ORPHEUSDB: Bolt-on Versioning for Relational Databases

Silu Huang[1], Liqi Xu[1], Jialin Liu[1], Aaron J. Elmore[2], Aditya Parameswaran[1]
[1]University of Illinois (UIUC) [2]University of Chicago
{shuang86,liqixu2,jialin2,adityagp}@illinois.edu aelmore@cs.uchicago.edu

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. 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". We develop and evaluate multiple data models for representing versioned data, as well as a light-weight partitioning scheme, LYRESPLIT, to further optimize the models for reduced query latencies. 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. LYRESPLIT can be applied in an online fashion as new versions are added, alongside an intelligent migration scheme that reduces migration time by $10\times$ on average.

1. INTRODUCTION

When performing data science, teams of data scientists repeatedly transform their datasets in many ways, by normalizing, cleaning, editing, deleting, and updating one or more data items at a time; 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 [12]. 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, performing joins across versions, or computing some aggregate statistics across versions [12]. 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, 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.

A concrete example of this phenomena occurs with biologists who operate on shared datasets, such as a protein-protein interaction dataset [41] or a gene annotation dataset [16], both of which are rapidly evolving, by periodically checking out versions, performing local analysis, editing, and cleaning operations, and committing these versions into a branched network of versions. This network of versions is also often repeatedly explored and queried for global statistics and differences (e.g., the aggregate count of protein-protein tuples with confidence in interaction greater than 0.9, for each version) and for versions with specific properties (e.g., versions with a specific gene annotation record, or versions with "a bulk delete", ones with more than 100 tuples deleted from their parents).

While recent work has outlined a vision for collaborative data analytics and versioning [12], and has developed solutions for dataset versioning from the ground up [33, 13], these papers offer partial solutions, require redesigning the entire database stack, and as such cannot benefit from the querying capabilities that exist in current database systems. Similarly, while temporal databases [42, 21, 36, 27] offer functionality to revisit instances at various time intervals on a linear chain of versions, they do not support the full-fledged branching and merging essential in a collaborative data analytics context, and the temporal functionalities offered and concerns are very different. We revisit related work in Section 6.

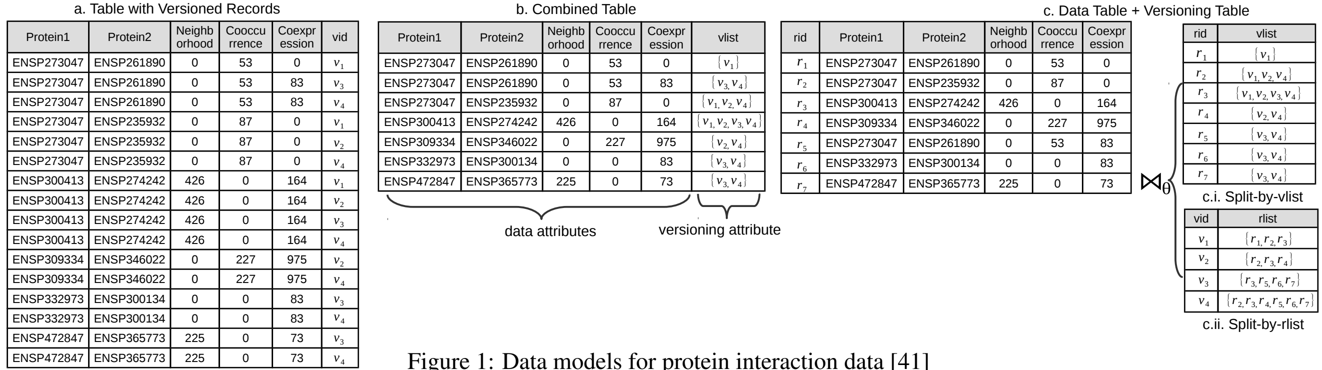
The question we ask in this paper is: *can we have the best of both worlds—advanced querying capabilities, plus effective and efficient versioning in a mature relational database*? More specifically, ***can traditional relational databases be made to support versioning for collaborative data analytics effectively and efficiently?***

To answer this question we develop a system, ORPHEUSDB[1], to *"bolt-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:

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

[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.

This work is licensed under the Creative Commons Attribution-NonCommercial-NoDerivatives 4.0 International License. To view a copy of this license, visit http://creativecommons.org/licenses/by-nc-nd/4.0/. For any use beyond those covered by this license, obtain permission by emailing info@vldb.org.
Proceedings of the VLDB Endowment, Vol. 10, No. 10
Copyright 2017 VLDB Endowment 2150-8097/17/06.



a. Table with Versioned Records

Protein1	Protein2	Neighborhood	Cooccurrence	Coexpression	vid
ENSP273047	ENSP261890	0	53	0	v_1
ENSP273047	ENSP261890	0	53	83	v_3
ENSP273047	ENSP261890	0	53	83	v_4
ENSP273047	ENSP235932	0	87	0	v_1
ENSP273047	ENSP235932	0	87	0	v_2
ENSP273047	ENSP235932	0	87	0	v_4
ENSP300413	ENSP274242	426	0	164	v_1
ENSP300413	ENSP274242	426	0	164	v_2
ENSP300413	ENSP274242	426	0	164	v_3
ENSP300413	ENSP274242	426	0	164	v_4
ENSP309334	ENSP346022	0	227	975	v_2
ENSP309334	ENSP346022	0	227	975	v_4
ENSP332973	ENSP300134	0	0	83	v_3
ENSP332973	ENSP300134	0	0	83	v_4
ENSP472847	ENSP365773	225	0	73	v_3
ENSP472847	ENSP365773	225	0	73	v_4

b. Combined Table

Protein1	Protein2	Neighborhood	Cooccurrence	Coexpression	vlist
ENSP273047	ENSP261890	0	53	0	$\{v_1\}$
ENSP273047	ENSP261890	0	53	83	$\{v_3, v_4\}$
ENSP273047	ENSP235932	0	87	0	$\{v_1, v_2, v_4\}$
ENSP300413	ENSP274242	426	0	164	$\{v_1, v_2, v_3, v_4\}$
ENSP309334	ENSP346022	0	227	975	$\{v_2, v_4\}$
ENSP332973	ENSP300134	0	0	83	$\{v_3, v_4\}$
ENSP472847	ENSP365773	225	0	73	$\{v_3, v_4\}$

c. Data Table + Versioning Table

rid	Protein1	Protein2	Neighborhood	Cooccurrence	Coexpression
r_1	ENSP273047	ENSP261890	0	53	0
r_2	ENSP273047	ENSP235932	0	87	0
r_3	ENSP300413	ENSP274242	426	0	164
r_4	ENSP309334	ENSP346022	0	227	975
r_5	ENSP273047	ENSP261890	0	53	83
r_6	ENSP332973	ENSP300134	0	0	83
r_7	ENSP472847	ENSP365773	225	0	73

$\bowtie_\theta$

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.i. Split-by-vlist

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	$\{r_2, r_3, r_4, r_5, r_6, r_7\}$

c.ii. Split-by-rlist

Figure 1: Data models for protein interaction data [41]

number, called *vid*, as shown in Figure 1(a) for simplified protein-protein interaction data [41]; the other attributes will be introduced later. The version number attribute allows us to apply selection operations to 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). To remedy this issue, one can use the array data type capabilities offered in current database systems, by replacing the version number attribute with an array attribute *vlist* containing all of the versions that each record belongs to, as depicted in Figure 1(b). This reduces storage overhead from replicating tuples. However, when adding a new version (e.g., 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 is to separate the data from the versioning information into two tables as in Figure 1(c), where the first table—the data table—stores 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 requires us to perform a join of these two tables to retrieve 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. 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, as we demonstrate in this paper, the latter approach outperforms other approaches (including those based on recording deltas) for most common operations.

Challenges in Balancing Storage and Querying Latencies. Unfortunately, the previous approach still requires a full theta join and examination of all of the data to reconstruct any given version. Our next question is if we can improve the efficiency of the aforementioned approach, at the cost of possibly additional storage. 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. To address this issue, we develop an efficient and light-weight approximation algorithm, LYRESPLIT, 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. In practice, this algorithm always leads to lower retrieval times for a given storage budget, than other schemes for partitioning, while being about $1000\times$ faster than these schemes. Further, we adapt LYRESPLIT to an online setting that incrementally maintains partitions as new versions arrive, and develop an intelligent migration approach to minimize the time taken for migration (by up to $10\times$).

Contributions. The contributions of this paper are as follows:

- We develop a dataset version control system, titled ORPHEUSDB, with the ability to effectively support both git-style version control commands and SQL-like queries. (Section 2)
- We compare different data models for representing versioned datasets and 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 4.2 and 4.1)
- We further adapt LYRESPLIT to be applicable to an online setting with new versions coming in, and develop an intelligent migration approach. (Section 4.3)
- We conduct extensive experiments using a versioning benchmark [33] and demonstrate that LYRESPLIT is on average $1000\times$ faster than competing algorithms and performs better in balancing the storage and version retrieval time. We also demonstrate that our intelligent migration scheme reduces the migration time by $10\times$ on average. (Section 5)

2. ORPHEUSDB OVERVIEW

ORPHEUSDB is a dataset version management system that 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. ORPHEUSDB has been developed as open-source software (orpheus-db.github.io). We now describe fundamental version-control concepts, followed by the ORPHEUSDB APIs, and finally, the design of ORPHEUSDB.

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 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. 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: 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 no two records can have the same values for the primary key attribute(s). 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 on 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. In our companion demo paper, we also describe an interactive user interface depicting the version graph, for users to easily explore and operate on dataset versions [44]. 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. 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 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 [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 *vid*s 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, 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 use 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]
```

The command does not need to specify the intended CVD since ORPHEUSDB internally maintains 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. 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 derived from, we alter the CVD to incorporate the new schema; we discuss this in Section 3.3.

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: -f for csv instead of -t. 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. 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) diff:* a standard differencing operation that compares two versions and outputs the records in one but not the other. *(b) init:* initialize either an external csv file or a database table as a new CVD in ORPHEUSDB. *(c) create_user, config, whoami:* allows users to register, login, and view the current user name. *(d) ls, drop*: list all the CVDs or drop a particular CVD. *(e) optimize:* as we will see later, ORPHEUSDB can benefit from intelligent incremental partitioning schemes (enabling operations to process much less data). Users can setup the corresponding parameters (e.g., storage threshold, tolerance factor, described later) via the command line; the ORPHEUSDB backend will periodically invoke the partitioning optimizer to improve the versioning performance.

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 statement as a string. Instead of 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. The details of the query syntax, translation, as well as examples can be found in our companion demo paper [44].

2.3 System Architecture

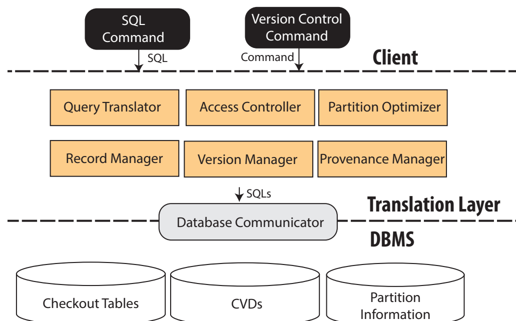


Figure 2: ORPHEUSDB Architecture

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. ORPHEUSDB consists of six core modules: the *query translator* is responsible for parsing input and translating it into SQL statements understandable by the underlying database system; the *access controller* monitors user permissions to various tables and files within ORPHEUSDB; the *partition optimizer* is responsible for periodically reorganizing and optimizing the partitions via a partitioning algorithm LYRESPLIT along with a *migration engine* to migrate data from one partitioning scheme to another, and is the focus of Section 4; the *record manager* is in charge of recording and retrieving information about records in CVDs; the *version manager* is in charge of recording and retrieving versioning information, including the *rid*s each version contains as well as the metadata for each version; and the *provenance manager* is responsible for the metadata of uncommitted tables or files, such as their parent version(s) and the creation time. At the backend, a traditional DBMS, we maintain CVDs that consist of versions, along with the records they contain, as well as metadata about versions. In addition, the underlying DBMS contains a temporary staging area consisting of all of the materialized tables that users can directly manipulate via SQL without going through ORPHEUSDB. Understanding how to best represent and operate on these CVDs within the underlying DBMS is an important challenge—this is the focus of the next section.

3. DATA MODELS FOR CVDS

In this section, we consider and compare methods to represent and operate on CVDs within a backend relational database, starting with the data within versions, and then the metadata about versions.

3.1 Versions and Data: The Models

To explore alternative storage models, we consider the array-based data models, shown in Figure 1, and compare them to a delta-based data model, which we describe later. The table(s) in Figure 1 displays simplified protein-protein interaction data [41], and has a composite primary key <*protein1, protein2*>, along with numerical attributes indicating sources and strength of interactions: *neighborhood* represents how frequently the two proteins occur close to each other in runs of genes, *cooccurrence* reflects how often the two proteins co-occur in the species, and *coexpression* refers to the level to which genes are co-expressed in the species.

One approach, as described in the introduction, is to augment the CVD's relational schema with an additional versioning attribute. For example, the tuple of <ENSP273047, ENSP261890, 0, 53, 83> in Figure 1(a) exists in two versions: v_3 and v_4. (Note that even though <protein1, protein2> is the primary key, it is only the primary key for any single version and not across all versions.) However, this approach implies that we would need to duplicate each record as many times as the number of versions it is in, 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 they 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 the *combined table approach*. As before, we augment the schema with an additional versioning attribute, but now, the versioning attribute is of type array and is named *vlist* (short for version list) as shown in Figure 1(b). For each record the *vlist* is the ordered list of version ids that the record is present in, which serves as an inverted index for each record. Returning to our example, there are two versions of records corresponding to <ENSP273047, ENSP261890>, with coexpression 0 and 83 respectively—these two versions are depicted as the first two records, with an array corresponding to v_1 for the first record, and v_3 and v_4 for the second.

Even though array is a non-atomic data type, it is commonly supported in many database systems [8, 3, 1]; thus ORPHEUSDB can be built with any of these systems as the back-end database. As our implementation uses PostgreSQL, we focus on this system for the rest of the discussion, even though similar considerations apply to the rest of the databases listed.

For the combined table approach, committing a new version to the CVD is time-consuming due to the expensive append operation for every record present in the new version. Consider the scenario where the user checks out version v_i into a materialized table T' and then immediately commits it back as a new version v_j. The query translator parses the user commands and generates the corresponding SQL queries for checkout and commit as shown in Table 1. 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.

Approach 2: The Split-by-vlist Approach. Our second approach addresses the limitations of the expensive commit operation for the combined table approach. We store two tables, keeping the versioning information separate from the data information, as depicted in Figure 1(c)—the *data table* and the *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. The *rid* attribute was not needed in the previous approach since it was not necessary to associate identifiers with the immutable records. 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 combined-table for commit, while it incurs the overhead of joining the data table with the versioning table for checkout. Specifically, we select

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

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'])

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.

Approach 3: The Split-by-rlist Approach. Alternatively, we can organize the versioning table with a primary key as *vid* (version id), and another attribute *rlist*, containing the array of the records present in that particular version, as in Figure 1(c.ii). We call this approach 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. 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.

So far, all our models support convenient rewriting of arbitrary and complex versioning queries into SQL queries understood by the backend database; see details in our demo paper [44]. However, our delta-based model, discussed next, does not support convenient rewritings for some of the more advanced queries, e.g., "find versions where the total count of tuples with *protein1* as ENSP273047 is greater than 50": in such cases, delta-based model essentially needs to recreate all of the versions, and/or perform extensive and expensive computation outside of the database. Thus, even though this model does not support advanced analytics capabilities "for free", we include it in our comparison to contrast its performance to the array-based models.

Approach 4: Delta-based Approach. Here, each version records the modifications (or deltas) from its precedent version(s). Specifically, each version is stored as a separate table, with an added tombstone boolean attribute indicating the deletion of a record. In addition, we maintain a precedent metadata table with a primary key *vid* and an attribute *base* indicating from which version *vid* stores the delta. When committing a new version v_j, a new table stores the delta from its previous version v_i. If v_j has multiple parents, we will store v_j as the modification from the parent that shares the largest common number of records with v_j. (Storing deltas from multiple parents would make reconstruction of a version complicated, since we would need to trace back multiple paths in the version graph. Here, we opt for the simpler solution.) A new record is then inserted into the metadata table, with *vid* as v_j and *base* as v_i. For the *checkout* command for version v_i, we trace the version lineage (via the *base* attribute) all the way back to the root. If an incoming record has occurred before, it is discarded; otherwise, if it is marked as "insert", we insert it into the checkout table T'.

Approach 5: The A-Table-Per-Version Approach. Our final array-based data model is impractical due to excessive storage, but is useful from a comparison standpoint. In this approach, we store each version as a separate table. We include a-table-per-version in our comparison; we do not include the approach in Figure 1a, containing a table with duplicated records, since it would do similarly in terms of storage and commit times to a-table-per-version, but worse in terms of checkout times.

3.2 Versions and Data: The Comparison

We perform an experimental evaluation between the approaches described in the previous section on storage size, and commit and checkout time. We focus on the commit and checkout times since they are the primitive versioning operations on which the other more complex operations and queries are built on. It is important that these operations are efficient, because data scientists checkout a version to start working on it immediately, and often commit a version to have their changes visible to other data scientists who may be waiting for them.

In our evaluation, we use four versioning benchmark datasets SCI_1M, SCI_2M, SCI_5M and SCI_8M, each with $1M$, $2M$, $5M$ and $8M$ records respectively, that will be 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. Our experiment involves first checking out the latest version v_i into a materialized table T' and then committing 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 and 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. For the delta-based approach, the storage size is similar to or even slightly smaller than split-by-vlist and split-by-rlist. This is because our versioning benchmark contains only a few deleted tuples (opting instead for updates or inserts); in other cases, where deleted tuples are more prevalent, the storage in the delta-based approach is worse than split-by-vlist/rlist, since the deleted records will be repeated.

Commit. From Figure 3(b), we can see that the combined-table and split-by-vlist take multiple orders of magnitude more time than split-by-rlist for commit. We also notice that the commit time when using combined-table is almost $10^4 s$ as the dataset size increases: 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 to the versioning table, thus getting rid of the expensive array appending operations. A-table-per-version also has higher latency for commit than split-by-rlist since it needs to insert all the records in T' into the CVD. For the delta-based approach, the commit time is small since the new version v_j is exactly the same as its precedent version v_i. It only needs to update the precedent metadata table, and create a new empty table. The commit time of the delta-based approach is not small in general when there are extensive modifications to T', as illustrated by other experiments (not displayed); For instance, for a committed

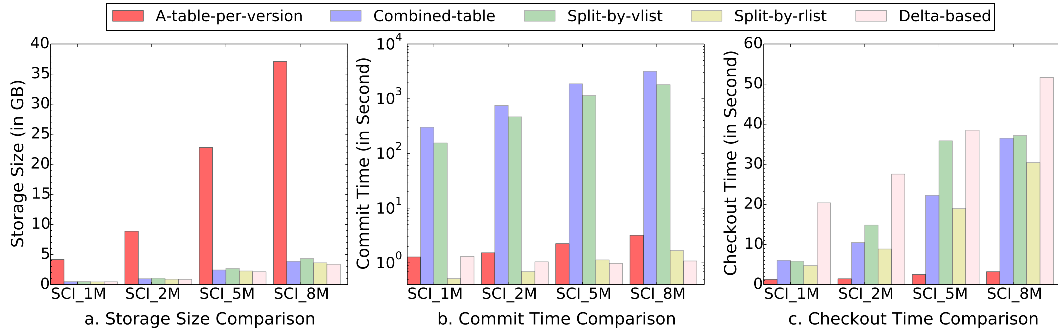


Figure 3: Comparison Between Different Data Models

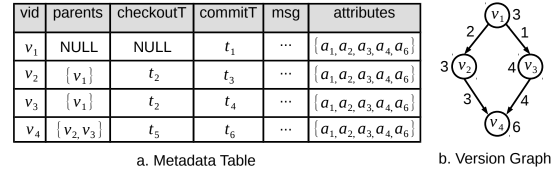
version with 250K records of which 30% of the records are modified, delta-based takes 8.16s, while split-by-rlist takes 4.12s.

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). 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. For the delta-based approach, the checkout time is large since it needs to probe into a number of tables, tracing all the way back to the root, remembering which records were seen.

Takeaways. Overall, considering the space consumption, the commit and checkout time, plus the fact that delta-based models are inefficient in supporting advanced queries as discussed in Section 3.1, 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; it only needs to access the relevant records instead of all records as in split-by-rlist. This motivates the need for the partition optimizer module in ORPHEUSDB, which tries to attain the best of both worlds by adopting a hybrid representation of split-by-rlist and a-table-per-version, described in Section 4.

3.3 Version Derivation Metadata

Version Provenance. 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*; Figure 4(a) depicts the metadata table for the example in Figure 1. It contains attributes including version id, parent/child versions, creation time, commit time, a commit message, and an array of attributes present in the version. 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* we can obtain each version's derivation information and visualize it as 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 merged into version v_4.



vid	parents	checkoutT	commitT	msg	attributes
v_1	NULL	NULL	t_1	···	$\{a_1, a_2, a_3, a_4, a_6\}$
v_2	$\{v_1\}$	t_2	t_3	···	$\{a_1, a_2, a_3, a_4, a_6\}$
v_3	$\{v_1\}$	t_2	t_4	···	$\{a_1, a_2, a_3, a_4, a_6\}$
v_4	$\{v_2, v_3\}$	t_5	t_6	···	$\{a_1, a_2, a_3, a_4, a_6\}$

a. Metadata Table

b. Version Graph

Figure 4: Metadata Table and Version Graph (Fixed Schema)

Schema Changes. During a commit, if the schema of the table being committed is different from the schema of the CVD it was derived from, we update the schema of CVD to incorporate the changes. We describe details in our technical report [9]. All of our array-based models can adapt to changes in the set of attributes: a simple solution for new attributes is so use the ALTER command to add any new attributes to the model, assigning NULLs to the records from the previous versions that do not possess these new attributes. Attribute deletions only require an update in the version metadata table. This simple mechanism is similar to the single pool method proposed in a temporal schema versioning context by De Castro et al. [17]. Compared to the multi pool method where any schema change results in the new version being stored separately, the single pool method has fewer records with duplicated attributes and therefore has less storage consumption overall. Even though ALTER TABLE is indeed a costly operation, due to the partitioning schemes we describe later, we only need to ALTER a smaller partition of the CVD rather than a giant CVD, and consequently the cost of an ALTER operation is substantially mitigated. In our technical report, we describe how our partitioning schemes (described next) can adapt to the single pool mechanism with comparable guarantees; for ease of exposition, for the rest of this paper, we focus on the static schema case, which is still important and challenging. There has been some work on developing schema versioning schemes [18, 35, 34] and we plan to explore these and other schema evolution mechanisms (including hybrid single/multi-pool methods) as future work.

4. PARTITION OPTIMIZER

In this section, we introduce the concept of partitioning a CVD by breaking up the data and versioning tables, in order to reduce the number of irrelevant records during checkout. All of our proofs can be found in our technical report [9].

[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.

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 edges—an edge between v_i and r_j exists if the version v_i contains the record r_j. The bipartite graph in Figure 5(a) captures the relationships between records and versions in Figure 1.

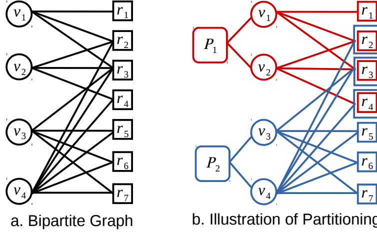


Figure 5: Version-Record Bipartite Graph & Partitioning

The goal of our partitioning problem is to partition G into smaller subgraphs, denoted as $\mathcal{P}_k$. 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. 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 $\mathcal{P}_2$.

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

The overall storage costs involves the cost of storing all of the partitions of the data and the versioning table. However, we observe that the versioning table simply encodes the bipartite graph, and as a result, its cost is fixed. 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 will 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 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 in our technical report [9]. So we define the *checkout cost of a version* v_i, $\mathcal{C}_i$, to be $\mathcal{C}_i = |\mathcal{R}_k|$, where $v_i \in \mathcal{V}_k$. The *checkout cost*, denoted as $\mathcal{C}_{avg}$, 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, our algorithms generalize to the *weighted* case [9]. On rewriting the expression for $\mathcal{C}_{avg}$ above, we get:

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

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, equivalent to $\sum_{i=1}^{n} \mathcal{C}_i$.

Formal Problem. 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 $\mathcal{C}_{avg}$ will be large. Typically, storage is under our control; thus, our problem can be stated as:

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 Problem 1 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.

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

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, to minimize computation, ORPHEUSDB does not compare the committed version against all of the ancestor versions, instead only comparing it to its parents. Therefore, if some records are deleted and then re-added later, these records would be assigned different *rid*s, and are treated as different. As it turns out, Problem 1 is still NP-HARD when the space of instances are those that can be 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

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; there are in total $K = |V| = n$ partitions, and 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; the storage cost is $\mathcal{S} = |R|$ and $\mathcal{C}_{avg} = |R|$. We illustrate these schemes in Figure 6.

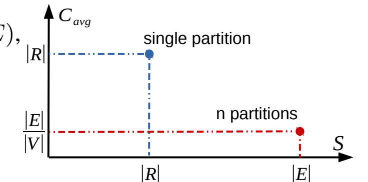


Figure 6: Extreme Schemes

Version Graph Concept. Our goal in designing our partitioning algorithm, LYRESPLIT[3], is to trade-off between these two extremes. Instead of operating on the version-record bipartite graph, which may be very large, LYRESPLIT operates on the much smaller version graph instead, which makes it a lot more lightweight. 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 $w(v_i, v_j)$ equals 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[4] of the graph—the root has depth 1. For example, in Figure 4, version v_2 has $|R(v_2)| = 3$ since it has

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

[4] In each iteration r, topological sorting algorithm finds vertices V' with in-degree equals 0, removes V', and updates in-degree of other vertices. $l(v_i) = r, \forall v_i \in V'$.

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).

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.

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 $\frac{1}{\delta} \cdot \frac{|E|}{|V|}$.*

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 an 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 the technical report [9], 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 $|R||V| < \frac{|E|}{\delta}$. If yes, then we terminate; otherwise, we pick one edge e^* with weight $e^*.w \leq \delta|R|$ 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; the guarantees hold independent of this method. For instance, we can pick the edge with the smallest weight; or the one such that after splitting, the difference in the number of versions in the two partitions is minimized. In our experiments, we use the latter. In our example in Figure 7(a), 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).

After each edge split, we update the number of records, versions and bipartite edges, and then we recursively call the algorithm on each partition. 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 Figure 7 (a) (b) and (c), $\ell = 0$, $\ell = 1$ and $\ell = 2$ respectively. We will use this notation in the performance analysis next.

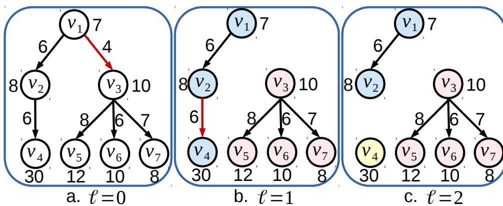


Figure 7: Illustration of LyreSplit ($\delta = 0.5$)

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.

Performance Analysis. Overall, the lowest storage cost is $|R|$ and the lowest checkout cost is $\frac{|E|}{|V|}$ respectively. We now analyze the performance in terms of these quantities: an algorithm has an approximation ratio of (X, Y) if its storage cost $\mathcal{S}$ is no larger than $X \cdot R$ while its checkout cost $\mathcal{C}_{avg}$ is no larger 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* LyreSplit*, then after splitting the storage cost $\mathcal{S}$ must be within $(1 + \delta)|R|$.*

Now, overall, we have:

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

Complexity. The total time complexity is $O(n\ell)$, where ℓ is the recursion level number when the algorithm 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 our technical report [9].

4.3 Incremental Partitioning

LyreSplit can be explicitly invoked by users or by OrpheusDB when there is a need to improve performance or a lull in activity. We now describe how the partitioning identified by LyreSplit is incrementally maintained during the course of normal operation, and how we reduce the migration time when LyreSplit identifies a new partitioning. We only describe the high-level ideas here; details and guarantees can be found in the technical report [9].

Online Maintenance. When a new version v_i is committed, OrpheusDB applies the same intuition as LyreSplit to determine whether to add v_i to an existing partition, or to create a new partition: if v_i shares a large number of records in common with one of its parent versions v_j, then v_i is added to the partition $\mathcal{P}_k$ that parent is in, or else a new partition is created. Specifically, if $w(v_i, v_j) \leq \delta^*|R|$ and $\mathcal{S} < \gamma$, where δ^* was the splitting parameter used during the last invocation of LyreSplit, then we create a new version. This way, the added storage cost is minimized, and the added checkout cost is guaranteed to be small, as in Lemma 1.

Even with the proposed online maintenance scheme, the checkout cost tends to diverge from the best checkout cost that LyreSplit can identify under the current constraints. This is because LyreSplit performs global partitioning using the full version graph as input, while online maintenance makes small changes to the existing partitioning. To maintain the checkout performance, OrpheusDB allows for a tolerance factor μ on the current checkout cost (users can also set μ explicitly). We let $\mathcal{C}_{avg}$ and $\mathcal{C}^*_{avg}$ be the current checkout cost and the best checkout cost identified by LyreSplit respectively. If $\mathcal{C}_{avg} > \mu\mathcal{C}^*_{avg}$, the *migration engine* is triggered, and we reorganize the partitions by migrating data from the old partitions to the new ones; until then, we perform online maintenance. In general, when μ is small, the migration engine is invoked more frequently. Next, we discuss how migration is performed.

Migration Approach. Given the existing partitioning $P = \{\mathcal{P}_1, \mathcal{P}_2, \ldots, \mathcal{P}_\alpha\}$ and the new partitioning $P' = \{\mathcal{P}'_1, \mathcal{P}'_2, ..., \mathcal{P}'_\beta\}$ identified by LyreSplit, we need an algorithm to efficiently migrate the data from P to P' without dropping all existing tables and recreating the partitions from scratch, which could be very costly. To do so, OrpheusDB needs to identify, for every $\mathcal{P}'_i \in P'$, the closest partition $\mathcal{P}_j \in P$, in terms of modification cost, defined as $|\mathcal{R}'_i \setminus \mathcal{R}_j| + |\mathcal{R}_j \setminus \mathcal{R}'_i|$, where $\mathcal{R}'_i \setminus \mathcal{R}_j$ and $\mathcal{R}_j \setminus \mathcal{R}'_i$ are the records needed to be inserted and deleted respectively to transform $\mathcal{P}_j$ to $\mathcal{P}'_i$. Since this is expensive to calculate, OrpheusDB instead approximates this quantity by using the number of versions

in common between $\mathcal{P}_i$ and $\mathcal{P}'_i$, along with operating on the version graph to identify common records, and then greedily identifying the "closest" partitions.

5. PARTITIONING EVALUATION

While Section 3.2 explores the performance of data models, this section evaluates the impact of partitioning. In Section 5.2 we evaluate if LYRESPLIT can be more efficient than existing partitioning techniques; in Section 5.3, we ask whether versioned databases strongly benefit from partitioning; and lastly, in Section 5.4 we evaluate how LYRESPLIT performs for online scenarios.

5.1 Experimental Setup

Datasets. We evaluated the performance of LYRESPLIT using the versioning benchmark datasets from Maddox et al. [33]; see details in [9]. The versioning model used in the benchmark is similar to git, where a branch is a working copy of a dataset. 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. 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, resulting in a DAG of versions. We varied the following parameters when we generated the benchmark datasets: the number of branches $\mathbb{B}$, the total number of records $|R|$, as well as the number of inserts (or updates) from parent version(s) $\mathbb{I}$. We list our configurations 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. In all of our datasets, each record contains 100 attributes, each of which is a 4-byte integer.

Table 2: Dataset Description

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

Setup. We conducted our evaluation on a HP-Z230-SFF workstation 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, where we set the memory for sorting and hash operations as $1GB$. In addition, we set the buffer cache size to be minimal to eliminate the effects of caching on performance. In our evaluation, for each dataset, we randomly sampled 100 versions and used them to get an estimate of the checkout time. Each experiment was repeated 5 times, with the OS page cache being cleaned before each run. Due to 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 compare LYRESPLIT against two partitioning algorithms in the NScale graph partitioning project [37]: the Agglomerative Clustering-based one (Algorithm 4 in [37]) and the KMeans Clustering-based one (Algorithm 5 in [37]), denoted as AGGLO and KMEANS respectively: KMEANS had the best performance, while AGGLO is an intuitive method for clustering versions. After mapping their setting into ours, like LYRESPLIT, NScale [37]'s algorithms group versions into different partitions while allowing the duplication of records. However, the NScale algorithms are tailored for arbitrary graphs, not for bipartite graphs (as in our case).

We implement AGGLO and KMEANS as described. AGGLO starts with each version as one partition and then sorts these partitions based on a shingle-based 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 a constraint BC. 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 assigned to partitions. 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 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 cost 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 datasets where the version graph is a tree, i.e., there are no merges (SCI_5M and SCI_10M), and datasets where the version graph is a DAG (CUR_5M and CUR_10M). Experiments on additional datasets can be found in [9]. 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.

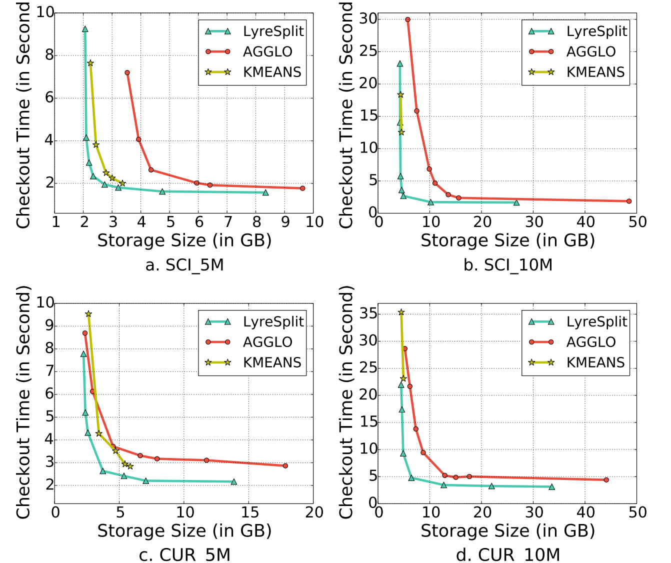


Figure 8: Storage Size vs. Checkout Time

Effectiveness Comparison.

> *Summary of Trade-off between Storage Size and Checkout Time.* LYRESPLIT dominates AGGLO and KMEANS with respect to the storage size and checkout time after partitioning, i.e., with the same storage size, LYRESPLIT's partitioning scheme 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. 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(b) and 8(d) 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 a constant value—the average checkout time when each version is stored as a separate table.

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(a), 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. Thus, with equal or lesser 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 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.

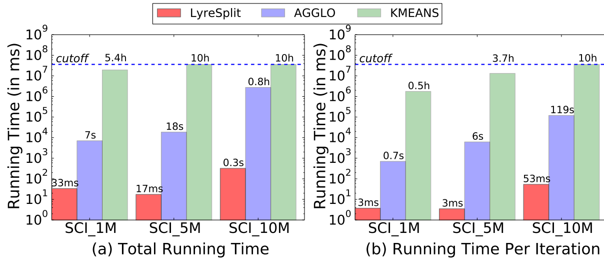


Figure 9: Algorithms' Running Time Comparison (SCI_*)

Efficiency Comparison.

Summary of Comparison of Running Time of Partitioning Algorithms. When minimizing the checkout time under a 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.

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: $0.99\gamma \leq \mathcal{S} \leq \gamma$. We discuss the results for the SCI datasets: the CUR dataset performance is similar [9]. Figure 9a shows the total running time during the end-to-end binary search process, while Figure 9b shows the running time per binary search iteration. Again, we terminate Kmeans and Agglo when the running time exceeds 10 hours. Consider the largest dataset SCI_10M in Figure 9 as an example: with LyreSplit the entire binary search procedure and each binary search iteration took 0.3s and 53ms respectively; Agglo takes 50 minutes in total; while Kmeans does not even finish a single iteration in 10 hours.

Overall, LyreSplit is $10^2\times$ faster than Agglo for SCI_1M, $10^3\times$ faster for SCI_5M, and $10^4\times$ faster for SCI_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. Thus our proposed LyreSplit is much more scalable than Agglo and Kmeans. Even if 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 respectively.

We now study the impact of partitioning and demonstrate that with a relatively small increase in storage, the checkout time can be substantially reduced. 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 average checkout time with and without partitioning. Figure 10(a) illustrates the comparison on the checkout time for different datasets, and Figure 10(b) displays the corresponding storage size comparison. Each collection of bars in Figure 10 corresponds to one dataset. Consider SCI_5M in Figure 10 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 10, 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 for CUR is similar and can be found in the technical report [9].

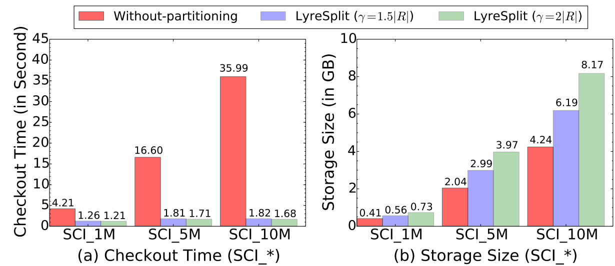


Figure 10: Comparison With and Without Partitioning

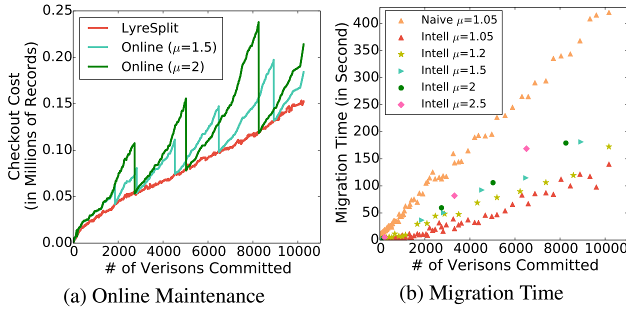
5.4 Maintenance and Migration

We now evaluate the performance of OrpheusDB's partitioning optimizer over the course of an extended period with many versions being committed to the system. We employ our SCI_10M dataset, which contains the largest number of versions (i.e. $10k$). Here, the versions are streaming in continuously; as each version commits, we perform online maintenance based on the mechanism described in Section 4.3. When $\frac{\mathcal{C}_{avg}}{\mathcal{C}^*_{avg}}$ reaches the tolerance factor μ, the migration engine is automatically invoked, and starts to perform the migration of data from the old partitions to the new ones identified by LyreSplit. We first examine how our online maintenance performs, and how frequently migration is invoked. Next, we test the latency of our proposed migration approach. The storage threshold is set to be $\gamma = 1.5|R|$. Similar results on $\gamma = 2|R|$ can be found in our technical report [9].

Online Maintenance.

Summary of Online Maintenance Compared to LyreSplit. With our proposed online maintenance mechanism, the checkout cost $\mathcal{C}_{avg}$ diverges slowly from the best checkout cost $\mathcal{C}^*_{avg}$ identified by LyreSplit. When $\mu = 1.5$, our migration engine is triggered only 7 times across a total of 10,000 committed versions .

As shown in Figure 11a, the red line depicts the best checkout cost $\mathcal{C}^*_{avg}$ identified by LYRESPLIT (note that LYRESPLIT is lightweight and can be run very quickly after every commit), while the blue and green lines illustrate the current checkout cost $\mathcal{C}_{avg}$ with tolerance factor $\mu = 1.5$ and $\mu = 2$, respectively. We can see that with online maintenance, the checkout cost $\mathcal{C}_{avg}$ (blue and green lines) starts to diverge from $\mathcal{C}^*_{avg}$ (red line). When $\frac{\mathcal{C}_{avg}}{\mathcal{C}^*_{avg}}$ exceeds the tolerance factor μ, the migration engine is invoked, and the blue and green lines jump back to the red line once migration is complete. As depicted in Figure 11a, when $\mu = 1.5$, migration is triggered 7 times, while it is only triggered 3 times when $\mu = 2$, across a total of 10000 versions committed. Thus, our proposed online maintenance performs well, diverging slowly from LYRESPLIT.



(a) Online Maintenance (b) Migration Time

Figure 11: Online Partitioning and Migration (SCI_10M)

Migration Time.

Summary of Comparison of Running Time of Migration. When $\mu = 1.05$, the migration time with our proposed method is on average $\frac{1}{10}$ of that with naive approach of rebuilding the partitions from scratch . As μ decreases, the migration time with our proposed method decreases.

Figure 11b depicts the migration time when the *migration engine* is invoked. Figure 11b is in correspondence with Figure 11a sharing the same x-axis. For instance, with $\mu = 2$, when the 5024th version commits, the *migration engine* is invoked as shown by the green line in Figure 11a. Correspondingly, the migration takes place, and we record the migration time with the green circle ($\mu = 2$) in Figure 11b. Hence, there are three green circles in Figure 11b, corresponding to the three migrations in Figure 11a.

We now compare our intelligent migration approach from Section 4.3, denoted *intell*, with the naive approach of rebuilding partitions from scratch, denoted *naive*. The points with upward triangles in Figure 11b all have $\mu = 1.05$, with the red points representing *intell*, and the brown representing *naive*: we see that *intell* takes at most $\frac{1}{3}$, and on average $\frac{1}{10}$ of the time of *naive*. For the sake of clarity, we omit the migration times for different μ using *naive*, since they roughly fall on the same line as that of $\mu = 1.05$. Next, consider the migration time with different μ using *intell*. Overall, as μ decreases, the migration time decreases. To see this, one can connect the points corresponding to each μ (denoted using different markers) to form lines in Figure 11b. When μ is smaller, migration takes place more frequently, due to which the new partitioning scheme identified by LYRESPLIT is more similar to the current one, and hence fewer modifications need to be performed. Essentially, we are amortizing the migration cost across multiple migrations.

6. RELATED WORK

We now survey work from multiple areas related to ORPHEUSDB.

Dataset Version Control. A recent vision paper on Datahub [12] acknowledges the need for database systems to support collaborative data analytics—we execute on that vision by supporting collaborative analytics using a traditional relational database. Decibel [33] 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 at all layers of the stack. For example, the eventual solution requires the system to log and query tuple membership on compressed bitmaps, reason about and operate on "delta files", 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, such as the parser, the transaction manager, and the query optimizer. We are approaching the problem from a different angle—the angle of *reuse*: how do we leverage relational databases to support versioning without any substantial changes to existing databases, which have massive adoption and open-source development that we can tap into. Starting from this perspective, the novelty of ORPHEUSDB lies in evaluating various designs of the representation scheme for capturing versioning information, and the partitioning algorithm for faster version control operations. Recent work on the principles of dataset versioning is also relevant [13] 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 system. Lastly, Chavan et al. [15] 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.

The problem of incremental view maintenance, e.g., [10], is also related since it implicitly considers the question of storage versus query efficiency, which is one of the primary concerns in data versioning. However, the considerations and challenges are very different, making the solutions not applicable to data versioning. Finally, Buneman et al. [14] introduce a range encoding approach to track the versioning of hierarchical data in scientific databases, but their method focuses on XML data and is not applicable to the relational datasets.

Temporal Databases. There is a rich body of work on time travel (or temporal) databases, e.g., [21, 36, 42], focusing on data management when the state of the data at a specific time is important. Temporal databases support a linear clock, or a linear chain of versions, whereas our work focuses on enabling non-linear histories. There has been some work on developing temporal databases by "bolting-on" capabilities to a traditional database [43], with DB2 [40] and Teradata [11] supporting time-travel in this way. Other systems adopt an "in-database" approach [25]. Kaufmann et al. [26] provide a good summary of the temporal features in databases, while Kulkarni et al. [27] describe the temporal features in SQL2011.

The canonical approach to recording time in temporal databases is via attributes indicating the start and end time, which differs a bit depending on whether the time is the "transaction time" or the "valid time". In either case, if one extends temporal databases to support arrays capturing versions instead of the start and end time, we will end up as a solution like the one in Figure 1b, which as shown severely limits performance. Thus, the techniques we describe in the paper on evaluating efficient data models and partitioning are still relevant and complement this prior work.

Most work in this area focuses on supporting constructs that do not directly apply to ORPHEUSDB, such as: (a) queries that probe interval related-properties, such as which tuples were valid in a specific time interval, via range indexes [38], or queries that roll back to specific points [31]; (b) temporal aggregation [25] to aggregate some attributes for every time interval granularity, and temporal join [20] to join tuples if they overlap in time; (c) queries that involve time-related constructs such as AS OF, OVERLAPS, PRECEDES.

There has been limited work on branched temporal databases [29, 39], with multiple chains of linear evolution as opposed to arbitrary branching and merging. While there has been some work on developing indexing [30, 22] techniques in that context, these techniques are specifically tailored for queries that select a specific branch, and a time-window within that branch, which therefore have no correspondences in our context. Moreover, these techniques require substantial modifications to the underlying database.

Restricted Dataset Versioning. There have been some open-source projects on versioning topics. LiquiBase [6] tracks schema evolution as the only applicable modifications giving rise to new versions: in our case, we focus primarily on the data-level modifications, but also support schema modifications as described in [9]. 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 [7] integrates various analytics tools into a collaborative data analysis platform. However, neither of the tools are focused on providing advanced querying and versioning capabilities. In addition, git and svn can be made to support dataset versioning, however, recent work shows these techniques are not efficient [33], and do not support sophisticated querying.

Graph Partitioning. There has been a lot of work on graph partitioning [24, 32, 19, 23], with applications ranging from distributed systems and parallel computing, to search engine indexing. The state-of-the-art in this space is NScale [37], which proposes algorithms to pack subgraphs into the 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 ones in terms of performance, while also providing a $10^3\times$ speedup. Kumar et al. [28] study 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. We also adapt LYRESPLIT to operate in an incremental fashion as new versions are introduced. Our experimental results demonstrate that LYRESPLIT is $10^3\times$ faster in finding the effective partitioning scheme compared to other algorithms, can improve the checkout performance up to $20\times$ relative to schemes without partitioning, and is capable of operating efficiently (with relatively few and efficient migrations) in a dynamic setting.

Acknowledgements. We thank the anonymous reviewers for their valuable feedback. We acknowledge support from ISTC for Big Data, grant IIS-1513407, IIS-1633755, and IIS-1652750, awarded by the National Science Foundation, grant 1U54GM114838 awarded by NIGMS and 3U54EB020406-02S1 awarded by NIBIB through funds provided by the trans-NIH Big Data to Knowledge (BD2K) initiative (www.bd2k.nih.gov), and funds from Adobe, Google, and the Siebel Energy Institute. The content is solely the responsibility of the authors and does not necessarily represent the official views of the funding agencies and organizations.

8. REFERENCES

[1] Array in MySql. 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] Liquibase. http://www.liquibase.org/.
[7] Mode. https://about.modeanalytics.com/.
[8] PostgreSQL9.5. www.postgresql.org/docs/current/static/intarray.html.
[9] ORPHEUSDB: Bolt-on versioning for relational databases. In *Technical Report*, Available at: http://data-people.cs.illinois.edu/papers/orpheus-tr.pdf.
[10] Y. Ahmad et al. Dbtoaster: Higher-order delta processing for dynamic, frequently fresh views. *VLDB Endowment*, 5(10):968–979, 2012.
[11] M. Al-Kateb et al. Temporal query processing in teradata. In *EDBT'13*.
[12] A. Bhardwaj et al. Datahub: Collaborative data science & dataset version management at scale. *CIDR*, 2015.
[13] S. Bhattacherjee et al. Principles of dataset versioning: Exploring the recreation/storage tradeoff. *VLDB*, 8(12):1346–1357, 2015.
[14] P. Buneman et al. Archiving scientific data. *TODS*, 29(1):2–42, 2004.
[15] A. Chavan et al. Towards a unified query language for provenance and versioning. In *TaPP*, 2015.
[16] G. O. Consortium et al. Gene ontology consortium: going forward. *Nucleic acids research*, 43(D1):D1049–D1056, 2015.
[17] C. De Castro, F. Grandi, and M. R. Scalas. On schema versioning in temporal databases. In *Recent advances in temporal databases*, pages 272–291. 1995.
[18] C. De Castro, F. Grandi, and M. R. Scalas. Schema versioning for multitemporal relational databases. *Information Systems*, 22(5):249–290, 1997.
[19] U. Feige, D. Peleg, and G. Kortsarz. The dense k-subgraph problem. *Algorithmica*, 29(3):410–421, 2001.
[20] D. Gao, S. Jensen, T. Snodgrass, and D. Soo. Join operations in temporal databases. *The VLDB Journal*, 14(1):2–29, 2005.
[21] C. S. Jensen and R. T. Snodgrass. Temporal data management. *IEEE Transactions on Knowledge and Data Engineering*, 11(1):36–44, 1999.
[22] 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.
[23] G. Karypis et al. Multilevel k-way hypergraph partitioning. *VLSI design*, 11(3).
[24] G. Karypis and V. Kumar. A fast and high quality multilevel scheme for partitioning irregular graphs. *SISC*, 20(1):359–392, 1998.
[25] M. Kaufmann et al. Timeline index: A unified data structure for processing queries on temporal data in sap hana. In *SIGMOD 2013*, pages 1173–1184.
[26] M. Kaufmann et al. Benchmarking bitemporal database systems: Ready for the future or stuck in the past? In *EDBT*, pages 738–749, 2014.
[27] K. Kulkarni and J.-E. Michels. Temporal features in sql: 2011. *ACM Sigmod Record*, 41(3):34–43, 2012.
[28] K. A. Kumar et al. Sword: workload-aware data placement and replica selection for cloud data management systems. *The VLDB Journal*, 23(6).
[29] G. M. Landau et al. Historical queries along multiple lines of time evolution. *The VLDB Journal*, 4(4):703–726, 1995.
[30] S. Lanka and E. Mays. *Fully persistent B+-trees*, volume 20. ACM, 1991.
[31] 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.
[32] D.-R. Liu and S. Shekhar. Partitioning similarity graphs: A framework for declustering problems. *Information Systems*, 21(6):475–496, 1996.
[33] M. Maddox et al. Decibel: The relational dataset branching system. *VLDB*, 9(9).
[34] H. J. Moon et al. Scalable architecture and query optimization fortransaction-time dbs with evolving schemas. In *SIGMOD 2010*.
[35] H. J. Moon et al. Managing and querying transaction-time databases under schema evolution. *VLDB*, 2008.
[36] G. Ozsoyoglu et al. Temporal and real-time databases: A survey. *TKDE*, 7(4).
[37] A. Quamar, A. Deshpande, and J. Lin. Nscale: neighborhood-centric large-scale graph analytics in the cloud. *The VLDB Journal*, pages 1–26, 2014.
[38] B. Salzberg and V. J. Tsotras. Comparison of access methods for time-evolving data. *ACM Computing Surveys (CSUR)*, 31(2):158–221, 1999.
[39] B. J. Salzberg and D. B. Lomet. *Branched and Temporal Index Structures*. College of Computer Science, Northeastern University, 1995.
[40] C. M. Saracco, M. Nicola, and L. Gandhi. A matter of time: Temporal data management in db2 for z. *IBM Corporation, New York*, 2010.
[41] D. Szklarczyk et al. The string database in 2011: functional interaction networks of proteins, globally integrated and scored. *Nucleic acids research*.
[42] A. U. Tansel et al. *Temporal databases: theory, design, and implementation*. Benjamin-Cummings Publishing Co., Inc., 1993.
[43] K. Torp, C. S. Jensen, and R. T. Snodgrass. Stratum approaches to temporal dbms implementation. In *IDEAS'98*, pages 4–13. IEEE, 1998.
[44] L. Xu, S. Huang, S. Hui, A. Elmore, and A. Parameswaran. ORPHEUSDB: A lightweight approach to relational dataset versioning. In *SIGMOD'17 Demo*.