Generalized Partitioning for Dataset Versions in OrpheusDB

Vincent Truong

Electrical Engineering and Computer Sciences
University of California, Berkeley

Technical Report No. UCB/EECS-2021-183
http://www2.eecs.berkeley.edu/Pubs/TechRpts/2021/EECS-2021-183.html

August 12, 2021

Generalized Partitioning for Dataset Versions in OrpheusDB

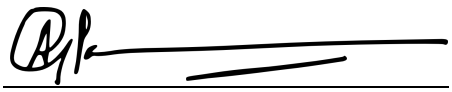
by Vincent Truong

Research Project

Submitted to the Department of Electrical Engineering and Computer Sciences, University of California at Berkeley, in partial satisfaction of the requirements for the degree of **Master of Science, Plan II**.

Approval for the Report and Comprehensive Examination:

Committee:



Professor Aditya Parameswaran
Research Advisor

08/04/2021

(Date)

* * * * * * *

Professor Aaron Elmore
Second Reader

08/10/2021

(Date)

Generalized Partitioning for Dataset Versions in OrpheusDB

by

Vincent Truong

A dissertation submitted in partial satisfaction of the

requirements for the degree of

Master of Science

in

Computer Science

in the

Graduate Division

of the

University of California, Berkeley

Committee in charge:

Aditya Parameswaran, Chair
Professor Aaron Elmore

Spring 2021

Generalized Partitioning for Dataset Versions in OrpheusDB

Abstract

Generalized Partitioning for Dataset Versions in OrpheusDB

by

Vincent Truong

Master of Science in Computer Science

University of California, Berkeley

Aditya Parameswaran, Chair

OrpheusDB is a lightweight dataset version management system designed to be integrated into data science workflows, akin to standard code version control systems. By using a relational database system as the backend, the Orpheus system leverages its querying and storage capabilities while remaining agnostic to the particular database system used.

Previous work by Huang et al. [7] explored different designs for implementing versioned storage effectively. However, in practice, the current implementation of OrpheusDB does not support all of the features described that would improve performance, such as efficient support for schema changes, which can occur often during data science. This report improves on the original design by adjusting how OrpheusDB handles checkouts and commits. In addition to some direct improvements to these operations, we implement the partitioning algorithm, Lyresplit, to offer the same level of performance shown in the original paper, as part of the open source version.

To my friends and family

Contents

List of Figures

Chapter 1

Introduction

Currently, many tools exist for managing and iterating on datasets as part of a data science workflow. This ranges from minimalist solutions that simply tie an ID to each dataset version, which can be used for retrieval, to systems that capture an entire machine learning workflow for reproducibility [2] [8] [9]. Thus, there are a variety of different approaches to storing and representing datasets and how they change over time. For example, one could save all iterations of the dataset in their entirety, allowing users to easily revisit and explore previous versions. However, such a scheme could store redundant rows and potentially duplicate entire dataset versions, leading to a large storage overhead that could have been saved through tracking and removing duplicates. On the other end of the spectrum, one could use a complex scheme to minimize the overall storage overhead and redundancy but this approach might compromise on efficient retrieval and analytical ability.

One system under development that attempts to solve this problem is OrpheusDB, a versioning system that leverages traditional relational database management systems to track, store, and retrieve versions of a dataset in an efficient manner while providing useful querying capabilities across the version history [3]. OrpheusDB seeks to find a reasonable "middle ground" in the storage-retrieval trade off by using a partitioning scheme to separate groups of related versions based on overlap in records. However, while introduced in the original paper, these partitioning schemes have not been implemented in their entirety into the open source system, and, before our work, the system lacked the performance as promised by the original paper [7].

Determining such a partitioning scheme and associated algorithms for creating, maintaining, and utilizing the partitions effectively is difficult. For example, we could partition all of the versions separately, resulting in each version being a table. While having fast retrieval times, this approach results in high storage overhead due to replicating data across multiple tables. The other extreme would be to attempt to compress all of the versions together into one table. Such a scheme would minimize the storage overhead but would result in expensive updates. Any scheme used would additionally need to be maintainable when new versions are being added; saving new versions should not require unnecessarily expensive commit operations and future checkout operations should not be heavily impacted

by previously committed versions.

To this end, OrpheusDB implements a partitioning scheme which groups versions of a dataset into partitions based on how similar the versions are to each other. It utilizes Lyresplit, an algorithm introduced in the original paper to determine which versions should be grouped together and how many partitions to create. This partitioning is automatically done without any direct user specification and is maintained even when new versions are added. Partitioning reduces the overall amount of data that the system has to access for each operation, improving performance, allowing OrpheusDB to handle certain operations more easily, particularly schema changes, compared to the previous implementation.

This report first starts with an overview of the OrpheusDB system at a high level and relevant subsystems, as described in the original paper [7]. We highlight improvements to the checkout and commit commands as well as the implementation of the partition and migration subsystem. Finally, we conclude with an analysis of different partitioning schemes and demonstrate different scenarios that can occur during a data science workflow with OrpheusDB.

Chapter 2

Architecture and Implementation Details

In this chapter, we present a brief overview of the OrpheusDB open source implementation. The design philosophy, such as the particular table layout and choice of representation, is discussed in the original OrpheusDB paper [7].

2.1 Overview

Internally, OrpheusDB acts as a translation layer between the user and the underlying relational database. The user interfaces with this layer through git-style commands or via a visual user interface [4] [3]. The translation layer communicates to the executor which validates the arguments passed in and runs the associated logic to execute the requested command. The functionality of OrpheusDB is split into different managers to handle each of the various facets of storing versioned datasets. OrpheusDB allows the user to store and revisit previous versions of their dataset efficiently, as well as provides a query interface to perform more interesting analyses across different versions.

OrpheusDB is written in Python with the Click and psycopg2 libraries to manage the command line interface and the database connection respectively [1] [5] [6]. It is backed by PostgreSQL 12.6; however, the implementation of OrpheusDB is designed to be portable to other database backends.

2.2 Data Representation

The key to OrpheusDB's design is how each of the dataset versions are laid out in the underlying relational database. Instead of storing each of the versions of the dataset as individual tables, we store a collaborative versioned dataset (CVD) capturing information across multiple tables. A CVD is comprised of four different types of tables: data, attribute, index, and version tables. An example of the layout is shown in Figure 2.1.

Table	Index Table				
vid	alist	rlist	partition	closest_parent	score
1	{1, 2, 3}	{1...10}	1	NULL	-1
2	{1, 2, 3}	{4...10}	1	1	7
3	{1, 2, 3}	{1...3, 11...15}	2	1	3
4	{1, 2, 3}	{1...3, 11...15}	2	3	8

Table	Version Table				
vid	author	num_records	parents	children	commit_time
1	User	10	NULL	{2, 3}	T1
2	User	6	1	NULL	T2
3	User	8	1	{3}	T3
4	User	8	3	NULL	T4

Table	Attribute Table	
aid	attname	attype
1	name	text
2	age	int
3	salary	int

Table	dataset_part_1		
rid	name	age	salary
1	...		
⋮	⋮		
10	...		

Table	dataset_part_2		
rid	name	age	salary
1	...		
⋮	⋮		
3	...		
11	...		
⋮	⋮		
15	...		

Figure 2.1: Example CVD

Data Table(s)

The data tables is where the actual versioned data is stored. It is comprised of records with unique record ids (rids) corresponding to the rows of data. The data in a particular dataset is partitioned into many smaller tables to improve performance. This is elaborated further in Chapter 4.

Attribute Table

The attribute table records the schema for each version in the CVD. This table stores information about the columns in a CVD such as the attribute name and typing. This allows OrpheusDB to correctly associate column names to the attribute ids (aids) used to retrieve and store versions of a dataset and track how columns evolve over time.

Index Table

The index table tracks which rids and aids are associated with which version in the CVD. With the introduction of partitioning, the index table now also contains partitioning meta-

data such as which partition a version is stored in and how similar a version is to other versions in the same partition. The partitioning metadata will be elaborated further in Chapter 4 of the report.

Version Table

The version table stores associated metadata information about a particular version. This table stores the vids with metadata such as commit user, commit message, and commit time, similar to git.

2.3 Managers

In order to facilitate safe operation on the associated database tables, each table is created by and updated by different managers. This section highlights some of the key managers in general operation.

Relation Manager

The relation manager directly manipulates the datasets inside the relational database that is backing OrpheusDB. This manager handles any updates to the data tables during the checkout and commit phase as well as creates and maintains the partitioning scheme.

Index Manager

The index manager keeps track of the version history and how similar each version is to the version from which it was derived. Additionally, it returns the associated metadata necessary to the relation manager to perform the checkout and commit such as the rids and attributes that a particular version contains.

Partition Manager

The partition manager monitors partition health and runs the partition decision boundary algorithm, Lyresplit, described in Chapter 4, Section 4.2. While this manager does not write any information directly back to the database, it reads from the index table to construct a version history tree for partitioning purposes. After deciding on the partitions for the CVD, it returns a plan to the relation and index manager to perform the partitioning operation and update the metadata respectively.

Chapter 3

General Operation

In this chapter, we go into greater detail about how two of the main operations in OrpheusDB, checkout and commit, work under the hood. While these commands existed in the previous version of OrpheusDB, they have been modified to handle partitioning with potentially multiple data tables per dataset. Users would primarily interact with their stored, versioned datasets through the checkout and commit commands.

3.1 Checkout

In order to materialize and use a particular version of the CVD, the user invokes the checkout command. OrpheusDB retrieves the associated rids, aids, and the name of the partition this particular version is located within. After translating the aids to attribute names using the CVD's attribute manager and its corresponding table, we construct and execute a SQL query that would fetch the requested version from the data table partition. OrpheusDB then writes the query results to either a csv file or to an table inside PostgreSQL per the user's request.

From the example CVD in Figure 2.1, in order to checkout version 2, OrpheusDB would execute the following SQL command to perform the checkout operation:

> SELECT name, age, salary FROM dataset_part_1 WHERE rid = ANY(ARRAY[4, 5, 6, 7, 8, 9, 10])

Pseudocode for the checkout operation is shown in Figure 3.1.

3.2 Commit

After the user has made changes to their exported version of the dataset, they can save these changes to OrpheusDB with the commit command. Commit can be broadly broken down into four stages: Schema Detection, Record Detection, Partition Scoring, and Storage.

Algorithm 1: Checkout

Input : Dataset Name, Vid, and Destination (D, V, dest)
Output: Destination (dest)

```
1 meta ← load_meta()
2 aids, rids, part_num ← index_mgr.fetch(D, V)
3 attr ← attribute_mgr.fetch(D, aids)
4 relation_mgr.create_table(D, rids, attr, part_num)
5 relation_mgr.copy(dest)
6 metadata_mgr.write(D, V)
7 return dest
```

Figure 3.1: Pseudocode of the Checkout Process

Schema Detection

During this phase, we want to ensure that each attribute of the committed version is accounted for. We denote attributes as either deleted, edited, added, or retained from the parent version with the edited category specifically being reserved for type changes. If OrpheusDB detects a new column being added, we update the schema of all of the data tables in the CVD with the new column in order to make the partitioning step — which could require moving records from one partition to another — easier.

Currently, Orpheus allows for specification of a schema file that describes the name and type of each column in the committed dataset. While this method is preferred due to the explicit declaration, Orpheus is also equipped with a rudimentary schema parser, allowing csv files with headers to be committed without an explicit schema file. However, this parser only checks the names of the columns from the header, potentially resulting in records already stored in the system to be recognized as new incoming records and becoming duplicated when they otherwise should not have been. Robust schema support remains as an interesting potential improvement which will be discussed further in Chapter 7.

Record Detection

To determine the overall changes in the dataset, we need to check which records are currently in the parent version of the dataset and which ones are being added. We perform an inner join on the attributes that are present in both the parent and child tables derived from the schema detection step to get the intersection. This determines both the new records and returning rids in this phase. Note that if there is more than one parent for the newly committed version, we check all of the parents to find the one with the highest overlap and prioritize the highest number of matched columns first.

Previously, record detection was implemented such that any schema changes resulted in

Algorithm 2: Commit

Input : Dataset Name, Data, Schema, Partition Paramter (D, d, s, δ)
Output: Partition P

```
1  P ← relation_mgr.fetch_part(D, head)
2  p_vid ← relation_mgr.checkout(D, head, P)
3  p_aids ← index_mgr.fetch_(D,p_vid)
4  if No schema file provided then
5  |  // Detecting Schema from header
6  |  s ← attr_mgr.schema_parse(data)
7  end
8  Copy data into temp table t using s → db
9  del, add, edit, same ← attr_mgr.diff(s, p_aids)
10 intersect ← relation_mgr.get_intersect(p_vid, d)
11 same_count ← count(intersect)
12 if same_count ≤ δ—R— then
13 |  // Store in new partition
14 |  relation_mgr.add_part(intersect, D)
15 |  relation_mgr.update_rids(intersect)
16 else
17 |  // Store in same partition
18 |  relation_mgr.update_existing(D, intersect)
19 |  relation_mgr.add_new_rows(D, data)
20 |  relation_mgr.drop(intersect)
21 end
22 return P
```

Figure 3.2: Pseudocode of the Commit Process

all incoming records being appended to the large data table as new records. By adding an additional scan over the smaller data table partition during the commit phase (during the join to create the intersection table), we are able to reduce future checkout times and decrease storage.

Partition Scoring

After we have determined the intersection between the parent table(s) and child table, we score the incoming version using a partitioning metric. By default, this metric is the size of the overlap between the closest parent and the child version but can be altered and specified by the user through the partitioning command. Additionally, we determine if the incoming

version is better suited to be added to the parent partition or stored separately as a new partition. This is further elaborated in Section 4.4 of the paper.

Storing

In the storing phase, we write the new records into the appropriate partition. In order to save computation, we reuse the intersection table created during the Record Detection phase. If we are creating a new partition, we simply rename the intersection table to the appropriate partition name before adding and assigning the new rids for rows that do not have one already. If we are appending to the current partition, we use the intersection table to update the columns in the current partition and then drop the intersection table. Afterwards, we update the CVD's index table to record the addition of the new version, the new rids, and which partition this version resides in.

Figure 3.2 details the modified commit algorithm.

In the example shown in Figure 2.1, if we were to commit a version into the same dataset as a child to vid 4, the following SQL commands are run:

> CREATE TABLE temp AS COPY FROM data.csv;
>
> CREATE VIEW part_view AS SELECT A, B, C FROM dataset_part_2 WHERE rid = ANY(ARRAY[1...3,11...15]);
>
> CREATE TABLE intersect_table AS SELECT * FROM part_view INNER JOIN temp USING (name, age, salary)

If a new partition is created:

> ALTER TABLE intersect_table RENAME TO dataset_part_3
>
> INSERT INTO dataset_part_3 SELECT name, age, salary FROM temp EXCEPT SELECT name, age, salary FROM part_view

If we are updating a new partition:

> INSERT INTO dataset_part_2 SELECT name, age, salary FROM temp EXCEPT SELECT name, age, salary FROM part_view

How OrpheusDB chooses to update an existing partition or create a new partition is discussed in the next chapter.

Chapter 4

Partitioning

If the data table becomes too large and encompasses too many distinct versions, the commit and checkout performance is heavily impacted, even if we maintain an index on rid. Thus, OrpheusDB has been designed to work on smaller subsets of the data table called data table partitions. In the example CVD shown in Figure 2.1, the CVD contains two data table partitions: data_part_1 and data_part_2. Each partition would contain all the records for a group of similar partitions. For simplicity, we ensure that a version cannot be in multiple partitions and cannot have records in multiple partitions. However, some of the data may be duplicated due to this partitioning scheme: for some record r_i that is shared across versions $v_1, ..., v_n$ with each version in its own partition, r_i is duplicated n times for the n different partitions.

To facilitate partitioning, we implement a partition manager that is closely linked to the relation manager. While the relation manager is designated with creating and updating tables in the database, the partition manager makes partitioning decisions based on a given dataset's version history.

OrpheusDB currently only considers row-wise partitioning (partitioning on rids). Consideration for other partitioning strategies such as across attributes will be discussed in the Future Work Chapter.

4.1 Version Representation

OrpheusDB stores all of the information necessary for partitioning a CVD in its index table. The index table has been updated to contain two additional fields: the closest parent and the score.

The closest parent denotes which parent vid, if there are multiple, is the most similar to the child version. In order to assign the closest parent, we score each parent version using a metric function. Users can specify which function to use when the dataset is initialized and by invoking the partition command. By default, OrpheusDB counts the rid overlap between the parent and child version as the metric.

Algorithm 3: Lyresplit

```
Input  : VersionNode Array (V)
Output: Partition Plan (P)
if size(P) = 1 —— partition_check(P) then
    return P
else
    split_node ← min(P, key=VersionNode.score)
    P1 ← split_node + all_child_nodes(split_node)
    P2 ← P / P1                                   // P2 = Rest of the nodes
    return Lyresplit(P1) + Lyresplit(P2)
end
```

Figure 4.1: Pseudocode of Lyresplit Implementation

A VersionNode class has been implemented to allow the partition manager to represent and manipulate a CVD's version tree. The class is used primarily with the Lyresplit algorithm from Huang et al. [7] with the whole version tree for a CVD being constructed when the CVD requires a full repartitioning. While a version graph has been implemented for the visual interface, the partition manager's VersionNodes only keeps track of the closest parent and the corresponding scores, represented by the weights of the edges in the graph.

4.2 Lyresplit

In order to decide where to partition, OrpheusDB uses Lyresplit, a lightweight recursive partitioning algorithm, designed to find a middle ground in the storage-computation trade off. After we construct the version tree as described in the Section 4.1, we begin by selecting a VersionNode based on the lowest weighed score in the current partition which indicates the smallest overlap between a parent and child version in the partition. The small overlap makes it a good candidate for partitioning as it would minimize the amount of duplicated data in the partitions. We split the version tree into two subtrees, representing two partitions, by removing the edge with the smallest weight. We, then, recursively call Lyresplit on the two smaller partitions, stopping when each partition either contains only one VersionNode or if the partition is of the desired size. Isolated experiments on the validity of the Lyresplit algorithm are shown in the original paper [7].

Figure 4.1 shows a rough sketch of the algorithm.

Algorithm 4: Migration

Input : Array of Planned Partitions P_p, Array of Current Partitions P_c
Output: Updated Partitions

```
part_pairing ← pair_partitions(P_p, P_c)
for pair in part_pairing do
    additions, deletions ← diff(pair)
    perform_add(pair, P_c)
end
perform_part_drop(part_pairing, P_c)
for pair in part_pairing do
    perform_drop(pair, P_c)
end
```

Figure 4.2: Pseudocode of Migration Algorithm

4.3 Migration

After finding the partitions as described in the previous section, OrpheusDB will attempt to migrate the existing partitions into the new partitions by moving rids from one data table partition to another. We first attempt to match each of the existing partitions to the desired partitions based on how similar they are. This reduces the number of operations required to create the new partition from the old partition. If the number of operations required to transform a partition to another exceeds the cost it would take to create a new table, or if there is no existing tables to match to the planned partition, we elect to create a new table instead.

For each planned partition, we mark the rids that need to be added or removed from the existing partition to transform it into the new partition. We proceed to update the partitions by adding the records that each partition requires. Afterwards, we delete all of the records that are not required in each partition. This approach of processing the additions before the deletions in all of the partitions allows the system to not use a temporary table to hold deleted data that could be required in a future partition. Finally, after the rids have been migrated to their desired partitions, the index table is updated to match the new partition scheme.

A sketch of the migration algorithm can be found in Figure 4.2.

4.4 Incremental Partitioning

As the user commits more versions to the particular dataset, performance will degrade over time until OrpheusDB repartitions the CVD. However, if the full partitioning and

migration operations are run after every commit, we incur a large overhead of recreating and manipulating the version tree. Thus, OrpheusDB uses an online partitioning algorithm by only comparing the current partition and the incoming version. If the incoming version's overlap with its closest parent is high, we store the version in the current partition, otherwise we store it as a new partition.

Chapter 5

Partitioning Schemes

For any particular dataset, there exists an exponential amount of different ways we can split the dataset versions into multiple partitions. In addition to how we decide to partition the data, factors such as the version history and the database used makes some partitioning schemes perform better than others. This chapter highlights and compares some of the basic schemes that are possible.

5.1 Experimental Setup

For each partitioning scheme, we performed two different experiments to simulate different potential workloads. Combining both of these tests would lead to a more realistic workload, but we isolated them to observe the performance of each scheme. The experiments were run on an i5-4250U running Ubuntu 20.04.1 and PostgreSQL 12.6. Each experiment assumes that OrpheusDB currently stores a CVD with exactly one version V with $n = 10000$ number of records. We attempt to commit a new version v_{new} to the dataset with the appropriate modification as discussed above. After the commit operation, we then attempt to retrieve the first version committed. Additionally, we assume that if partitioning is used in the scheme, the new version will sit in its own partition.

We evaluated two different scenarios: adding rows to the dataset or adding a new column to the dataset. Any update to the dataset can be comprised of these two operations; deletions can be done by simply updating the index table.

5.2 Evaluated Approaches

No Partitioning

In this scheme, we do not perform any partitioning and instead create one large data table to store all of the data. Additionally, any changes to the schema results in all records being considered as new regardless if it was derived from the previous versions. Before the

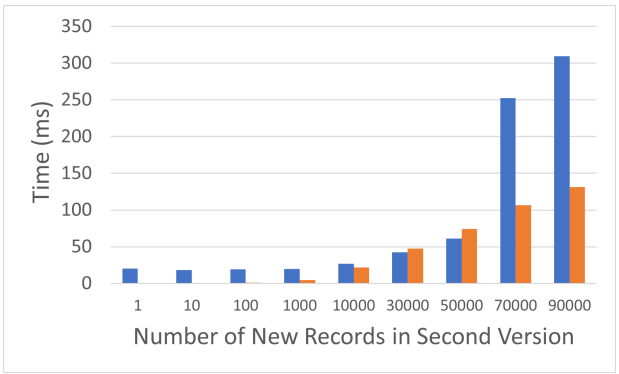
changes described by this paper, OrpheusDB utilizes this scheme due to the simplicity of implementation.

Horizontal Partitioning

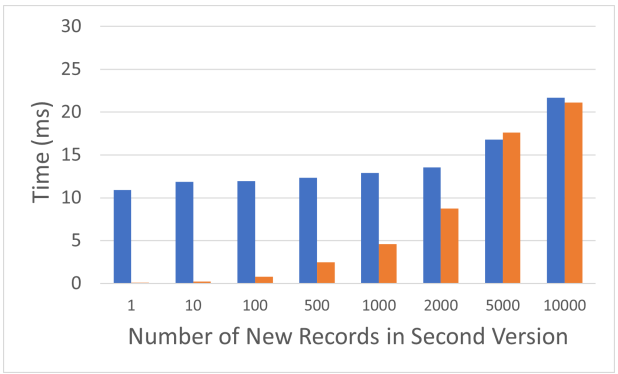
In this scheme, we partition the data table across the rows of the dataset. A partition P with versions $v_1...v_n$ would contain the records for all of the versions in the partition. We would duplicate overlapping records between partitions if necessary.

Vertical Partitioning

In this scheme, we focus on partitioning across columns. If a version adds a new column, we create a smaller table containing the column and the rid. In order for the system to checkout a version that spans multiple tables, we perform a join on the rids before outputting the result. If no columns are being added or if columns are removed, this reduces to the same scheme as having no partitioning at all.

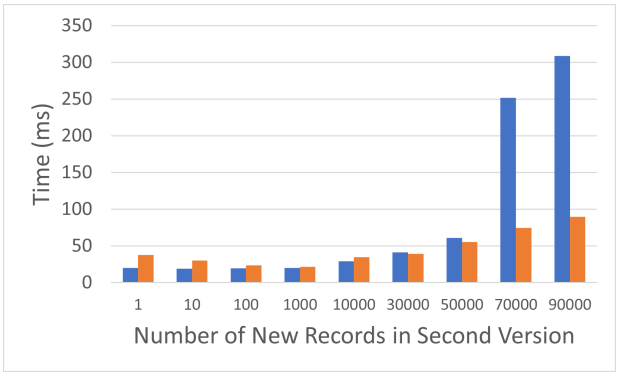


Adding Rows

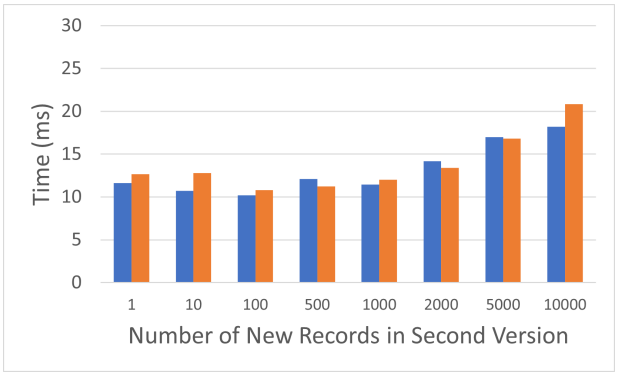


Adding a Column

Figure 5.1: Time (ms) taken on checkout (blue) and commit (orange) by the no partitioning scheme

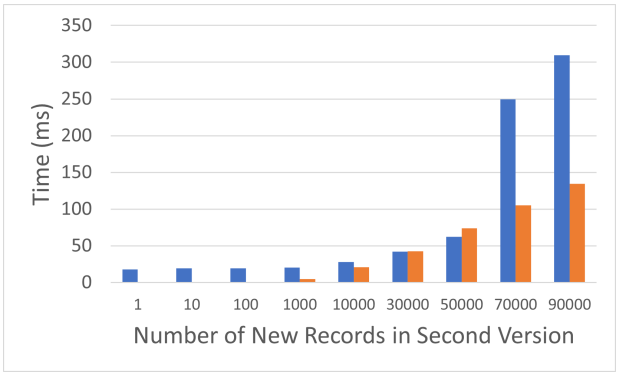


Adding Rows

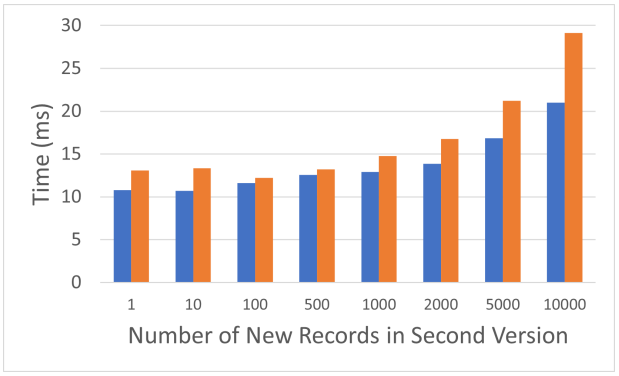


Adding a Column

Figure 5.2: Time (ms) taken on checkout (blue) and commit (orange) by the row partitioning scheme



Adding Rows



Adding a Column

Figure 5.3: Time (ms) taken on checkout (blue) and commit (orange) by the column partitioning scheme

Performance

Graphs on performance (time taken for each operation) for each partitioning scheme are shown in Figure 5.1, 5.2, and 5.3. We vary the amount of new data being added and see how these schemes perform as the amount of data increases.

In the first scenario where we are adding rows to the dataset, there are no significant differences between each of the schemes; when adding 90,000 new records to the table, the no partitioning, row partitioning, and column partitioning scheme took 309.686ms, 308.621ms, 309.423ms respectively. However, when attempting to checkout the first version, the row

partitioning scheme performs the best (89.877ms) with the other two schemes having about the same performance (∼130ms). This difference is due to that scheme holding the first version as its own partition, reducing the amount of data that the scheme has to search through to checkout the first version.

In the second scenario where we are adding a column to the dataset with a different amount of records changed, the difference between the partitioning schemes becomes more apparent. With no partitioning, adding a new column is treated as adding in new records, similar to the previous version of OrpheusDB. This results in duplicating the data but an extremely fast commit time accordingly; for example, commiting a new column with 5000 records using the no partitioning scheme took $\sim 5.674ms$ compared to row partitioning which took $\sim 14.542ms$ for the same operation. However, checkout times for the first version increase as the system has to search through more data to fetch the version. The row partitioning scheme performs similarly to the no partitioning scheme, however, due to having to update the entire data partition, the time taken to perform the update has a higher overhead when compared to no partitioning. This can be seen in the experiment, even if the number of records are added is small in comparison to the size of the table, the commit time hovers ∼12ms for when $N < 2000$ where N is the number of new records. The column partitioning scheme performed the worst with the costly overhead of creating a new table and arranging the rids appropriately as a join key. The scheme had similar performance to row partitioning scheme at an average of ∼13ms when $N < 2000$ and increases to 28.94ms at $N = 10000$.

For the OrpheusDB implementation, we chose to implement a horizontal partitioning scheme. As shown in our experiements, this scheme has as short, if not shorter, commit times and checkout times compared to no partitioning. This scheme is also easily maintained when compared to the vertical partitioning.

Chapter 6

Workflow

In this section, we elaborate on the current implementation of horizontal partitioning in OrpheusDB. We depict different scenarios that can occur and highlight how the implementation handles them, depicting the CVD and version tree after the update.

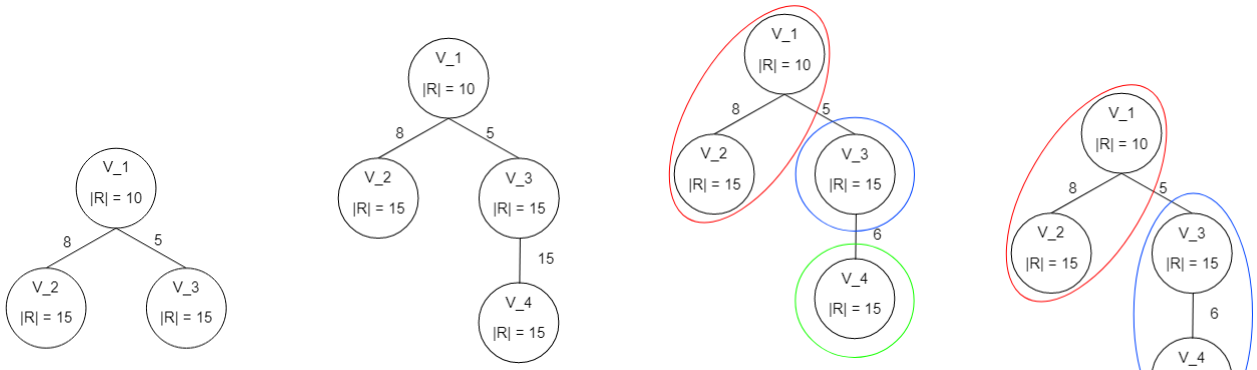
For each scenario, suppose we have three different versions V_1, V_2, V_3 of dataset D. V_1 has 10 records, V_2 has 15 records and share 8 records with its parent, V_1. V_3 has 15 records and share 5 of them with V_1. V_1 and V_2 are in "data_part_1" while V_3 is in "data_part_2". All three versions have the same attributes (A, B, C). The partitioning parameter is set to the default value: $\delta = 0.5$. Figure 6.1(a) shows the version tree with the associated weights. Figure 6.2(a) shows the initial CVD layout. We assume that the user has currently exported version V_3 of the dataset and is in the process of committing another one as a child to V_3.

Adding New Rows

Suppose the user attempts to commit V_4 by adding three new records to V_3 and retaining the 15 records found in V_3. OrpheusDB's relation manager would create an intersection table containing 15 records. Comparing V_4 to its parent, we add it to the current partition ($Overlap > \delta|R_{parent}|$). After updating the data partition, the index manager updates the index table associated with the dataset with the new version information with a score of 15. The updated version tree is shown in Figure 6.1(b) with $|R| = 18$ in V_3 instead of $|R| = 15$. Figure 6.2(b) depicts the table layout.

Adding New Columns

Suppose the user has altered the schema of the table and added a new column to the dataset. The previous implementation of OrpheusDB would have considered all of these rows as new records, making the partition scheme ineffective. The current implementation of OrpheusDB is able to correctly determine that the records are not entirely new and append the new column accordingly. The relation manager performs an update operation on the



(a) Initial State

(b) Adding New Row or Columns w/o Partitioning

(c) Adding Row w/ Incremental Partitioning

(d) Full Partition

Figure 6.1: Version Tree of the Difference Scenarios. Partitions are the different colored circles surrounding version nodes

data partition and the index manager marks the new version being committed with a score of 15. The updated version tree is shown in Figure 6.1(b) and the CVD is shown in 6.2(c).

Adding New Rows with Partitioning

Suppose the user attempts to commit a version that only inherits six records from V_3 and contains nine additional records for a total of 15 records. After creating the intersection table containing the six records in common, OrpheusDB chooses to create a new partition: "data_part_3." The relation manager inserts the nine new records into the intersection table and assigns new rids to each of these records. Finally, the intersection table is renamed to "data_part_3" and the index manager updates its corresponding table with the new partition and version information with a score of six. The update version tree with the partitions drawn out is shown in Figure 6.1(c).

Note that using Lyresplit would have resulting in creating different partitions when compared to the incremental partitioning used here. If the repartitioning command is invoked, the CVD would be partitioned as follows: $\{V_1, V_2\}, \{V_3, V_4\}$. This removes from some duplicated data and the overall number of partition tables used. The version tree if OrpheusDB fully repartitions the CVD is shown in 6.1(d). The CVDs for both version trees are found in Figure 6.2(d) and (e) respectively.

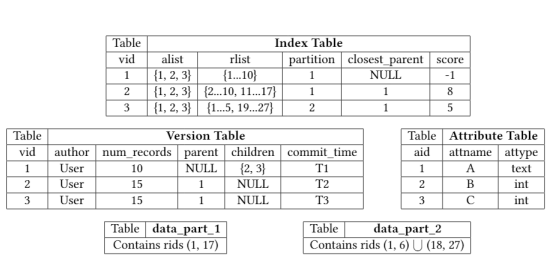


Table	Index Table				
vid	alist	rlist	partition	closest_parent	score
1	{1, 2, 3}	{1...10}	1	NULL	-1
2	{1, 2, 3}	{2...10, 11...17}	1	1	8
3	{1, 2, 3}	{1...5, 19...27}	2	1	5

Table	Version Table				
vid	author	num_records	parent	children	commit_time
1	User	10	NULL	{2, 3}	T1
2	User	15	1	NULL	T2
3	User	15	1	NULL	T3

Table	Attribute Table	
aid	attname	attype
1	A	text
2	B	int
3	C	int

Table	data_part_1
Contains rids (1, 17)	

Table	data_part_2
Contains rids (1, 6) ⋃ (18, 27)	

(a) Initial State

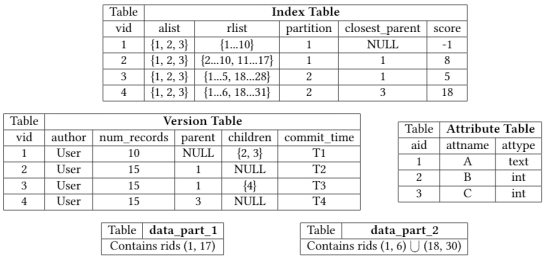


Table	Index Table				
vid	alist	rlist	partition	closest_parent	score
1	{1, 2, 3}	{1...10}	1	NULL	-1
2	{1, 2, 3}	{2...10, 11...17}	1	1	8
3	{1, 2, 3}	{1...5, 18...28}	2	1	5
4	{1, 2, 3}	{1...6, 18...31}	2	3	18

Table	Version Table				
vid	author	num_records	parent	children	commit_time
1	User	10	NULL	{2, 3}	T1
2	User	15	1	NULL	T2
3	User	15	1	{4}	T3
4	User	15	3	NULL	T4

Table	Attribute Table	
aid	attname	attype
1	A	text
2	B	int
3	C	int

Table	data_part_1
Contains rids (1, 17)	

Table	data_part_2
Contains rids (1, 6) ⋃ (18, 30)	

(b) Adding Rows without Partitioning

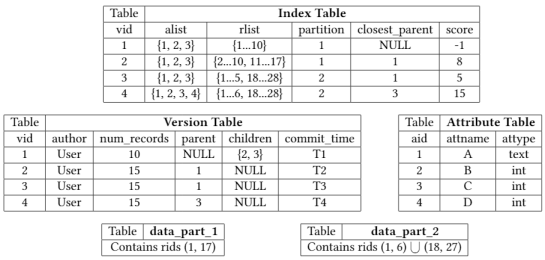


Table	Index Table				
vid	alist	rlist	partition	closest_parent	score
1	{1, 2, 3}	{1...10}	1	NULL	-1
2	{1, 2, 3}	{2...10, 11...17}	1	1	8
3	{1, 2, 3}	{1...5, 18...28}	2	1	5
4	{1, 2, 3, 4}	{1...6, 18...28}	2	3	15

Table	Version Table				
vid	author	num_records	parent	children	commit_time
1	User	10	NULL	{2, 3}	T1
2	User	15	1	NULL	T2
3	User	15	1	NULL	T3
4	User	15	3	NULL	T4

Table	Attribute Table	
aid	attname	attype
1	A	text
2	B	int
3	C	int
4	D	int

Table	data_part_1
Contains rids (1, 17)	

Table	data_part_2
Contains rids (1, 6) ⋃ (18, 27)	

(c) Adding Columns

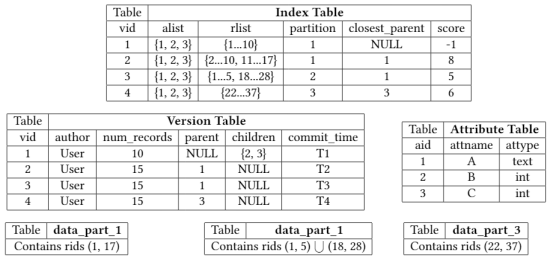


Table	Index Table				
vid	alist	rlist	partition	closest_parent	score
1	{1, 2, 3}	{1...10}	1	NULL	-1
2	{1, 2, 3}	{2...10, 11...17}	1	1	8
3	{1, 2, 3}	{1...5, 18...28}	2	1	5
4	{1, 2, 3}	{22...37}	3	3	6

Table	Version Table				
vid	author	num_records	parent	children	commit_time
1	User	10	NULL	{2, 3}	T1
2	User	15	1	NULL	T2
3	User	15	1	NULL	T3
4	User	15	3	NULL	T4

Table	Attribute Table	
aid	attname	attype
1	A	text
2	B	int
3	C	int

Table	data_part_1
Contains rids (1, 17)	

Table	data_part_1
Contains rids (1, 5) ⋃ (18, 28)	

Table	data_part_3
Contains rids (22, 37)	

(d) Adding Rows With Incremental Partitioning

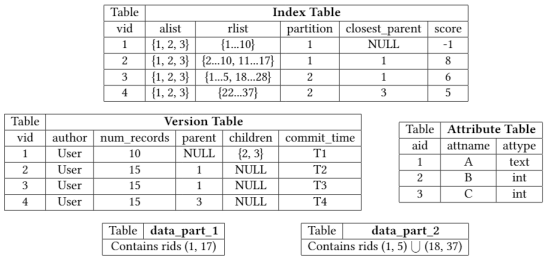


Table	Index Table				
vid	alist	rlist	partition	closest_parent	score
1	{1, 2, 3}	{1...10}	1	NULL	-1
2	{1, 2, 3}	{2...10, 11...17}	1	1	8
3	{1, 2, 3}	{1...5, 18...28}	2	1	6
4	{1, 2, 3}	{22....37}	2	3	5

Table	Version Table				
vid	author	num_records	parent	children	commit_time
1	User	10	NULL	{2, 3}	T1
2	User	15	1	NULL	T2
3	User	15	1	NULL	T3
4	User	15	3	NULL	T4

Table	Attribute Table	
aid	attname	attype
1	A	text
2	B	int
3	C	int

Table	data_part_1
Contains rids (1, 17)	

Table	data_part_2
Contains rids (1, 5) ⋃ (18, 37)	

(e) Adding Rows With Full Partitioning

Figure 6.2: CVDs of the different scenarios

Adding New Columns With Partitioning

Suppose the user attempts alter the schema of a table by adding a new column while adding enough additional rows that OrpheusDB decides to create a new partition. OrpheusDB will update the schema of all data table partitions inside the CVD. After the update, OrpheusDB returns to adding the new rows as described in the previous section.

Chapter 7

Conclusion and Future Work

7.1 Conclusion

In this report, we have detailed the implementation of OrpheusDB, highlighting the changes to the main operations. In particular, we adjusted the checkout and commit procedures to leverage all of the benefits of partitioning. Additionally, OrpheusDB now utilizes horizontal partitioning to operate on smaller data partitions instead of the entire table. The system also maintains the partitions, partitioning it on the fly or migrating the data across partitions once performance degrades with new data being added. We have also highlighted the performance of the different partitioning schemes, showing that it is highly dependent on the particular workload used.

7.2 Future Work

Weighted Partitioning Metric

In order to determine the partition boundaries in a CVD's history, OrpheusDB uses a metric function to score how similar each child version is to its closest parent version in the Lyresplit algorithm. However, a weighted function could be used on a particular CVD's history to work towards improved partition boundaries, translating into better performance overall. An example of a weighted function would be a popularity metric which would bias towards versions that are used more often, increasing performance by considering and optimizing for how often a version is used.

Modified Partitioning Schemes

Currently, OrpheusDB only considers partitioning horizontally across the rids, using the Lyresplit algorithm to determine how to partition the versions into different tables. While this partitioning scheme was shown to be more efficient in checkout operations, commit

operations do not perform as well on certain workloads such as when many different columns are added in a version. Introducing a dynamic, hybrid scheme would be one interesting improvement to the partitioning scheme. This scheme would allow the system to adapt more effectively to a dataset's workload and consider more facets of how we can partition a CVD such as column overlaps.

Schema Support

OrpheusDB has a built–in schema parser to help it identify which columns are being introduced in a particular version. This is primarily used during the commit phase to determine how to adjust the data partitions appropriately for proper operation. However, it is not robust to operations that modify the schema. While we addressed concerns about OrpheusDB handling schema updates, it still requires the user to specify what the update is. Ideally, we would want to implement a lightweight system that can detect how evolution of these columns occur without requiring a pairwise comparison between all of the preexisting columns of the dataset. This would allow OrpheusDB to track these changes and potentially store an virtual column that can be recomputed on the fly. Some examples of operations that would be interesting to support are normalization or combining two or more columns together to create a derived column.

Bibliography

[1] *Click*. May 2021. URL: https://click.palletsprojects.com/en/7.x/.

[2] *Open-source Version Control System for Machine Learning Projects*. May 2021. URL: https://dvc.org/.

[3] *OrpheusDB Website*. May 2021. URL: https://orpheus-db.github.io/.

[4] "OrpheusDB: A Lightweight Approach to Relational Dataset Versioning". en. In: 10 (2017). URL: https://people.eecs.berkeley.edu/~adityagp/papers/orpheus-tr.pdf.

[5] *PostgreSQL: The World's Most Advanced Open Source Relational Database*. May 2021. URL: https://www.postgresql.org/.

[6] *psycopg2*. May 2021. URL: https://pypi.org/project/psycopg2/.

[7] Huang Silu et al. "OrpheusDB: Bolt-on Versioning for Relational Databases". en. In: *Proceedings of the VLDB Endowment* 10.10 (2017), pp. 1427–1441. DOI: 10.14778/3342263.3342278. URL: http://www.vldb.org/pvldb/vol10/p1130-huang.pdf.

[8] *Splitgraph*. May 2021. URL: https://www.splitgraph.com/.

[9] Azari Sun and Turakhia. "Gallery: A Machine Learning Model Management System at Uber". In: *Proceedings of the 22nd International Conference on Extending Database Technology*. 2020, pp. 474–485.