ORPHEUSDB: Bolt-on Versioning for Relational Databases

Silu Huang[1], Liqi Xu[1], Jialin Liu[1], Aaron Elmore[2], Aditya Parameswaran[1]
[1]U Illinois (Urbana-Champaign) [2]U Chicago

ABSTRACT

Data science teams often collaboratively analyze datasets, generating dataset versions at each stage of iterative exploration and analysis. There is a pressing need for a system that can support dataset versioning, enabling such teams to efficiently store, track, and query across dataset versions. While git and svn are highly effective at managing code, they are not capable of managing large unordered structured datasets efficiently, nor do they support analytic (SQL) queries on such datasets. We introduce ORPHEUSDB, a dataset version control system that "*bolts on*" versioning capabilities to a traditional relational database system, thereby gaining the analytics capabilities of the database "for free", while the database itself is unaware of the presence of dataset versions. We develop and evaluate multiple data models for representing versioned data within a database. We additionally develop a light-weight partitioning scheme, titled LYRESPLIT, that further optimizes the data models for reduced storage consumption and query latencies. We demonstrate that with LYRESPLIT, ORPHEUSDB is on average $10^3\times$ faster in finding effective (and better) partitionings than competing approaches, while also reducing the latency of version retrieval by up to $20\times$ relative to schemes without partitioning.

1. INTRODUCTION

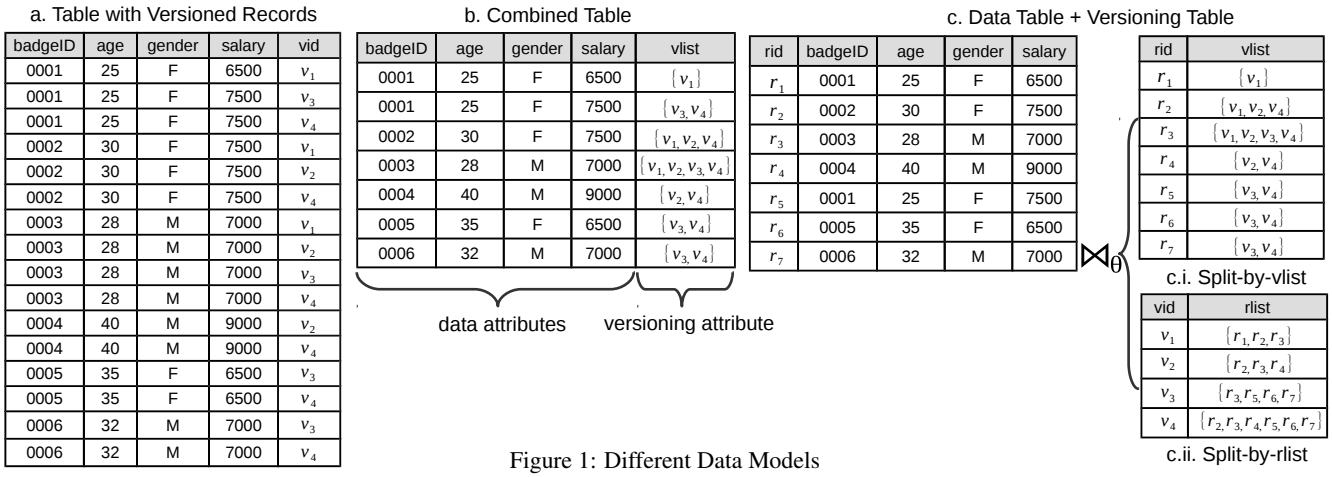
From large-scale physical simulations, to high-throughput genomic sequencing, and from conversational agent interactions to sensor data from the Internet of Things, the need for data science and extracting insights from large datasets has never been greater. To do this, teams of data scientists repeatedly transform their datasets in many ways: thus, the New York Times defines data science as a *step-by-step process of experimentation on data* [5]. The dataset versions generated, often into the hundreds or thousands, are stored in an ad-hoc manner, typically via copying and naming conventions in shared (networked) file systems. This makes it impossible to effectively manage, make sense of, or query across these versions. One alternative is to use a source code version control system like git or svn to manage dataset versions. However, source code version control systems are both inefficient at storing unordered structured datasets, and do not support advanced querying capabilities (e.g. querying for versions that satisfy some predicate). Therefore, when requiring advanced (SQL-like) querying capabilities, data scientists typically store each of the dataset versions as independent tables in a traditional relational database. This approach results in massive redundancy and inefficiencies in storage and query performance, as well as manual supervision and maintenance to track versions. As a worse alternative, they only store the most recent versions—thereby losing the ability to retrieve the original datasets or trace the provenance of the new versions. The question we ask in this paper is: *can we have the best of both worlds—advanced querying capabilities, plus effective and efficient versioning in a mature relational database*? More specifically, ***can traditional relational databases be made to support versioning?***

To answer this question we develop a system, titled ORPHEUSDB[1], by *"bolting-on"* versioning capabilities to a traditional relational database system that is unaware of the existence of versions. By doing so, we seamlessly leverage the analysis and querying capabilities that come "for free" with a database system, along with efficient versioning capabilities.

Developing ORPHEUSDB comes with a host of challenges, centered around the choice of the representation scheme or the data model used to capture versions within a database, as well as effectively balancing the storage costs with the costs for querying and operating on versions. We describe the challenges associated with the data model first.

Challenges in Representation. One simple approach of capturing dataset versions would be to represent the dataset as a relation in a database, and add an extra attribute corresponding to the version number, called *vid*, as shown in Figure 1(a). The version number attribute allows us to apply selection operations to extract or retrieve specific versions. However, this approach is extremely wasteful as each record is repeated as many times as the number of versions it belongs to. It is worth noting that a timestamp is not sufficient here, as a version can have multiple parents (a merge) and multiple children (branches). Therefore, a single timestamp value cannot capture which versions a tuple belongs to. To remedy this issue, one can take advantage of the array data type capabilities offered in current database systems, by replacing the version number attribute with an array attribute containing all of the versions that each record belongs to, as depicted in Figure 1(b). This reduces some of the storage overheads from replicating tuples. However, when adding a new version (e.g. a duplicate or a clone of an existing version) this approach leads to extensive modifications across the entire relation, since the array will need to be updated for every single record that belongs to the new version. Another strategy would be to separate the data from the versioning information into two tables as shown in Figure 1(c), where the first table—the data table—stores all of the records appearing in any of the versions, while the second table—the versioning table—captures the versioning information, or which version contains which records. This strategy, however, requires us to perform a join of these two tables to retrieve or recreate any versions. Further, there are two ways of recording the versioning information: the first involves using an array of versions, the second involves using an array of records; we illustrate this in Figure 1(c.i) and Figure 1(c.ii) respectively.

[1]Orpheus is a musician and poet from ancient Greek mythology with the ability to raise the dead with his music, much like ORPHEUSDB has the ability to retrieve old ("dead") dataset versions on demand.



a. Table with Versioned Records

badgeID	age	gender	salary	vid
0001	25	F	6500	v_1
0001	25	F	7500	v_3
0001	25	F	7500	v_4
0002	30	F	7500	v_1
0002	30	F	7500	v_2
0002	30	F	7500	v_4
0003	28	M	7000	v_1
0003	28	M	7000	v_2
0003	28	M	7000	v_3
0003	28	M	7000	v_4
0004	40	M	9000	v_2
0004	40	M	9000	v_4
0005	35	F	6500	v_3
0005	35	F	6500	v_4
0006	32	M	7000	v_3
0006	32	M	7000	v_4

b. Combined Table

badgeID	age	gender	salary	vlist
0001	25	F	6500	$\{v_1\}$
0001	25	F	7500	$\{v_3, v_4\}$
0002	30	F	7500	$\{v_1, v_2, v_4\}$
0003	28	M	7000	$\{v_1, v_2, v_3, v_4\}$
0004	40	M	9000	$\{v_2, v_4\}$
0005	35	F	6500	$\{v_3, v_4\}$
0006	32	M	7000	$\{v_3, v_4\}$

c. Data Table + Versioning Table

rid	badgeID	age	gender	salary
r_1	0001	25	F	6500
r_2	0002	30	F	7500
r_3	0003	28	M	7000
r_4	0004	40	M	9000
r_5	0001	25	F	7500
r_6	0005	35	F	6500
r_7	0006	32	M	7000

$\bowtie_\theta$

c.i. Split-by-vlist

rid	vlist
r_1	$\{v_1\}$
r_2	$\{v_1, v_2, v_4\}$
r_3	$\{v_1, v_2, v_3, v_4\}$
r_4	$\{v_2, v_4\}$
r_5	$\{v_3, v_4\}$
r_6	$\{v_3, v_4\}$
r_7	$\{v_3, v_4\}$

c.ii. Split-by-rlist

vid	rlist
v_1	$\{r_1, r_2, r_3\}$
v_2	$\{r_2, r_3, r_4\}$
v_3	$\{r_3, r_5, r_6, r_7\}$
v_4	$\{r_2, r_3, r_4, r_5, r_6, r_7\}$

Figure 1: Different Data Models

The latter approach allows easy insertion of new versions, without having to modify existing version information, but may have slight overheads relative to the former approach when it comes to joining the versioning table and the data table. Overall, however, as we demonstrate in this paper, the latter approach outperforms other approaches for most common operations.

Challenges in Balancing Storage and Querying Latencies. The next question is if we can improve the efficiency of the aforementioned approach even further, at the cost of possibly additional storage, since it requires a full theta join and examination of all of the data to reconstruct any given version. One approach is to partition the versioning and data tables such that we limit data access to recreate versions, while keeping storage costs bounded. However, as we demonstrate in this paper, the problem of identifying the optimal trade-off between the storage and version retrieval time is NP-HARD, via a reduction from the 3-PARTITION problem. That said, we can develop an efficient and light-weight approximation algorithm that enables us to trade-off storage and version retrieval time, providing a guaranteed $((1+\delta)^\ell, \frac{1}{\delta})$-factor approximation under certain reasonable assumptions—where the storage is a $(1+\delta)^\ell$-factor of optimal, and the average version retrieval time is $\frac{1}{\delta}$-factor of optimal, for any value of parameter $\delta \leq 1$ that expresses the desired trade-off. The parameter ℓ depends on the complexity of the branching structure of the version graph, and will be described later. In practice, this algorithm always performs better, i.e., has better retrieval time for a given storage budget, than other schemes for partitioning, and is about $1000\times$ faster than these schemes.

Related Work. As previously noted, ORPHEUSDB satisfies an unmet need in the recent growing body of literature on efficient dataset versioning. Work on time-travel databases [30] supports a linear chain of versions as opposed to arbitrary branches and merges as is the case in source code version control and in dataset versioning. We share the vision of the vision paper on the DataHub project [13] in supporting collaborative data analytics; we execute on that vision by supporting collaborative data analytics using a traditional relational database, thereby seamlessly leveraging the sophisticated analysis capabilities. Work on Decibel [25] describes techniques to build a standalone storage engine to support dataset versioning "from the ground up"—as such, their solution would not mesh well with a traditional relational database, since it requires extensive changes at all layers of the stack. Furthermore, their solution does not benefit from full-fledged query processing and optimization, logging, the ability to write UDFs, and all other benefits that come "for free" with a relational database. In this paper, we are approaching the problem from a different angle—the angle of *reuse*—how do we leverage the decades of effort in relational databases to support versioning without any substantial changes to the database. Recent work on the principles of dataset versioning is also relevant [14] in that it shares the concerns of minimizing storage and recreation cost; however, the paper considered the unstructured setting from an algorithmic viewpoint, and did not aim to build a full-fledged dataset versioning coupled within a relational database. Unlike all of these lines of work, we are focusing on working within the constraints of existing relational database systems, all of which have massive adoption and open-source development we can tap into. We describe related work in more detail in Section 6.

Contributions. The contributions of this paper are as follows:

- We develop a dataset version control system, titled ORPHEUSDB, with the ability to support both git-style version control commands and SQL-like queries. (Section 2)
- We compare different data models for representing versioned datasets and experimentally evaluate their performance in terms of storage consumption and time taken for querying. (Section 3)
- To further improve query efficiency, we formally develop the optimization problem of trading-off between the storage and version retrieval time via partitioning and demonstrate that this is NP-HARD. We then propose a light-weight approximation algorithm for this optimization problem, titled LYRESPLIT, providing a $((1+\delta)^\ell, \frac{1}{\delta})$-factor guarantee. (Section 4)
- We conduct extensive experiments using a versioning benchmark in [25] and demonstrate that LYRESPLIT is on average $1000\times$ faster than competing algorithms and performs better in balancing the storage and version retrieval time. (Section 5)

2. ORPHEUSDB OVERVIEW

ORPHEUSDB is a hosted system that supports dataset version management. Since ORPHEUSDB is built on top of standard relational databases, it inherits much of the same benefits of relational databases, while also compactly storing, tracking, and recreating versions on demand. In this section, we describe the user-facing interfaces of ORPHEUSDB, followed by the ORPHEUSDB system design. We begin by describing fundamental version-control concepts within ORPHEUSDB. ORPHEUSDB has been developed as open-source software; code is available at [10].

2.1 Dataset Version Control

The fundamental unit of storage within ORPHEUSDB is a *collaborative versioned dataset* (CVD) to which one or more users can contribute. Each CVD corresponds to a relation with a fixed schema, and implicitly contains many *versions* of that relation. A

version is an instance of the relation, specified by the user and containing a set of records—we will elaborate on how users can create versions subsequently. Versions within a CVD are related to each other via a *version graph*—a directed acyclic graph—representing how the versions were derived from each other: a version in this graph with two or more parents is defined to be a *merged version*. Records in a CVD are *immutable*, i.e., any modifications to any record attributes result in a new record, and are stored and treated separately within the CVD. Overall, there is a many-to-many relationship between records and versions that is captured within the CVD: each record can belong to many versions, and each version can contain many records. Each version has a unique version id, *vid*, and each record has its unique record id, *rid*. The record ids are used to identify immutable records within the CVD and are not visible to end-users of ORPHEUSDB. In addition, the relation corresponding to the CVD may have primary key attribute(s); this implies that for any version—an instance of the relation—no two records can have the same values for the primary key attribute(s). However, across versions, this needs not be the case. ORPHEUSDB can support multiple CVDs at a time. However, in order to better convey the core ideas of ORPHEUSDB, in the rest of the paper, we focus our discussion with a single CVD.

2.2 ORPHEUSDB APIs

Users interact with ORPHEUSDB via the command line, using both SQL queries, as well as git-style version control commands. To make modifications to versions, users can either use SQL operations issued to the relational database that ORPHEUSDB is built on top of, or can alternatively operate on them using programming or scripting languages, as we will describe subsequently. We begin by describing the version control commands.

Version control commands. Users can operate on CVDs much like they would with source code version control. The first and most fundamental operation is *checkout*: this command materializes a specific version of a CVD as a newly created regular table within a relational database that ORPHEUSDB is connected to. The table name is specified within the checkout command, as follows:

checkout -f [cvd] -v [vid] -t [table name]

Here, the version with id vid is materialized as a new table [table name] within the database, to which standard SQL statements can be issued, and which can later be added to the CVD as a new version. The version from which this table was derived—i.e., vid—is referred to as the *parent version* for the table.

Instead of materializing one version at a time, users can materialize multiple versions, by listing multiple vids in the command above, essentially *merging* multiple versions to give a single table. When merging, the records in the versions are added to the table in the precedence order listed after -v: for any record being added, if another record with the same primary key has already been added, it is omitted from the table. This ensures that the eventual materialized table also respects the primary key property. There are other conflict-resolution strategies, such as letting users resolve conflicted records manually; for simplicity, we use a precedence based approach. Internally, the *checkout* command records the versions that this table was derived from (i.e., those listed after -v), along with the table name. Note that only the user who performed the checkout operation is permitted access to the materialized table, so they can perform any analysis and modification on this table without interference from other users, only making these modifications visible when they add this table back as a new version to the CVD using the *commit* operation described next.

The *commit* operation adds a new version to the CVD, by making the local changes made by the user on their materialized table visible to others. The commit command has the following format:

commit -t [table name] -m [commit message]

Note that the commit message does not need to specify the intended CVD since ORPHEUSDB internally keeps a mapping between the table name and the original CVD. In addition, since the versions that the table was derived from originally during checkout are internally known to ORPHEUSDB, the table is added to the CVD as a new version with those versions as parent versions. During the commit operation, ORPHEUSDB compares the (possibly) modified materialized table to the parent versions. If any records were added or modified these records are treated as new records and added to the CVD as such. (Recall that records are immutable within a CVD.) Note that an alternative is to compare the new records with all of the existing records in the CVD to check if any of the new records have existed in any version in the past, which would take longer to execute. At the same time, the latter approach would identify records that were deleted then re-added later. Since we believe that this is not a common case, we opt for the former approach, which would only lead to modest additional storage at the cost of much less computation during commit. We call this the *no cross-version diff* implementation rule. Lastly, if the schema of the table that is being committed is different from the CVD it derives from, then it becomes part of a new CVD: a CVD has a single schema.

In order to support data science workflows, we additionally support the use of *checkout* and *commit* into and from csv (comma separated value) files via slightly different flags: -c for csv instead -t for table. The csv file can be processed in external tools and programming languages such as Python or R, not requiring that users perform the modifications and analysis using SQL. However, during commit, the user is expected to also provide a schema file via a -s flag so that ORPHEUSDB can make sure that the columns are mapped in the correct manner. An alternative would be to use schema inference tools, e.g., [26, 18], which could be seamlessly incorporated if need be. Internally, ORPHEUSDB also tracks the name of the csv file as being derived from one or more versions of the CVD, just like it does with the materialized tables.

In addition to checkout and commit, ORPHEUSDB also supports other commands, described very briefly here: *(a) List:* List the contents of a version without materializing it. *(b) Diff:* A standard differencing operation that compares two versions and outputs the records in one but not the other. *(c) Log:* Display metadata related to one or more versions, including parent and child versions, commit times, and commit messages. *(d) Optimize:* As we will see in the following, ORPHEUSDB can benefit from intelligent partitioning schemes (enabling other operations to access and process much less data), as we will describe in Section 4. While these partitioning algorithms can be called periodically by the system, they can also be invoked explicitly by the user.

SQL commands. ORPHEUSDB supports the use of SQL commands on CVDs via the command line using the *run* command, which either takes a SQL script as input or the SQL as a string. Apart from materializing a version (or versions) as a table via the checkout command and explicitly applying SQL operations on that table, ORPHEUSDB also allows users to directly execute SQL queries on a specific version, using special keywords VERSION, OF, and CVD via syntax

SELECT ... FROM VERSION [vid] OF CVD [cvd], ...

without having to materialize it. Further, by using renaming, users can operate directly on multiple versions (each as a relation) within a single SQL statement, enabling operations such as joins across multiple versions.

However, listing each version individually as described above may be cumbersome for some types of queries that users wish to

run, e.g., applying an aggregate across a collection of versions, or identifying versions that satisfy some property. For this, ORPHEUSDB also supports constructs that enable users to issue aggregate queries across CVDs grouped by version ids, or select version ids that satisfy certain constraints. Internally, these constructs are translated into regular SQL queries that can be executed by the underlying database system. In addition, ORPHEUSDB provides shortcuts for several types of queries that operate on the version graph, e.g., listing the descendant or ancestors of a specific version, or querying the metadata, e.g., identify the last modification (in time) to the CVD. We omit details of these query constructs due to space limitations.

2.3 System Architecture

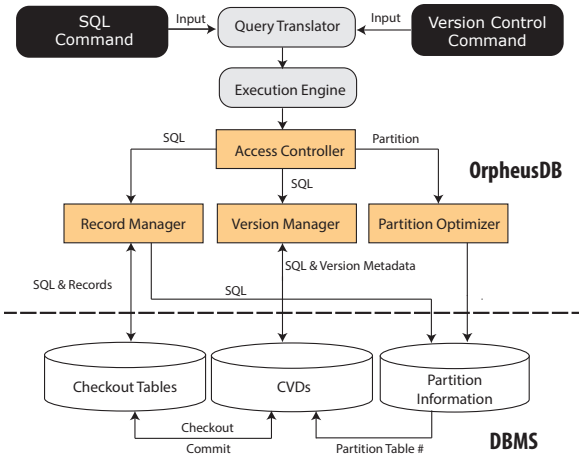


Figure 2: ORPHEUSDB Architecture

We implement ORPHEUSDB as a middleware layer or wrapper between end-users (or application programs) and a traditional relational database system—in our case, PostgreSQL. PostgreSQL is completely unaware of the existence of versioning, as versioning is handled entirely within the middleware. Figure 2 depicts the overall architecture of ORPHEUSDB. It includes a query translator and an execution engine. A command issued to ORPHEUSDB is first parsed by the translator, and then handed over to the executor. The executor interfaces with the access controller, version manager, record manager, and partition optimizer. The access controller manages the users' permissions to various CVDs and temporary materialized tables. The version manager is responsible for keeping track of the (i) derivation relationships between versions, (ii) the relationship between versions and materialized tables or files, and (iii) other metadata, such as checkout/commit times, users involved in the commit, and commit messages. The record manager is responsible for retrieving records from versions, materializing records, and adding new records to CVDs. The partition optimizer is periodically called to reorganize and optimize the data storage in the backend, and will be the focus of Section 4.

At the backend, a traditional relational database, we maintain CVDs, consisting of information of two types: versions, along with the records they contain, as well as metadata about versions. A primary focus of this paper is to understand how best to store this information, and that will be the bulk of our exploration in the next section. In addition, the backend contains a temporary staging area consisting of all of the materialized tables that users can directly manipulate via SQL without going through ORPHEUSDB.

In brief, we now describe how these components work with each other for the basic checkout and commit commands, once the command is parsed. For checkout, the executor generates SQL queries to retrieve records from the relevant versions, which are then handled by the record manager; the executor tasks the version manager with logging the related derivation information and other metadata; the executor materializes the table containing the retrieved records in the temporary staging area; and finally the executor invokes the access controller to grant permissions to the relevant user. On commit, the executor invokes the record manager to append new records to the CVD, and the version manager to update the metadata of the newly added version, and also performs cleanup by removing the table from the staging area.

3. DATA MODELS FOR CVDS

In this section, we consider and compare various methods to represent CVDs within a backend relational database. We first consider how data contained within versions may be represented and operated on (what the record manager interacts with), followed by how the version metadata may be represented (what the version manager interacts with).

3.1 Versions and Data: The Models

We now describe how a collection of versions can be represented within a database. We consider a simple schema for a CVD, where the schema is a primary key of badgeID, age, gender, and salary.

One approach as described in the introduction is to augment the CVD's relational schema with an additional versioning attribute. For example, in Figure 1(a) the combination of badgeID 0001, age 25, gender F and salary 7500 is in two versions: v_3 and v_4. Note that even though badgeID is the primary key, it is only the primary key for *any single version and not across all versions*. Here, there are two records with badgeID 0001 that have different values for other attributes: one with (25, F, 7500) that is present in v_3 and v_4, and another with (25, F, 6500) that is present in v_1.

However, as is evident from this figure, this approach implies that we would need to duplicate each record as many times as the number of versions, leading to severe storage overhead due to redundancy, as well as inefficiency for several operations, including checkout and commit. We focus on alternative approaches that are more space efficient and discuss how these approaches can support the two most fundamental operations—commit and checkout—on a single version at a time. Considerations for multiple version checkout is similar to that for a single version; our findings generalize to that case as well.

Approach 1: The Combined Table Approach. Our first approach of representing the data and versioning information for a CVD is called the Combined Table approach. Here, as before, we augment the schema with an additional versioning attribute, but now, the versioning attribute is of type array, and is named *vlist* as shown in Figure 1(b). Specifically, for each record the *vlist* (short for version list) is the ordered list of version ids that the record is present in; this serves as an inverted index for each record in the CVD. Returning to our example of the records corresponding to badgeID 0001, of which we identified two versions—one with salary 6500, and one with salary 7500—these two versions are depicted as the first two records, along with an array corresponding to v_1 for the first version, and v_3 and v_4 for the second version.

Even though array is a non-atomic data type, it is commonly supported in most database systems, including PostgreSQL [11], DB2 [3], OracleRDBMS [9], and MySQL [6, 1]—thus ORPHEUSDB can be built with any of these systems as the back-end database. Our current implementation uses PostgreSQL; we focus on PostgreSQL for the rest of the discussion, even though similar considerations apply to the rest of the databases listed. PostgreSQL provides a number of useful functions and operators for manipulating

Command	SQL Translation with combined-table	SQL Translation with Split-by-vlist	SQL Translation with Split-by-rlist
CHECKOUT	SELECT * into T' FROM T WHERE ARRAY[v_i] <@ vlist	SELECT * into T' FROM dataTable, (SELECT rid AS rid_tmp FROM versioningTable WHERE ARRAY[v_i] <@ vlist) AS tmp WHERE rid = rid_tmp	SELECT * into T' FROM dataTable, (SELECT unnest(rlist) AS rid_tmp FROM versioningTable WHERE vid = v_i) AS tmp WHERE rid = rid_tmp
COMMIT	UPDATE T SET vlist=vlist+v_j WHERE rid in (SELECT rid FROM T')	UPDATE versioningTable SET vlist=vlist+v_j WHERE rid in (SELECT rid FROM T')	INSERT INTO versioningTable VALUES (v_j, ARRAY[SELECT rid FROM T'])

Table 1: SQL Queries for *Checkout* and *Commit* Commands with Different Data Models

arrays, including append operations, set operations, value containment operations, and sorting and counting functions.

Now, we discuss how we can support commit and checkout operations using this representation. For the combined table approach, committing a new version to the CVD is time-consuming due to the expensive appending operation to every record that is present in the new version. We illustrate using a simple example where the user first checks out version v_i into a materialized table T' and then immediately commits it back to the CVD as a new version v_j. The query translator parses user's command and generates the corresponding SQL queries for *checkout* and *commit* respectively as shown in Table 1. In the checkout statement, the containment operator 'int[] <@ int[]' returns true if the array on the left is contained within the array on the right. When checking out v_i into a materialized table T', the array containment operator 'ARRAY[v_i] <@ vlist' first examines whether v_i is contained in *vlist* for each record in CVD, then all records that satisfy that condition are added to the materialized table T'. Next, when T' is committed back to the CVD as a new version v_j, for each record in the CVD, if it is also present in T' (i.e., the WHERE clause), we append v_j to the attribute *vlist* (i.e., vlist=vlist+v_j). In this case, since there are no new records that are added to the CVD, no new records are added to the combined table. However, even this process of appending v_j to *vlist* can be expensive especially when the number of records in v_j is large, as we will demonstrate later.

Approach 2: The Split-by-vlist Approach. Our second approach is targeted at addressing the limitations of the expensive commit operation for the combined table approach. Here, we store two tables, as opposed to one, keeping the versioning information separate from the data information in the relation corresponding to the CVD, as depicted in Figure 1(c). We create two tables, *data table* and *versioning table*. The data table contains all of the original data attributes along with an extra primary key *rid*, while the versioning table maintains the mapping between versions and *rid*s. Note that the *rid* attribute was not necessary in the previous approaches since it was not necessary to associate identifiers with the immutable records. Specifically, the relation primary key—badgeID—is not sufficient to distinguish between multiple copies of the same record. For example, $r1$ and $r5$ are two versions of the same record (i.e., the record with a given badgeID). There are two ways we can store the versioning data. The first approach is to store the *rid* along with the *vlist*, as depicted in Figure 1(c.i). We call this approach *split-by-vlist*. Split-by-vlist has a similar SQL translation as the combined-table for *commit*, while it incurs the overhead of joining the data table with the versioning table for *checkout*. Specifically, we select the *rid*s that are in the version to be checked out and store it in the table tmp, followed by a join with the data table. For example, when checking out version v_1, tmp will comprise the relevant *rid*s r_1, r_2, r_3, which are identified by looking at the *vlist* for each record in the versioning table and checking if v_1 is present, which is then joined with the data table to extract the appropriate results into the materialized table T'.

Approach 3: The Split-by-rlist Approach. Alternatively, we can organize the versioning table in a different manner, with a primary key as *vid* (version id), and another attribute *rlist*, containing the array of the records present in that particular version, as shown in Figure 1(c.ii). We call this approach as the *split-by-rlist* approach. When committing a new version v_j from the materialized table T', we only need to add a single tuple in the versioning table with *vid* equal to v_j, and *rlist* equal to the list of record ids in T'. This eliminates the expensive array appending operations that are part of the previous two approaches, making the *commit* command much more efficient, as we will see later on. For the *checkout* command for version v_i, we first extract the record ids associated with v_i from the versioning table, by applying the unnesting operation: unnest(*rlist*), following which we join the *rid*s with the data table to identify all of the relevant records. For example, for checking out v_1, instead of examining the entire versioning table, we simply need to examine the tuple corresponding to v_1, unnest those *rid*s—r_1, r_2, r_3, followed by a join.

3.2 Versions and Data: The Comparison

Now, we perform an experimental evaluation between the three approaches described in the previous section, along with another approach where we store each version as a separate table on storage costs, and *commit* and *checkout* time. We call the latter approach the *a-table-per-version* approach. Compared to a-table-per-version, the approach shown in Figure 1(a) does similarly in terms of storage and *commit* times, but worse in terms of *checkout*, and as a result, we chose not to include it in our results.

In our evaluation, we use datasets SCI_1M, SCI_2M, SCI_5M and SCI_8M, each with $1M$, $2M$, $5M$ and $8M$ records respectively, described in detail in Section 5.1. For split-by-vlist, a physical primary key index is built on *rid* in both the data table and the versioning table; for split-by-rlist, a physical primary key index is built on *rid* in the data table and on *vid* in the versioning table. When calculating the total storage size, we count the index size as well. Additionally, in order to fix the size of the version that we operate on across different datasets, we add a dummy version v_i at the end of each dataset with $500K$ records, randomly selected across all versions. In our experiments, we first checkout v_i into a materialized table T' and then commit T' back into the CVD as a new version v_j. We depict the experimental results in Figure 3.

Storage. From Figure 3(a), we can see that a-table-per-version takes $10\times$ more storage than the other data models. This is because each record exists on average in 10 versions. Compared to a-table-per-version, combined-table, split-by-vlist and split-by-rlist deduplicate the common records across versions and therefore have roughly similar storage. In particular, split-by-vlist and split-by-rlist share the same data table, and thus the difference can be attributed to the difference in the size of the versioning table.

Commit. From Figure 3(b), we can see that the combined-table and split-by-vlist take much more time—multiple orders of magnitude more—than split-by-rlist for the *commit* operation. We also notice that the *commit* time when using combined-table is almost $10^4 s$ when committing a version with $500K$ records. This is mainly

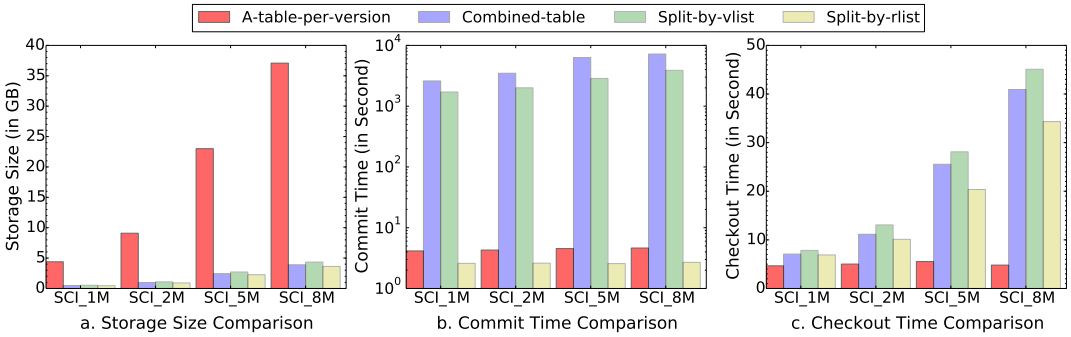


Figure 3: Comparison Between Different Data Models

because when using combined-table, we need to add v_j to the attribute *vlist* for each record in the CVD that is also present in T'. Similarly, for split-by-vlist, we need to perform an append operation for several tuples in the versioning table. On the contrary, when using split-by-rlist, we only need to add one tuple in the versioning table, thus getting rid of the expensive array appending operations, especially for versions with a large number of records. Furthermore, a-table-per-version also has higher latency for *commit* than split-by-rlist since it needs to insert $500K$ records into the CVD.

Checkout. From Figure 3 (c), we can see that split-by-rlist is a bit faster than combined-table and split-by-vlist for *checkout*. Not surprisingly, a-table-per-version is the best for this operation since it simply requires retrieving all the records in a specific table (corresponding to the desired version). We now dive into the query plan for other three data models. Combined-table requires one full scan over the combined table to check whether each record is in version v_i. On the other hand, split-by-vlist needs to first scan the versioning table to retrieve the *rid*s in version v_i, and then join the *rid*s with the data table. Lastly, split-by-rlist retrieves the *rid*s in version v_i using the primary key index on *vid* in the versioning table, and then joins the *rid*s with the data table. For both split-by-vlist and split-by-rlist, we used a hash-join, which was the most efficient[2], where a hash table on *rid*s is first built, followed by a sequential scan on the data table by probing each record in the hash table. Overall, combined-table, split-by-vlist and split-by-rlist all require a full scan on the combined table or the data table, and even though split-by-rlist introduces the overhead of building a hash table, it reduces the expensive array operation for containment checking as in combined-table and split-by-vlist.

Overall Takeaways. Overall, considering the space consumption, *commit* and *checkout* time, we claim that split-by-rlist is preferable to the other data models in supporting versioning within a relational database. Thus, we pick split-by-rlist as our data model for representing CVDs. That said, from Figure 3(c), we notice that the *checkout* time for split-by-rlist grows with dataset size. For instance, for dataset SCI_8M with $8M$ records in the data table, the *checkout* time is as high as 30 seconds. On the other hand, a-table-per-version has very low *checkout* times on all datasets. This is because it only needs to access the relevant records instead of all records as in split-by-rlist. Thus, this motivates the need for the partition optimizer module in ORPHEUSDB, which tries to attain the best of both worlds—trading off a bit more storage for reduced checkout and commit times. We will dive into the partition optimizer and its underlying algorithms in Section 4.

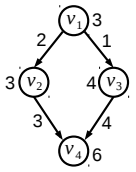
3.3 Version Derivation Metadata

[2]We also tried alternative join methods—the findings were unchanged; we will discuss this further in Section 4.1. We also tried using an additional secondary index for *vlist* for split-by-vlist which reduced the time for checkout but increased the time for commit even further.

As discussed in Section 2.3, the version manager in ORPHEUSDB keeps track of the derivation relationships among versions and maintains metadata for each version. We store version-level provenance information in a separate table called the *metadata table*. As depicted in Figure 4(a), the metadata table for the example described in Figure 1 contains attributes including version id, its parent versions, creation time, commit time, children versions (the versions derived from the present version) and a commit message. Using the data contained in this table, users can easily query for the provenance of versions and for other metadata. In addition, using the attribute *parents* in the metadata table, we can obtain each version's derivation information and visualize it using a directed acyclic graph that we call a *version graph*. Each node in the version graph is a version and each directed edge points from a version to one of its children version(s). An example is depicted in Figure 4(b), where version v_2 and v_3 are both derived from version v_1, and version v_2 and v_3 are merged into version v_4. We will return to the version graph concept in Section 4.2.

vid	parents	checkoutT	commitT	commitMsg
v_1	NULL	NULL	t_1	Initialize
v_2	$\{v_1\}$	t_2	t_3	Filter,Insert
v_3	$\{v_1\}$	t_2	t_4	Filter,Update,Insert
v_4	$\{v_2, v_3\}$	t_5	t_6	Merge

a. Metadata Table



b. Version Graph

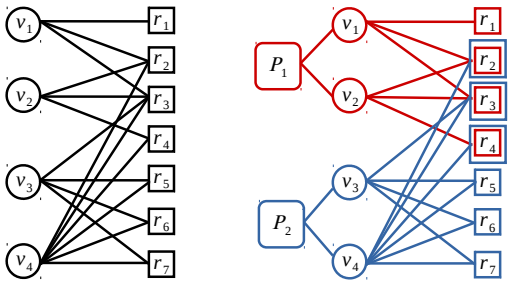
Figure 4: Metadata Table and its Corresponding Version Graph

4. PARTITION OPTIMIZER

Recall that Figure 3(c) indicated that as the number of records within a CVD increases, the checkout latency of our data model (split-by-rlist) increases—this is because the number of "irrelevant" records, i.e., the records that are not present in the version being checked out, but nevertheless require processing increases. In this section, we introduce the concept of partitioning a CVD by breaking up the data and versioning tables, in order to reduce the the number of irrelevant records during checkout. We formally define our partitioning problem, demonstrate that this problem is NP-HARD, and identify a light-weight approximation algorithm. We provide a convenient table of notation in the Appendix (Table 3).

4.1 Problem Overview

The Partitioning Notion. Let $V = \{v_1, v_2, ..., v_n\}$ be the n versions and $R = \{r_1, r_2, ..., r_m\}$ be the m records in a CVD. We can represent the presence of records in versions using a version-record bipartite graph $G = (V, R, E)$, where E is the set of bipartite edges—there is an edge between v_i and r_j if the version v_i contains the record r_j. For instance, the bipartite graph in Figure 5(a) captures the relationships between records and versions in Figure 1.



a. Bipartite Graph b. Illustration of Partitioning

Figure 5: Version-Record Bipartite Graph & Partitioning

The goal of our partitioning problem is to partition G into smaller subgraphs, each denoted as $\mathcal{P}_k$. Formally, we let $\mathcal{P}_k = (\mathcal{V}_k, \mathcal{R}_k, \mathcal{E}_k)$, where $\mathcal{V}_k$, $\mathcal{R}_k$ and $\mathcal{E}_k$ represent the set of versions, records and bipartite graph edges in partition $\mathcal{P}_k$ respectively. Note that $\cup_k \mathcal{E}_k = E$, where E is the set of edges in the original version-record bipartite graph G. We further constrain each version in the CVD to exist in only one partition, while each record can be duplicated across multiple partitions. In this manner, we only need to access one partition when checking out a version, consequently simplifying the *checkout* process by reducing the overhead from accessing multiple partitions. (While we do not consider it in this paper, in a distributed setting, it is even more important to ensure that as few partitions are consulted during a checkout operation.) Thus, our partition problem is equivalent to partitioning V, such that each partition ($\mathcal{P}_k$) stores all of the records corresponding to all of the versions assigned to that partition. Figure 5(b) illustrates a possible partitioning strategy for Figure 5(a). Partition $\mathcal{P}_1$ contains version v_1 and v_2, while partition $\mathcal{P}_2$ contains version v_3 and v_4. Note that records r_2, r_3 and r_4 are duplicated in $\mathcal{P}_1$ and $\mathcal{P}_2$, while each of the other records are present in only one partition.

Metrics. We consider two criteria while partitioning: the storage cost and the checkout cost. Recall that the cost for commit is fixed and small—see Figure 3(b), so we only focus on the cost for checkout.

Let us consider storage first. The overall storage involves the cost of storing all of the partitions of the data table, and of the versioning table. However, we observe that the versioning table simply encodes the bipartite graph, and as a result, its cost is fixed, no matter which partitioning scheme is used. Furthermore, since all of the records in the data table have the same (fixed) number of attributes, so instead of optimizing the actual storage, we simply optimize for the number of records in the data table across all the partitions. Thus, we define the *storage cost*, $\mathcal{S}$, to be the following:

$$\mathcal{S} = \sum_{k=1}^{K} |\mathcal{R}_k| \tag{4.1}$$

Next, we consider checkout. First, we note that the time taken for checking out version v_i is proportional to the size of the data table in the partition $\mathcal{P}_k$ that contains version v_i, which in turn is proportional to the number of records present in that data table partition. We theoretically and empirically justify this observation in Appendix C.1. So we define the *checkout cost of a version* v_i, $\mathcal{C}_i$, to be $\mathcal{C}_i = |\mathcal{R}_k|$, where $v_i \in \mathcal{V}_k$. Then, the *checkout cost*, denoted as $\mathcal{C}_{avg}$, which is what we optimize for, is defined to be the average of $\mathcal{C}_i$, i.e., $\mathcal{C}_{avg} = \frac{\sum_i \mathcal{C}_i}{n}$. While we focus on the average case, which assumes that each version is checked out with equal frequency—a reasonable assumption when we have no other information about the workload—our algorithms generalize to the weighted case, which we describe in Appendix B.2. On rewriting the equation regarding $\mathcal{C}_{avg}$ above, we get the following equation:

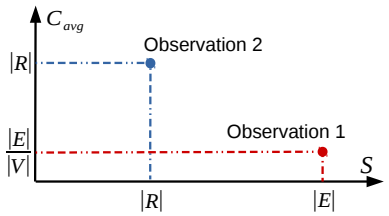


Figure 6: Extreme Partitioning Schemes

$$\mathcal{C}_{avg} = \frac{\sum_{k=1}^{K} |\mathcal{V}_k||\mathcal{R}_k|}{n} \tag{4.2}$$

The numerator is simply sum of the number of records in each partition, multiplied by the number of versions in that partition, across all partitions—this is the cost of checking out all of the versions.

Formal Problem. Note that our two metrics $\mathcal{S}$ and $\mathcal{C}_{avg}$ interfere with each other. If we want a small $\mathcal{C}_{avg}$, then we need more storage, and if we want the storage to be small, then consequently, $\mathcal{C}_{avg}$ will be large—we will discuss this again with examples in the next section. Typically, the storage is under our control and we want to optimize the checkout cost. Thus, our formal problem can be stated as the following:

PROBLEM 1 (MINIMIZE CHECKOUT COST). *Given a storage threshold γ and a version-record bipartite graph $G = (V, R, E)$, find a partitioning of G that minimizes $\mathcal{C}_{avg}$ such that $\mathcal{S} \leq \gamma$.*

We can show that the problem above is NP-HARD using a reduction from the 3-PARTITION problem, whose goal is to decide whether a given set of n integers can be partitioned into $\frac{n}{3}$ sets with equal sum. 3-PARTITION is known to be strongly NP-HARD, i.e., it is NP-HARD even when its numerical parameters are bounded by a polynomial in the length of the input.

THEOREM 1. *Problem 1 is* NP-HARD.

The proof for this theorem can be found in Appendix A.

We now clarify one complication between our formalization so far and our implementation. ORPHEUSDB uses the *no cross-version diff rule*: that is, while performing a commit operation, ORPHEUSDB does not compare the committed version against all of the ancestor versions, in order to keep that time bounded, and instead only compares the version to its parents. Therefore, if some records have been deleted and then re-added later, these records are actually identical, but would have been assigned different *rid*s, and are treated as different within the set R and the CVD. As it turns out, Problem 1 is still NP-HARD when the space of instances of version-record bipartite graphs are only those that can be conceivably generated when this rule is applied. For the rest of this section, we will use the formalization with the no cross-version diff rule in place, since that relates more closely to practice.

4.2 Partitioning Algorithm

Before we introduce our algorithm titled LYRESPLIT[3], we first describe two observations—these observations will help us formalize our algorithm's guarantees. Given a version-record bipartite graph $G = (V, R, E)$, there are two extreme cases for partitioning. At one extreme, we can minimize the checkout cost by storing each version in the CVD as one partition. In this scheme, there are in total $K = |V| = n$ partitions. The storage cost is $\mathcal{S} = \sum_{k=1}^{n} |\mathcal{R}_k| = |E|$ and the checkout cost is $\mathcal{C}_{avg} = \frac{1}{n}\sum_{k=1}^{n} (|\mathcal{V}_k||\mathcal{R}_k|) = \frac{|E|}{|V|}$. At another extreme, we can minimize the storage by storing all versions in one single partition. Then, the storage cost is $\mathcal{S} = |R|$ and $\mathcal{C}_{avg} = |R|$. We illustrate these two schemes in Figure 6, and list them as formal observations below:

[3]A lyre was the musical instrument of choice for Orpheus.

OBSERVATION 1. *Given a bipartite graph $G = (V, R, E)$, the checkout cost $\mathcal{C}_{avg}$ is minimized by storing each version as one separate partition: $\mathcal{C}_{avg} = \frac{|E|}{|V|}$.*

OBSERVATION 2. *Given a bipartite graph $G = (V, R, E)$, the storage cost $\mathcal{S}$ is minimized by storing all versions in a single partition: $\mathcal{S} = |R|$.*

Version Graph Concept. Instead of operating on the version-record bipartite graph, which may be very large when the versions contain a large number of tuples, LYRESPLIT operates on the version graph instead, which makes it a lot more lightweight. We recall the concept of a *version graph* from Section 3.3, and depicted in Figure 4. We denote a version graph as $\mathbb{G} = (\mathbb{V}, \mathbb{E})$, where each vertex $v \in \mathbb{V}$ is a version and each edge $e \in \mathbb{E}$ is a derivation relationship. Note that $\mathbb{V}$ is essentially the same as V in the version-record bipartite graph. An edge from vertex v_i to a vertex v_j indicates that v_i is a parent of v_j; this edge has a weight $w(v_i, v_j)$ equal to the number of records in common between v_i and v_j. We use $p(v_i)$ to denote the parent versions of v_i. For the special case when there are no merge operations, $|p(v_i)| \leq 1, \forall i$, and the version graph is a tree, denoted as $\mathbb{T} = (\mathbb{V}, \mathbb{E})$. Lastly, we use $R(v_i)$ to be the set of all records in version v_i, and $l(v_i)$ to be the depth of v_i in the version graph $\mathbb{G}$ in a topological sort of the graph—the root has depth 1. For example, in Figure 4, version v_2 has $|R(v_2)| = 3$ since it has three records, and is at level $l(v_2) = 2$. Further, v_2 has a single parent $p(v_2) = v_1$, and shares two records with its parent, i.e., $w(v_1, v_2) = 2$.

Next, we describe the algorithm for LYRESPLIT when the version graph is a tree (i.e., no merge operations). We then naturally extend our algorithm to other settings, as we will describe next.

The Version Tree Case. Our algorithm is based on the following lemma, which intuitively states that if every version v_i shares a large number of records with its parent version, then the checkout cost is small, and bounded by some factor of $\frac{|E|}{|V|}$, where $\frac{|E|}{|V|}$ is the lower bound on the optimal checkout cost (from Observation 1).

LEMMA 1. *Given a bipartite graph $G = (V, R, E)$, a version tree $\mathbb{T} = (\mathbb{V}, \mathbb{E})$, and a parameter $\delta \leq 1$, if the weight of every edge in $\mathbb{E}$ is larger than $\delta|R|$, then the checkout cost $\mathcal{C}_{avg}$ when all of the versions are in one single partition is less than $\frac{1}{\delta} \cdot \frac{|E|}{|V|}$.*

PROOF. Consider the nodes of the version tree $\mathbb{T}$ level-by-level, starting from the root. That is, all of a version's ancestors are considered before it is evaluated. Now, given a version v_i, the number of new records added by v_i is $R(v_i) - w(v_i, p(v_i))$. Thus, we have:

$$\begin{aligned} |R| &= |\cup_{i=1}^{|V|} R(v_i)| \\ &= R(v_1) + \sum_{l(v_i)=2} (R(v_i) - w(v_i, p(v_i))) \\ &\quad + \sum_{l(v_i)=3} (R(v_i) - w(v_i, p(v_i))) + \cdots \\ \implies |R| &= \sum_{i=1}^{|V|} R(v_i) - \sum_{i=2}^{|V|} (w(v_i, p(v_i))) \end{aligned}$$

Since each edge weight is larger than $\delta|R|$, i.e., $w(v_i, p(v_i)) > \delta|R|, \forall 2 \leq i \leq |V|$, we have:

$$|R| < |E| - \delta(|V| - 1)|R| \leq |E| - \delta|V||R| + |R|$$

where the last inequality is because $\delta \leq 1$. Thus, we have $|R| < \frac{1}{\delta} \cdot \frac{|E|}{|V|}$. Since $\mathcal{C}_{avg} = |R|$ when we have only one partition, the result follows. □

Algorithm 1: LYRESPLIT ($\mathbb{G}, |R|, |V|, |E|, \delta$)

Input : Version tree $\mathbb{G} = (\mathbb{V}, \mathbb{E})$ and parameter δ
Output : Partitions $\{\mathcal{P}_1, \mathcal{P}_2, \cdots, \mathcal{P}_K\}$

1 **if** $|R| \times |V| < \frac{|E|}{\delta}$ **then**
2 return V
3 **end**
4 **else**
5 $\Omega \leftarrow \{e | e.w \leq \delta \times |R|, e \in \mathbb{E}\}$
6 $e^* \leftarrow$ PickOneEdgeCut(Ω)
7 Remove e^* and split $\mathbb{G}$ into two parts $\{\mathbb{G}_1, \mathbb{G}_2\}$
8 Update the number of records, versions and bipartite edges in $\mathbb{G}_1$, denoted as $|R_1|, |V_1|$ and $|E_1|$
9 Update the number of records, versions and bipartite edges in $\mathbb{G}_2$, denoted as $|R_2|, |V_2|$ and $|E_2|$
10 $\mathcal{P}_1$=LYRESPLIT ($\mathbb{G}_1, |R_1|, |V_1|, |E_1|, \delta$)
11 $\mathcal{P}_2$=LYRESPLIT ($\mathbb{G}_2, |R_2|, |V_2|, |E_2|, \delta$)
12 return $\{\mathcal{P}_1, \mathcal{P}_2\}$
13 **end**

Lemma 1 indicates that when $\mathcal{C}_{avg} \geq \frac{1}{\delta} \cdot \frac{|E|}{|V|}$, there must exist some version v_j that only shares a small number of common records with its parent version v_i, i.e., $w(v_i, v_j) \leq \delta|R|$; otherwise $\mathcal{C}_{avg} < \frac{1}{\delta} \cdot \frac{|E|}{|V|}$. Intuitively, such edge (v_i, v_j) with $w(v_i, v_j) \leq \delta|R|$ is a potential edge for splitting since the overlap between v_i and v_j is small.

LYRESPLIT **Illustration.** We describe a version of LYRESPLIT that accepts as input a parameter δ, and then recursively applies partitioning until the overall $\mathcal{C}_{avg} < \frac{1}{\delta} \cdot \frac{|E|}{|V|}$; we will adapt this to Problem 1 later. The pseudocode is provided in Algorithm 1, and we illustrate its execution on an example in Figure 7.

As before, we are given a version tree $\mathbb{T} = (\mathbb{V}, \mathbb{E})$. We start with all of the versions in one partition. We first check whether $|R||V| < \frac{|E|}{\delta}$ (line 1). If yes (line 1), then we terminate; otherwise, we pick one edge e^* with weight $e^*.w \leq \delta|R|$ (lines 5–6) to cut in order to split the partition into two. According to Lemma 1, if $|R||V| \geq \frac{|E|}{\delta}$, there must exist some edge whose weight is no larger than $\delta|R|$. The algorithm does not prescribe a method for picking this edge if there are multiple such edges; the guarantees for the algorithm hold independent of this method. For instance, we can pick the edge with the smallest weight; or we can pick the edge such that after splitting, the difference in the number of versions in the two partitions is minimized. In our experiments, we use the latter, and break a tie by selecting the edge that balances the records between two partitions in addition to the number of versions.

In our example in Figure 7(a), we first find that having the entire version tree as a single partition violates the property, and we pick the red edge to split the version tree $\mathbb{T}$ into two partitions—as shown in Figure 7(b), we get one partition $\mathcal{P}_1$ with the blue nodes (versions) and another $\mathcal{P}_2$ with the red nodes (versions).

For each partition, we update the number of records, versions and bipartite edges (lines 8–9). And then we recursively call the algorithm on each partition (lines 10–11). In the example, we terminate for $\mathcal{P}_2$ but we split the edge (v_2, v_4) for $\mathcal{P}_1$, and then terminate with three partitions—Figure 7(c). We define ℓ be the recursion level number in Algorithm 1. In Figure 7 (a) (b) and (c), $\ell = 0$, $\ell = 1$ and $\ell = 2$ respectively. We will use this notation in the performance analysis next.

Now that we have an algorithm for the δ case, we can simply apply binary search on δ and obtain the best δ for Problem 1. We can show that for two δ such that one is smaller than the other, the edges cut in the former is a superset of the latter. This makes binary search feasible. We omit these details due to space limitations.

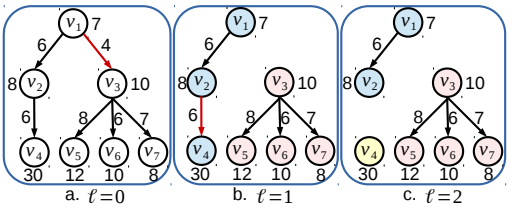


Figure 7: Illustration of Algorithm 1 ($\delta = 0.5$)

Performance Analysis. As stated in Observation 1 and 2, the lowest storage cost is $|R|$ and the lowest checkout cost is $\frac{|E|}{|V|}$ respectively. We now analyze the performance of LYRESPLIT in terms of these quantities: an algorithm has an approximation ratio of (X, Y) if its storage cost $\mathcal{S}$ is no bigger than $X \cdot R$ while its checkout cost $\mathcal{C}_{avg}$ is no bigger than $Y \cdot \frac{|E|}{|V|}$. We first study the impact of a single split edge.

LEMMA 2. *Given a bipartite graph $G = (V, R, E)$, a version tree $\mathbb{T} = (\mathbb{V}, \mathbb{E})$ and a parameter δ, let $e^* \in \mathbb{E}$ be the edge that is split in Algorithm 1, then after splitting the storage cost $\mathcal{S}$ must be within $(1+\delta)|R|$.*

PROOF. First according to Lemma 1, if $|R||V| \geq \frac{|E|}{\delta}$, there must exist some edge $e^* = (v_i, v_j)$ whose weight is less than $\delta|R|$, i.e., $e^*.w \leq \delta|R|$. Then, we remove one such e^* and split $\mathbb{G}$ into two parts $\{\mathbb{G}_1, \mathbb{G}_2\}$ as depicted in line 7-9 in Algorithm 1. The current storage cost $\mathcal{S} = |R_1| + |R_2|$. The common records between $\mathbb{G}_1$ and $\mathbb{G}_2$ is exactly the common records shared by version v_i and v_j, i.e., $e^*.w$. Thus, we have:

$$|R| = |R_1 \cup R_2| = |R_1| + |R_2| - e^*.w \geq |R_1| + |R_2| - \delta|R|$$
$$\implies \mathcal{S} = |R_1| + |R_2| \leq (1+\delta)|R|$$

Hence proved. □

Now, overall, we have:

THEOREM 2. *Given a parameter δ, Algorithm 1 results in a $((1+\delta)^{\ell}, \frac{1}{\delta})$-approximation for partitioning.*

PROOF. Let us consider all partitions when Algorithm 1 terminates at level ℓ. Each partition (e.g., Figure 7(c)) corresponds to a subgraph of the version tree (e.g., Figure 7(a)). According to Lemma 1, the total checkout cost $\mathcal{C}_k$ in each partition $\mathcal{P}_k = (\mathcal{V}_k, \mathcal{R}_k, \mathcal{E}_k)$ must be smaller than $\frac{|\mathcal{E}_k|}{\delta}$, where $|\mathcal{E}_k|$ is the number of bipartite edges in partition $\mathcal{P}_k$. Since $\sum_{k=1}^{K} |\mathcal{E}_k| = |E|$, we prove that the overall average checkout cost $\mathcal{C}_{avg}$ is $\frac{\sum \mathcal{C}_k}{|V|} < \frac{1}{\delta} \cdot \frac{|E|}{|V|}$.

Next, we consider the storage cost. The analysis is similar to the complexity analysis for quick sort. Our proof uses a reduction on the recursive level number ℓ. First, when $\ell = 0$, all versions are stored in a single partition (e.g. Figure 7(a)). Thus, the storage cost is $|R|$. Next, as the recursive algorithm proceeds, there can be multiple partitions at each recursive level ℓ. For instance, there are two partitions at level $\ell = 1$ and three partitions at level $\ell = 2$ as shown in Figure 7(b) and (c). Assume that there are τ partitions $\{\mathcal{P}_1, \mathcal{P}_2, \cdots, \mathcal{P}_\tau\}$ at level $\ell = \alpha$, and the storage cost for these partitions is no bigger than $(1+\delta)^{\alpha} \cdot |R|$. Then according to Lemma 2, for each partition $\mathcal{P}_k$ at level $\ell = \alpha$, after splitting the storage cost at level $(\alpha+1)$ will be no bigger than $(1+\delta)$ times that at level α. Thus, we have the total storage cost at level $(\alpha+1)$ must be no bigger than $(1+\delta)^{\alpha+1} \cdot |R|$. □

Complexity. At each recursive level of Algorithm 1, it takes $O(n)$ time for checking the weight of each edge in the version tree (line 5). The update in line 8–9 can also be done in $O(n)$ using one pass of tree traversal for each partition. Thus, the total time complexity is $O(n\ell)$, where ℓ is the recursive level number when Algorithm 1 terminates.

Generalizations. We can naturally extend our algorithms for the case where the version graph is a DAG: in short, we first construct a version tree $\hat{\mathbb{T}}$ based on the original version graph $\mathbb{G}$, then apply LYRESPLIT on the constructed version tree $\hat{\mathbb{T}}$. We describe the details for this algorithm in Appendix B.1.

5. PARTITIONING EVALUATION

In Section 3.2, we evaluated various data models on their storage size as well as time for commit and checkout. In this section, we focus our experimental evaluation on partitioning. We first compare LYRESPLIT with existing algorithms for partitioning in Section 5.2. Then, we evaluate the impact of partitioning itself—we run LYRESPLIT with different storage threshold constraints, and evaluate the checkout time and compare this to the checkout time without partitioning in Section 5.3.

5.1 Experimental Setup

Datasets. We evaluated the performance of LYRESPLIT using the versioning benchmark datasets from Maddox et al. [25]. The versioning model used in the benchmark is similar to git, where a branch is a working copy of a dataset. For simplicity, we can think of branches as different users' working copies. In particular, we selected the Science (SCI) and Curation (CUR) workloads since they are most representative of real-world use cases. The SCI workload simulates the working patterns of data scientists, who often take copies of an evolving dataset for isolated data analysis. The version graph here can be visualized as a mainline (i.e., a single linear version chain) with various branches at different points—both from different points on the mainline as well as from other already existing branches. Thus, the version graph is analogous to a tree with branches. The CUR workload simulates the evolution of a canonical dataset that many individuals are contributing to—these individuals not just branch from the canonical dataset but also periodically merge their changes back in. Branches can be created from existing branches, and then merged back into the parent branch. As a result, the version graph is a DAG consisting of versions checked out from and merged back to various versions.

We varied the following parameters when we generated the versioning benchmark datasets: the number of branches (denoted as $\mathbb{B}$), the total number of records $|R|$, as well as the number of inserts (or updates) from parent version(s) (denoted as $\mathbb{I}$). We list the detailed statistics of our datasets in Table 2. For instance, dataset SCI_1M represents a SCI workload dataset where the input parameter corresponding to $|R|$ in the dataset generator is set to $1M$ records. Note that due to the inherent randomness in the dataset generator, the actual number of records generated did not perfectly match the value of $|R|$ we input to the generator. Last but not least, in all of our datasets, each record contains 100 attributes, each of which is a 4-byte integer.

Dataset	$\|V\|$	$\|R\|$	$\|E\|$	$\|\mathbb{B}\|$	$\|\mathbb{I}\|$
SCI_1M	1K	944K	11M	100	1000
SCI_2M	1K	1.9M	23M	100	2000
SCI_5M	1K	4.7M	57M	100	5000
SCI_8M	1K	7.6M	91M	100	8000
SCI_10M	10K	9.8M	556M	1000	1000
CUR_1M	1.1K	966K	31M	100	1000
CUR_5M	1.1K	4.8M	157M	100	5000
CUR_10M	11K	9.7M	2.34G	1000	1000

Table 2: Dataset Description

Setup. We conducted our evaluation on a HP-Z230-SFF worksta-

tion with an Intel Xeon E3-1240 CPU and 16 GB memory running Linux OS (LinuxMint). We built ORPHEUSDB as a wrapper written in C++ over PostgreSQL 9.5[4], where we set the memory for sorting and hash operations as $1GB$ (i.e., work_mem=1GB) to reduce external memory sorts and joins. In addition, we set the buffer cache size to be minimal (i.e., shared_buffers =128KB) in PostgreSQL to eliminate the caching effect on performance. In our evaluation, for each dataset, we randomly sampled 100 versions and used that to get an estimation of the checkout time. Moreover, we performed each experiment 5 times and before each experiment we cleaned the OS page cache. Due to the experimental variance, we discarded the largest and smallest number among the five trials, and then took the average of the remaining three trials.

Algorithms. We compared LYRESPLIT against two graph partitioning algorithms in the state-of-the-art graph partitioning paper, NScale [28]: the Agglomerative Clustering-based Algorithm (Algorithm 4 in NScale) and the KMeans Clustering-based Algorithm (Algorithm 5 in NScale), denoted as AGGLO and KMEANS respectively in our paper. After mapping their setting into our context, like LYRESPLIT, NScale [28]'s algorithms group versions into different partitions (or bins) while allowing the duplication of records. However, the focus and algorithms within NScale are tailored for arbitrary graph partitioning, not for bipartite graph partitioning (like in our case). We selected AGGLO and KMEANS algorithms as our baselines since AGGLO is an intuitive method for clustering versions, while KMEANS had the best performance in NScale.

We implemented AGGLO and KMEANS as described in [28]. Specifically, AGGLO starts with each version as one partition and then sorts these partitions based on a shingle-based[5] ordering. Then, in each iteration, each partition is merged with a candidate partition that it shares the largest number of common shingles with. The candidate partitions have to satisfy two conditions (1) the number of the common shingles is larger than a threshold τ, which is set via a uniform sampling-based method, and (2) the number of records in the new partition after merging is smaller than BC, a pre-defined maximum number of records per partition. To address Problem 1 with storage threshold γ, we conduct a binary search on BC and find the best partitioning scheme under the storage constraint.

For KMEANS, there are two input parameters: partition capacity BC as in AGGLO, and the number of partitions K. Initially, K random versions are picked and assigned to K partitions, the centroid of which is initialized as the set of records in each partition. Next, we assign the remaining versions to their nearest centroid based on the number of common records, after which each centroid is updated to the union of all records in the corresponding partition. In subsequent iterations, each version is moved to a partition, such that after the movement, the total number of records across partitions is minimized, while respecting the constraint that the number of records in each partition is no larger than BC. The number of KMEANS iterations is set to 10. In our experiment, we vary K and set BC to be infinity. We tried other values for BC; the results are similar to that when BC is infinity. Overall, with the increase of K, the total storage increases and the checkout cost decreases. Again, we use binary search to find the best K for KMEANS and minimize the checkout cost under the storage constraint γ for Problem 1.

5.2 Comparison of Partitioning Algorithms

In these experiments, we consider both the datasets where the version graph is a tree, i.e., there are no merges (SCI_1M, SCI_5M and SCI_10M), and the datasets where the version graph is a DAG (CUR_1M, CUR_5M and CUR_10M). We first compare the effectiveness of different partitioning algorithms: LYRESPLIT, AGGLO and KMEANS, in balancing the storage size and the checkout time. Then, we compare the efficiency of these algorithms by measuring their running time.

Effectiveness Comparison.

Summary of Trade-off between Storage Size and Checkout Time. In all datasets, LYRESPLIT dominates AGGLO and KMEANS with respect to the storage size and checkout time after partitioning, i.e., with the same storage size, the partition scheme by LYRESPLIT provides a smaller checkout time.

In order to trade-off between $\mathcal{S}$ and $\mathcal{C}_{avg}$, we vary δ for LYRESPLIT, BC for AGGLO and K for KMEANS to obtain the overall trend between the storage size and the checkout time. The results are shown in Figure 8, where the x-axis depicts the total storage size for the data table in gigabytes (GB) and the y-axis depicts the average checkout time in seconds for the 100 randomly selected versions. Recall that for a CVD, its versioning table is of constant storage size for different partitioning schemes, so we do not include this in the storage size computation. Each point in Figure 8 represents a partitioning scheme obtained by one algorithm with a specific input parameter value. We terminated the execution of KMEANS when its running time exceeded 10 hours for each K, which is why there are only two points with star markers in Figure 8(c) and 8(f) respectively. The overall trend for AGGLO, KMEANS, and LYRESPLIT is that with the increase in storage size, the average checkout time first decreases and then tends to be some constant value. This constant equals the average checkout time when each version is stored as a separate table, which in fact corresponds to the smallest possible checkout time. For instance, in Figure 8(f) with LYRESPLIT, the checkout time decreases from 22s to 4.8s as the storage size increases from 4.5GB to 6.5GB, and then converges at around 2.9s.

Furthermore, LYRESPLIT has better performance than the other two algorithms in both the SCI and CUR datasets in terms of the storage size and the checkout time, as shown in Figure 8. For instance, in Figure 8(b), with 2.3GB storage budget, LYRESPLIT can provide a partitioning scheme taking 2.9s for checkout on average, while both KMEANS and AGGLO give schemes taking more than 7s for checkout. Thus, with equal or less storage size, the partitioning scheme selected by LYRESPLIT achieves much less checkout time than the ones proposed by AGGLO and KMEANS, especially when the storage budget is small. The reason for this is that LYRESPLIT takes a "global" perspective to partitioning, while AGGLO and KMEANS take a "local" perspective. Specifically, each split in LYRESPLIT is decided based on the derivation structure and similarity between various versions, as opposed to greedily merging partitions with partitions in AGGLO, and moving versions between partitions in KMEANS.

Efficiency Comparison.

Summary of Comparison of Running Time of Partitioning Algorithms. When minimizing the checkout time under a given storage constraint (Problem 1), LYRESPLIT is on average $10^3\times$ faster than AGGLO, and more than $10^5\times$ faster than KMEANS for all SCI_* and CUR_* datasets.

As discussed, given a storage constraint in Problem 1, we use binary search to find the best δ, BC, and K for LYRESPLIT, AGGLO and KMEANS respectively. In this experiment, we set the storage threshold as $\gamma = 2|R|$, and terminate the binary search process when the resulting storage cost $\mathcal{S}$ meets the constraint:

[4]PostgreSQL's version 9.5 added the feature of dynamically adjusting the number of buckets for hash-join.

[5]Shingles are calculated as signatures of each partition based on a min-hashing based technique.

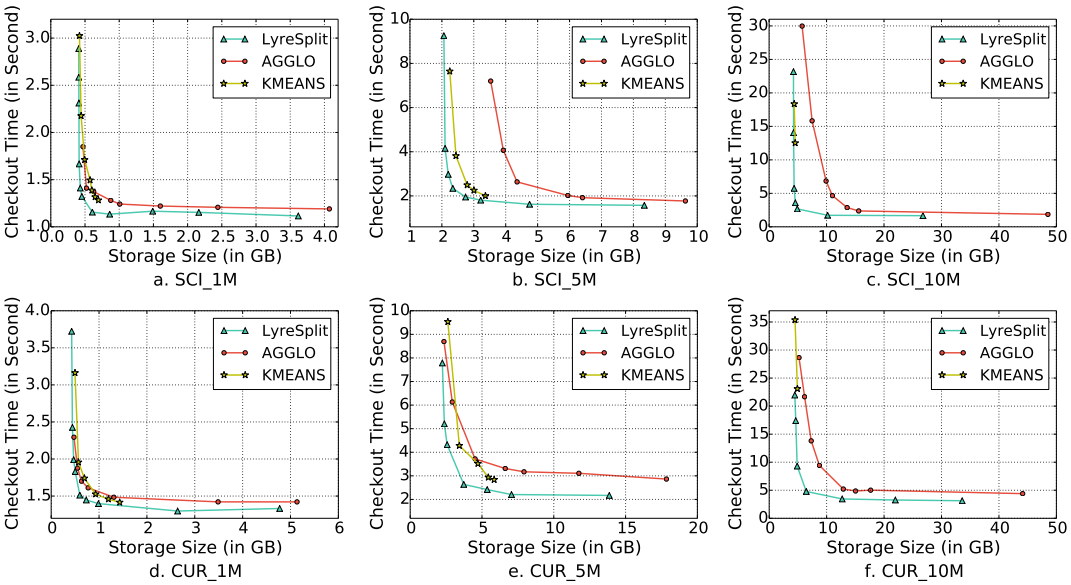


Figure 8: Storage Size vs. Checkout Time

$0.99\gamma \leq \mathcal{S} \leq \gamma$. Figure 9a and 10a report the total running time during the end-to-end binary search process, while Figure 9b and 10b display the running time per binary search iteration for different algorithms. Again, we terminate KMEANS and AGGLO when the running time exceeds 10 hours, thus we cap the running time in Figure 9 and 10 at 10 hours. We can see that LYRESPLIT takes much less time than AGGLO and KMEANS. Consider the largest dataset CUR_10M in Figure 10 as an example, with LYRESPLIT the entire binary search procedure and each binary search iteration took 0.3s and 47ms respectively; AGGLO does not terminate within 10 hours, and takes 0.6h on average for each completed iteration; while KMEANS does not even finish a single iteration in 10 hours.

In summary, LYRESPLIT is $10^2\times$, $10^3\times$, $10^4\times$ and $10^5\times$ faster than AGGLO for SCI_1M, SCI_5M (CUR_1M, CUR_5M), SCI_10M and CUR_10M respectively, and more than $10^5\times$ faster than KMEANS for all datasets. This is mainly because LYRESPLIT only needs to operate on the version graph while AGGLO and KMEANS operate on the version-record bipartite graph, which is much larger than the version graph. Furthermore, KMEANS can only finish the binary search process within 10 hours for SCI_1M and CUR_1M. This algorithm is extremely slow due to the pairwise comparison between each version with each centroid in each iteration, especially when the number of centroids K is large. Referring back to Figure 8(f), the running times for the left-most point on the KMEANS line takes 3.6h with $K = 5$, while the right-most point takes 8.8h with $K = 10$. Thus our proposed LYRESPLIT is much more scalable than AGGLO and KMEANS. Even though KMEANS is closer to LYRESPLIT in performance (as seen in the previous experiments), it is impossible to use in practice.

5.3 Benefits of Partitioning

Summary of Checkout Time Comparison with and without Partitioning: With only a $2\times$ increase on the storage, we can achieve a substantial $3\times$, $10\times$ and $21\times$ reduction on checkout time for SCI_1M, SCI_5M, and SCI_10M, and $3\times$, $7\times$ and $9\times$ reduction for CUR_1M, CUR_5M, and CUR_10M respectively.

In this section, we study the impact of partitioning and demonstrate that with a relatively small increase in storage, the checkout time can be reduced to a very small number even for large datasets. We conduct two sets of experiments with the storage threshold as $\gamma = 1.5 \times |R|$ and $\gamma = 2 \times |R|$ respectively, and compare the

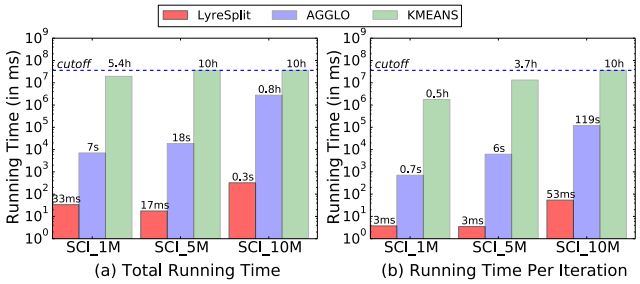


Figure 9: Algorithms' Running Time Comparison (SCI_*)

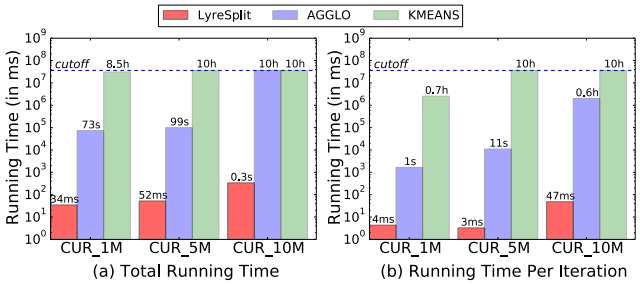


Figure 10: Algorithms' Running Time Comparison (CUR_*)

average checkout time with and without partitioning. Figure 11 illustrates the comparison on the checkout time for different datasets, and Figure 12 displays the corresponding storage size comparison. Each collection of bars in Figure 11 and Figure 12 corresponds to one dataset. Consider SCI_5M in Figure 11a and 12a as an example: the checkout time without partitioning is 16.6s while the storage size is 2.04GB; when the storage threshold is set to be $\gamma = 2 \times |R|$, the checkout time after partitioning is 1.71s and the storage size is 3.97GB. As illustrated in Figure 11a and 12a, with only $2\times$ increase in the storage size, we can achieve $3\times$ reduction on SCI_1M, $10\times$ reduction on SCI_5M, and $21\times$ reduction on SCI_10M for the average checkout time compared to that without partitioning. Thus, with partitioning, we can eliminate the time for accessing irrelevant records. Consequently, the checkout time remains small even for large datasets.

The results shown in Figure 11b and 12b are similar to those in Figure 11a and 12a: with $2\times$ increase on the storage size, we can achieve $3\times$ reduction on CUR_1M, $7\times$ reduction on CUR_5M, and $9\times$ reduction on CUR_10M for average checkout time compared to

that without partitioning. However, the reduction in Figure 11b is smaller than that in Figure 11a. The reason is the following. We can see that the checkout time without partitioning is similar for SCI and CUR datasets, but the checkout time after partitioning for CUR dataset is greater than the corresponding SCI dataset. This is because the average number of records in each version, i.e., $\frac{|E|}{|V|}$, in CUR is around 3 to 4 times greater than that in the corresponding SCI, as depicted in Table 2. Recall that $\frac{|E|}{|V|}$ is the minimal checkout cost $\mathcal{C}_{avg}$ after partitioning as stated in Observation 1. Thus, the smallest possible *checkout* time for CUR, which is where the blue lines with triangle markers (corresponding to LYRESPLIT) in Figure 8(d)(e)(f) converges to, is typically larger than that for the corresponding SCI in Figure 8(a)(b)(c). Overall, as demonstrated in Figure 11 and 12, with a small increase in the storage size, we can reduce the average *checkout* time to within a few seconds even when the number of records in a CVD increases dramatically. Referring back to our motivating experiment in Figure 3(c), we claim that with partitioning the *checkout* time using split-by-rlist is comparable to that by a-table-per-version.

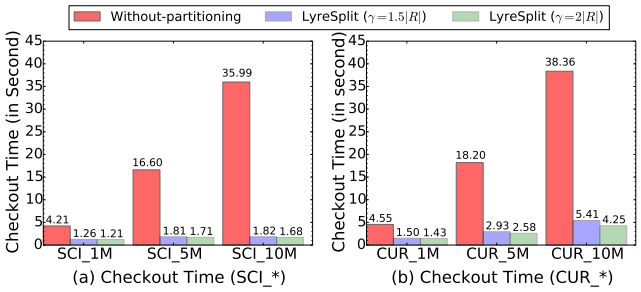


Figure 11: Checkout Time With and Without Partitioning

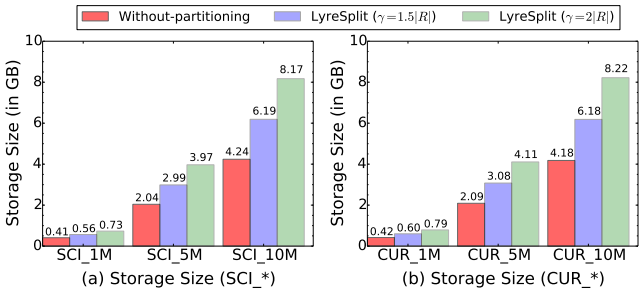


Figure 12: Storage Size With and Without Partitioning

6. RELATED WORK

We now survey work from multiple areas related to ORPHEUSDB.

Time-travel Databases. Time-travel databases support versioning for the restricted case when the version graph is a linear chain [12, 15, 29, 27, 30]. As a concrete example of an implementation, Oracle Flashback [23] provides users the ability to "roll back" the database to a previous state, or query historical information at a particular timestamp or within a certain time interval. Unfortunately, this line of work does not readily adapt to the more complex case where the version graph is a directed acyclic graph, which naturally arises as a result of collaborative data analysis.

Dataset Version Control. A recent vision paper on Datahub [13] acknowledges the need for a database system that can support collaborative analytics—we build on that vision in this paper by developing a database system with versioning capabilities. Decibel [25] describes a new version-oriented storage engine designed "from the ground up" to support versioning. Unfortunately, the architecture involves several choices that make it impossible to support within a traditional relational database without substantial changes. For example, the eventual solution requires the system to reason about and operate on "delta files", log and query tuple membership on compressed bitmaps, and execute new and fairly complex algorithms for even simple operations such as branch (in our case checkout) or merge (in our case commit). It remains to be seen how this storage engine can be made to interact with other components of the stack, such as the parser, the transaction manager, and the query optimizer. Since ORPHEUSDB is instead built on top of a traditional relational database, we inherit all of those benefits "for free". Other work considers how to best trade off storage and retrieval [14] for a setting involving unstructured data as opposed to the structured setting we consider herein. Jiang et al. [19] describe a new index for tree-oriented versioned data that could not support merges; their method would also require substantial changes to the underlying database indexing layer. Lastly, Chavan et al. [16] describe a query language for versioning and provenance, but do not develop a system that can support such a language—our system can support an important subset of this language already.

Restricted Dataset Versioning. There have been some open-source projects on versioning topics related to ORPHEUSDB. For example, LiquiBase [7] tracks schema evolution as the only applicable modifications giving rise to new versions: in our case, we focus on the data-level modifications; schema changes result in new CVDs. On the other hand, DBV [4] is focused on recording SQL operations that give rise to new versions such that these operations can be "replayed" on new datasets—thus the emphasis is on reuse of workflows rather than on efficient versioning. As other recent projects, Dat [2] can be used to share and sync local copies of dataset across machines, while Mode [8] integrates various analytics tools into a collaborative data analysis platform. However, neither of the tools are focused on providing advanced querying and versioning capabilities. Lastly, git and svn can be made to support dataset versioning, however, as recent work identified, these techniques are not efficient [25], and they do not support sophisticated querying of the type we describe herein.

Graph Partitioning. There has been a lot of work on graph partitioning [20, 24, 17, 21], with applications ranging from distributed systems and parallel computing, to search engine indexing. The state-of-the-art in this space is NScale [28], which proposes algorithms to pack subgraphs into minimum number of partitions while keeping the computation load balanced across partitions. In our setting, the versions are related to each other in very specific ways, and by exploiting these properties, our algorithms are able to beat the NScale algorithms in terms of performance, while also providing $10^3\times$ speedup. Kumar et al. [22] studies workload-aware graph partitioning by performing balanced k-way cuts on the tuple-query hypergraph for data placement and replication on the cloud; in their context, however, queries are allowed to touch multiple partitions.

7. CONCLUSIONS

We presented ORPHEUSDB, a dataset version control system that is "bolted on" a relational database, thereby seamlessly benefiting from advanced querying as well as versioning capabilities. We proposed and evaluated four data models for storing CVDs in a database. We further optimized the best data model (split-by-rlist) via the LYRESPLIT algorithm that applies intelligent but lightweight partitioning to reduce the amount of irrelevant data that is read during checkout. Our experimental results demonstrate that LYRESPLIT is $10^3\times$ faster in finding the effective partitioning scheme compare to other algorithms, and can improve the checkout performance up to $20\times$ relative to schemes without partitioning.

8. REFERENCES

[1] Add array data type in MySql 7.1. https://dev.mysql.com/worklog/task/?id=2081.

[2] Dat. http://datproject.org/.

[3] DB2 9.7 array. https://www.ibm.com/support/knowledgecenter/SSEPGG_9.7.0/com.ibm.db2.luw.sql.ref.doc/doc/r0050497.html.

[4] dbv. https://dbv.vizuina.com/.

[5] For big-data scientists, 'janitor work' is key hurdle to insights. http://www.nytimes.com/2014/08/18/technology/for-big-data-scientists-hurdle-to-insights-is-janitor-work.html?_r=0.

[6] How to store array in MySql. http://99webtools.com/blog/how-to-store-array-in-mysql/.

[7] Liquibase. http://www.liquibase.org/.

[8] Mode. https://about.modeanalytics.com/.

[9] Oracle DB array. https://docs.oracle.com/javase/tutorial/jdbc/basics/array.html.

[10] ORPHEUSDB open-source software: Details omitted for anonymity.

[11] PostgreSQL 9.5 intarray. https://www.postgresql.org/docs/current/static/intarray.html.

[12] I. Ahn and R. Snodgrass. Performance evaluation of a temporal database management system. In *ACM SIGMOD Record*, volume 15, pages 96–107. ACM, 1986.

[13] A. Bhardwaj, S. Bhattacherjee, A. Chavan, A. Deshpande, A. J. Elmore, S. Madden, and A. G. Parameswaran. Datahub: Collaborative data science & dataset version management at scale. *CIDR*, 2015.

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

[15] P. Buneman, S. Khanna, K. Tajima, and W.-C. Tan. Archiving scientific data. *ACM Transactions on Database Systems (TODS)*, 29(1):2–42, 2004.

[16] A. Chavan, S. Huang, A. Deshpande, A. Elmore, S. Madden, and A. Parameswaran. Towards a unified query language for provenance and versioning. In *7th USENIX Workshop on the Theory and Practice of Provenance (TaPP 15)*, 2015.

[17] U. Feige, D. Peleg, and G. Kortsarz. The dense k-subgraph problem. *Algorithmica*, 29(3):410–421, 2001.

[18] K. Fisher, D. Walker, K. Q. Zhu, and P. White. From dirt to shovels: fully automatic tool generation from ad hoc data. In *ACM SIGPLAN Notices*, volume 43, pages 421–434. ACM, 2008.

[19] L. Jiang, B. Salzberg, D. B. Lomet, and M. B. García. The bt-tree: A branched and temporal access method. In *VLDB*, pages 451–460, 2000.

[20] G. Karypis and V. Kumar. A fast and high quality multilevel scheme for partitioning irregular graphs. *SIAM Journal on scientific Computing*, 20(1):359–392, 1998.

[21] G. Karypis and V. Kumar. Multilevel k-way hypergraph partitioning. *VLSI design*, 11(3):285–300, 2000.

[22] K. A. Kumar, A. Quamar, A. Deshpande, and S. Khuller. Sword: workload-aware data placement and replica selection for cloud data management systems. *The VLDB Journal*, 23(6):845–870, 2014.

[23] J. W. Lee, J. Loaiza, M. J. Stewart, W.-M. Hu, and W. H. Bridge Jr. Flashback database, Feb. 20 2007. US Patent 7,181,476.

[24] D.-R. Liu and S. Shekhar. Partitioning similarity graphs: A framework for declustering problems. *Information Systems*, 21(6):475–496, 1996.

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

[26] R. J. Miller, M. A. Hernández, L. M. Haas, L.-L. Yan, C. H. Ho, R. Fagin, and L. Popa. The clio project: managing heterogeneity. *SIgMOD Record*, 30(1):78–83, 2001.

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

[28] A. Quamar, A. Deshpande, and J. Lin. Nscale: neighborhood-centric large-scale graph analytics in the cloud. *The VLDB Journal*, pages 1–26, 2014.

[29] R. Snodgrass and I. Ahn. A taxonomy of time databases. *ACM Sigmod Record*, 14(4):236–246, 1985.

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

APPENDIX

A. PROOF OF THEOREM 1

PROOF. We reduce the well known NP-HARD 3-PARTITION problem to our Problem 1. The 3-PARTITION problem is defined as follows: Given an integer set $\mathcal{A} = \{a_1, \cdots, a_n\}$ where n is divisible by 3, partition $\mathcal{A}$ into $\frac{n}{3}$ sets $\{A_1, A_2, A_j \cdots A_{\frac{n}{3}}\}$ such that for any A_j, $\sum_{a_i \in A_j} a_i = \frac{B}{n/3}$ where $B = \sum_{a_i \in \mathcal{A}} a_i$.

Symb.	Description	Symb.	Description
G	bipartite graph	E	bipartite edge set in G
V	version set in G	n	total number of versions
R	record set in G	m	total number of records
v_i	version i in V	r_j	record j in R
$\mathcal{P}_k$	k^{th} partition	$\mathcal{V}_k$	version set in $\mathcal{P}_k$
$\mathcal{R}_k$	record set in $\mathcal{P}_k$	$\mathcal{E}_k$	bipartite edge set set in $\mathcal{P}_k$
$\mathcal{S}$	total storage cost	γ	storage threshold
$\mathcal{C}_i$	checout cost for v_i	$\mathcal{C}_{avg}$	average checkout cost
$\mathbb{G}$	version graph	$\mathbb{V}$	version set in $\mathbb{G}$
$\mathbb{E}$	edge set in $\mathbb{G}$	e	$e = (v_i, v_j)$: v_i derives v_j
$\mathbb{T}$	version tree	$e.w$	# of common records on e
$l(v_i)$	level # of v_i in $\mathbb{G}$	$p(v_i)$	parent version(s) of v_i in $\mathbb{G}$
$R(v_i)$	record set in v_i	ℓ	# of recursive levels in Alg 1

Table 3: Notations

To reduce 3-PARTITION to our Problem 1, we first construct a version-record bipartite graph $G = (V, R, E)$ (Figure 13) that consists of B versions and $(B + D)$ records, where D is the number of dummy records and can be any positive integer. Specifically:

- For each integer $a_i \in \mathcal{A}$:
  - Create a_i versions $\{v_i^1, v_i^2, \cdots, v_i^{a_i}\}$ in V;
  - Create a_i records $\{r_i^1, r_i^2, \cdots, r_i^{a_i}\}$ in R;
  - Connect each v_i^j with r_i^τ in E, where $1 \leq j \leq a_i$ and $1 \leq \tau \leq a_i$. This forms a biclique between $\{v_i^1, \cdots, v_i^{a_i}\}$ and $\{r_i^1, \cdots, r_i^{a_i}\}$.
- We also create dummy records R_D and edges E_D:
  - R_D: create D dummy records $R_D = \{r_0^1, r_0^2, \cdots, r_0^D\}$ in R, where $D \geq 1$;
  - E_D: connect each dummy record with every version $v \in V$.

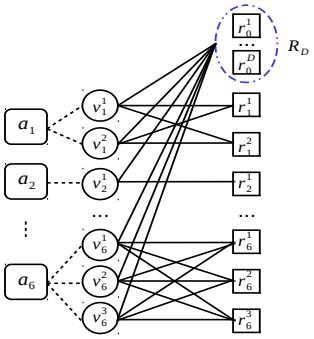


Figure 13: An Example of a Constructed Graph G

As inputs to Problem 1, we take the constructed graph G and set storage threshold $\gamma = \frac{n}{3} \cdot D + B$. We have the following two claims for the optimal solution to Problem 1:

Claim 1. For each a_i, its corresponding versions $\{v_i^1, v_i^2, \cdots, v_i^{a_i}\}$ must be in the same partition.

Claim 2. The optimal solution must have $\frac{n}{3}$ partitions, i.e, $K = \frac{n}{3}$.

We prove our first claim by contradiction. For a fixed a_i, if $\{v_i^1, v_i^2, \cdots, v_i^{a_i}\}$ are in different partitions, denoted as $P' = \{\mathcal{P}_{\tau_1}, \mathcal{P}_{\tau_2}, \cdots\}$, we can reduce the average checkout cost while maintaining the same storage cost by moving all these versions into the same partition $\mathcal{P}_{k^*} \in P'$ with the smallest $|\mathcal{R}_{k^*}|$. Furthermore, the only common records between v_i^x and v_j^y, where $i \neq j$, are the dummy records in R_D, thus only these dummy records will be duplicated across different partitions. Consequently, the total storage cost from records except the dummy record, i.e., $R \setminus R_D$, in all partitions is a constant B, regardless of the partitioning scheme.

Based on the first claim, we have $|\mathcal{R}_k| = |\mathcal{V}_k| + D, \forall k$ and our optimization objective function can be represented as follows:

$$\mathcal{C}_{avg} = \frac{1}{B}\sum_{k=1}^{K}|\mathcal{V}_k| \times (|\mathcal{V}_k| + D) = \frac{1}{B}(\sum_{k=1}^{K}|\mathcal{V}_k|^2 + B \cdot D) \quad \text{(A.1)}$$

Next, we prove the correctness of our second claim. First, we show that keeping the total storage cost $\sum_{k=1}^{K}|\mathcal{R}_k| \leq \frac{n}{3} \times D + B$ is equivalent to keeping the number of partitions $K \leq \frac{n}{3}$. From our first claim, we know that no record in $R \setminus R_D$ will be duplicated and the total number of records that corresponds to $R \setminus R_D$ in all of the partitions is B. On the other hand, each partition $\mathcal{P}_k$ must include all dummy records R_D, which is of size D. Thus, the number of partitions K must be no larger than $\frac{n}{3}$. Furthermore, we claim that the optimal solution must have $\frac{n}{3}$ partitions, i.e., $K = \frac{n}{3}$; otherwise, we can easily reduce the checkout cost by splitting any partition into multiple partitions.

Lastly, we prove that the optimal $\mathcal{C}_{avg}$ equals $B/K + D$ if and only if the decision problem to 3-PARTITION is correct. First, since $\sum_{k=1}^{K}|\mathcal{V}_k| = B$, $\mathcal{C}_{avg}$ in Equation A.1 is minimized when all $|\mathcal{V}_k| = B/K, \forall k$. Returning to the 3-PARTITION problem, if our decision to 3-PARTITION is true, then we can partition the versions in the constructed graph G accordingly and $\mathcal{C}_{avg} = B/K + D$ with each $|\mathcal{V}_k| = \frac{B}{K} = \frac{B}{n/3}$. Second, if the decision problem is false, then $\mathcal{C}_{avg}$ must be larger than $B/K + D$. Otherwise, all $|\mathcal{V}_k|$ must be the same and equal to B/K. Subsequently, we can easily partition $\mathcal{A}$ into $\frac{n}{3}$ sets with equal sum for 3-PARTITION, which contradicts the assumption that the decision problem is false. □

B. EXTENSIONS

B.1 Version graph is a DAG

When there are merges between versions, the version graph $\mathbb{G} = (\mathbb{V}, \mathbb{E})$ is a DAG. We can simply transform the $\mathbb{G}$ to a version tree $\hat{\mathbb{T}}$ and then apply LYRESPLIT as before. Specifically, for each vertex $v_i \in \mathbb{V}$, if there are multiple incoming edges, we retain the edge with the highest weight and remove all other incoming edges. In other words, for each merge operation in the version graph $\mathbb{G}$, e.g., where v_i is merged with v_j to obtain v_k, the corresponding operation in $\hat{\mathbb{T}}$ with the removed edge (v_j, v_k) is to inherit records only from one parent v_i and (conceptually) create new records in the CVD for all other records in v_k even though some records have exactly the same value as that in v_j.

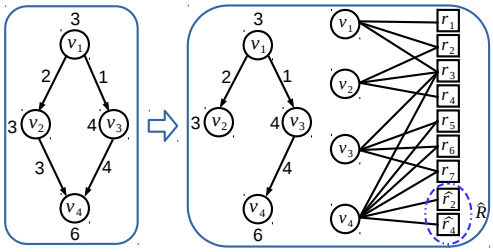


Figure 14: $\hat{\mathbb{T}}$ and $\hat{G}$ for $\mathbb{G}$ in Figure 4

For example, for the version graph $\mathbb{G}$ shown in Figure 14(a), its version v_4 has two parent versions v_2 and v_3. Since $3 = w(v_2, v_4) < w(v_3, v_4) = 4$, we remove edge (v_2, v_4) from $\mathbb{G}$ and obtain the version tree $\hat{\mathbb{T}}$ in Figure 14(b). Moreover, conceptually, we can draw a bipartite graph $\hat{G}$ corresponding to $\hat{\mathbb{T}}$ as shown in Figure 14(b) with two duplicated records, i.e., $\{\hat{r_2}, \hat{r_4}\}$. That is, v_4 in $\hat{\mathbb{T}}$ inherits 4 records from v_3 and creates two new records $\hat{R} = \{\hat{r_2}, \hat{r_4}\}$ even though $\hat{r_2}$ ($\hat{r_4}$) is exactly the same as r_2 (r_4). Thus, we have 9 records with $|\hat{R}| = 2$ and 16 bipartite edges in Figure 14(b).

Performance analysis. The number of bipartite edges in the bipartite graph $\hat{G}$ (corresponding to $\hat{\mathbb{T}}$) is the same as that in G (corresponding to $\mathbb{G}$), i.e., $|E|$. However, compared to G, the number of records in $\hat{G}$ is larger, i.e., $|R| + |\hat{R}|$, where R is the set of records in the original version-record bipartite graph G and $\hat{R}$ is the set of duplicated records. According to Theorem 2, given δ, LYRESPLIT provides a partitioning scheme with the checkout cost within $\frac{1}{\delta} \cdot \frac{|E|}{|V|}$ and the storage cost within $(1+\delta)^{\ell}(|R| + |\hat{R}|)$. Moreover, this analysis is obtained by treating $\hat{R}$ as different from R when calculating the storage cost and checkout cost. In post-processing, we can combine $\hat{R}$ with R when calculating the real storage cost and checkout cost, making the real $\mathcal{S}$ and $\mathcal{C}_{avg}$ even smaller.

B.2 Weighted Checkout Cost

In this section, we focus on the weighted checkout cost case, where versions are checked out with different frequencies.

Problem formulation. Let $\mathcal{C}_w$ denote the weighted checkout cost; say version v_i is checked out with probability or frequency f_i Then the weighted checkout cost $\mathcal{C}_w$ can be represented as $\mathcal{C}_w = \frac{\sum_{i=1}^{n}(f_i \times \mathcal{C}_i)}{\sum_{i=1}^{n} f_i}$. With this weighted checkout cost, we can modify the problem formulation for Problem 1 by simply replacing $\mathcal{C}_{avg}$ with $\mathcal{C}_w$.

Proposed Algorithm. Without the loss of generality, we assume that f_i for any version v_i is an integer. Given a version tree[6] $\mathbb{T} = (\mathbb{V}, \mathbb{E})$ and the frequency f_i for each version v_i, we construct a version tree $\mathbb{T}' = (\mathbb{V}', \mathbb{E}')$ in the following way:

- For each version $v_i \in \mathbb{V}$:
  - $\mathbb{V}'$: Create f_i versions $\{v_i^1, v_i^2, \cdots, v_i^{f_i}\}$ in $\mathbb{V}'$;
  - $\mathbb{E}'$: Connect v_i^j with v_i^{j+1} to form a chain in $\mathbb{E}'$, where $1 \leq j < f_i$
- For each edge $(v_i, v_j) \in \mathbb{E}$:
  - $\mathbb{E}'$: Connect $v_i^{f_i}$ with v_j^1 in $\mathbb{E}'$

The basic idea of constructing $\mathbb{T}'$ is to duplicate each version $v_i \in \mathbb{V}$ f_i times. Afterwards, we apply LYRESPLIT directly on $\mathbb{T}'$ to obtain the partitioning scheme. However, after partitioning, $v_i^j \in \mathbb{V}'$ with the same i may be assigned to different partitions, denoted as P'. Thus, as a post process, we move all v_i^j $(1 \leq j \leq f_i)$ into the same partition $\mathcal{P} \in P'$ that has the smallest number of records. Correspondingly, we get a partitioning scheme for $\mathbb{V}$, i.e., for each $v_i \in \mathbb{V}$, assign it to the partition where $v_i^j \in \mathbb{V}'$ $(1 \leq j \leq f_i)$ is in.

Performance analysis. At one extreme, when each version is stored in a separate table, the checkout cost $\mathcal{C}_w$ for $\mathbb{T}$ is the lowest with each $\mathcal{C}_i = |R(v_i)|$, the number of records in version v_i; thus, $\mathcal{C}_w = \frac{\sum_{i=1}^{n}(f_i \times |R(v_i)|)}{\sum_{i=1}^{n} f_i}$, denoted as ζ. At another extreme, when all versions are stored in a single partition, the total storage cost is the smallest, i.e., $|R|$. In the following, we study the performance of the extended algorithm in the weighted case, and compare the storage cost and weighted checkout cost with $|R|$ and ζ respectively.

First, consider the bipartite graph $G' = (V', R', E')$ corresponding to the constructed version tree $\mathbb{T}'$. The number of versions $|V'|$ equals $\sum_{i=1}^{n} f_i$, since there are f_i replications for each version v_i; the number of records $|R'|$ is the same as $|R|$, since there are no new records added; the number of bipartite edges $|E'|$ is $\sum_{i=1}^{n}(\sum_{j=1}^{f_i}|R(v_i^j)|) = \sum_{i=1}^{n}(f_i \times |R(v_i)|)$, since the number of records in each version v_i^j with the same i is in fact $|R(v_i)|$. Next,

[6] if the version graph is a DAG instead, we first transform it into a version tree as discussed in Appendix B.1.

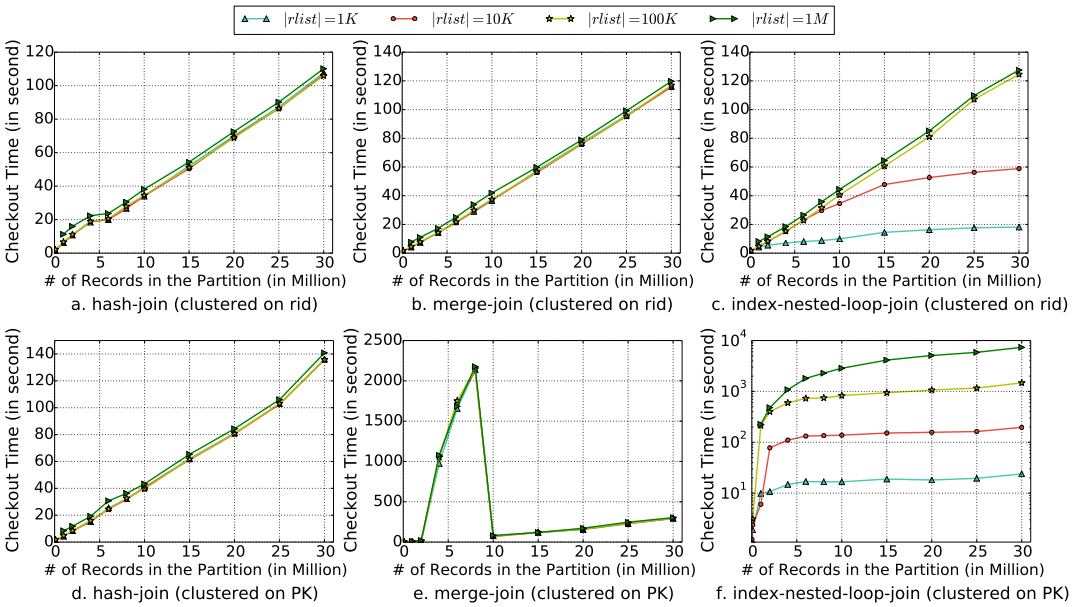


Figure 15: Checkout Cost Model Validation

based on Theorem 2, the average checkout cost after appyling Algorithm 1 is within $\frac{1}{\delta} \cdot \frac{|E'|}{|V'|} = \frac{1}{\delta} \cdot \frac{\sum_{i=1}^{n}(f_i \times |R(v_i)|)}{\sum_{i=1}^{n} f_i} = \frac{1}{\delta} \cdot \zeta$, while the storage cost is within $(1+\delta)^{\ell} \cdot |R'| = (1+\delta)^{\ell} \cdot |R|$, where ℓ is the termination level in Algorithm 1. After post processing, the total storage cost as well the average checkout cost decreases since we pick the partition with the smallest number of records for all v_i^j with a fixed i. At last, note that after mapping the partitioning scheme from $\mathbb{T}'$ to $\mathbb{T}$, the total storage cost and the average (unweighted) checkout cost for $\mathbb{T}'$ are in fact the total storage cost and the weighted checkout cost for $\mathbb{T}$ respectively. Thus, with the extended algorithm, we achieve the same approximation bound as in Theorem 2 with respect to the lowest storage cost and weighted checkout cost, i.e., in the weighted checkout case, our algorithm also results in $((1+\delta)^{\ell}, \frac{1}{\delta})$-approximation for partitioning .

C. ADDITIONAL EXPERIMENTS

C.1 Verification of Checkout Cost Model

In this section, we both analyze and experimentally evaluate the checkout cost model proposed in Section 4.1. We demonstrate that the checkout cost $\mathcal{C}_i$ of a version v_i grows linearly with the number of records in the partition $\mathcal{P}_k$ that contains v_i, i.e., $\mathcal{C}_i \propto |\mathcal{R}_k|$.

As depicted in the SQL query in Table 1, the checkout cost is impacted by the cost of two operations: (a) obtaining the list of records *rlist* associated with v_i; (b) joining data table with *rlist* to get all valid records. The cost from part (a) is a constant regardless of the partitioning scheme we use, and it is small since *rlist* can be obtained efficiently using a physical primary key index on *vid*. Thus, we focus our analysis on the cost from part (b).

We focus on three important types of join operations: hash-join, merge-join and nested-loop-join. In the following, we evaluate the checkout cost model for all these join algorithms and provide a detailed analysis. We vary the number of records in the checkout version ($|rlist|$) and the number of records in its corresponding partition ($|\mathcal{R}_k|$) in our experiments. The parameter $|\mathcal{R}_k|$ is varied from 1K to 30M and $|rlist|$ is varied from 1K to 1M, where *rlist* is a sorted list of randomly sampled *rid*s from $\mathcal{R}_k$. In addition, we have two different physical layouts for the data table, one clustered on *rid* and another clustered on its original relation primary key (PK)—badgeID in Figure 1. For each of the three join types, we compare the checkout time (in seconds) vs. the estimated checkout cost (in millions of records). Note that we build an index on *rid* in the data table, otherwise, the nested-loop-join would be very time-consuming since each outer loop requires a full scan on the inner table. The results are presented in Figure 15, where each line is plotted with a fixed $|rlist|$ (1K, 10K, 100K, and 1M respectively) and varying $|\mathcal{R}_k|$. We now describe the performance of the individual join algorithms below.

Hash-join. No matter which physical layout is used, the query plan for a hash-join based approach is to first build a hash table for *rlist* and then sequentially scan the data table with each record probing the hash table. By benefiting from the optimized implementation of the hash-join in PostgreSQL, the cost of probing each *rid* in the hash table is almost a constant. With fixed $|rlist|$, the building phase in hash-join is the same, while the running time in the probing phase is proportional to $|\mathcal{R}_k|$. Hence, as depicted in Figure 15(a) and (d), with a fixed $|rlist|$, the running time increases linearly with the growth of $|\mathcal{R}_k|$.

Merge-join. When the data table is clustered on *rid*, the query plan for a merge-join based approach is to first sort *rlist* obtained from the versioning table, then conduct an index scan using *rid* index on the data table and merge with the *rlist* from the versioning table. First, since *rlist* from the versioning table has already been sorted, quicksort can immediately terminate after the first iteration. Second, since the data table is physically clustered on *rid*, an index scan on *rid* is equivalent to a sequential scan in the data table. Thus, with fixed $|rlist|$, the running time grows linearly with the increase of $|\mathcal{R}_k|$, which is experimentally verified in Figure 15(b).

On the other hand, when the data table is clustered on the relation primary key, PostgreSQL gives different query plans for different $|\mathcal{R}_k|$. When $|\mathcal{R}_k|$ is equal to 4M, 6M and 8M, the query plan is the same as the above—sort *rlist*, conduct an index scan on *rid* and merge with *rlist*. However, since the physical layout is no longer clustered on *rid*, having an index scan on *rid* is equivalent to performing random access $|\mathcal{R}_k|$ times into the data table, which is very time-consuming as illustrated in Figure 15(e). For other $|\mathcal{R}_k|$ except 4M, 6M and 8M, the query plan is to first sort *rlist* from the versioning table, conduct a sequantial scan on the data ta-

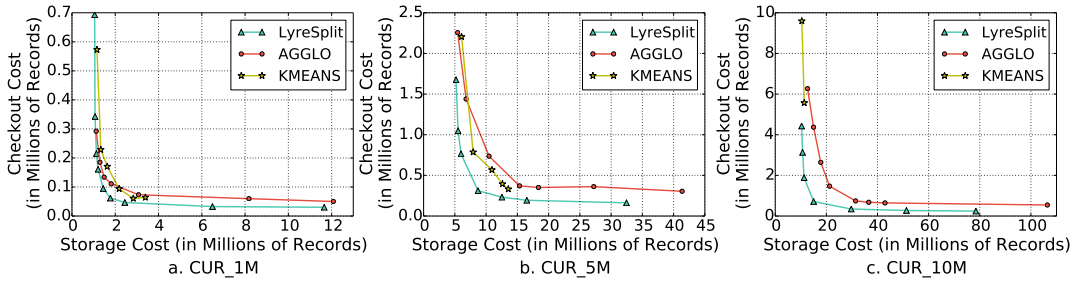


Figure 16: Estimated Storage Cost vs. Estimated Checkout Cost

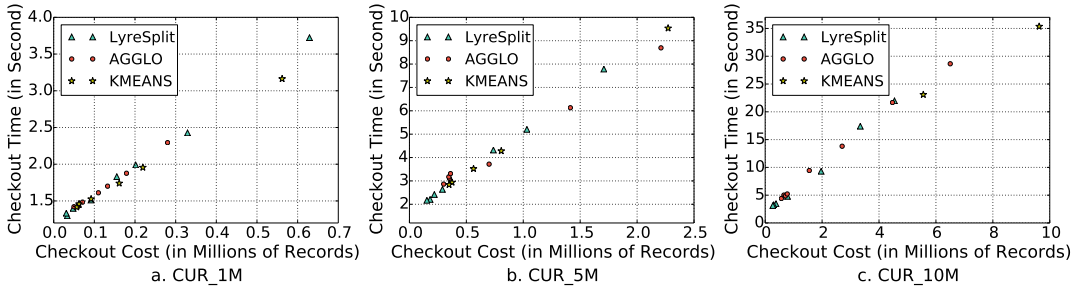


Figure 17: Estimated Checkout Cost vs. Real Checkout Time

ble, sort the *rid*s, and then finally merge *rid*s with *rlist*. Thus, with fixed $|rlist|$, the running time is proportional to $|\mathcal{R}_k|$, but greater than the hash-join based approach due to the overhead of sorting, as shown by the last five points in Figure 15(e).

Index-nested-loop-join. No matter which physical layout is used, the query plan for an index-nested-loop-join based approach is to perform a random I/O in the data table for each *rid* in *rlist* from the versioning table. Consider the scenario where $|rlist|$ is fixed and the data table is clustered on *rid*. When $|rlist|$ is much smaller than $|\mathcal{R}_k|$, the running time is almost the same since each random I/O is a constant and $|rlist|$ is fixed. This is also verified by the right portion of the blue line ($|rlist|$=1K) and red line ($|rlist|$=10K) in Figure 15(c). However, when $|rlist|$ is comparable to $|\mathcal{R}_k|$, the running time is proportional to $|\mathcal{R}_k|$ as illustrated in the green ($|rlist|$=1M) and yellow ($|rlist|$=100K) line in Figure 15(c). This is because hundreds of thousands of random I/Os are eventually reduced to a full sequential scan on the data table when $\mathcal{R}_k$ is clustered on *rid*. Returning to the checkout cost model, since partitioning algorithms tend to group similar versions together, after partitioning, $|rlist|$ is very likely to be comparable to $|\mathcal{R}_k|$ and thus the checkout time can be quantified by $|\mathcal{R}_k|$. Furthermore, the yellow line ($|rlist|$=100K) in Figure 15(c) indicates that even when $\frac{|rlist|}{|\mathcal{R}_k|} = \frac{1}{300}$, random I/Os will still be reduced to a sequential scan, consequently the running time grows linearly with $|\mathcal{R}_k|$.

However, note that when the data table is not clustered on rid, each random I/O takes almost constant time as shown in Figure 15(f). Since random I/O is more time-consuming than sequential I/O, the index-nested-loop-join performs much worse than hash-join as shown in Figure 15(d) and (f).

Overall Takeaways. When the data table is clustered on *rid*, the checkout cost can be quantified by $|\mathcal{R}_k|$ for hash-join and merge-join based approaches; while for index-nested-loop-join, the checkout cost can also be quantified by $|\mathcal{R}_k|$ when $\frac{|rlist|}{|\mathcal{R}_k|} \geq \frac{1}{300}$, which is typically the case in the partitions after partitioning. On the other hand, when the data table is not clustered on *rid*, the checkout cost for the hash-join based approach can still be quantified by $|\mathcal{R}_k|$, while the merge-join and the index-nested-loop-join based approaches perform worse than that of hash-join for most cases. Overall, a hash-join based approach has the following advantages:*(a)* the checkout time using hash-join does not rely on any index on *rid*; *(b)* hash-join based approach has good and stable performance regardless of the physical layout; *(c)* the checkout cost using hash-join is easy to model, laying foundation for further optimization on *checkout* time. Thus, throughout our paper we focus on hash-join for the *checkout* command and model the checkout cost $\mathcal{C}_i$ as linear in the number of records $|\mathcal{R}_k|$ in the partition that contains v_i.

C.2 Estimated Storage Cost and Checkout Cost

In the main body of the paper, we performed an experimental evaluation of the trade-off between the actual checkout time and the storage size in Figure 8. One question we may have is whether our cost model has a similar trade-off. Due to space limitations, we focus on the CUR dataset since they are the harder datasets; results on SCI are similar. In Figure 16, we report the estimated checkout cost versus the estimated storage cost according to our model. We can see that the trend in Figure 16 is very similar to that in Figure 8. However, the absolute reduction on checkout cost in Figure 16 is typically greater than that in Figure 8. This is because we do not count the constant overhead in the estimated checkout cost. We again check the correctness of our checkout cost model by comparing the estimated checkout cost with the actual checkout time. We present the result in Figure 17. We can see that the points in Figure 17(a)(b)(c) roughly form a straight line, validating that the checkout time is linearly correlated to our cost model. Hence, we conclude that our cost modeling and problem formulation are accurate.