ORPHEUSDB: A Lightweight Approach to Relational Dataset Versioning

Liqi Xu[1], Silu Huang[1], Sili Hui[1], Aaron J. Elmore[2], Aditya Parameswaran[1]
[1]University of Illinois (UIUC) [2]University of Chicago
{liqixu2,shuang86,silihui2,adityagp}@illinois.edu aelmore@cs.uchicago.edu

ABSTRACT

We demonstrate ORPHEUSDB, a lightweight approach to versioning of relational datasets. ORPHEUSDB is built as a thin layer on top of standard relational databases, and therefore inherits much of their benefits while also compactly storing, tracking, and recreating dataset versions on demand. ORPHEUSDB also supports a range of querying modalities spanning both SQL and git-style version commands. Conference attendees will be able to interact with ORPHEUSDB via an interactive version browser interface. The demo will highlight underlying design decisions of ORPHEUSDB, and provide an understanding of how ORPHEUSDB translates versioning commands into commands understood by a database system that is unaware of the presence of versions. ORPHEUSDB has been developed as open-source software; code is available at `http://orpheus-db.github.io`.

1. INTRODUCTION

The ever-rising ubiquity of data science has led to individuals and teams of various sizes to analyze and manipulate data at scale for commercial, scientific, and medical domains. This engenders the proliferation of dataset versions from various stages of analysis, which are often stored and maintained in an ad-hoc manner—typically with each version stored as a separate file independent of others. Such ad-hoc versioning mechanisms result in an explosion in storage, and simultaneously make it impossible to effectively manage and query across these dataset versions. While source code version control systems like `git` and `svn` may seem like appealing alternatives, they are both inefficient for data versioning and lack advanced querying capabilities [5, 6].

We present ORPHEUSDB[1], a system for relational dataset versioning, where datasets exist as an directed acyclic graph of versions with each version having zero or more parents. ORPHEUSDB is built as a thin wrapper "bolted" on top of a traditional (unmodified) relational database, with all of the versioning logic captured within the wrapper. The underlying relational database is completely unaware of the existence of versions. In this manner, ORPHEUSDB can seamlessly benefit from improvements to the underlying relational database, and also decouples and isolates the versioning components and logic from basic data management. Moreover, by operating on top of a relational database, ORPHEUSDB inherits all of the benefits of advanced querying capabilities "for free", along with efficient versioning capabilities.

Overall, ORPHEUSDB not only supports a large group of git-style commands (e.g., commit and checkout), but also supports a rich syntax of SQL queries, including queries on a specific set of versions, or queries to identify versions that satisfy certain properties. Consider a protein-protein interaction dataset [12], where teams of scientists continuously check out, update, and commit scores encoding interactions between proteins, based on various forms of evidence, including *neighborhood, cooccurrence,* and *coexpression*. Given this versioned *Interaction* dataset, the following SQL query finds all versions that have more than 100 protein-protein pairs with coexpression attribute greater than 80:

```
SELECT vid FROM CVD Interaction
WHERE coexpression > 80
GROUP BY vid HAVING count(*) > 100;
```

Note that `vid` (short for version id) and `CVD` (short for collaborative versioned dataset) are keywords within ORPHEUSDB that we will define later in Section 2. ORPHEUSDB also supports version graph traversal via special functional primitives, with operations such as listing the ancestors or descendants of a specific version or group of versions.

Underneath, ORPHEUSDB represents versioned data in a relational database using a simple but powerful representation scheme, coupled with intelligent partitioning algorithms to provide an efficient balance between version control performance and storage overhead over large datasets. At one extreme, for fast version retrieval, we may prefer to store each version as an independent relation as some records may appear in multiple versions; at the other extreme, for less storage overhead, we may want to store each record exactly once, independent of the number of versions it exists in. Our partitioning algorithms allow us to navigate this trade-off to gain the best of both worlds: fast version retrieval and compact storage. Our experimental evaluation [9] demonstrates that, compared to other alternative data models without partitioning, ORPHEUSDB achieves up to 10× less storage consumption, and up to 1000× less time for version control commands.

Related Work. Time-travel databases support versioning for linear chains of versions [4, 10, 7], as opposed to branched evolution of versions with merges, which is more natural in collaborative data science; the concerns, objectives (e.g., temporal joins, interval operators), and techniques are also fundamentally different [9]. We

[1]Orpheus is a musician from 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.

SIGMOD'17, May 14-19, 2017, Chicago, IL, USA

DOI: http://dx.doi.org/10.1145/3035918.3058744

build on the vision of Datahub [5] for collaborative data analytics; Decibel [11] also executes on the Datahub vision, but instead takes an approach "from the ground up", re-engineering all components of a version-oriented storage engine. This prototype is incomplete, and does not support full-fledged querying and optimization capabilities. Moreover, the Decibel storage and indexing methods (e.g., compressed bitmaps with deltas), as well as query processing algorithms (e.g., traverse multiple paths in the version graph just to create a version), require substantial changes to all layers of the database, making it challenging to modify or adapt existing databases for versioning purpose. Other work considers how to best trade off storage and retrieval for unstructured data versioning [6]; and the design of a prototypical versioning query language, without any actual implementation [8]. Our technical report [9] provides a detailed description of related work.

2. DATA REPRESENTATION

We begin by describing the basic notions of version control within ORPHEUSDB, and then describe how versioned data is represented.

The basic 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 conceptually contains all versions of that relation. Each row in CVD is an *immutable* record: any modification of a record generates a new version of that record and is treated and stored as a new record in the CVD. Overall, each record can be present in many versions of the relation, and each version can contain many records. Each record is identified by its unique record id, *rid*, and each version is identified by its unique version id, *vid*.

In ORPHEUSDB, for each CVD we separate the data from the versioning information into two tables: the *data table* and the *index table*. The data table stores all of the records appearing in any of the versions. We add an additional *rid* attribute in the data table to differentiate records in multiple versions that have the same primary key attribute(s); this attribute is invisible to end-users. In the index table, we track the records present in each version. In order to minimize the storage overhead from storing *vid* multiple times in (*vid*, *rid*) pairs, we instead take advantage of the data type `array` implemented in most modern database systems and maintain an attribute *rlist* of type `array` per *vid*. Thus, the attributes in the index table are *vid* and *rlist*. Readers may be able to identify other alternate designs for the index table; We have experimentally evaluated these designs in our paper and found this design to provide the best trade-offs from a storage and efficiency standpoint [9]. In particular, this design allows us to significantly reduce latencies during insertions of new versions, by avoiding extensive modifications across the entire relation. We return to the performance implications in Section 3.3.

In addition to the versioning information, in ORPHEUSDB, we also maintain version-level metadata in a separate *version table*. The table contains attributes such as the *vid*, an `array` of *vid*s it is derived from (i.e., parent versions), an `array` of *vid*s it is used to derive (i.e., children versions), creation time, commit time, committer and a commit message. Conceptually, we can view the derivation relationship in the version table as a *version graph*, where each node represents one version and each edge represents the derivation of version relations.

3. ORPHEUSDB ARCHITECTURE

We describe the functionality and syntax of our command line interface and SQL capabilities, followed by an overview of the ORPHEUSDB system architecture.

3.1 Version Control Commands

Checkout. Users can *checkout* all records within a specific version from a CVD via the command: `checkout [cvd] -v [vid] -t [table name]`. All records associated with the version are materialized and stored in a newly created table, whose name is specified in the command. Users can also checkout from multiple versions. In this scenario, records within these versions are *merged* together in the precedence order listed after `-v`. Checked-out versions are analogous to private working branches, where the owner can perform any analysis and modification on this table without interference from other users. This is effectively a composite of the `git` commands for checkout, branch, and merge.

Commit. Users are also capable of committing their local tables back to the CVD, making the modifications visible to other users via the syntax: `commit -t [table name] -m [commit message]`. For the records that are newly inserted or modified from its parent version(s), we append them to the data table as new records.

In order to support data science workflows, we also allow users to checkout and commit into and from `csv` files, by replacing the flag `-t` for `table` with `-f` for `file`. The `csv` file can be processed in external tools and programming languages such as Python or R. During commit, we require users to include a schema representation for the csv file.

Other commands. Besides checkout and commit, we briefly describe other commands supported in ORPHEUSDB: *(a) diff:* A standard differencing operation that compares two versions and outputs the records in one but not the other. *(b) init:* Load an external `csv` file or a structured table into ORPHEUSDB as a initial CVD and also create the corresponding versioning information. *(c) create_user:* Prompt a user to register as an ORPHEUSDB user. *(d) ls, drop:* Output a list of CVDS, or delete a CVD in ORPHEUSDB that the current user has access to. *(e) optimize:* Partition the data table within a CVD into a group of small tables, enabling other operations to access and process much less data for version retrieval. The partitioning algorithm can be executed periodically by the system or explicitly by a user; we will describe this further in Section 3.3.

3.2 SQL Commands

If the user has checked out one or more versions as a PostgreSQL table, then they are free to apply vanilla SQL to that table; if they have checked it out as a `csv`, then they are free to operate on that `csv` via external programming or scripting tools. In addition, users can run SQL commands on CVDS without having to materialize the appropriate versions. This happens via the command line using the *run* command, which either takes a SQL script as input or the SQL query as a string. These SQL commands use the special keywords: `VERSION`, `OF`, and `CVD` via syntax: `SELECT ... FROM VERSION [vids] OF CVD [cvd], ...`. For example, scientists can quickly overview a small number of (e.g., 50) records within the first two versions of the *Interaction* CVD whose *coexpression* attribute is greater than 80 via the following SQL command:

```
SELECT * FROM VERSION 1, 2 OF CVD Interaction
WHERE coexpression > 80 LIMIT 50;
```

Moreover, users can use SQLs to explore versions that satisfy some property by applying aggregation grouped by version ids. The corresponding syntax can be written as: `SELECT vid, ... FROM CVD [cvd], ... GROUP BY vid, ...`. Recall that in ORPHEUSDB, for each CVD, there are three related tables: the data table, the index table, and the version table. When writing SQL queries, users can be entirely unaware of the exact representation, and instead refer to attributes as if they are all present in one large CVD table. Internally, ORPHEUSDB translates these queries to those that are appropriate for the underlying representation.

In addition, ORPHEUSDB provides shortcuts for certain operations, such as traversing the version graph (e.g., listing the descendant or ancestors of a specific version) or comparing records between versions (e.g., identify records that coexists in two specific versions). These operations are accessible via functional primitives that can be included as predicates within a query: *(a) ancestor(*vid*)/descendant(*vid*), parent(*vid*):* The function takes a *vid* as the input and returns an array of all the ancestors/descendant, or its parent(s) of the *vid* in the version graph. *(b) v_diff(*vid*/*`ARRAY(`vid*)),* vid*/*`ARRAY(`vid*))):* The function takes two arguments, each of which could be either a *vid* integer or an array of *vid*s. It returns records in the data table that exist in the first argument but not in the second argument. *(c) v_intersect(* `ARRAY(`vid*)):* This is an aggregation function which takes an array of versions as the input and returns records in the data table that exist in all of these input versions.

We show a SQL example in Section 3 with its translation to the underlying representation in Figure 3(left). Another example of the version graph API is shown in Figure 3(right), where the query aims to find all version ids and their corresponding commit time such that the delta from the parent version(s) is greater than 100 records.

3.3 System Design

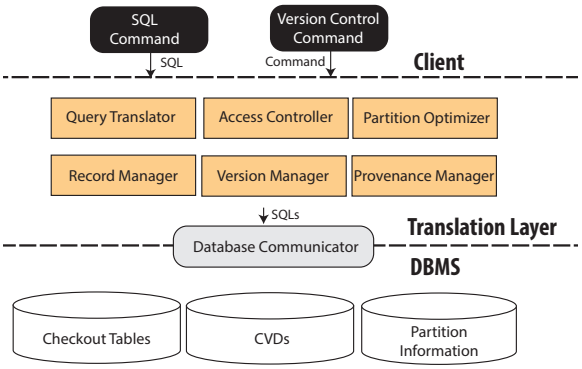

Figure 1: ORPHEUSDB *Architecture*

As shown in Figure 1, ORPHEUSDB is built as a lightweight layer on top of a traditional relational database, PostgreSQL. This layer handles the versioning logic in its entirety and the PostgreSQL backend is completely unaware of the existence of versioning. We now describe each of the modules in the translation layer of ORPHEUSDB. The query translator is responsible for parsing and transforming the input SQL to the one that are executable over our data model. The query translator is implemented by extending the `sqlparse` library [3] to extract the semantics of the SQL issued to ORPHEUSDB while the command line capabilities are instrumented using the Python package Click [1]. The access controller tracks the user information of the current session and manages the users' permissions to various CVDS and temporary materialized tables. The partition optimizer [9] supports a light weight approximation algorithm called LYRESPLIT, which, at a high level, recursively identifies partitioning opportunities on the version graph, until the average number of records per partition table is smaller than a theoretically guaranteed bound. The partition optimizer also logs the partition table each version resides in. Moreover, the record manager is responsible for record updates and retrieval within/from the data table, while the version manager is responsible for updates to or retrieval of versioning information from the version table and index table. The provenance manager logs all of the the metadata information for each private table/file that has not been committed, including the create time, parent versions, and the derived CVD name. Lastly, the database communicator acts as an intermediary between ORPHEUSDB and underlying database, passing SQL commands and returning results.

4. DEMONSTRATION DESCRIPTION

To demonstrate the functionality of ORPHEUSDB, and to make it easy for users to issue versioning commands and examine dataset versions, we have built a web-based front-end interface. We first describe the design of this interface and then describe the demonstration scenarios.

4.1 User Interface and Functionality

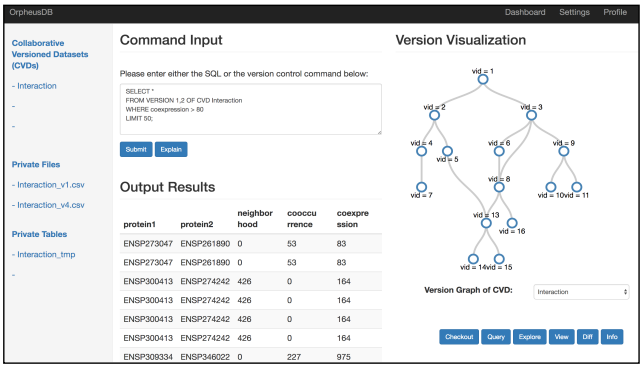

Figure 2: ORPHEUSDB *User Interface*

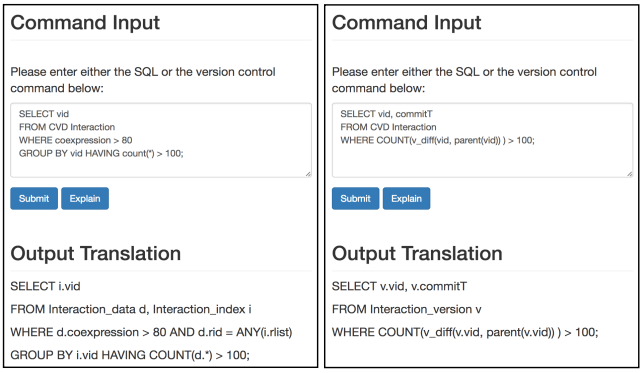

Figure 3: ORPHEUSDB *Translation Zoom-In for two queries*

As depicted in Figure 2, our web-based Javascript frontend consists of: *(a)* [Left-hand-side panel] A dataset explorer that lists the public CVDS, as well as all of the private tables and csv files that the current user has access to. *(b)* [Center, top] An command text box that takes either a SQL or git-like version control command. *(c)* [Center, bottom] An output window that displays either (i) a translation of the command into the ones understood by the backend if the 'Explain' button is selected; or (ii) the output of the command or query along with other messages (e.g. error messages) if the 'Submit' button is selected. *(d)* [Right-hand-side panel] An interactive version graph explorer that displays the version graph of the current CVD, allowing zoom-in and zoom-out. Users can select a set of versions via point-and-click, and apply various operations to these versions listed below the version graph.

The most primitive way to interact with this interface is to issue a git-style or a SQL style command into the command text box. If the user clicks the Explain button, ORPHEUSDB will display the translated SQL for the command that can be understood by the PostgreSQL backend. We show two examples of the output for the Explain button in Figure 3. If the user clicks on the Submit button instead, ORPHEUSDB will display the results of the command (if

correct), as is shown in Figure 2 for the command issued in the command text box. In addition, ORPHEUSDB will highlight, in the version graph of the version explorer, the nodes (i.e., versions) that participated in the output.

Another starting point to explore the versions is the version graph explorer. If the version graph is large, the user can avail of the zoom-in and zoom-out capabilities to examine the version graph in more or less detail. The user can get "quick facts" about a specific version by hovering over the node corresponding to that version. Then, the user can either use right click or drag a box to select a set of versions, following which the user can click one of the options listed below the version graph as shortcuts to express commands or explore versions in more detail: *(a) Checkout, Query, Explore*: clicking these options will prepopulate the text command box with the query templates for checking out one or more versions, querying one or more versions, and identifying versions that satisfy some property, for the set of versions that the user has selected. The user can start from this pre-populated template and then modify it to suit their needs instead of starting from scratch. *(b) View, Diff, Info*: clicking these options will display more information about the selected versions, whose output will be displayed below the version graph explorer. *View* will show the contents of the versions, *diff* will compare the contents of two or more versions, and *info* will list metadata pertaining to the selected versions.

4.2 Demonstration Scenarios

The goals of our demonstration scenarios are to *(a)* illustrate that the ORPHEUSDB front-end provides an effective and interactive mechanism to explore and operate on dataset versions; *(b)* show that ORPHEUSDB goes beyond `git` and `svn` to support both vanilla versioning commands as well as advanced SQL commands on versions; *(c)* demonstrate how ORPHEUSDB manages to support these commands, tracing the end-to-end execution of ORPHEUSDB; *(d)* illustrate how ORPHEUSDB can be embedded into a data science workflow; and *(e)* validate the design choices of ORPHEUSDB, via alternative data model designs and partitioning choices.

Next, we introduce the datasets we plan to use for our demonstration, following which we detail the demonstration scenarios that meet the above goals.

Datasets Description. We will consider two versioning schemes from Maddox et al. [11] that we will modify using real world datasets:

- **Analysis dataset**: This dataset, derived from the science workload in [11], simulates the working patterns of data scientists, who periodically take copies of an evolving dataset for isolated data analysis. The version graph here is a tree and can be visualized as a mainline (i.e., a single linear version chain) with various versions at different points. For instance, the evolving gene association dataset [2] contains the gene ontology (GO) assignments for various proteins in a given species. Multiple data science teams periodically check out and perform analysis on this dataset. Usually, data cleaning, normalization and featurization are conducted before each data mining task. This process generates various new versions of the same dataset, which are in turn committed and shared with the teammates.
- **Curation dataset**: This dataset simulates the evolution of a canonical dataset that many individuals are contributing to—these individuals don't just checkout from the canonical dataset but also periodically merge their changes back in. As a result, the version graph is a DAG . For instance, the protein interaction dataset [12] records the evolution of protein-protein interaction evidence over time across different organizations. Each attribute represents an evidence type, e.g., *neighborhood, cooccurrence and coexpression*. Typically, organizations first check out some existing version, update the evidence scores based on biological experiments or some curated knowledge base, and then periodically commit or merge back to create a new version of the protein-protein interaction dataset.

Scenario 1: Exploring Dataset Versions (Goals a–c). We will allow conference attendees to operate on dataset versions via the command text box as well as the version graph explorer, as described in the previous subsection. Attendees will get a feel for the querying capabilities of ORPHEUSDB via a spectrum of commands, both SQL and basic. Examples of SQL commands can be found in Figure 3. Attendees will be able to issue these queries via prepopulated templates from the version browser, as well as via the command line. For each of these queries, we will display the corresponding SQL translation that is understood by the PostgreSQL backend; in addition, we will show how each CVD is represented internally within PostgreSQL to allow attendees to get an intuitive feel for the data representation scheme adopted by ORPHEUSDB.

Scenario 2: End-to-End Data Science Workflow (Goal d). Next, we will demonstrate one way how a data scientist might use ORPHEUSDB: we will use the command-line interface to checkout one or more versions as a `csv` file, following which we update that `csv` file by performing some simple data cleaning operations within an external Python script, and then commit this `csv` file as a new dataset version back in ORPHEUSDB. We will demonstrate how ORPHEUSDB records the fact that a `csv` file is under checkout mode, and automatically infers the parent versions, and makes the appropriate changes (e.g., adding new records and the new version) to the underlying representation of the CVD within PostgreSQL.

Scenario 3: Evaluating the Design Choices (Goal e). To enable conference attendees to gain an understanding for the design choices of ORPHEUSDB, namely the impact of the data representation scheme and the partitioning algorithms, we will use a live A-B test with three server instances. One server will implement our data model with partitioned CVDS, while the other two will implement a naive data model with no partitioning, and the ORPHEUSDB data model with no partitioning, respectively. We will then allow attendees to study the performance of checking out or committing a version for these three settings, with performance metrics shown.

Acknowledgements. We acknowledge support from grant IIS-1513407 and IIS-1633755 awarded by the National Science Foundation and grant 1U54GM114838 awarded by NIGMS through funds provided by the trans-NIH BD2K initiative.

5. REFERENCES

[1] Click: a command line library for python. http://click.pocoo.org/5/.
[2] Go annotation. ftp://ftp.ebi.ac.uk/pub/databases/GO/goa/old/HUMAN/.
[3] sqlparse 0.2.2: Non-validating sql parser. https://pypi.python.org/pypi/sqlparse.
[4] I. Ahn and R. Snodgrass. Performance evaluation of a temporal database management system. In *SIGMOD Record*, volume 15, pages 96–107, 1986.
[5] A. Bhardwaj et al. Datahub: Collaborative data science & dataset version management at scale. *CIDR*, 2015.
[6] S. Bhattacherjee et al. Principles of dataset versioning: Exploring the recreation/storage tradeoff. *VLDB*, 8(12):1346–1357, 2015.
[7] P. Buneman, S. Khanna, K. Tajima, and W.-C. Tan. Archiving scientific data. *ACM Transactions on Database Systems (TODS)*, 29(1):2–42, 2004.
[8] A. Chavan et al. Towards a unified query language for provenance and versioning. In *TaPP*, 2015.
[9] S. Huang, L. Xu, J. Liu, A. Elmore, and A. Parameswaran. Orpheusdb: Bolt-on versioning for relational databases. *arXiv preprint arXiv:1703.02475*, 2017.
[10] 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.
[11] M. Maddox et al. Decibel: The relational dataset branching system. *VLDB*, 9(9):624–635, 2016.
[12] D. Szklarczyk et al. The string database in 2011: functional interaction networks of proteins, globally integrated and scored. *Nucleic acids research*, 39(suppl 1):D561–D568, 2011.