting. Note the rest of $T1$'s changes were still incorporated atomically and it is as if $T2$ is occurring after $T1$ even though they ran concurrently, but $T2$'s data is already in place (the record was already appended and it was assigned a unique bitmap location), all that needs to be done to preform the atomic overwrite of multiple records is by flipping the appropriate bits. The use of the index can be mitigated a bit if a temporary in-memory index is built (as in 2.4) but only holds the locations of new records added by the tcing transactions.

The power and speed of this come from using bitmaps to atomically switch over to new records atomically in an efficient way by simply flipping bits (assuming the underlying system can do this atomically and transactionally, the system we implemented as discussed in 4.2 can). This is only one advantage of using bitmaps for isolation, but one could conceive of many more advantages around similar ideas and speeding up query processing.

3.4 Concurrent Versioned Benchmark

To analyze transaction performance of a concurrent and transactional versioned system requires building on the versioning benchmark presented in 2.5.

3.4.1 Overview

In the concurrent case, the utility of branching strategies is unclear since maintaining the invariant of a particular branching strategy is a synchronization problem of its own and yields no additional insight about the performance of the underlying concurrency control scheme of the underlying versioned database system. For this reason we augment the versioned benchmark described in 2.5 so that concurrency control and transactions can also be effectively analyzed. The original versioned benchmark is utilized to load and build the dataset with a particular ancestry defined by the branching strategy using a single thread, this is called a *Versioned Scenario*. Then the system is opened up to multiple client threads that issue requests to the versioned DBMS as defined in a *Versioned Workload* similar to that of YCSBs Core Workload [21].

3.4.2 Versioned Scenario

As we build the base version structure (described in 2.5), we record information about the structure being built to help the workload select relevant data to read/write per branch (e.g., ensure that most of the keys being selected for update are actually part of that branch and do not become inserts). Specifically, we track:

- The branches created.
- The keys in every branch. The indexing scheme described in 2.4 is not employed and instead a full copy of the keys in a branch is made. This is primarily for simplicity in the key generation mechanisms and scales to moderate dataset sizes and number of branches.
- All the *vc*s created (required for historical query generation).

- Since the benchmark supports reading data from specific *vc*s, if point reads from historical *vc*s is enabled it may be the case that a lot of keys requested based on branch may not exist in the commit requested. To enabled *accurate vc* reads, keys in the *vc*s can also be tracked if requested by the user, this is called *accurate commit mode*. Note: this employs the same mechanisms used to track keys in branches and so can consume more memory in large workloads.

All of this information is encapsulated in a *build report* object that is then passed to the *Versioned Workload*.

3.4.3 Versioned Workload

Once the versioned structure is built a user-specified number of client threads are created that continuously call into the *Versioned Workload* to get queries to execute. The *Versioned Workload* is a transaction factory: it creates *Transaction* objects that encapsulate all the information required to execute the desired transaction. The *Versioned Workload* selects the next type of query and makes a call to either the selected *Contention Strategy* or the *Historical Query Generator* to get operands for that query in the `getNextTransaction()` method. These calls are lightweight and not a bottleneck. Because the *Versioned Workload* object only generates the transactions and does not execute them, the `getNextTransaction()` method can be entirely protected by a mutual exclusion lock.

Transactions

We now describe the transactions supported by the benchmark. For each of the following transactions, the branches and keys involved when referring to transactions that affect the workspace of a branch (as opposed to historical queries) are derived from the *Contention Strategy*. The rest of the queries that query historical *vc*s created during the Versioned Scenario/build of the structure rely on the *Historical Query Generator* since these reads do not produce any conflicts.

- `update` to a set of records in a branch.

- `read` a set of records from a branch.

- `read-modify-update` a set of records in a branch. The transaction first reads the records and the updates them.

- `scan` a branch, retrieving up to *max scan length* records

- `diff` a branch, retrieving up to *max scan length* records

- `branch` a branch, this transaction first makes a *vc* to the branch being branched to make the resultant branch *non-historical*

- `merge` a set of branches. The transaction is started with an intent to merge so it grabs exclusive locks as the start, *vc*s the workspaces of the parent branches, and then executes the merge query.

- `branch-modify-merge`: select a branch, branch it (creating a historical branch), read and modify a few records from the parent branch, and then merge the changes back into the parent branch.

- `branch-historical`, creates a *historical branch* derived from the *vc* returned by the *Historical Query Generator*.

- `read-historical`: read a set of records from a historical *vc*

- `scan-historical`: scan a historical *vc*, retrieving up to *max scan length* records

- `diff-historical`: diff a pair of historical *vc*s, retrieving up to *max scan length* records.

The workload parameter vc *frequency* controls the frequency at which branch modification operations (e.g. update, rmw, branch) result in a *vc* at the end of the transaction.

Contention Strategies

Contention strategies are designed to vary the level of contention between mutator transactions (e.g. update) in two dimensions, the branches selected and the records (primary keys) within that branch that are selected. The contention strategy operation selection functions take as argument client information so that the contention strategy can make decisions based on the client requesting operands for a particular type of operation. The contention strategies make use of the *build report* from the *Versioned Scenario* to generate branch names and selected record (keys). New branches and *vc*s introduced as a result of transactions from the workload and not the *Versioned Scenario* are ignored, so that the set of branches selected remains constant and the selection distributions over these branches make sense to create the appropriate level of contention.

The first type of contention strategy is the called *independent writers contention strategy.* This strategy models no contention between the transactions generated between clients (and client can only generate one transaction at a time, so no contention between transactions). Every client gets a disjoint sets of branches to operate on, at most two (to support merge queries). This type of workload can give a baseline for the best possible performance a versioned system can achieve.

The second type of contention strategy is called the *basic contention strategy.* It selects branches based on a user specified distribution (e.g. uniform, zipfian) and then based on the branch selects (records) within the branch using another user specified distribution. With these two parameters, the benchmark can evaluate performance with varying levels of contention. For instance, a high contention workload selects the same set of branches and the same set of keys from each branch while lower contention workloads could select the same braches, but different keys within the branch or different branches entirely.

Contention Levels:

The following configurations provide *minimal*, *moderate*, and *high* contention.

- **minimal:** *uniform* distribution for branch selection and *uniform* distribution for key selection within a branch

- **moderate:** *uniform* distribution for branch selection and *zipfian* distribution for key selection within a branch, or *zipfian* distribution for branch selection and *uniform* distribution for key selection within a branch

- **high:** *zipfian* distribution for branch selection and *zipfian* distribution for key selection within a branch

Historical Query Generation

The *Historical Query Generator* makes use of the *build report* to get all the *vc*s created, then using a user specified distribution selects *vc*s for operations at random (the default is a zipfian, so more recent *vc*s are selected for analysis). For point reads from a *vc*, if *accurate* vc *mode* is not enabled the keys assumed to be in the *vc* are the keys currently in the branch the *vc* was made on. The keys are selected from a uniform distribution by default since selecting keys to read from a historical *vc* never creates contention.

3.5 Evaluation

Since Decibel is the first of its kind, there is nothing to compare to so the system is evaluated with respect to varying the number of client threads for each parameter analyzed. We evaluate Decibel with transaction support against sequential Decibel (a single thread that issues commands as quickly as it can) when appropriate. The reason to not evaluate against sequential Decibel for everything is that sequential Decibel is naturally faster with one thread since it does no log writes and has no serialization/isolation overhead between transactions, but cannot support concurrent updates. They are fundamentally different types of systems. One could compare transactional Decibel with a trivial implementation of Decibel that supports concurreny by grabbing a global database lock, a global lock does not scale, limits concurrency, and prevents readers from being concurrent with writers, and so it is not an interesting comparison. Thus, we just analyze how transactional Decibel handles increasing concurrency. Each of the following sections focuses on evaluating the system with respect to a single or small subset of parameters. Unless otherwise specified the default parameters for each experiment are as follows. The experiments were run on a computer with an Intel Xeon 7500 8 core processor and 25 GB/sec memory bandwidth. We use 4MB pages with 1KB records. When a record is read all of its fields are read and when a record is updated all of its field are updated. Most of the dataset is in memory to prevent random reads from incurring a random I/O cost and skewing read/scan performance. However, for durability, transactions that mutate the database state must incur log writes which is forced to disk at *tc* time. Finally, to further evaluate the user of bitmaps in indexing, the primary key index used for point/indexed reads and writes is in-memory.

Versioned Scenario Configuration

- **Branching Strategy:** Scientific
- **Number of branches:** 1
- **Dataset Size:** 1GB
- **Insert/Update ration:** 80%, 20%
- **Aggregate number of *vc*s (evenly spaced over the dataset):** 10000

Versioned Workload Configuration

- **Update/Read Ratio:** 50%, 50%
- **Contention Level:** minimal to moderate (zipfian for branch selection, uniform for key selection)
- ***vc* (versioned commit) frequency:** 20%
- **Number of transactions to complete:** 100000

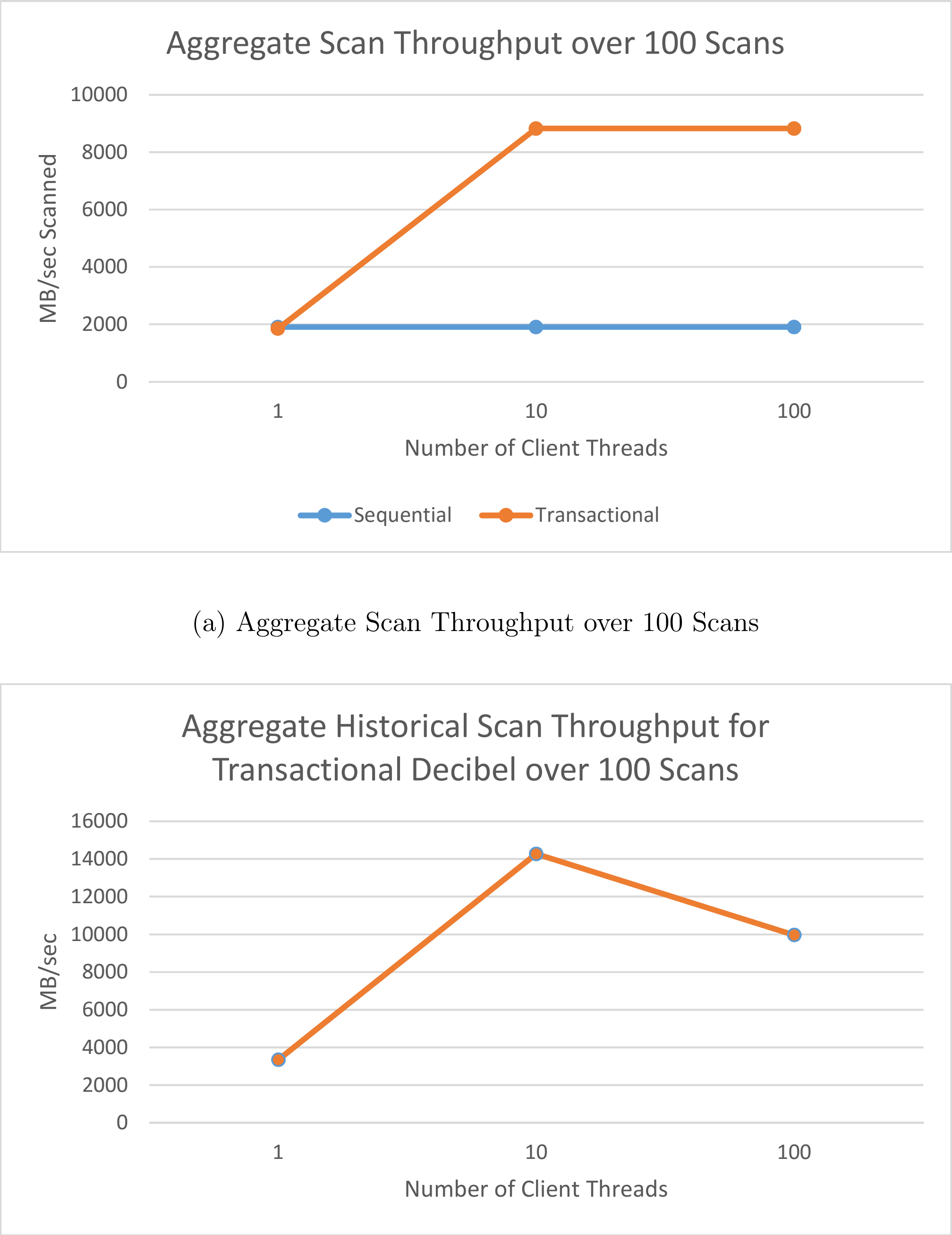
For experiments that use the default number of branches (1), there is only one segment that contains every record and one bitmap column that tracks changes to those records. This is to assess the viability of using bitmaps for isolation in this very simple case.

3.5.1 Read Only Queries

In this section we evaluate transactional Decibel on a set of read only queries, showing Decibel's high performance.

Scans

In the section we evaluate parallel scan performance of the workspace of a branch. The experiment involves a transaction workload of 100 scans and the reports the aggregate scan throughput (total amount of data scanned divided by amount of time taken) varying the number of threads. Figure 3-5 shows the results and compares against sequential Decibel. In this case this is acceptable since the data is immutable. This also gives a sense of the high parallel read performance. Throughput increases dramatically with the number of threads, with virtually no overhead and produces close to 5x as much throughput as sequential Decibel, capping out at data transfer speeds close to that of the memory bus. This is because each scan can go on in parallel on each core (scan throughput does not increase when going out to 100 threads because this is much larger than the number of cores) and so the total time it takes to scan all the records is much lower than sequential Decibel that must execute each scan one at a time. This makes the aggregate throughput higher. 3-5 also shows the scan performance of scanning random historical *vc*s; again this is immutable data that incurs no overhead to scan.



(a) Aggregate Scan Throughput over 100 Scans

(b) Aggregate Historical Scan Throughput over 100 Scans

Figure 3-5: Read Only Transaction Scan Performance

Index Reads

Figure 3-6 shows the number of indexed read transactions per second that transactional Decibel can perform. The results are similar to scan performance. However, single threaded transactional Decibel performance indicates that there is some overhead for basic transaction maintenance, but the system scales as the number of threads increases. Since point reads consult the bitmap for every read the reason that the performance is so high is that because the branch workspace bitmap is not changing and so the transaction does not write to the transaction local workspace bitmap (causing the branch workspace bitmap that it got from the global snapshot to be copied). In turn, every transaction shares the same branch workspace bitmap and the majority of this bitmap is cached by each processor so index lookups incur little overhead.

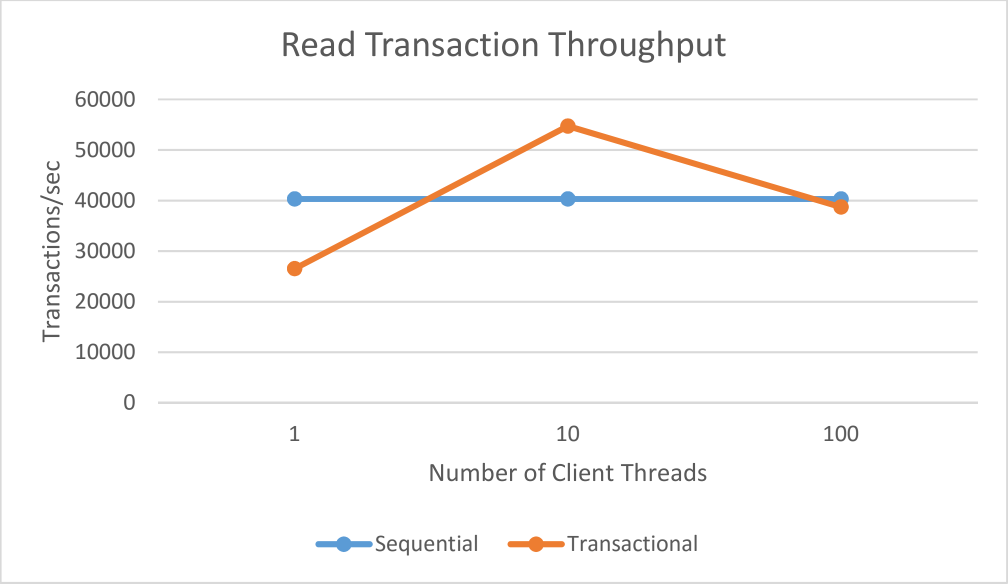


Figure 3-6: Indexed Read Transaction Throughput (transactions/second)

3.5.2 Update Performance

In this section we see how transactional Decibel scales with as the ratio of updates to reads increases, ending with a transaction workload that is 100% updates. Figure 3-7 shows the results. As the update ratio increases, the transaction throughput goes down. Even though read queries can be answered without synchronization, writes must be incorporated into the branch workspace and each transaction touches the same branch (since there is only one). Furthermore, for writes, each bitmap must be copied so that each transaction has its own local bitmap, which creates a bottleneck. The runtime is dominated by the updates, individual read latencies are negligible (0 ms), so reads proceed at the throughput specified in 3.5.1. Average update latencies range from 1-20 ms but as the number of client threads increase more updates are batched together increasing the throughput. Nonetheless, performance still generally increases with the number of threads. This is because the batch transaction committer is batch rebasing updates by many transactions at once and the the batch transaction committer thread can have parts of the bitmaps cached for even better performance. Furthermore, since no locks are involves in updating the bitmap to build the next global snapshot there is minimal cache coherence overhead.

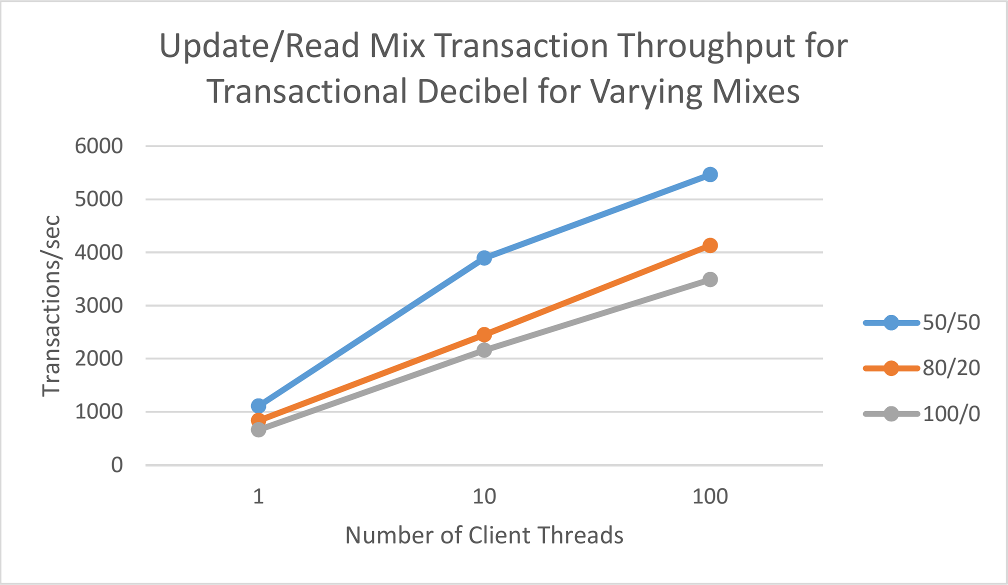


Figure 3-7: Update Transaction Throughput (transactions/second)

3.5.3 Scalability: Varying *vc* Frequency

In this set of experiments we evaluate how increasing the *vc* frequency affects performance. Every update has a certain probability of creating a *vc* in addition to its point update and we sweep this probability from 0 to 100. This shows that making a *vc*

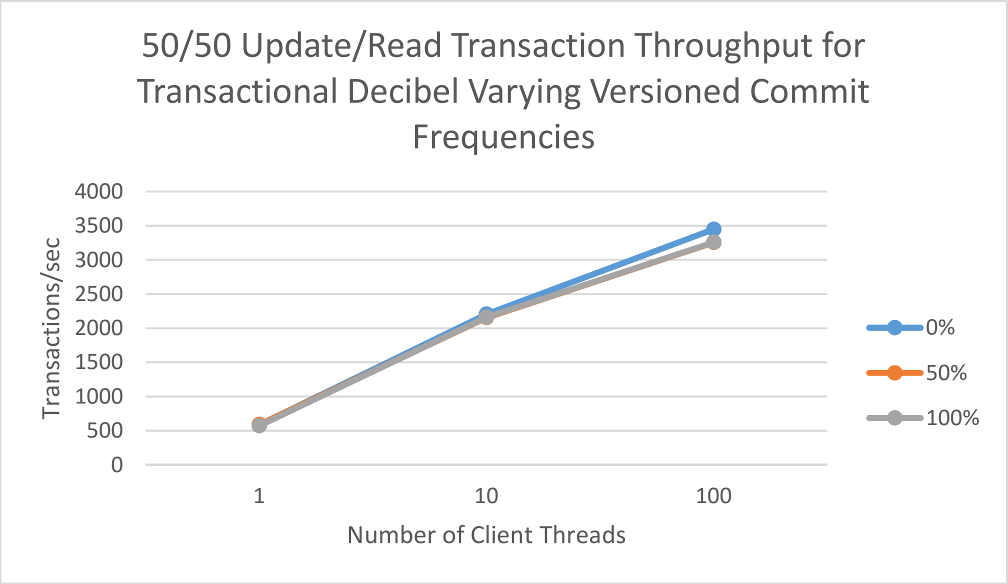


Figure 3-8: Transaction Throughput (transactions/second) Varying the Frequency at which Update Transactions Version Commit

incurs very little overhead. This is because of two reasons. First, having each transaction just store the deltas on the relevant bitmaps for each *vc* created eliminates the overhead of figuring out what additional changes need to be saved to recover the *vc*. In addition, the rebasing cost of a *vc* is minimal. Furthermore, saving these bitmap deltas as the new global snapshot is being built is cheap because they are not written out to the corresponding pack files at *tc* time, instead they are buffered in memory and so saving *vc* deltas involves just manipulating a few objects in memory. Second, regardless of whether a *vc* is made or not, the branch workpsace must be updated to include the changes made by a transaction and that is where the time is being spent. Specifically whether a transaction just updates the branch workspace or updates the branch workspace and makes a *vc*, the local changes that the transaction introduced in either of those cases need to be incorporated in the new global snapshot's branch workspace so that these changes are visible to subsequent transactions.

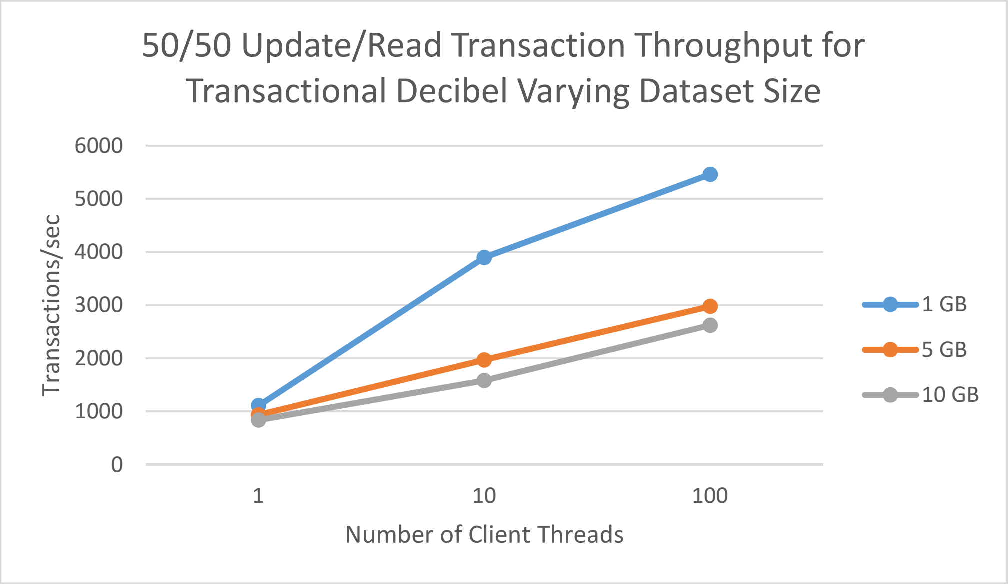


Figure 3-9: Transaction Throughput (transactions/second) Varying Dataset Size

3.5.4 Scalability: Dataset Size

In these set of experiments we vary the dataset size. These set of experiments specifically show how the idea of using bitmaps for isolation scales. Figure 3-9 shows that as the dataset size goes from 1GB to 5GB throughput drops by a little less than a factor of 2. What this shows is that even though these bitmaps are in-memory and run length encoded (compressed), giving each transaction that makes an update its own copy of the bitmap is moderately expensive. Furthermore, if the access pattern is random, more and more runs of 1s are are interrupted, reducing the effectiveness of run length encoding. This may also cause run optimizing the bitmap to take more time. Also, these bitmaps may no longer fit in the processor's cache and so reading and writing to the bitmap incurs a memory access. We did not attempt a 100GB dataset since the bitmap alone would have been 12.5 MB, which is larger than L1 and L2 caches and would occupy half of the L3 cache. Furthermore, each transaction may need its own slightly modified copy of the bitmap, incurring more overhead. It is future work to determine how to reduce this.

3.5.5 Scalability: Branches

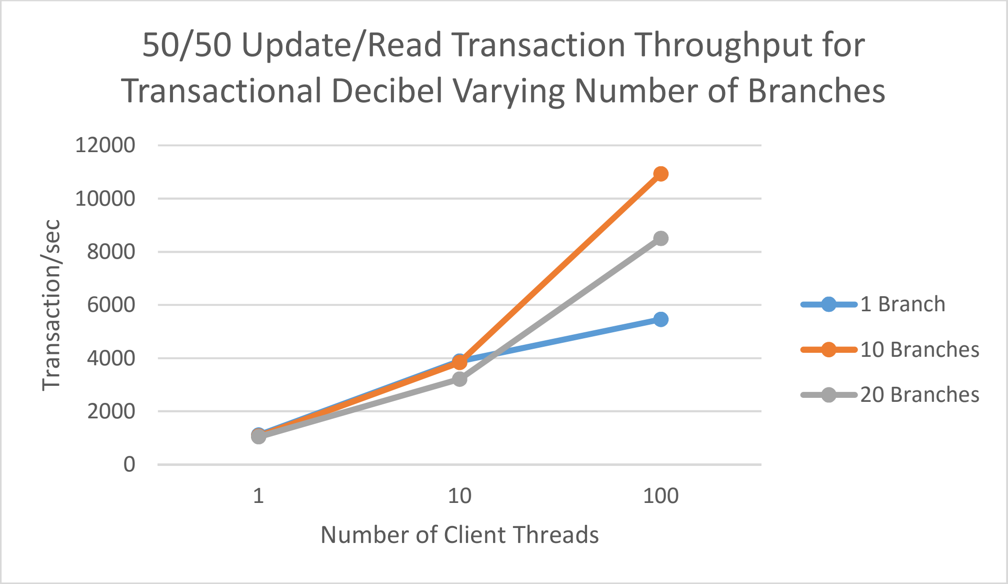


Figure 3-10: Transaction Throughput (transactions/second) Varying the Number of Branches

Here we see how the system scales with multiple branches. The performance roughly matches that of single branch performance and maintains the same general trend of increasing performance with the number of client threads going out to the number of cores in the machine. This performance can be enhanced if the rebase procedure between branches in the batch transaction comitter is parallelized as described in 4.2.7. However, there is a boost in performance at 100 threads because more transactions are being batched together and the group of transactions is touching a small set of smaller bitmaps than in the single branch case. Also, run length optimizing these smaller bitmaps is faster.

Figure 3-11 shows indexed read only transaction throughput varying the number of branches. This roughly matches the single branch case and its high performance. However, another scaling issue related to copying of Hybrid meta data takes over at 20 branches, reducing performance.

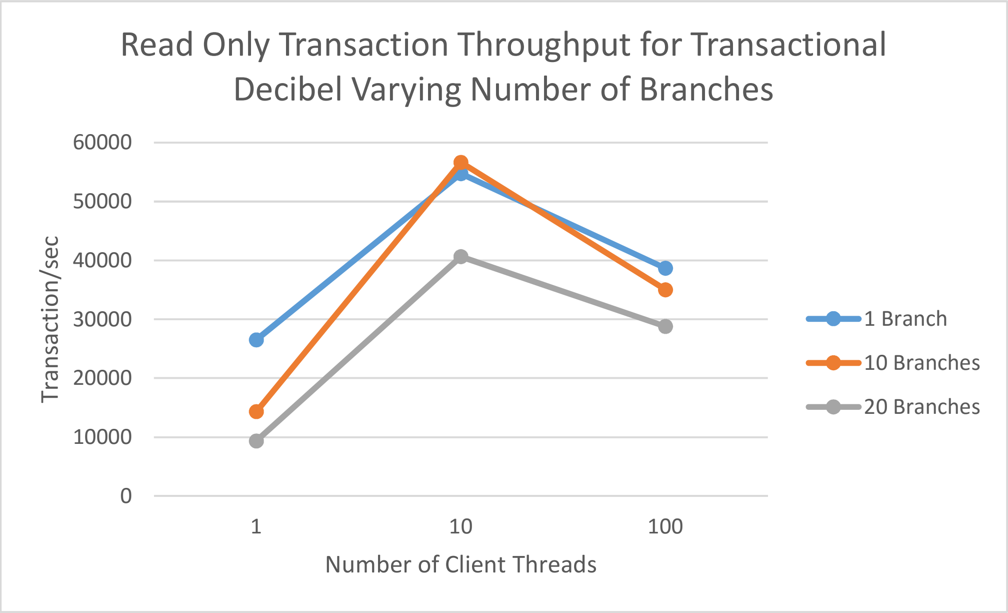


Figure 3-11: Read only Transaction Throughput (transactions/second) Varying the Number of Branches

Chapter 4

Versioned Transaction Manager Implementation

In this chapter we discuss the implementation of the Versioned Transaction Manager. We first introduce some components in the versioned DBMS that required careful attention to support high concurrency and then we move on to discuss the architecture of the versioned transaction manager and how bitmap state is controlled.

4.1 Mechanisms in Decibel for High Concurrency and Recovery

In this section we discuss the implementation details of some core components of the underlying storage system and how they deal with and enable high concurrency in the face of isolation.

4.1.1 Highly Concurrent Append Only Heap File

In an append only file there is a lot of contention on the last few pages of the file and so even holding a short duration lock on the entire page to ensure that a transaction's inserted record gets a unique slot and does not get overwritten by another transaction becomes unfeasible. Furthermore, threads reading these last few pages should not see a partially inserted record (the record could be *tc*ed but because of the java memory model the changes are not in memory they are in the store buffer) so the requirement for a memory fence, locks is still required. However, after a page has all its slots filled it becomes immutable and so locks are never required. Thus, the goal is to have each thread appending a record to get its own slot without locks, ensure that the contents of the record are written to memory, and allow readers once the page becomes full to access the data without locks. The `AppendOnlyHeapPage` has an atomic counter that and when a thread wants to insert a record into a page it calls *getAndIncrement()* to atomically get an index on the page that no other thread has ever received and no other thread will get. The thread is then free to write to this slot in the page and no other thread will over write it. Since the page is just an array of bytes each page has a thread local *ByteBuffer* object that it used to access the page's bytes. Each page also has a lock bank, one lock per record slot (the minimum possible granularity in the page) to act as a memory barrier for the inserted record, the thread acquires an uncontended lock while writing the data to the page in memory. Readers of data on a page also grab this lock if the page has been modified since it was retrieved from disk (not been filled yet). After the page has been filled there are no more needs for locks and reads access the record slots without locking. The `AppendOnlyHeapFile` keeps track of which pages have free slots using an atomic integer array that keeps track of the free count on each page, if a thread fails to get a slot (the counter exceeds the number of slots on the page) on a page then it consults the `AppendOnlyHeapFile` free count integer array around locations that correspond to the end of the file and tries again. The page counter and the integer array at the file level do not have to exactly be in sync the file integer array is more a shortcut and the thread will just keep trying to insert if it observes that the page is full. After an insert the inserting thread updates the spot corresponding to the page it modified in the file level integer array atomically. These make reads and inserts extremely fast because multiple threads can access and insert their records in parallel, writing to disjoint memory locations. When the end of the file is reached, several new pages are written in batch by one thread (it acquires an exclusive lock on the page) in batch to minimize page initialization cost and prevent persisted high contention on the last page.

4.1.2 BufferPool Page Eviction during Concurrent Modification

Threads currently on a page should not keep a reference to this page when it is evicted and a page should not be evicted or written to disk when there are threads currently inserting records into it as those may be captured in the data written to disk. However, the system should not be prevented from writing the page to disk for a long duration as this slows down threads waiting for the new page to be brought into memory. The way this is handled is that prior to any operation on any page it increments an atomic counter (called the modification counter) particular to that page and then checks to see if the page is *disabled*, if so the thread decrements the counter, does not perform the operation, and waits for the page to be re-enabled. Prior to writing the page to disk the page is quiesced, the thread writing the page out sets a *disabled* flag and then checks the counter and waits for it to become 0. Since the threads check the other's variables in reverse order they will never miss each other and once the *disabled* flag is set the threads will drain out and the thread can proceed to write the page out to disk and not miss a modification. This is called *quiescing* a page. If the page is evicted it is marked as *dead* and the next time the thread checks to see if the page is enabled it will also check to see if the page is dead and if so it will stop waiting and cause the operation to be re-tried at the next higher level, e.g. for insert find another empty page and attempt the insert again.

Since pages can be evicted while being read and these pages should not continue to be referenced for proper memory management, the page iterators must handle a *PageEvictedException* that may be thrown when accessing a slot on page in which case the iterator re-fetches the page and continues iteration right where it left off on the page.

4.1.3 Transaction Managed Page

The system wraps a pages returned by the database files in a *Transaction Managed Page* that handles the problems defined in 4.1.2 (encapsulates the access counter, *disabled* flag, waiting, and quiesce functionality) so that regardless of the page implementation it will still always work with the buffer pool. The *Transaction Managed Page* also handles the record level WAL logging functionality (logs before and after images of the record modified) if the concurrency control/recovery scheme requires it. Since accessing the page pins it from eviction it is ok if the transaction and so the changes will not make it to disk until a log entry is made.

4.1.4 Logging Module

The `DbLogFile` is a a general purpose log implementation built to log record level changes to pages, segment meta data, and version graph meta data. It buffers the head of the log file in memory using a concurrent queue and also manages log sequence numbers and the dirty page table required for recovery as per the AIRES recovery algorithm [62] that is used as part of the Decibel recovery algorithm specified in 4.2.10.

4.1.5 Lock Manager

The system also has a generic multi-granularity hierarchical lock manager [52]. It is generic (has a generic type that specified the type of object being locked) so any type of object can be locked and it can be used to support page level locks and record level locks. The lock manager is deadlock free in that it guarantees deadlock free mutual exclusion according to the specific of multi-granularity locks (that is if the lock is able to be acquired then at least one transaction can acquire it, the locking process does not deadlock itself, this is different from transaction deadlock where transactions deadlock each other because they are requesting the same locks in different order). However, it does not have any other progress guarantees besides, and transactions can be starved, repeatedly fail to acquire the lock. Future work is to make this first come first served, which is stronger than starvation freedom. This is particularly useful in Decibel's locking protocol to ensure `merge` transactions are not infinitely starved.

To support efficient range locking based on key ranges, this lock manager was used as a primitive to build a key partitioned multi-granularity hierarchical lock [56]. This separates locking key ranges from physical index key-range locking which makes it easier to develop locking protocols when the underlying index implementation is opaque to the developer of the database.

4.1.6 *vc* Bitmap Cache

To speed up checkout times there is a *vc* bitmap cache in the buffer pool that contains the materialized bitmaps of recently requested *vc*s managed in LRU fashion. However, it is also beneficial to cache *vc* deltas from the pack files in memory as well since this aids the reconstruction time of *vc*s and is less memory intensive. The *Versioned Transaction Manager* does this to some extent in the *global snapshot* it maintains, it caches recent *vc* deltas in memory.

4.2 Versioned Transaction Manager Implementation Details

In this section we discuss the implementation of the versioned transaction manager that provides transaction isolation. The implementation is in Java. We assume the version graph and the workspace bitmaps of all branches fit in memory. The latter is more a requirement since these bitmaps are going to be used for transaction isolation and so need to be consulted prior to every database access.

4.2.1 Overview

The `HybridVersionedTransactionManager` is the class encapsulating all the code for versioned transaction management based on the Hybrid scheme. A transaction begins by calling `transactionBegin()` on the `HybridVersionedTransactionManager` which creates the transaction local workspace, the `HybridVersionedWorkspace`, for the transaction based on the current `GlobalSnapshot`. The `GlobalSnapshot` encapsulates a Hybrid storage model object that exposes a transactionally consistent version graph and branch workspace bitmaps via `SnapshotManager` objects that represent the bitmaps. When the transaction's workspace is created, local workspace versions of these `SnapshotManager` objects are created (in copy on write fashion) that the transaction can modify directly. Every time a transaction tries to access the database it makes the corresponding call into the `HybridVersionedTransactionManager` (e.g. `insert,` *vc*`, etc.`) and the transaction manager then forwards this call to its `HybridVersionedWorkspace` that operates on a local Hybrid storage model object with the appropriate transaction local version graph and `SnapshotManager` objects loaded into it. From that point on operations run as in sequential Decibel in 2 with the addition of conflict trakcing provided by the `HybridVersionedWorkspace` and the `VersionedReadWriteSet`. Finally, when the transaction *tc*s, it calls `transactionComplete()` on the `HybridVersionedTransactionManager`. The thread then executing the transaction validates the transaction's changes according to the OCC protocol. If validation fails, the transaction aborts, otherwise it enqueues its changes for the `BatchTransactionComitter` to take its bitmap and version graph changes and build a new `GlobalSnapshot` so that is changes will be exposed to subsequent transactions. The `BatchTransactionComitter` drains the queue and builds a new `GlobalSnapshot` containing all the changes of the *tc*ing transactions. For durability the transacton waits for its log entries to be flushed to disk.

4.2.2 Transaction Manager Hierarchy

The `HybridVersionedTransactionManager` extends the `AbstractOCCTransactionManager` which extends the `AbstractTransactionManager`. The `AbstractTransactionManager` intercepts all calls to modify the database. It manages moving pages from memory to disk after the `BufferPool` evicts them, e.g. quiescing the page and then writing it to disk. It is also responsible from wrapping retrieved pages from disk in the `TransactionManagedPage`. It also provides configuration options for all transaction managers that subclass it, such as enabling logging and how to handle transaction *tc* and abort. The `AbstractTransactionManager` creates a `BatchTransactionComitter` thread that runs in the background and *tc*s (or aborts, in the case of locking protocols it is useful to batch abort to minimize the number of times parts of the log on disk are accessed) the changes of all the *tc*ing transactions according to the semantics of the subclass. Transactions that want to *tc* submit the request to the `BatchTransactionComitter`, which returns a `TransactionFuture` and the transactions wait *synchronously* (to support scalability in the number of threads waiting threads go to sleep and wait using Java's `wait()` and `notify()`) for the `BatchTransactionComitter` to process their request and ensure that their changes are durable and reflected in the current database state. This includes forcing the log to disk and by batching several transactions together when the log is forced, the per transaction cost of durability goes down. Note that not all transactions have to be batch *tc*ed or aborted, for instance, in the case of the `HybridVersionedTransactionManager`, read only transactions do change the database and so may not have to wait. Again, this is configurable by and dependent on the subclass.

The `AbstractOCCTransactionManager` provides the basic functionality to handle transaction management according to the Optimistic Concurrency Control protocol. This include keeping track of read/write sets for *active* and *completed transactions*, and parallel validation for transactions that wish to *tc*. The `AbstractOCCTransactionManager` also manages transaction `Workspace` objects for each transaction via a `WorkpsaceManager` object. The `WorkpsaceManager` generates new `Workspace` objects for transactions according to semantics of the specific implementation. All operations intercepted by the transaction manager are directed to the transaction's workspace and then after validation those changes are exposed to transactions that subsequently start. It also provides the ability for transactions that retry after a validation failure to proceed without conflict by causing any transaction that is attempting to read or modify a value that the failed transaction attempted to modify to block, waiting for the earlier transaction to complete.

The `HybridVersionedTransactionManager` has a `HybridVersionedWorkspace-`

Manager that encapsulates a GlobalSnapshot object which encapsulates all the information required to access the current serializable snapshot of the database state. It is on top of this that the HybridVersionedTransactionManager *tc* procedure incorporates all the changes of transactions that have passed validation to create a new serializable GlobalSnapshot. This new GlobalSnapshot is atomically released to new transactions via the HybridVersionedWorkspaceManager using a AtomicReference, the partial results of a transaction are never exposed.

4.2.3 Hybrid State

Before we discuss the Global Snapshot in detail, an understanding of the state required by the Hybrid storage model is required. To reduce complications, no new *head* segment for the parent branch is created upon branch. This means that there are O(number of branches) segments and all updates to a branch go to its original segment. New branches get their own segments to store data particular to that branch. We continue to use the columnar approach to bitmap storage and representation. Because of the columnar approach, not creating a new segment on branch of the parent branch is justifiable since no vacuous 0s are introduced into the columns for the child branches, the 0's are implicit since no new records for the child branch are inserted into the parent segment or ancestor. Thus, full Hybrid performance is still achievable and complexity is reduced.

Snapshot Managers

The most fundamental component is the SnapshotManager, it encapsulates and manages snapshots of the workspace bitmaps, corresponding to different *vc*s. There is one SnapshotManager per branch that tracks the segment membership in a branch, which segments belong to a particular branch as discussed in 2.3.4. This is called the *branch-membership* snapshot manager. After branch creation (where the new branch inherits all the segments of all its parent), this may change as a result of a merge where a branch must now track segments from a branch in a disjoint subtree of the version graph. There is also one SnapshotManager per branch (bitmap column) on every segment that tracks records for that branch. As records are updated and moved from parent segments to the target branch's segment, the ancestor segment's snapshot managers' workspaces for the target branch are updated (flipping the corresponding bits to 0) and the snapshot manager for the target branch on its own segment is updated to reflect the addition of a new record. Controlling the bits exposed by these SnapshotManagers controls the snapshot of the database a transaction can view. Operations on a SnapshotManager include that ability to add() and remove() tracked locations in a segment for a branch (remember that the bits in these bitmap columns correspond to physical locations in a segment file that represent records that belong to a particular branch in that segment). These modifications go to the *workspace* for the SnapshotManager that reflects the current state (accumulated changes across all of time) of the bitmap column. These *workspace*s are snapshotted during a *vc* (versioned commit) operation (to enabled reconstruction of a particular *vc*). How the snapshot process works is dependent on the snapshot manager implementation. The SnapshotManager used in 2.3.4 stored a snapshot to a *pack file* by only writing out the delta between the last snapshot and the current workspace. The snapshoted workspace can then be reconstructed by replaying the deltas on the initial *vc* (initial workspace), which is also written to the *pack file*. Finally, the SnapshotManager supports retrieving particular snapshots based on a *vc* identifier.

***vc*s (Versioned Commits)**

*vc*s on a branch are totally ordered by a *commit tag* number, representing the *vc* number on that branch. So a globally unique *vc* is identified via *(branch id, commit tag)*. A *vc* involves snapshotting all the SnapshotManagers on the relevant segments for a branch. The SnapshotManager's use these *commit tag*s to determine what snapshot to re-create. Since a *vc* involves a snapshot of every SnapshotManager belonging to a branch, it would seem that the *commit tag* corresponds exactly with the snapshot number to reconstruct on every segment. This is true if the branch is never merged, if the set of segments included in a branch never changed. After a merge procedure, the set of segments included in a branch may grow and if the branch had previous *vc*s then the commit tag 1 has no meaning for the `SnapshotManager` on the newly tracked segments since the segments' `SnapshotManager`s do not have a snapshot for commit tag 1. Thus, these *global commit tags* must be translated to `segment local commit tags` that map to an actual snapshot number that the `SnapshotManager` can reconstruct. This is done by keeping a per segment `tracking commit tags` map that specifies the *global commit tag* at which this segment started to be included in the branch (the map maps from branch identifier to global commit tag at which this segment started to be a part of the branch). This acts as an offset so that subsequent *vc*s map to the correct snapshot numbers per segment `SnapshotManager` for that branch. For instance, if branch B has global commit tag 10 (and so B's *head* segment `SnapshotManager` has 10 snapshots) and it is merged with branch C that introduces a new segment, call it 2. At global commit 10 segment 2 has no snapshots, but at commit 11 for branch B, branch B's `SnapshotManager` on segment 2 will have 1 snapshot, but the tracking commit tag for branch B on segment 2 is 10. Thus, to restore to commit tag 11 we restore B's head segment `SnapshotManager` to snapshot 11 and then translate the current global commit tag to a segment local commit tag for segment 2 as follows, the snapshot number for branch B on segment 2 is: (global commit tag for the desired snapshot) - (tracking commit tag for branch B on segment 2 (10)) = 1, mapping to the correct snapshot number that the `SnapshotManager` on segment 2 can reconstruct for branch 2.

Transaction Manged Master Workspace Snapshot Manager

Though the snapshot managers provide the ability to create new dataset snapshots, an additional layer is required to support transactions and control how workspace changes and new snapshots are exposed. The `TransactionMasterWorkspaceSnapshotManager` wraps a `SnapshotManager` object (that is implemented to store version information efficiently). It is used to atomically incorporate or rebase bitmap changes made by concurrent transactions and expose bitmap column changes to subsequent transactions. Specifically, it has an atomic reference that contains the current transactionally consistent state of the bitmap column. It then has a workspace bitmap that is an exact duplicate that can be written to by the `BatchTransactionCommitter` thread to take changes from concurrent transactions that have modified that bitmap locally in their workspace and build a new workspace bitmap that reflects the new transactionally consistent state when building the new global snapshot. Once the new workspace is completely updated the atomic reference switches over to a copy of the current workspace state to atomically expose all the bitmap changes. This will be discussed in more detail, but this is description just gives a sense of what is happening to update bitmaps. The key point is there is an isolation between the currently publicly visible state (i.e. visible to new transactions) and the new transactionally consistent state being built to ensure that new changes can be incorporated without blocking transactions that just want to read the state.

To enhance performance the `SnapshotManager` wrapped by the `TransactionMasterWorkspaceSnapshotManager` is a `ConcurrentBufferedSnapshotManager`. This snapshot manager buffers recent *vc* snapshots in memory in a manner equivalent to 2.3.4, it buffers deltas in memory and an aggregate delta of all the deltas currently on disk. This speeds up checkout, or snapshot recreation time for recent snapshots since the pack file on disk does not need to be consulted. Also, since these deltas are immutable, it admits the possibility of concurrent snapshot reconstruction of recent snapshots. The `TransactionMasterWorkspaceSnapshotManager` also caches recently reconstructed and accessed snapshots in their entirety in the BufferPool to further increase performance.

Additional Hybrid State

The additional state maintained by the Hybrid storage model is:

- *versionGraph* object which contains all of the version graph information, i.e. *vc* nodes and edges between them representing full lineage tracking.

- *segmentIdGenerator* which is an atomic integer used to generate unique segment numbers that can be referenced in the *branch-membership*. This variable needs to be atomic because it is shared by all concurrent transactions that are generating new segments for new branches and so need to atomically generate a unique segment number.

- *branchMembershipMap* which maps from a branch id to its *branch-membership* snapshot manager.
- *branchIdToInitialSegmentNum* which maps from a branch id to the original/dedicated segment that should hold new records/updates for that branch.
- *segmentIdToSegment* maps a particular segment id to the corresponding `Segment` object.

The additional state maintained per `Segment` object is:

- `AppendOnlyHeapFile` object that represents the underlying data file for that segment that records can be concurrently appended to by concurrent transactions. This is thread safe and shared by all transactions inserting data into this segment. Again, what controls visibility of these records are the bitmap column snapshot managers.
- *columnNumberGenerator* is an atomic integer that generates column numbers for new snapshot managers for this segment. New snapshot managers for a segment are introduced when branches are created or merged. This is shared between all concurrent transactions and needs to be globally unique to ensure that concurrent branches of the same branch get disjoint columns, which makes reconciling concurrent changes easier.
- *trackingCommitTags* as discussed in 4.2.3.
- *branchIdToColumn* which maps a branch id to the corresponding unique column.
- `SegmentBitMap` which holds the relevant `SnapshotManager` objects for the branches currently visible to a transaction and uses a *columnNumberToSnapshotManager* to get the appropriate snapshot manager corresponding to a branch that has a

particular column on a segment. It can be the case that a branch has a different column number on different segments, but that is ok. All that is required is that we can map a branch to the correct column on each segment.

It is important to note that the system adopts a "share what can be shared" philosophy. Some objects are shared to minimize transaction start cost and ensure synchronization (e.g., with the *columnNumberGenerator* between the transactions), while other objects are copied to avoid exposing new effects produced by concurrent transactions, giving each transaction an isolated state. This is discussed more in 4.2.4.

4.2.4 Global Snapshots and Transaction Start

Now the reader is able to understand the essence of a `GlobalSnapshot`. We walk through the basic elements of a `GlobalSnapshot` and how transactions convert this immutable snapshot of the database state into a *workspace* to which they can write and read their own changes that are isolated from concurrently running transactions. At the most basic level, the `GlobalSnapshot` is a Hybrid storage model object where each snapshot manager is a `TransactionMasterWorkspaceSnapshotManager`. It also holds a snapshot number which is the system defined transaction number (according to the OCC protocol) of the last transaction that had an effect in building the `GlobalSnapshot`. The *tc*ed transactions that this transaction's changes must be validated against include only the transactions with transactions numbers later than this snapshot number of the snapshot the transaction started with (the *tc*ed transactions since it started) and the transactions running concurrently with it. This `GlobalSnapshot` is immutable and replaced when a new `GlobalSnapshot` is created.

When a transaction begins a `HybridVersionedWorkspace` is generated to hold all the isolated changes for the transaction. A `HybridVersionedWorkspace` contains a *lightweight mutable copy* of the Hybrid storage model object in the current `GlobalSnapshot`.

The following hybrid state is shared by all copies: `AppendOnlyHeapFile`, *segmen-*

tIdGenerator, *columnNumberGenerator*. All the TransactionMasterWorkspaceSnapshotManager are converted to TransactionWorkspaceSnapshotManager that are initialized from the currently publicly visible bitmap exposed by the TransactionMasterWorkspaceSnapshotManager and hold all the changes made by the transaction isolated from other transactions (the initial bitmap is copy on write). The rest of the state is copied to ensure that transactions get a transactionally consistent view of those data structures. For instance, the *branchIdToColumn* and *branchIdToInitialSegmentNum* are copied to prevent exposing new branches from concurrent transactions. This also means that these structures do not have be thread safe, reducing the synchronization cost. However, it is important to note that the AppendOnlyHeapFile per segment is shared by all transactions so that the transactions all insert their data into the same, correct file backing the segment. This is why the AppendOnlyHeapFile had to be very high performance and support high throughput concurrent inserts at the page level. Remember, though the transaction's data is physically inserted to the same file, isolation is guaranteed by the workspace bitmaps of the corresponding TransactionWorkspaceSnapshotManager which is local to the transaction and initialized from a transactionally consistent global snapshot.

As for the *versionGraph*, the version graph is not copied since after several thousand *vc*s this becomes a bottleneck. Instead, the version graph is split up into two parts, a *commonVersionGraph* and a *workspaceVersionGraph*, whose *union* represents the complete version graph view with respect to the transaction. The *commonVersionGraph* is shared directly by all transactions and all new additions to the version graph are directed to the *workspaceVersionGraph*, a separate version graph object. Both of these graphs are wrapped in a VersionGraph object that is also augmented with a map that maps from the a branch identifier to the current *head vc* identifier for that branch as of when the transactionally consistent GlobalSnapshot created and exposed. When a transaction needs a complete view of the version graph (e.g. for merge operations), it looks at the graph union of the *workspaceVersionGraph* and the *commonVersionGraph*. To ensure connectivity, the *workspaceVersionGraph* is initialized with nodes corresponding to the head *vc*s of all branches. To discover the version graph changes made by a transaction, only the *workspaceVersionGraph* needs to be explored, excluding the *head vc*s that existed at the time the transaction started.

Given a transactionally consistent, isolated *lightweight mutable copy* of the Hybrid storage model object in the `GlobalSnapshot`, the transaction can proceed as in sequential Decibel applying the same algorithms to this snapshot. The `TrasnactionWorkspace` then forwards all writes/reads to this local snapshot and tracks the changes for validation purposes when the transaction tries to *tc*.

4.2.5 Transaction Workspace and Dependency Tracking

As discussed, the transaction workspace is primarily comprised of a *lightweight mutable copy* of the Hybrid storage model object in the *GlobalSnapshot* and additional dependency tracking to ensure serializability. We now describe this process in more detail.

TransactionWorkspaceSnapshotManager

As discussed in 4.2.4 all the `TransactionMasterWorkspaceSnapshotManager` are converted to `TransactionWorkspaceSnapshotManager` that are initialized from the currently publicly visible bitmap exposed by the `TransactionMasterWorkspaceSnapshotManager` and hold all the changes made by the transaction isolated from other transactions. We now discuss this in more detail.

A `TransactionWorkspaceSnapshotManager` is a new `SnapshotManager` object that is designed to act like a traditional `SnapshotManager` with respect to the concurrently executing transaction. It provides a transactionally consistent view of that `SnapshotManager`'s current state that allows a transaction to read its own changes, but have its own changes be isolated from other transactions. It does this by starting in a transactionally consistent state, as per the snapshot of the bitmap column's workspace exposed by the corresponding `TransactionMasterWorkspaceSnapshotManager`. This initial snapshot is *copy-on-write.* That is all read-only transactions share the same immutable initial snapshot, while transactions that mutate the state need their own local copy so that its changes to the bitmap workspaces are isolated. Now this is primarily an optimization so that transactions that make multiple changes and then want to scan a branch can do so easily and efficiently. That being said, the transaction's changes across all its *vc*s are not stored in their entirety as that would require a substantial amount of memory. This also slows down the reconciliation procedure that incorporates a transaction's local changes when building a new `GlobalSnapshot`, discussed in 3.3.6 and 4.2.6 since to figure out the non-conflicting changes (as determined by validation) made by a transaction requires XORing large bitmaps that are mostly the same (specifically the one corresponding to the *vc* and the workspace bitmap the `TransactionWorkspaceSnapshotManager` got from the `TransactionMasterWorkspaceSnapshotManager` when its lightweight copy of the `GlobalSnapshot` was created). Instead, only the deltas made to this base bitmap workspace by the transaction are stored in memory. Specifically, every time a bit is flipped from an insert, update, or delete to a record, that delta is recorded in a set of deltas since the workspace was initialized or last reset, called the *workspace delta*. When a *vc* is made, this delta set is snapshotted and stored in a list of *vc*s made by this transaction and the *workspace delta* is reset. It is important to note that because these updates may be sparse, it is better to use a structure like `HashSet<Integer>` to store these detlas. This list of *vc* deltas and the *workspace delta* is stored by the `TransactionWorkspaceSnapshotManager` in a `BufferedDeltaSnapshotManager` object.

To support retrieving a particular snapshot based on a snapshot number, the `TransactionWorkspaceSnapshotManager` is initialzied with the current last snapshot number that was installed in the `TransactionMasterWorkspaceSnapshotManager`, this represents the last valid snapshot of a *vc* that was solidified, existed prior to the transaction's start, call this *lastSnapNum*. The next snapshot number generated by the `TransactionWorkspaceSnapshotManager` *vc* procedure will be *lastSnapNum* + 1 (the next snapshot number had this transaction run serially given the current state, note that these snapshots may have to be rebased and so the snapshot numbers will change when they are incorporated to build the next global snapshot) and subsequent snapshots continue to increment this. Thus, when a snapshot with $snapshotNum \leq lastSnapNum$ is requested, the request is forwarded to the `TransactionMasterWorkspaceSnapshotManager` object that in turn forwards the request to the `SnapshotManager` that holds all of the previously created snapshots. The reason it is forwarded to the `TransactionMasterWorkspaceSnapshotManager` object is so that the `TransactionMasterWorkspaceSnapshotManager` can control access to the `SnapshotManager` that backs its, providing necessary synchronization if necessary (the `SnapshotManager` that backs it does not necessarily have to be thread safe). Also, to reduce snapshot reconstruction time for this solidified snapshots, the `TransactionMasterWorkspaceSnapshotManager` caches the snapshots it reconstructed in the `BufferPool` which it can later retrieve. To retrieve a snapshot with $snapshotNum > lastSnapNum$, requires first retrieving the snapshot corresponding to $lastSnapNum$ and then applying all the deltas buffered by the `TransactionWorkspaceSnapshotManager`'s `BufferedDeltaSnapshotManager` between $lastSnapNum + 1$ and the desired $snapshotNum$.

When a new branch is created, the `SnapshotManager` created for each relevant segment is a `TransactionWorkspaceSnapshotManager`. Then when the transactions local workspace is merged to build a new global snapshot, this is upgraded to a `TransactionMasterWorkspaceSnapshotManager`. This also happens when a `merge` causes a new segment to be tracked for a particular branch (remember a new `SnapshotManager` has to be generated in this case).

Given this type of workspace for bitmap column `SnapshotManager` workspace isolation and the fact that new records appended to the same *AppendOnlyHeapFile* for a particular segment are not visible unless they appear in a bitmap column visible to the transaction, bitmaps have effectively and Decibel's natural versioning primitives have been leveraged to provide isolation.

Tracking Dependencies: Versioned Read Write Set

Now we delve into how serialization conflicts are tracked, based on the theory discussed in 3.2. Before we introduce the notion of a `VersionedReadWriteSet`, we discuss a typical `ReadWriteSet`. A standard `ReadWriteSet` simply have two sets of key, the set of primary keys of records that the transaction read, called the *read set*, and the set of primary keys of records that the transaction wrote, called the *write set*. It is worth noting that if primary keys are integers and non-negative (which they are likely to be in most relational databases), then the `ReadWriteSet` can be represented compactly using compressed bitsets (bitmaps).

A `VersionedReadWriteSet` extends this by having a `ReadWriteSet` per branch to determine point read/write conflicts per branch. Validation of point reads and writes in this case are by looking at the intersection of the branches accessed between the two transactions being validated and then for each branch that both of them accessed validate the corresponding `ReadWriteSet`s in the typical manner. It also tracks branches created and branch level data (e.g. `diff`) and version graphs reads (e.g. `listCommits()` or `listBranches`) using `HashSet`s of branch identifiers. The same branch access intersection detection and validation is applied again here when validating these branch level operations. To handle *Non-Historical Branch Conflicts* the `VersionedReadWriteSet` maintains a map that maps the *Non-Historical Branch* to the first ancestor branch that made a branch in its lineage *Non-Historical* (it may be useful to this point to review 3.2). This is the branch with which that *Non-Historical* branch has to check conflicts with. When validating a transaction's `VersionedReadWrite` against that of another transaction, it must check the *Non-Historical* `ReadWriteSet` and branch level operations against the other transaction's `ReadWriteSet` corresponding to the ancestor witch which the *Non-Historical* conflicts. It also needs to validate against the branch level operations as well in the same way. The validation must do this for every non-historical branch, so validation in this case is expensive.

Tracking Dependencies: Hybrid Versioned Workspace

The `HybridVersionedWorkspace` detects places where possible conflicts may occur and then adds them to the `VersionedReadWriteSet` accordingly. Specifically it maintains two sets, *branchesToTrack* and *commitsToTrack* which contain the branch and *vc* identifiers, respectively, of the branches and *vc*s that should be tracked for possible data conflicts, i.e. operations that involve these *vc*s need to *report* what was accessed or *non-historical* branch conflicts to the `VersionedReadWriteSet`. For instance, every scan or point read from a *branchesToTrack* or *commitsToTrack* must have the primary keys of the records read added to the read set for the corresponding branch in the `VersionedReadWriteSet` (similar reporting is done from records updated in a branch, except those primary keys are added to the write set for the branch). This is done by the `TransactionManagedVersionedScanner` that wraps the versioned iterator that scans over the data in the desired *vc* or branch. Every record that is accessed by the underlying iterator is passed to the `HybridVersionedWorkspace` which extracts the primary key and adds it to the read set for the corresponding branch. To handle multi-*vc* and multi-branch scans, the ability for the Hybrid to efficiently report all the versions in which a record belongs is used so that if a record belongs to multiple branches it is added to the read set for all of those branches. Note that none of this overhead is required to scan *vc*s that existed prior to the start of the transaction since they have been solidifed and are serializable so do not need to be tracked, in turn the underlying file iterator can be access directly and full scan/read performance can be achieved.

branchesToTrack is initialized to hold every branch in the global snapshot copy the transaction acquired. *commitsToTrack* is initially empty. When a *vc* is added to a branch in *branchesToTrack*, the *vc* is added to *commitsToTrack*. To detect *non-historical* branches, the `HybridVersionedWorkspace` maintains a map from branch identifier to ancestor branch for which conflicts may arise, called the *branchToConflictingAncestor* map. It is initialized with every branch that existed in the global snapshot copy being mapped to itself. When a branch operation branches a *vc* in *commitsToTrack*, the branch this *vc* was made on is retrieved (it is actually stored in the *vc* identifier object) and the ancestor branch it conflicts with is looked up in *branchToConflictingAncestor*. Then, a new entry is added that also maps the new branch to this ancestor branch. This constraint/conflict is also added to the `VersionedReadWriteSet` for checking during validation against other transactions.

Note that the *vc*s created before the start of transaction are not in *commitsToTrack* and so *historical* branches never have to undergo this added dependency tracking and scans, reads, or updates of their data or *vc*s never incur any overhead and the underlying file iterator can be access directly and full scan/read performance can be achieved. Also, note that branching of a branch directly without a *vc* first to that branch is not allowed because it breaks the lineage tracking of the branch as discussed in 3.2, so the system only needs to concern itself with branches derived from *vc*s.

Merge Locking

The system has branch locks, utilizing the lock manager presented in 4.1.5. Whenever a branch is accessed for a merge operation, it uses *branchToConflictingAncestor* to forward lock requests to the correct conflicting branch as specified in 3.3.9. Once the exclusive lock is acquired the new global snapshot is retrieved and checked for any changes on either of the branches that are being merged and the transaction is aborted in a change is detected. A change can be detected using a version vector with one slot per branch that is incremented (by the `BatchTransactionComitter` if a *tc*ing transaction had a modification to that branch. The end version vector is then associated with the `GlobalSnapshot` that can be retrieved after the merge lock is acquired. Analogously, if a transaction that just wants to modify a branch (i.e., `branch`, `vc`, `update`), it uses *branchToConflictingAncestor* to get the correct lock to grab and checks for a merge *vc* on the desired branch (or conflicting ancestor) and aborts if one is detected (merge *vc*s are stored in a sorted map per branch in the `VersionGraph` object so the transaction can use the current *head vc* commit identifier to determine if there have been any subsequent merge *vc*s after this).

4.2.6 Global Snapshot Reconciliation In Detail: Merging a Transaction Local Snapshot into Global Snapshot

Now we discuss the implementation of the algorithm for taking a transactions local workspace changes and incorporating them into the current global snapshot to build a new global snapshot which was presented in 3.3.6 (the reader is encouraged to review this before proceeding). Remember this procedure just takes the changes of a single transaction and incorporates it into the new global file.

This process starts by creating a *lightweight copy for replacement* of the current `GlobalSnapshot`. This is different from the copy acquired by a new transaction. For the most part the same things are copied and shared, except the *commonVersionGraph* and `TransactionMasterWorkspaceSnapshotManager`s can be accessed directly (no `TransactionWorkspaceSnapshotManager`s are produced).

Updating the Version Graph

After *vc*s are rebased, if necessary, new *vc* identifiers are generated and then the *commonVersionGraph* is updated to include these new *vc*s and the edges between (based on how the old *vc*s where connected in the transaction's worksapce). To prevent these *vc*s becoming visible to transactions that are using the *commonVersionGraph* in their workspaces, the transaction's `VersionGraph` object masks the *commonVersionGraph* such that no *vc* added after the transaction started is visible in the union of the *commonVersionGraph* and the *workspaceVersionGraph* (this is done through mechanisms provided by the graph library being used) [1].

Rebasing Regular *vc*s

A *regular* vc is a *vc* that is not a merge or branch creation *vc*, those cases are discussed subsequently. Rebasing a *vc* involves simply rebasing all the snapshot managers corresponding to this branch on every segment tracked by this branch. At this point when a new *vc* (that is not a merge or a branch creation *vc*) is being added all dependencies that could have affected *branch-membership* for the branch the *vc* is on have been applied (this is true because *vc*s are processed in topologically sorted order). Thus, all the segments that are relevant to this *vc* are in the new `GlobalSnapshot`'s (that is being built) *branch-membership* `TransactionMasterWorkspaceSnapshotManager`'s workspace for the branch the *vc* is on. Thus, using the segments listed here, we rebase each `TransactionMasterWorkspaceSnapshotManager` per segment as follows.

1) Given a delta corresponding to a *vc* that the transaction made, this delta is XORed into the workspace for the TransactionMasterWorkspaceSnapshotManager, then a new snapshot is created. To optimize, the *vc* delta is passed along to the SnapshotManager that backs the TransactionMasterWorkspaceSnapshotManager. The backing SnapshotManager may also be keeps an aggregate delta of all the workspace changes from previous transactions since the last *vc* processed. Before creating the snapshot based on the delta reported by the current transaction, the SnapshotManager will XOR the aggregate delta into the reported delta for the *vc* and this creates the new effective *vc* delta for that *vc*. The new *vc* needs to include all the previously un*vc*ed changes by previous transactions that may have executed concurrently, but did not produce update conflicts, so that the resultant snapshot is a valid snapshot with respect to the history of branch changes by previous transactions. Once a *vc* delta is created that includes all of these changes the deltas for subsequent *vc*s do not need to include the aggregate delta and the aggregate delta is reset.

2) 1) is repeated for every *vc* delta that the transaction produced.

Remember updating the workspace of the TransactionMasterWorkspaceSnapshotManager is ok because the TransactionMasterWorkspaceSnapshotManager has a separate snapshot object that represents the last transactionally consistent snapshot of the workspace that is currently exposed to all transactions as part of the previous GlobalSnapshot and this is immutable and separate from the workspace of the TransactionMasterWorkspaceSnapshotManager that is currently being written to.

Updating Workspaces

After rebasing *vc*s for a transaction, if the transaction made any changes to any of the workspaces of the TransactionWorkspaceSnapshotManagers that were not part of any *vc* then this final workspace delta is XORed into the workspace for the corresponding `TransactionMasterWorkspaceSnapshotManager` so that next global workspace released reflects the most recent un*vc*ed changes of the the transaction.

Branch Initialization

Whenever a new branch is detected, the parent *vc* identifier from which the branch was derived, the *derivation vc*, and the transaction's Hybrid storage model object, called a `HybridDbFile` are analyzed and the `HybridDbFile` of the current `GlobalSnapshot` is updated in the following way:

- Based on the `Segment` object local to the `HybridDbFile` of the transaction, a new `Segment` object is created that is essentially empty, but refers to the same underlying `AppendOnlyHeapFile` that contains the branch's new data. Then *branchIdToInitialSegmentNum* is updated so that it refers to the new segment.

- A new *branch-membership* `TransactionMasterWorkspaceSnapshotManager` is created and initialized based on the *branch-membership* SnapshotManager of the parent branch at the snapshot specified by the *derivation vc*. Then the workspace for the *branch-membership* is updated to include the new segment id and then a new snapshot is created.

- Now based on the complete initial *branch-membership*, the branch is initialized on every segment that has data belonging to that branch (more segments could be added at a later *vc* as a result of a merge).

Branch initialization per `Segment` proceeds as follows. Again the parent *vc* identifier from which the branch was derived, the *derivation* vc, and the corresponding `Segment` from the transaction local `HybridDbFile`'s file, call it `other`, are referenced during this process.

- Creating a new `TransactionMasterWorkspaceSnapshotManager` initialized based on the parent branch's bitmap for this segment at the *derivation* vc. Note that subsequent rebasing of this branch will bring its state up to date on all SnapshotManagers. Thus, the TransactionWorkspaceSnapshotManager has been upgraded to a TransactionMasterWorkspaceSnapshotManager.

- The SegmentBitmap is updated to include the new TransactionMasterWorkspaceSnapshotManager at the appropriate column (*branchIdToColumn* is also updated).

- trackingCommitTags is initialized so that the new branch is tracked on this segment starting at *vc* 1.

Merges

Handling merge *vc*s proceeds like branch initialization (for the primary parent, the branch that was modified as a result of the merge) for new segments that now need to be tracked as part of the branch except the new TransactionMasterWorkspaceSnapshotManager is initialized to the bitmap of the corresponding TransactionWorkspaceSnapshotManager at the *merge vc* snapshot. Since a transaction had an exclusive lock on the branch at the time the merge was conducted, the bitmaps can be transferred over directly to the new TransactionMasterWorkspaceSnapshotManager. Also, the tracking *vc* tag on this segment is set to the value of the tracking *vc* tag on the transaction's local segment object.

If a segment already had a TransactionMasterWorkspaceSnapshotManager for the primary parent then the current workspace is rebased based on the merge *vc* delta and a new *vc* snapshot created.

The *branch-membership* for the primary parent is also rebased so that it includes the new segments that it tracks.

Exposing Changes

After the changes from all transactions are rebased and the workspaces of all branches updated (all the corresponding TransactionMasterWorkspaceSnapshotManager workspaces are updated), the changes introduced by all the transactions must be exposed to subsequent transactions. The challenge here is that the changes to every TransactionMasterWorkspaceSnapshotManager for every segment must be released atomically. Remember, when a new snapshot is built, a lightweight copy of the current GlobalSnapshot is created. For simplicity, we stated that the TransactionMasterWorkspaceSnapshotManager was passed along directly. However, a lightweight copy of the TransactionMasterWorkspaceSnapshotManager was actually created with its data backed by its parent copy: the data was not copied, a new object was just created so it could be manipulated independently. This is so a new atomic reference that holds the current transactionally consistent state of the bitmap column will be created and will be separate from the one that transactions can access via the current GlobalSnapshot. Thus, the new snapshot that will be available to transactions can be placed in this new reference and not yet exposed to concurrent transactions. Once all these snapshots per TransactionMasterWorkspaceSnapshotManager are in place, the GlobalSnapshot is atomically replaced itself via the globally accessible atomic reference, and now the changes of all transactions have been incorporated to build a new GlobalSnapshot and have been released to subsequent transactions.

4.2.7 Batch Transaction Committer

When transactions are ready to *tc* they submit their workspace to the BatchTransactionCommitter thread that makes a *lightweight copy for replacement* of the current GlobalSnapshot and then merges in all the changes of every transaction that is read to *tc* to build a new GlobalSnapshot that reflects the changes of all the transactions. Then all changes are exposed atomically by replaced the current global snapshot with the new one.

This is simple and justifiable because all of the transactions data changes are already installed, all that needs to be done is patch the version graph and update the bitmaps on the affected segments. Furthermore, since these bitmaps are compressed and are in a representation where they are stored contiguously in-memory, the BatchTransactionCommitter has cache locality when accessing an updating the bitmap. It also in a sense batch rebases all the *vc*s of all the transactions of a *tc*ing transaction.

This is a general purpose solution which is designed to handle arbitrary version graph changes in an efficient way by batching changes. The `BatchTransactionCommitter` also forces the head of the log to disk for durability and so amortizes the cost of disk writes over many transactions.

However, this means that writes to different branches need to wait on each other, but controlled parallelism can be applied here. There can be a committer thread per group of branches and if a transaction writes to only the branches in one of those sets, then its changes can be processed in parallel with the other batch committers. However, if a transaction's changes involve changes to branches in two different sets then the transaction's changes have to be processed after the other batch committers have finished by a master batch comitter than integrates the final changes. However, for the common case of independent branch writes, full performance can be achieved and rebases to that branch are effectively batched. This also simplified complexity in handling transactions that affect many branches.

As a final note, there is also a background log flusher thread the flushes the log to disk in the background. This helps speed up transaction *tc* time because as the committer is processing the bitmap changes, the log flusher flushes and forces the log to disk so that when the committer finishes building the next global snapshot, all the required changes are already on disk and the comitter does not have to force the log to disk.

4.2.8 Transaction Abort

Since a transaction's deltas are buffered in memory (which is reasonable since these deltas can be compressed) and the deltas are never exposed until *tc* (merged into a new `GlobalSnapshot`), transaction abort simply involves discarding the in-memory changes to bitmaps that the transaction performed, no-undo is required. However, this does leave some slots in the segments that the transaction added data to vacant. The reason for this is so that write performance in the common case is just an append to the main file which is faster than traditional OCC schemes that use separate files or pages to store temporary version of modified records that must then be copied to the correct locations corresponding to the records modified to make those record visible to subsequent transactions. This problem can be partially solved by putting the locations of wasted slots of an aborted transaction in a pool of slots that can be re-used by subsequent transactions, and transactions check this pool for and atomically get independent slots to write to first, before consulting the `AppendOnlyHeapFile`. Though this re-uses storage, it may make a transaction incur a random I/O to fetch the page holding the empty slot from disk. Exploring better strategies for space reclamation is an element of future work.

4.2.9 Cleaning up Versioned Read/Write Sets

Though the `VersionedReadWriteSet`s stored for validation only contain the *primary keys* of the records touched by the transaction they consume more and more memory as the system proceeds. Each `GlobalSnapshot` is assigned a *snapshot number*. Every transaction's `VersionedReadWriteSet` is available in a *readWriteSetMap* that maps from transaction identifier to its `VersionedReadWriteSet`. The system also maintains a *snapshotNumToTransactionSet* that maps a *snapshot number* to the transactions whose changes were incorporated into that `GlobalSnapshot`, this is a sorted map so ranges can be scanned. Every transaction helps clean these up at transaction start. Before a transaction starts it records the *snapshot number* of the current `GlobalSnapshot` and places this is another map *tidToEarliest* that specifies the earliest *snapshot number* that a transaction has found. The transaction then tries to clean up unneeded `VersionedReadWriteSet` as discussed shortly. After this is done, it proceeds to grab its copy of the current `GlobalSnapshot`, which may have a later *snapshot number*. Note that a transaction does not grab a snapshot at the time it records the earliest snapshot number it has seen, it just records the earliest possible snapshot number it could get so if it is delayed adding an entry to *tidToEarliest* and the `GlobalSnapshot` advances this is ok because those more recent `VersionedReadWriteSet`s is what the transaction will be validated against, the system just needs to ensure that no transactions to validate against are missed. Thus, after the addition to *tidToEarliest*, the transaction finds the minimum snapshot number in *tidToEarliest* then uses *snapshot-*

NumToTransactionSet to get all of the transactions identifiers of all the transactions that belong to a snapshot with a snapshot number strictly smaller than the minimum snapshot number found. Then the corresponding `VersionedReadWriteSet` are removed from the *readWriteSetMap* and the memory is freed.

4.2.10 Recovery

Recovery is straight forward. Each transaction is assigned a global write number or serialization number by the `BatchTransactionCommitter`. Before releasing the new `GlobalSnapshot`, the `BatchTransactionCommitter` logs a summary record that includes the write numbers of all the transactions it processed. This summary is essentially a batch *tc* log record, *tc* log records are added to the log and then the log is forced so that transactions are only *tc*ed if the head of the log and the corresponding *tc* log records made it to disk. This combined with the fact that the system is append only and data is not modified in place, redo only recovery is sufficient. Now all that is required is for all transaction to log all changes to the Hybrid state of their local `GlobalSnapshot` (see 4.2.3). This includes the records themselves and the bitmap deltas for every `TransactionMasterWorkspaceSnapshotManager` (assuming the backing `SnapshotManager` stores snapshots by storing deltas). Standard WAL logging mechanism are utilized. When a transaction creates a *vc*, each delta created in the relevant `TransactionWorkspaceSnapshotManagers` (remember there is one per segment for every branch tracked by that segment) is assigned a *local write number* and this is logged with the delta created on that `TransactionWorkspaceSnapshotManager`. If the transaction made un-*vc*ed changes to the workspace those are logged at the end of the transaction and assigned the next write number (it is also noted that this last delta does not correspond to a *vc*).

To recover the state of the database simply requires replaying the all the changes of *tc*ed transactions in serialization number order and when processing the bitmap changes of a particular transaction for a particular `TransactionMasterWorkspaceSnapshotManager`, apply those changes in *local write number* order (i.e., writing exactly the required deltas in the right order to the pack file on disk and keeping an aggregate delta in-memory that includes every delta introduced by every *tc*ed transaction, so as to recover the entire workspace for that `TransactionMasterWorkspaceSnapshotManager`). The version graph can be recovered similarly (by logging nodes and edges and offseting node identified properly based on the local write number, serialization number of the transactions, and the final node identifiers of the previous transactions).

Checkpointing

Though recovering the entire data from a log ensure recovery, after a long period of time the log will grow. Thus, a checkpointer thread periodically forces pages to disk and writes changes (bitmap deltas) for every `TransactionMasterWorkspaceSnapshotManager` to disk (to the pack file that backs them on disk). To perform this atomically, the checkpointer grabs a global snapshot and using a index structure maintained by the system to determine the highest write number in this snapshot. It then forces all the deltas of every `TransactionMasterWorkspaceSnapshotManager` to disk and records a map of `TransactionMasterWorkspaceSnapshotManager` identifier (i.e. pack file name) to the EOF (end of file) location after flushing and forcing. Then the checkpointer writes this information to a checkpoint record that is in a separate checpoint meta data file. This checkpoint record also specifies the the highest write (serialization) number of the `GlobalSnapshot` it checkpointed to disk and the minimum of the log sequence numbers of the transaction start log entry of all running transactions and the transactions whose changes are reflected in the current global snapshot and any subsequent snapshot created since the checkpointer captured its global snapshot. This *min lsn* specifies the earliest point in the log record to look for relevant log records of *tc*ed transactions. To recover the records the systems uses the AIRES recovery algorithm. During recovery, the system starts log replay from *min lsn* of the most recent checkpoint and filters out log records belonging to transactions with write numbers less than the highest write number of the `GlobalSnapshot` it checkpointed to disk since their changes are already in the corresponding on disk structures (the write numbers of the relevant transactions can be found by rolling forward in the log and examining the batch commit records logged by the `BatchTransactionCommitter`). For changes that are missing the recovery procedure uses the *map of* `TransactionMasterWorkspaceSnapshotManager` *identifier (i.e. pack file name) to the EOF location* from the most recent checkpoint as the start location to start applying new bitmap deltas. The checkpoint meta data file is append only and the checkpoint record has a checksum, so either the new checkpoint record makes it to disk or it does not and if the recovery algorithm starts from an earlier checkpoint all it will be doing is overwriting the data that is already in the on disk structures (e.g. pack files) with the same data, which does not affect correctness.

For performance reasons during *tc*, the on disk structures that hold the records and the pack files should be on a separate disk from the log.

Chapter 5

Related Work

5.1 Related Work in Dataset Versioning

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 [5, 92, 82, 76] support "time-travel", i.e., the ability to query point-in-time snapshots of a linear chain of database versions. Lomet et al. introduced *ImmortalDB*, a temporal versioning system built directly into SQLServer [54]. 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 [55]. Recent work has looked into linear versioning for specialized databases, e.g., graph and scientific array databases. Khurana et al. [46] develop snapshot retrieval schemes for graph databases, while Seering et al. [79] develop compression schemes for array databases. Soroush et al. [83] 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 [65] presents a logic-based approach to representing valid time in RDF and OWL. Several papers (e.g., [2, 93]) 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 [18, 49, 59]. 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 [9, 69]. However, these methods only store enough history to preserve transactional semantics, whereas Decibel preserves historical records to maintain branched lineage.

Some operations in Decibel include provenance tracking at the record or version level. Provenance tracking in databases and scientific workflow systems has been studied extensively as well (see, e.g., [31, 25]). But those systems do not include any form of collaborative version control, and do not support unified querying and analysis over provenance and versioning information [20].

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. Instead, Decibel ports the broad API and workflow model of these systems to a traditional relational database management system.

There exists a considerable body of work on "fully-persistent" data structures, B+Trees in particular [27, 51, 72]. 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. [40] 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 [84] considers branchable clones which leverage existing copy-on-write algorithms for creating cloneable B-Trees [27]. 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.

Even discounting the inherent differences between key-value and relational storage models, none of the aforementioned work on multi-versioned B-Trees considers the full range of version control operations and ad hoc analytics queries that we consider with Decibel. In general, B-Trees are appropriate for looking up individual records in particular versions, but are unlikely to be useful in performing common versioning operations like scan, merge, and difference, which are our focus in this paper.

5.2 Related Work in Transactions

Transaction management is a well studied field in database systems. There has been much work in concurrency control protocols that are optimistic or maintain multiple versions of records (MVCC) so that writes do not block reads and vice versa [74]. Decibel is a mix of an Optimistic Concurrency Control and Multi-version Concurrency Control system to maintain transaction isolation. It uses the built in versioning to ensure that transactions get isolated transactionally consistent snapshots of the dataset to operate on, and then it employs the well known Optimistic Concurrency Control protocol [48] to serialize concurrent changes. Though Decibel does not attempt to revise these protocols to reduce conflicts (such as in [109]), it takes a new approach of changing the underlying versioning maintenance mechanism and provides a simpler model to reason about. In OCC each transaction gets its own workspace (changes buffered in memory and spilling over to an auxiliary file when they become too large) to hold new versions of updated records. Multi-version concurrency control protocols ([58] provides a survey of MVCC protocols) such as that in PostgreSQL [69] use append only data stores so that updates create a new record instead of updating records in place and timestamps to isolate changes, a transaction acquires a read time stamp at transaction start and all changes with a larger timestamp (or corresponding to transactions currently running) are invisible to the transaction. Other systems update records in place and store versions as before image deltas in *undo buffers* to enable reconstruction of older versions [67, 58] a transaction may be using. For timestamp protocols, the timestamp represents the snapshot the transaction is operating on, supporting high performance Snapshot Isolation ([10] provides a survey of all the *isolation levels* provided by modern databases) which is not serializable, even in the read only case [30]. Additional conflict tracking is then utilized to ensure serializability. This not only introduces complexity in providing transaction update serialization, but also presents additional challenges for updating indexes and finding records changed by a particular transaction since may versions of a record may exist concurrently. Indexes in PostgreSQL store no version information so only by examining the records themselves can it identify the ones relevant to the transaction [58], which can be inefficient if the records are scattered across multiple pages. Decibel leverages its natural versioning primitives and a simple indexing algorithm to solve all of these problems that scales a large number of concurrent readers.

Chapter 6

Conclusion

We presented Decibel, our database storage engine for managing and querying large numbers of relational dataset versions in a transactional manner. 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, 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. Finally we also showed how to leverage the natural versioning properties of Decibel to implement a highly concurrent versioned database that supports transactions and non-blocking, high throughput read only historical cross-versioned query transactions, the common use case of versioned databases.

Bibliography

[1] jGraphT.

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

[3] Daniel J. Abadi, Samuel Madden, and Nabil Hachem. Column-stores vs. row-stores: how different are they really? In *SIGMOD Conference*, pages 967–980, 2008.

[4] Daniel J. Abadi, Daniel S. Myers, David J. DeWitt, and Samuel Madden. Materialization Strategies in a Column-Oriented DBMS. In *ICDE*, pages 466–475, 2007.

[5] Ilsoo Ahn and Richard Snodgrass. Performance evaluation of a temporal database management system. In *SIGMOD*, pages 96–107, 1986.

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

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

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

[9] Hal Berenson, Phil Bernstein, Jim Gray, Jim Melton, Elizabeth O'Neil, and Patrick O'Neil. A critique of ANSI SQL isolation levels. In *SIGMOD*, pages 1–10, 1995.

[10] Hal Berenson, Phil Bernstein, Jim Gray, Jim Melton, Elizabeth O'Neil, and Patrick O'Neil. A critique of ansi sql isolation levels. In *Proceedings of the 1995 ACM SIGMOD International Conference on Management of Data*, SIGMOD '95, pages 1–10, New York, NY, USA, 1995. ACM.

[11] Philip A. Bernstein, Sudipto Das, Bailu Ding, and Markus Pilman. Optimizing optimistic concurrency control for tree-structured, log-structured databases.

In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, SIGMOD '15, pages 1295–1309, New York, NY, USA, 2015. ACM.

[12] Anant Bhardwaj, Amol Deshpande, Aaron Elmore, David Karger, Sam Madden, Aditya Parameswaran, Harihar Subramanyam, Eugene Wu, and Rebecca Zhang. Collaborative data analytics with datahub (demo). In *VLDB*, 2015.

[13] Anant P. Bhardwaj, Souvik Bhattacherjee, Amit Chavan, Amol Deshpande, Aaron J. Elmore, Samuel Madden, and Aditya G. Parameswaran. DataHub: collaborative data science & dataset version management at scale. In *CIDR*, 2015.

[14] Souvik Bhattacherjee, Amit Chavan, Silu Huang, Amol Deshpande, and Aditya G. Parameswaran. Principles of dataset versioning: Exploring the recreation/storage tradeoff. In *PVLDB*, 2015.

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

[16] Shawn Bowers, Timothy M McPhillips, and Bertram Ludäscher. Provenance in collection-oriented scientific workflows. *Concurrency and Computation: Practice and Experience*, 20(5):519–529, 2008.

[17] Paul G Brown. Overview of scidb: large scale array storage, processing and analysis. In *SIGMOD*, pages 963–968. ACM, 2010.

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

[19] Peter Buneman and Wang chiew Tan. Archiving scientific data. In *ACM SIGMOD*, pages 1–12, 2002.

[20] Amit Chavan, Silu Huang, Amol Deshpande, Aaron J. Elmore, Samuel Madden, and Aditya G. Parameswaran. Towards a unified query language for provenance and versioning. In *TaPP*, 2015.

[21] Brian F. Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. Benchmarking cloud serving systems with ycsb. In *SoCC*, 2010.

[22] Mark Craven, Johan Kumlien, et al. Constructing biological knowledge bases by extracting information from text sources. In *ISMB*, volume 1999, pages 77–86, 1999.

[23] Philippe Cudré-Mauroux, Hideaki Kimura, Kian-Tat Lim, Jennie Rogers, Roman Simakov, Emad Soroush, Pavel Velikhov, Daniel Wang, Magdalena Balazinska, Jacek Becla, et al. A Demonstration of SciDB: A Science-Oriented DBMS. *PVLDB*, 2(2):1534–1537, 2009.

[24] Philippe Cudré-Mauroux, Hideaki Kimura, Kian-Tat Lim, Jennie Rogers, Roman Simakov, Emad Soroush, Pavel Velikhov, Daniel L. Wang, Magdalena Balazinska, Jacek Becla, David J. DeWitt, Bobbi Heath, David Maier, Samuel Madden, Jignesh M. Patel, Michael Stonebraker, and Stanley B. Zdonik. A Demonstration of SciDB: A Science-Oriented DBMS. *PVLDB*, 2(2):1534–1537, 2009.

[25] Susan B Davidson, Sarah Cohen Boulakia, et al. Provenance in scientific workflow systems. *IEEE Data Eng. Bull.*, 30(4):44–50, 2007.

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

[27] James R. Driscoll, Neil Sarnak, Daniel D. Sleator, and Robert E. Tarjan. Making data structures persistent . In *STOC*, pages 109–121, 1986.

[28] Jose M. Faleiro and Daniel J. Abadi. Rethinking serializable multiversion concurrency control. *Proc. VLDB Endow.*, 8(11):1190–1201, July 2015.

[29] Alan Fekete, Dimitrios Liarokapis, Elizabeth O'Neil, Patrick O'Neil, and Dennis Shasha. Making snapshot isolation serializable. *ACM Trans. Database Syst.*, 30(2):492–528, June 2005.

[30] Alan Fekete, Elizabeth O'Neil, and Patrick O'Neil. A read-only transaction anomaly under snapshot isolation. *SIGMOD Rec.*, 33(3):12–14, September 2004.

[31] Juliana Freire, David Koop, Emanuele Santos, and Cláudio T Silva. Provenance for computational tasks: A survey. *Computing in Science & Engineering*, 10(3):11–21, 2008.

[32] Nathan Goodman, Steve Rozen, Lincoln Stein, and A. G. Smith. The LabBase system for data management in large scale biology research laboratories. *Bioinformatics*, 14(7):562–574, 1998.

[33] Todd J. Green, Grigoris Karvounarakis, and Val Tannen. Provenance semirings. In *Proceedings of the Twenty-sixth ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems*, PODS '07, pages 31–40, New York, NY, USA, 2007. ACM.

[34] Philip J. Guo, Sean Kandel, Joseph M. Hellerstein, and Jeffrey Heer. Proactive wrangling: mixed-initiative end-user programming of data transformation scripts. In *UIST*, pages 65–74, 2011.

[35] Tom Heath and Christian Bizer. Linked data: Evolving the web into a global data space. *Synthesis lectures on the semantic web: theory and technology*, 1(1):1–136, 2011.

[36] Jeffrey Heer and Joseph M. Hellerstein. Data visualization & social data analysis. *PVLDB*, 2(2):1656–1657, 2009.

[37] GD Held, MR Stonebraker, and Eugene Wong. Ingres: A relational data base system. In *Proceedings of the May 19-22, 1975, national computer conference and exposition*, pages 409–416. ACM, 1975.

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

[39] Zachary G. Ives, Nitin Khandelwal, Aneesh Kapur, and Murat Cakir. ORCHESTRA: Rapid, Collaborative Sharing of Dynamic Data. In *CIDR*, pages 107–118, 2005.

[40] Linan Jiang, Betty Salzberg, David Lomet, and Manuel Barrena. The BT-Tree: A branched and temporal access method. In *PVLDB*, pages 451–460, 2000.

[41] Robert Kallman, Hideaki Kimura, Jonathan Natkins, Andrew Pavlo, Alex Rasin, Stanley B. Zdonik, Evan P. C. Jones, Samuel Madden, Michael Stonebraker, Yang Zhang, John Hugg, and Daniel J. Abadi. H-store: A high-performance, distributed main memory transaction processing system. In *PVLDB*, volume 1, 2008.

[42] Sean Kandel, Andreas Paepcke, Joseph Hellerstein, and Jeffrey Heer. Wrangler: interactive visual specification of data transformation scripts. In *CHI*, pages 3363–3372, 2011.

[43] Sean Kandel, Ravi Parikh, Andreas Paepcke, Joseph M. Hellerstein, and Jeffrey Heer. Profiler: integrated statistical analysis and visualization for data quality assessment. In *AVI*, pages 547–554, 2012.

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

[45] Udayan Khurana and Amol Deshpande. Efficient snapshot retrieval over historical graph data. *CoRR*, abs/1207.5777, 2012.

[46] Udayan Khurana and Amol Deshpande. Efficient snapshot retrieval over historical graph data. *ICDE*, 2013.

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

[48] H. T. Kung and John T. Robinson. On optimistic methods for concurrency control. *ACM Trans. Database Syst.*, 6(2):213–226, June 1981.

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

[50] Gad M. Landau, Jeanette P. Schmidt, and Vassilis J. Tsotras. Historical queries along multiple lines of time evolution. In *PVLDB*, pages 703–726, 1995.

[51] Sitaram Lanka and Eric Mays. Fully persistent B+-Trees . In *SIGMOD*, 1991.

[52] Suh-Yin Lee and Ruey-Long Liou. A multi-granularity locking model for concurrency control in object-oriented database systems. *IEEE Trans. on Knowl. and Data Eng.*, 8(1):144–156, February 1996.

[53] Chunhyeok Lim, Shiyong Lu, Artem Chebotko, and Farshad Fotouhi. OPQL: A first opm-level query language for scientific workflow provenance. In *Services Computing (SCC), 2011 IEEE International Conference on*, pages 136–143. IEEE, 2011.

[54] David Lomet, Roger Barga, Mohamed F. Mokbel, German Shegalov, Rui Wang, and Yunyue Zhu. Transaction time support inside a database engine. In *ICDE*, 2006.

[55] David Lomet, Mingsheng Hong, Rimma Nehme, and Rui Zhang. Transaction time indexing with version compression. In *VLDB*, 2008.

[56] David Lomet and Mohamed F. Mokbel. Locking key ranges with unbundled transaction services. *Proc. VLDB Endow.*, 2(1):265–276, August 2009.

[57] Michael Maddox, David Goehring, Aaron Elmore, Samuel Madden, Aditya Parameswaran, and Amol Deshpande. Decibel: The relational dataset branching system. Technical Report, available at: `http://web.engr.illinois.edu/~adityagp/decibel-tr.pdf`, 2016.

[58] Dibyendu Majumdar. A Quick Survey of MultiVersion Concurrency Algorithms. November 2007.

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

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

[61] Eric Mays, Sitaram Lanka, Bob Dionne, and Robert Weida. A persistent store for large shared knowledge bases . In *IEEE Transactions on Knowledge and Data Engineering*, pages 33–41, 1991.

[62] C. Mohan, Don Haderle, Bruce Lindsay, Hamid Pirahesh, and Peter Schwarz. Aries: A transaction recovery method supporting fine-granularity locking and partial rollbacks using write-ahead logging. *ACM Trans. Database Syst.*, 17(1):94–162, March 1992.

[63] C Mohan, Bruce Lindsay, and Ron Obermarck. Transaction management in the R* distributed database management system. *TODS*, 11(4):378–396, 1986.

[64] Anthony Molinaro. *SQL Cookbook.* O'Reilly, 2005.

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

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

[67] Thomas Neumann, Tobias Mühlbauer, and Alfons Kemper. Fast serializable multi-version concurrency control for main-memory database systems. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, SIGMOD '15, pages 677–689, New York, NY, USA, 2015. ACM.

[68] Gultekin Ozsoyoglu and Richard T. Snodgrass. Temporal and Real-Time Databases: A Survey. *IEEE Transactions on Knowledge and Data Engineering*, 7:513–532, 1995.

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

[70] E. Rahm and A. Thomasian. Distributed optimistic concurrency control for high performance transaction processing. In *Databases, Parallel Architectures and Their Applications,. PARBASE-90, International Conference on*, pages 490–495, Mar 1990.

[71] Raghu Ramakrishnan and Johannes Gehrke. *Database Management Systems.* Osborne/McGraw-Hill, Berkeley, CA, USA, 2nd edition, 2000.

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

[73] Mark A Roth, Herry F Korth, and Abraham Silberschatz. Extended algebra and calculus for nested relational databases. *ACM TODS*, 1988.

[74] Mohammad Sadoghi, Mustafa Canim, Bishwaranjan Bhattacharjee, Fabian Nagel, and Kenneth A. Ross. Reducing database locking contention through multi-version concurrency. *Proc. VLDB Endow.*, 7(13):1331–1342, August 2014.

[75] B. Salzberg and V. Tsotras. Comparison of access methods for time-evolving data. *ACM Comput. Surv.*, 31(2), 1999.

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

[77] Sunita Sarawagi. User-Adaptive Exploration of Multidimensional Data. In *VLDB*, pages 307–316, 2000.

[78] Sunita Sarawagi and Gayatri Sathe. i^3: Intelligent, interactive investigaton of olap data cubes. In *SIGMOD Conference*, page 589, 2000.

[79] Adam Seering, Philippe Cudre-Mauroux, Samuel Madden, and Michael Stonebraker. Efficient versioning for scientific array databases. In *ICDE*. IEEE, 2012.

[80] Adam Seering, Philippe Cudre-Mauroux, Samuel Madden, and Michael Stonebraker. Efficient versioning for scientific array databases. In *ICDE*. IEEE, 2012.

[81] Richard Snodgrass. The temporal query language TQuel. *ACM Transactions on Database Systems (TODS)*, 12(2):247–298, 1987.

[82] Richard T. Snodgrass, editor. *The TSQL2 Temporal Query Language*. Kluwer, 1995.

[83] Emad Soroush and Magdalena Balazinska. Time travel in a scientific array database. In *ICDE*, 2013.

[84] Benjamin Sowell, Wojciech Golab, and Mehul A. Shah. Minuet: A scalable distributed multi-version B-Tree. In *PVLDB*, pages 884–895, 2012.

[85] Chris Stolte, Diane Tang, and Pat Hanrahan. Polaris: a system for query, analysis, and visualization of multidimensional databases. *Commun. ACM*, 51(11):75–84, 2008.

[86] Michael Stonebraker. Concurrency control and consistency of multiple copies of data in distributed ingres. In *Berkeley Workshop*, pages 235–258, 1978.

[87] Michael Stonebraker, Jacek Becla, David J DeWitt, Kian-Tat Lim, David Maier, Oliver Ratzesberger, and Stanley B Zdonik. Requirements for science data bases and scidb. In *CIDR*, volume 7, pages 173–184, 2009.

[88] Michael Stonebraker, Gerald Held, Eugene Wong, and Peter Kreps. The design and implementation of ingres. In *TODS*, volume 1, pages 189–222. ACM, 1976.

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

[90] Michael Stonebraker and Greg Kemnitz. The POSTGRES next generation database management system. In *CACM*, volume 34, pages 78–92. ACM, 1991.

[91] Mike Stonebraker, Daniel J Abadi, Adam Batkin, Xuedong Chen, Mitch Cherniack, Miguel Ferreira, Edmond Lau, Amerson Lin, Sam Madden, Elizabeth O'Neil, Pat O'Neil, Alex Rasin, Nga Tran, and Stan Zdonik. C-store: a column-oriented dbms. In *Proceedings of the 31st international conference on Very large data bases*, pages 553–564. VLDB Endowment, 2005.

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

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

[94] Irving L Traiger, Jim Gray, Cesare A Galtieri, and Bruce G Lindsay. Transactions and consistency in distributed database systems. *TODS*, 7(3):323–342, 1982.

[95] Stephen Tu, Wenting Zheng, Eddie Kohler, Barbara Liskov, and Samuel Madden. Speedy transactions in multicore in-memory databases. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, SOSP '13, pages 18–32, New York, NY, USA, 2013. ACM.

[96] `http://aws.amazon.com`. Amazon Web Services.

[97] `http://bigdata.csail.mit.edu/Living_Lab`. Living Lab.

[98] `http://git-scm.org`.

[99] `http://github.org`. GitHub.

[100] `http://ipython.org`. IPython.

[101] `http://mercurial.selenic.com`.

[102] `http://office.microsoft.com/en-us/excel/`. Microsoft Excel.

[103] `http://pandas.pydata.org`. Python Data Analysis Library (retrieved June 1, 2014).

[104] `http://www.bitbucket.org`. Bitbucket.

[105] `http://www.peterlundgren.com/blog/on-gits-shortcomings`. On Git's Shortcomings.

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

[107] Eugene Wu, Samuel Madden, and Michael Stonebraker. SubZero: A fine-grained lineage system for scientific databases. In *ICDE*, pages 865–876, 2013.

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

[109] Yuan Yuan, Kaibo Wang, Rubao Lee, Xiaoning Ding, Jing Xing, Spyros Blanas, and Xiaodong Zhang. Bcc: Reducing false aborts in optimistic concurrency control with low cost for in-memory databases. *Proc. VLDB Endow.*, 9(6):504–515, January 2016.