ORPHEUSDB: AN ATTEMPT TOWARDS DATA VERSION CONTROL ON RELATIONAL DATABASE

BY

SILI HUI

THESIS

Submitted in partial fulfillment of the requirements
for the degree of Master of Science in Computer Science
in the Graduate College of the
University of Illinois at Urbana-Champaign, 2016

Urbana, Illinois

Adviser:

Assistant Professor Aditya Parameswaran

ABSTRACT

Relational database management systems, such as PostgreSQL[1], have been widely used in production environment for over decades. With stable and convenient user interfaces, they offer means to manipulate and manage millions of records, storing data like medical records, user information, banking transactions and etc. Yet, with the advent of modern data analytic tools and the need of more complicated data analysis, relational database management systems are limited as many of them do not provide interactive ways to manage dynamic changes in dataset for data scientist's need. Inspired by version control system *git*[2], we proposed and implemented a data version control system called OrpheusDB that will enable data versioning capability for existing relational database management system. This thesis will cover the design, architecture and implementation details of OrpheusDB. By all means, this work should serve as a reference to OrpheusDB at its current release.

To my parents, for their love and support.

TABLE OF CONTENTS

CHAPTER 1

INTRODUCTION

1.1 What is OrpheusDB?

OrpheusDB is a data version control system specialized at managing and tracking datasets of multiple versions. In the current version, it provides command line interface in Unix-like system to manage and query versions of data with underlying PostgreSQL[1] data storage.

1.1.1 What is data version control system?

From fraud transaction detection to targeted advertisement, and from user behavior study to mining trending topics on social media, the knowledge of extracting hidden information from large volume of data has became more important than ever in modern era. On its center lie a group of trained individuals whose job is to extract and transform collected data from one way to another. These datasets can be quite diverse, ranging from structured (tabular) data to unstructured data, from tiny kilobytes of samples to gigabytes even terabytes of logs. What adds to the complexity of managing data at scale is the challenge of how to effectively mange them as they evolve. Common operations such as normalizing certain field, reducing dimensionalities, transforming data into different space and etc. are executed frequently that can potentially lead to hundreds, sometimes thousands, of versions of data. It can get even more complex when there are multiple individuals contributing to the same dataset. Inevitably, three problems emerge.

- how to support collaboration of dataset?
- how to manage versions of data effectively?

- how to query across versions efficiently?

Data version control system is trying to answer these three questions at once.

1.1.2 What are some of the existing methods?

In general, there are two main approaches to manage and store versions of data. One way would be to store these versions in an underlying databases to take advantage of the query capabilities supported. Assume the data is structured, relational database systems, such as PostgreSQL[1], can store each version as an individual table. In return, it supports both complex queries that are highly optimized and collaborative user management. The downside of representing version as table is it 1) loses the "history" of versions and thus management may require extra effort, and 2) can create massive redundancies between versions that may require additional storage space. An different approach to mange version of data would be using existing version control systems such as *svn*[3] or *git*[2], storing each version as individual files with careful naming convention. An example of such would be **data_v1.csv**, **data_v2.csv**. By leveraging the version control system, the task to manage massive amount of versions becomes straight-forward. On the other hand, such leverage itself requires additional cost of further processing when desired queries are initiated. It is easy to find information such as which version is most up-to-dated and who committed it, yet it requires extra effort, a script or a program, to perform the queries and transformations that are not supported by version control system. Last but not least, data redundancy remains the same.

1.1.3 What problem does OrpheusDB try to solve?

Many studies[4, 5, 6, 7, 8] are trying to solve the similar problem. OrpheusDB, on the other hand, started from a different perceptive. What if we can take the advantage of both aforementioned approaches? Then, the question becomes simpler. ***Can we enable relational databases with the capability of data versioning?*** In such way, not only OrpheusDB can provide answers to those questions mentioned in section 1.1, but the

underlying infrastructure for relational databases can remain unchanged for those users who have already setup so.

1.1.4 What OrpheusDB offers additionally?

Inspired by the semantics of *git*[2], OrpheusDB is expected to have the advantage of both relational database system and version control system. In a nutshell, it mainly supports the following features.

- collaborative data analysis
- data versioning capability and management
- support of queries across versions

In the current version, we adopt PostgreSQL[1] as the underlying relational storage for OrpheusDB.

1.2 Related works

In recent years, numerous studies[4, 5, 6, 8] and efforts[9, 10, 11] were put into the field of data version control system.

Storage trade-off between various data versioning schemes is studied and proposed in 2015[4]. In the paper, Chavan et al.[6] proposed VQuel, a query language that supports analysis on versions of data and provenance information. The project Datahub[5], together with its storage engine Decibel[8], proposed a promising storage layer that broadens the functionality of version control.

Enterprises and organizations are also investing into the field. Flashback[11] enables querying against history information of the database. Other databases, such as Liquibase[9] and dbv[10], grant user the ability of merging and branching from versions.

CHAPTER 2

SYSTEM DESIGN

This chapter will cover how OrpheusDB is designed from ground up. It will start from data model and data presentation, then move to system design and each component. In the current version, OrpheusDB only support **PostgreSQL**[1] as the underlying relational database storage.

2.1 Data models

Data model is crucial in database version control system. In the following subsections, the philosophy of how data model is represented in OrpheusDB is discussed, while the implementation can be found in Chapter 4.

2.1.1 How to capture version information in relational database?

To leverage the advantage of relational database, OrpheusDB adopts the following storage model: each record and each version is associated with unique id, with version information maintained in a many-to-many relationship between versions and records. There are two main reasons for this adoption: it is relatively easy to understand and implement without much overhead; the version information is therefore transformed into a bipartite graph where optimization on effective partition can be applied and tuned according to use case.

2.1.2 Record id

Record id, *rid*, is the unique key for each record (row) of dataset. Given a schema in relational database, OrpheusDB augments each record with a special attribute *rid*. Such id is guaranteed to be unique within the same dataset. Fig 2.1 gives an example based on simple employment information schema. There are seven records in this example, each with self-explanatory data. In addition to the common fields $employee\_id, gender, salary$, an additional field of *rid* is assigned to each row, which serves as the unique identifier of this row.

2.1.3 Version id

Version id, *vid*, is the unique identifier of a particular version of dataset. Each version id is associated with a list of records. In OrpheusDB, instead of keeping the full data, only the *rid* of each individual record will be stored in its corresponding versions. Fig 2.2 gives an simple example of each version and its corresponding records stored. The field *vid* represents the id of each version. In addition to store record information, OrpheusDB also stores historical information. **Version graph**, a directed acyclic graph where each edge represents a derivation from an old version to a new version, is embedded in OrpheusDB using *vid*. Fig 2.3 shows an example of version graph, while its implementation can be found in Chapter 4. Apart from its simplicity, such storage choice enable further compaction on version mapping to save storage spaces.

2.2 Architecture

The follow subsections will cover architecture and the idea behind them.

2.2.1 Overview

The overview of system architecture can be found in Fig 2.4. Data model and data representation, as discussed in section 2.1, are embedded in the

rid	employee_id	age	salary
r1	10001	40	10000
r2	10002	35	8000
r3	10003	38	8000
r4	10004	50	7000
r5	10005	30	8000
r6	10006	28	6000
r7	10007	26	6000

Figure 2.1: **Table with augmented rid**

vid	rlist
v1	{r1,r2,r3}
v2	{r2,r3,r4}
v3	{r2,r3,r4,r5}
v4	{r1,r2}
v5	{r1,r2,r3,r4,r5}

Figure 2.2: **Version with corresponding rid**

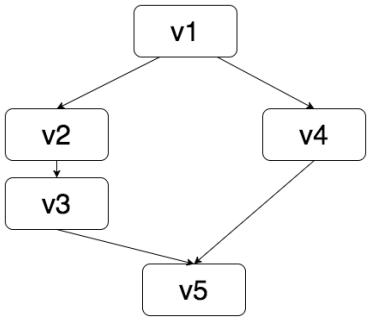

Figure 2.3: **Version Graph Illustration**

underlying relational database. Components that are on top of database are:

- Query translator: gets the input from user, either pre-defined command or SQL, parse it and send the parameters to Execution Engine
- Execution Engine: based on the parameters it receives, checks with Access Controller and invoke corresponding routine
- Access Controller: serves as a middle-ware before command or SQL are executed to check accessibility
- Record Manager: manipulates data records and *rid*
- Version Manager: manipulates version, version graph and *vid*

- Partition Optimizer: perform partition optimization

The design is based on couple of key ideas. Firstly, since the data volume can be huge, it is desired that the system is scalable enough to handle as volume increases. Decomposing SQL execution from the main parsing component not only grants the ability of parallel execution, but also provides flexibility and room for optimization to the parsing process. Secondly, such design also provides flexibility of the underlying relational database. If user decided to migrate from PostgreSQL to another database, the only changes need to be made in OrpheusDB are those who directly talk to the database, i.e record manager and version manager. Last but not least, this decomposition of component makes the system easy to maintain while leaves enough rooms for potential additional component in future releases.

2.2.2 Collaborative version dataset

The notion of **collaborative version dataset** (CVD), as is shown in 2.4, is the fundamental unit of operation in OrpheusDB. A single collaborative version dataset is initialized with a fixed schema and is shared by multiple authorized individuals. Each individual can either checkout a particular version, submit query, or commit changes he or she made. Since record in OrpheusDB is **immutable**, each commit of changes to CVD will result in a new version, with those changed records being appended with the new *rid*. The details of how CVD is implemented can be found in Chapter 4.

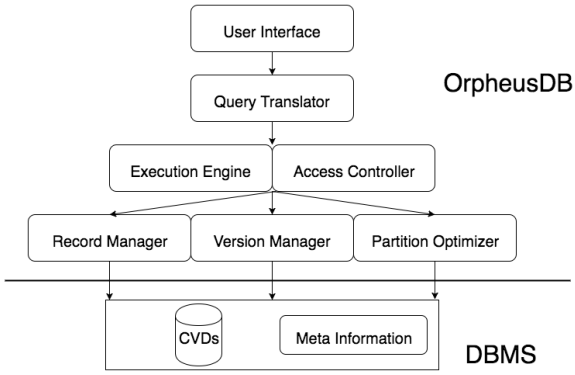

Figure 2.4: **Overview of OrpheusDB**

2.2.3 Query translator

Query translator is more than the layer between user and the execution engine. Since in the beginning of OrpheusDB , it is expected that complex queries on versions should be supported. Query translator, in addition regular SQL queries, should also have the ability to parse version-related queries and translate them to corresponding executable SQL statements. In the current version, most SQL queries are supported, including queries against data and version. Details on semantics and implementations can be found in section 3.4.6 and 4.4.5.

2.2.4 Execution engine

The execution engine is where all queries are gathered and send for execution. It takes in input from query translator, and invoke corresponding routines accordingly. This component is required since query can be related to version-related information as discussed in section 2.2.3 that cannot be directly fed into relational database.

2.2.5 Access controller

Access controller is the component responsible for user access. By default, the user who initialized the CVD is the owner and thus can grant access to other individual. In the current version, everyone can access CVD without permission check.

2.2.6 Record manager and version Manager

These two components are responsible for executing SQL statements to the underlying data storage. As operations coming in, record manager is responsible for updating *rid* related information, while version manager is for *vid*.

2.2.7 Partition optimizer

Partition optimizer is the component that periodically compact version to minimize storage cost. As more versions are committed, the cost to maintain version graph and version mapping will increase. This component will, ideally, compact version graph and version information while maintaining the effectiveness during queries. In the current version, the partition optimizer is not implemented.

CHAPTER 3

USER MANUAL

This chapter should serve as the manual of OrpheusDB in current release.

3.1 System requirements

The main OrpheusDB program comes in a python package. To successfully install, the following system requirements must be met.

- major Unix distribution
- python 2.7.x
- PostgreSQL 9.5.0 or later

The following are python dependencies.

- click 6.6 or later
- psycopg2 2.6.2 or later
- pandas 0.19.0 or later
- pyyaml 3.12 or later
- pyparsing 2.1.1 or later

The standard **setup.py** script comes with the release. Upon installation, needed dependencies should be automatically resolved.

3.2 Installation

An easier way to install OrpheusDB is through python package control **pip**. Any major release of **pip** should satisfy the need. To make sure it is correctly installed, one can type in the following command to terminal

```
pip --version
```

and it should print out the current version of installed **pip**. Package **setuptools** should appear in the list after typing in the following line.

```
pip list
```

Instructions on how to install **pip** can be found at its official website[12].

3.2.1 How to install?

Navigate to the OrpheusDB home directory, and type this command to terminal.

```
pip install .
```

The script will try to install OrpheusDB under the default python **sys.path** directory. To know where the current default python directory is, type the following command to terminal.

```
which python
```

The reason why we place OrpheusDB under python directory is simple. Since the main interface OrpheusDB exposed to user is command line interface, the main entry point should be placed in **PATH** so it can be invoked. After successful installation, the package name **orpheus** should appear in **pip** packages. To make sure this is the case, one can type in **pip list** to see if the package **orpheus** is in the list.

To make sure the installation is successful, a list of available command should be printed to the terminal by typing the following command.

```
dh --help
```

3.2.2 The $ORPHEUS_HOME$ parameter

The $ORPHEUS_HOME$ environmental parameter should have been set after following the above installation steps. The following command will print the current directory this parameter is pointing to.

```
echo $ORPHEUS_HOME$
```

By default, this parameter should be set in **.bashrc** file under $HOME$ directory. In case it is empty or not the correct one, user can manually add or change it to the correct directory in **.bashrc** file.

3.2.3 How to change installation parameters?

Installation parameters can be found in **setup.py** file. By default, the executable is named **dh**. In case it is taken, change it to any non-taken alias.

3.2.4 How to uninstall?

The advantage of installation through **pip** is it provides easy way to uninstall packages. By typing in the following command to terminal, **pip** will uninstall OrpheusDB.

```
pip uninstall orpheus
```

Again, by checking the **pip list** command, the package **orpheus** should not in there after successful un-installation.

3.3 Configurations

OrpheusDB will need the follow configurations for it to run.

3.3.1 OrpheusDB configuration

Before the start the program, system configuration should be set in **config.yaml** file. Particularly, the following fields should be set.

- host, the host name of PostgreSQL server or cluster
- port, the port number of the host
- orpheus_home, the home directory of Orpheus

Database name is not required. In fact, OrpheusDB provides interface to switch between databases under the same host. Any changes made to **config.yaml** will be reflected immediately, therefore if a user wants to use database from a different host, he or she only needs to change the corresponding line in **yaml** file.

3.3.2 PostgreSQL configuration

In the current version, OrpheusDB assumes security is addressed at system level. Since every command will construct the channel to PostgreSQL[1] server on the fly, the server itself should be free to access without the need to pass along password. By default, OrpheusDB assumes the server was initialized with the easiest setup with default **pg_hba.conf** file. An example of such initialization would is given as following.

```
postgres -D /path/to/database
```

However, if user needs to connect OrpheusDB with a more complicated setup, the connector object in **db.py** can be overwritten with customized connection parameters. More details can be found in section 4.4.2.

3.4 User interface

The user interface for OrpheusDB is a command line interface written in python package called **click**[13]. To start the interface, open terminal and input this line.

```
dh --help
```

This command will list all the predefined commands available. To run anyone of them, type this line to terminal with the word **command** replaced.

```
dh command
```

Add **--help** to the end will display what parameters this command takes in and its usage.

In the current version of implementation, OrpheusDB implements the following commands.

- checkout
- commit
- config
- create_user
- db
- drop
- init
- ls
- run
- whoami

The functionality of each command will be discussed further in the following subsections, together with what parameters it takes. Detailed documentation can be found through adding **--help** to each of the command or reading through **main.py** file.

3.4.1 create_user

Description:

This command promotes user to registered as a OrpheusDB user. The registered user name will then be pushed to the database server as with **SUPERUSER** credential. The command exists so that OrpheusDB knows to whom it should assign ownership of Collaborative Version Dataset (cvd). Discussion of cvd representation can be found in 2.2.2. In the current version, there is no password protection, yet this may change in a later version of release.

Examples:

Example 1. *A command line prompt will ask user to enter user name.*

```
dh create_user
```

3.4.2 config

Description:

This command behaves similar to the login function. It serves as the way to switch between different OrpheusDB users. As mentioned in 3.4.1, OrpheusDB fully trust the credential of current user and thus will not ask for password.

Examples:

Example 2. *A command line prompt will ask user to enter user name he or she wants to switch to.*

```
dh config
```

3.4.3 init

Description:

The **init** command initialize CVD in database from CSV file. Upon finishing execution, all records in the CSV file will be the first version of the CVD and the number of version in this dataset will be 1. OrpheusDB does not infer data type and column name from CSV. Instead, user needs to provide such information by either providing the name of another table within the same database server, from which data type and column name will be copied, or providing a schema file. Fig 3.1 gives an example of comma separated schema file with first field being the column name and second being the type. More complexed schema parsing is possible by overwriting the **schema_parser.py** file. When a schema is passed in as a file, it should not contain reserved column names. Otherwise an exception will be thrown.

Section 4.1.5 covers information about reserved column names. In current version, special column attributes such as **PRIMARY KEY**, **NULL** are not supported.

Arguments:

- **INPUT**, the file path to CSV file that user wish to initialized as CVD. Path can be either absolute path or relative path
- **DATASET**, the name of the new CVD

Options:

- **-t**, the name of the table from which the schema will be used
- **-s**, the file path to comma separated schema file. Example can be found in fig 3.1

```
employee_id, int
name, text
age, int
salary, int
```

Figure 3.1: **Sample Schema File**

Examples:

Example 3. *The following command initialize* ***test1.csv*** *file into CVD called* ***dataset1*** *using the schema file* ***sample_schema***

```
dh init test1.csv dataset1 -s sample_schema
```

Example 4. *If user want to initialized using the schema of an existing table, this example will load file* ***test2.csv*** *into CVD called* ***dataset2*** *using the schema of* ***exist_table****. Records stored in* ***exist_table*** *remains unchanged since this command will only copy its schema.*

```
dh init test1.csv dataset1 -t exist_table
```

3.4.4 checkout

Description:

Command **checkout** checks out version(s) of a CVD to a destination, either a CSV file on local disk or a table in PostgreSQL. This command is to materialize only a small portion of the dataset (the desired versions) without touching any records beyond specified. Version identifier is the **vid** of this CVD. When checking out to CSV, it will create the new file if the file path does not exist, or it will overwrite it. When check out to table, it will create new table if table with the same name does not exist, or it will raise exception and abort.

Upon finishing, OrpheusDB will remember the file or table it touched, and save it for later when the **commit** command is invoked. In case a checked-out file is overwritten by a later checkout, OrpheusDB will only honor the most recent operation, meaning "last write win". Detail implementation of how OrpheusDB keeps track of checked-out files can be found in section 4.2 and section 4.4.9. Only CSV file format is supported in the current version.

Arguments:

- **DATASET**, the name of the CVD user wishes to check out version(s) from

Options:

- **-v**, the version user wishes to checkout. If multiple versions checkout is desired, user can enter multiple -v. See examples below. This field is required.
- **-t**, the name of the table user wishes to checkout to. It will create a new table if name does not exist, or it will abort the process and an exception will be thrown. If this name is a reserved table name, an exception will also be thrown. For a complete list of reserved table name, please refer to section 4.1.4. Either this field or **-f** need to be set by user.

- **-f**, the path to the check-out file in CSV format. It the file does not exist it will create a new one, or it will overwrite that file. Either this field or **-t** need to be set by user.
- **-d**, the delimiter used when checking out versions to file. By default, it is set to comma.
- **-h**, the flag for header. If user wish to include header when checking out versions to file, including this flag will make the first line of checked out file the header.

Examples:

Example 5. *This line checks out version **1** from CVD called **dataset1** into table **v1_table**. After execution, the checked-out table will have the same schema as specified in **dataset1**.*

```
dh checkout dataset1 -v 1 -t v1_table
```

Example 6. *If user wants to checkout to a file, the next command will checks out version 2 from the same CVD into file **/Users/demo/v2.csv**.*

```
dh checkout dataset1 -v 2 -f /Users/demo/v2.csv
```

Example 7. *Absolute path is not required, for instance, this line will checkout version 2 from CVD called **dataset2** to the current working directory and create or overwrite a file called **v2.csv***

```
dh checkout dataset2 -v 2 -f v2.csv
```

Example 8. *Multiple version checkout can be done by adding more **-v** to the command line.*

```
dh checkout dataset2 -v 1 -v 2 -f /Users/demo/v2.csv
```

3.4.5 commit

Description:

This command will commit previous checked-out (through **checkout**) version, either a table or a CSV file, to its corresponding dataset. Detail

implementations of how OrpheusDB keeps track of checked-out files can be found in section 4.2 and section 4.4.9. If the source is not previously checked-out, an exception will be raised. Upon invoking, command will first check if there are any changes made by comparing the committing records against previous checked-out records. If there are at least one record changed, a new version will be committed to the corresponding CVD, with the changed records appended and version graph updated. If there is no change, this command will also commit the file or table as user wishes. Implementation of how this function is achieved can be found in section 4.3.2. One special case is the committing of a subset of previous checked-out records, which will also resolve a new version.

Options:

- **-m**, the the commit message. Message with space in between words can be wrapped in single quotation. This field is required.
- **-t**, the table user wishes to commit. If this table does not exist, or this name is a reserved table name, or this table is not previously checked-out table, an exception will be thrown. For a complete list of reserved table name, please refer to section 4.1.4. Either this field or **-f** needs to be set by user.
- **-f**, the CSV file (path) users wishes to commit. If this file is not previously checked-out file or this file does not exist, an exception will be thrown and the operation will be aborted. Either this field or **-t** needs to be set by user.
- **-d**, the delimiter used when committing file. By default, it is set to comma.
- **-h**, the flag for header. If first line of the committing file is header, including this flag will tell *Orpheus* do not parse the first line.

Examples:

Example 9. *This line will commit the previous checked-out table* ***v1_table*** *to CVD* ***dataset1*** *with the commit message according to example 5.*

```
dh commit -t v1_table -m 'first commit'
```

Example 10. *This line will commit the previous checked-out file* ***v2.csv*** *to CVD* ***dataset2*** *with the commit message according to example 7.*

```
dh commit -f v2.csv -m 'second commit'
```

3.4.6 run

Description:

The **run** command expose query interface to user. A command line prompt will ask user to input query statement. Only a single line separated by \n or **Enter** will be executed. If user wishes to execute more than a single line, use the **-f** option. As is discussed in 2.2.3, OrpheusDB supports major SQL queries with the following additional features.

- query on one or more versions of CVD without materialization.
- query on unknown versions of CVD with predicates
- query on version graph traversal

To support these queries, a special SQL semantic is implemented allowing a richer syntax to the regular SQL syntax. The new semantic includes key words like **VERSION** and **CVD**, together with a list of predefined functions to helper user navigate through version graph. Sections 4.4.5 will cover more information about the implementation details. In the current release, only **SELECT** statement is supported from only one **CVD**.

Options:

- **-f**, the file path to a SQL file. Upon invoking, each line, separated by **\n**, is executed sequentially. An exception will be thrown if the file does not exist or syntax is wrong.

Examples:

Example 11. *This example will select employee_id from version 1 and version 2 of the CVD* ***dataset1****.*

```
SELECT employee_id
FROM VERSION 1,2 OF CVD dataset1;
```

Example 12. *To select records that satisfy certain constraint, the* ***WHERE*** *clause can be useful.*

```
SELECT employee_id
FROM VERSION 1,2 OF CVD dataset1
WHERE salary > 5000;
```

Example 13. *To query against each version with in a* CVD*, user will need to specify a* ***GROUP BY*** *clause and the field* ***vid****. The following example counts the number of relations inside each version from dataset1.*

```
SELECT count(employee_id)
FROM CVD dataset1
GROUP BY vid;
```

Example 14. *To query versions with certain predicates from a* CVD*, the* ***SELECT*** *clause and the* ***GROUP BY*** *clause need to have the field* ***vid****. The next example finds all versions in dataset1 containing precisely 100 Employees with age 25.*

```
SELECT vid
FROM CVD dataset1
WHERE age = 25
GROUP BY vid
HAVING count(employee_id) = 100;
```

Example 15. OrpheusDB *also supports queries against version graph traversal. The predefined function* ***ancestor*** *will return a list of versions that is committed before. The following example will find all versions that are within 2 commits of version 1 of CVD dataset1 which have more than 100 employees.*

```
SELECT vid
FROM CVD dataset1
WHERE vid = ANY(ancestor(1, 2))
GROUP BY vid
HAVING count(employee_id) > 100;
```

3.4.7 db

Description:

Command **db** will display the database name OrpheusDB currently is connected to.

Options:

- **-d**, the database name user wishes to connect to. When set, OrpheusDB will switch to that particular database and remember it. An exception will be thrown when the given database does not exist or connection is not open.

Examples:

Example 16. *This command will set the database to* ***backup****.*

```
dh db -d backup
```

3.4.8 drop

Description:

Command **drop** will try to drop a CVD. In the current implementation, there is no way to revert back a dropped CVD. Extra caution may needed when invoking this command.

Arguments:

- **DATASET**, the name of the CVD the current user wishes to drop

Examples:

Example 17. *It will drop the* CVD *of name* ***dataset1****. A command line prompt will ask the user if he or she would like to proceed.*

```
dh drop dataset1
```

3.4.9 ls

Description:

Command **ls** takes no argument, and it will display list of CVD names the current user is owner of. It will show nothing if the current user does not own a CVD.

3.4.10 whoami

Description:

this command also takes no argument, and it will display the current user name OrpheusDB remembers.

CHAPTER 4

IMPLEMENTATIONS

This chapter will go over the implementation of OrpheusDB in detail.

4.1 PostgreSQL related

As is discussed before, OrpheusDB is built on top of PostgreSQL[1] storage. This section will go over what is actually stored and how they are stored.

4.1.1 Collaborative version dataset

Each collaborative version dataset (cvd) is decomposed in to three components and stored separately. These three components are **datatable**, **indextable**, and **versiontable**.

Datatable is the table where all the records are stored. In addition to the data fields from original data, it include a **SERIAL PRIMARY KEY** field called **rid**, which is guaranteed to be unique. Fig 4.1 gives an example of table representation. When **checkout** happens, however, OrpheusDB does not checkout **rid** field to confuse end user. How this is implemented can be found in section 4.3.1.

Indextable is the table where mapping between versions and records are stored. Each such table consist of only two fields, **rlist** and **vlist**. Both types are **ARRAY**. The field **rlist** stores **rid** from datatable, and the field **vlist** stores **vid** from versiontable. Fig 4.2 gives an example of such table representation.

Versiontable is the table that stores all meta information related to versions. Its attribute names and attribute types can be found in fig 4.4. **Version Graph**, as discussed in 2.1.3, is embedded using fields **parent**

rid	employee_id	age	salary
1	66000001	25	6500.0
2	66000002	30	7500.0
3	66000003	25	7000.0
4	66000004	21	5000.0

Figure 4.1: **Sample Datatable**

vlist	rlist
{1}	{1,2,3,4}
{2}	{1,2,3}

Figure 4.2: **Sample Indextable**

vid	author	num_records	parent	children	create_time	commit_time	commit_msg
2	abc	3	{1}	{}	2016-11-16 15:11:28.348151	2016-11-16 15:11:28.348151	first commit
1	abc	4	{-1}	{2}	2016-11-16 14:17:08.873155	2016-11-16 14:17:08.873175	init commit

Figure 4.3: **Sample Versiontable**

and **children**. Fig 4.3 shows a sample of versiontable representation in PostgreSQL. Version with parent field being **{-1}** is the source in of this CVD in version graph, meaning it is not derived from another version.

The naming convention for these three table is hard-coded and can be found in **const.py** file. For a CVD **dataset1**, its datatable, indextable, versiontable are named **dataset1_datatable**, **dataset1_indextable**, **dataset1_versiontable** respectively.

vid	integer
author	text
num_records	integer
parent	ARRAY
children	ARRAY
create_time	timestamp without time zone
commit_time	timestamp without time zone
commit_msg	text

Figure 4.4: **Version table and its attributes**

4.1.2 Data transfer

To support command such as **checkout** and **commit**, data transfer between local file system and PostgreSQL table must be done in a effective manner. Fig 4.5 shows how this is implemented in OrpheusDB. PostgreSQL provide SQL interface that support **COPY** statement. OrpheusDB takes advantage of this syntax and provides the most effective way to transfer data to and from database. Detail of syntax can be found at PostgreSQL COPY website[14].

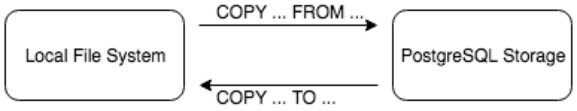

Figure 4.5: **Data transfer between PostgreSQl and file system**

4.1.3 User information in PostgreSQL

Each newly created OrpheusDB user is automatically created as a **SUPERUSER** in database. Whenever the newly created user login through command **config**, OrpheusDB will allocate space in database to stored user related information. Particularly, each user has its own **schema**[15] that stores such information. The name of the schema is named after the user name. In the current version, each user schema only one attributes, the table called **datasets**, which records all the CVD the current user is the owner of. The implementation of these functionalities can be found in **main.py** and **db.py** files, or section 4.4.1 and 4.4.2 below.

4.1.4 Reserved table name

As is discussed in 5.2.2, many table names are hard coded easier reference. In general, user would like to avoid naming tables after these reserved tables to avoid confusion. The following table names are reserved and used internally. Naming table after them will throw exception when trying to execute command **checkout** or **commit**.

- "[text]_datatable", used in CVD

- "[text]_indextable", used in CVD
- "[text]_versiontable", used in CVD
- "tmp_table", used internally for data transfer

4.1.5 Reserved column names

End user has the freedom to initialize a CVD with desired column names, except the following reserved column names.

- "vid"
- "rid"

Upon receiving these reserved column names during command **init**, an exception will be thrown.

4.2 Meta information

Like other major systems, ORPHEUSDB keeps track of its program execution information in a hidden folder. Such information, called **meta information** is updated every time when specific commands are executed. In order for the whole user experience to be connected, storing certain information on disk is required since every call to the ORPHEUSDB interface is a individual process that will be terminated upon finishing.

4.2.1 .meta folder

All meta information is placed under **.meta** folder under ORPHEUSDB home directory by default. It contains the follow properties.

- config
- logs
- tracker
- users

The **config** file contains context information for program execution. Reading from it is the first thing OrpheusDB will do whenever a new command is invoked. It contains information such as host, port, OrpheusDB home directory and etc. Additional discussion can be found in section 4.4.8.
The **logs** folder contains all the program execution logs. Like any standard program package, OrpheusDB will record its execution for trouble-shooting.
The **tracker** file keeps track of command **checkout**. As is discussed in both 3.4.4 and 3.4.5, this file is how OrpheusDB remembers what has been checked out and what is the checkout destination. During the **commit** command, OrpheusDB will first look into this file to find related information regarding to-be-committed file or table, and then execute commands accordingly. Additional discussion can be found in section 4.4.9
The **users** folder contains all registered user information though OrpheusDB. Since there is no password protection in the current version, only user name is stored. Plans and ideas of how to enhance the security of OrpheusDB will be discussed in Chapter 5.

4.3 Commands

This section will cover both **checkout** and **commit** implementations in details.

4.3.1 **checkout** command

The **Checkout** command discussed in section 3.4.4 provides simple interfaces for user to interact with version without the need to materialize the whole dataset. The flow of how it is implemented can be found in 4.6. Since each row in **datatable** is augmented with **rid** field, **checkout** will filter out this field, and checkout records based on the rest of attributes, i.e, the original data attributes.

First, OrpheusDB will take the input version ids to find corresponding **rid** in **indextable** of that CVD, and then find the corresponding records in **datatable**. Checking-out to table is using the following SQL statement.

```
SELECT ... FROM ... INTO ... WHERE rid = ANY( ... ::int[]);
```

When checking-out to file, OrpheusDB internally will first invoke the checkout routine to a temporary table called "tmp_table", and then use the **COPY** statement, discussed in section 4.1.2, to dump the "tmp_table" to the desired location. Upon finishing checking-out, OrpheusDB will write related information into **.meta/tracker** file.

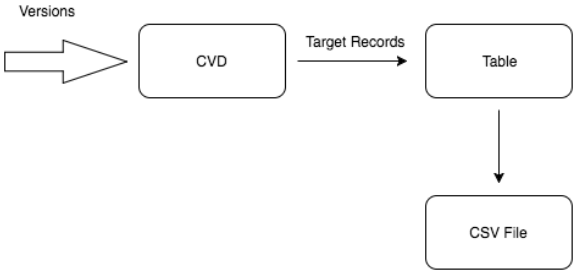

Figure 4.6: **The flow of Checkout**

4.3.2 **commit** command

The **commit** command discussed in section 3.4.5 is one of the most important features in OrpheusDB. It takes in a checked-out (through **Checkout** command) source, either file or table, finds out changes made to it, and commits these changes to the corresponding CVD. The flow of this command be found in fig 4.7. To commit a table, *Orpheus* will find the what versions were checked out through **.meta/tracker** file, find out what changes were made, and then commit a new version with the corresponding records to CVD. There are two steps to identify what records should be committed to a new version.

- find records that were changed
- find records that remain the same

The implementation to find out what records were changed is done through the SQL statement.

```
(SELECT ... from ...) EXCEPT (SELECT ... from ...);
```

Effectively, these records will be append to the **datatable** and assigned wit new **rid**. To find out what records remain the same, the following SQL statement is used.

```
SELECT ... FROM ... INNER JOIN ... on ...;
```

The records selected from this statement, combined with the records from appended new records, are all the records for this commit. Such information will then be fed back to to update the corresponding **versiontable** and **indextable**. Committing a file follows the similar procedure, except OrpheusDB will first convert the file into a temporary table called "tmp_table" through the **COPY** statement and proceed with the table, similar to the idea in **checkout** command.

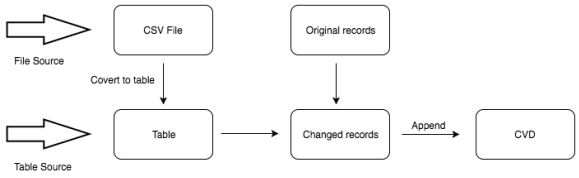

Figure 4.7: **The flow of Commit**

4.4 Files

Detailed documentation can be found in each corresponding code. In this section, a selected number of files and implementations will be further discussed.

4.4.1 main.py

This file the main entrance for *Orpheus*, which contains all the commands specifics. Below is the example of command **db**.

```
1 @cli.command()
2 @click.option('--database', '-d')
3 @click.pass_context
4 def db(ctx, database):
5     if database:
6         ctx.obj['database'] = database
```

```
7         UserManager.write_current_state(ctx.obj)
8 %     click.echo('using:%s' % ctx.obj['database'])
9 }
```

Listing 4.1: A simple example

OrpheusDB commands are included in a group named *cli*. Effectively, all new commands should also be placed in this group. To add more command, developer can refer to Click website [13]. After include line 3 above, the first parameter passed in each command will be the click context, where the object *ctx.obj* is the context of current execution for OrpheusDB. More information about what is in OrpheusDB context can be found in section 4.4.8.

4.4.2 db.py

This file contains class called **DatabaseManager**, which serves as the channel between OrpheusDB and underlying database, PostgreSQL[1] in this case. It takes in a OrpheusDB context dictionary as argument upon constructing and has the following import class fields.

- connect, the connector object from python package Psycopg2[16]
- cursor, part of the connector object that takes in SQL statements and execute them
- config, the OrpheusDB context dictionary. Refer to section 4.4.8 for more information.
- connect_str, the connecting string to construct connector object

Classes such as **RelationManager** and **VersionManager** depends on the **cursor** field to execute their respective SQL statements. Further discussion of these two classes will be found below.

4.4.3 relation.py

This file contains only one class called **RelationManager**, in which all methods that touch records in **datatables** are implemented. When

constructing, it takes in a **DatabaseManager** object, and use the channel to manipulate tables. The documentation within code has specifics about each method.

4.4.4 version.py

The file contains only one class called **VersionManager**, in which all methods that need to update **versiontable** and **indextable** are implemented. Similar to **RelationManager**, it takes in a **DatabaseManager** object as argument upon constructing, and will use the channel within it to manipulate table.

4.4.5 orpheus_sqlparser.py

This file contains SQL parser to support version query syntax. It contains a class called **SQLParser** with methods to transform richer SQL syntax, discussed in 3.4.6, to executable SQL statements.

In general, it follows the following steps.

1. identify all column attributes touched by this statement
2. for each column attribute, find the table this column resides in
3. replace CVD names with table names in **SELECT** and **FROM** clause
4. add **WHERE** clause statement if needed

The following are examples of what happened underneath.

Example 18. *The example 13 is given as*

```
SELECT count(employee_id)
FROM CVD dataset1
GROUP BY vid;
```

*Parser will first identify columns touched, in this case **vid** and **employee_id** respectively, and then alias the table they reside in as **i** and **d**. It then will change the **FROM** clause to aliases, and prefix an alias before every touched column. Since the statement touches more tables, a*

hard-coded ***WHERE*** *clause,* ***d.rid = ANY(i.rlist)****, is added. After such transformation, the original statement becomes executable SQL statements.*

```
SELECT count(d.employee_id)
FROM dataset1_datatable d, dataset1_indextable i
WHERE d.rid = ANY(i.rlist)
GROUP BY vid;
```

Example 19. *The example 15 is given as*

```
SELECT vid
FROM CVD dataset1
WHERE vid = ANY(ancestor(1, 2))
GROUP BY vid
HAVING count(employee_id) > 100;
```

Similar to the above example, parser will first identify ***vid*** *and* ***employee_id*** *and their corresponding table,* ***i*** *and* ***d****, then prefix every appearance of touched column with alias. Since this statement also touches two tables, an additional* ***WHERE*** *clause is added. After such transformation, the original statement becomes executable SQL statements.*

```
SELECT i.vid
FROM dataset1_indextable i, dataset1_datatable d
WHERE d.rid = ANY(i.rlist) AND i.vid = ANY(ancestor(1, 2))
GROUP BY i.vid
HAVING count(d.employee_id) > 2;
```

In addition to the parsing, the **SQLParser** class also provides a list of predefined functions that are loaded into database. The following lists their names and corresponding utilities. The documentation can be found within the file for each function.

- ancestor, return the **ancestor** of a particular *vid* in version graph
- descendant, return the **descendant** of a particular *vid* in version graph
- v_diff, return the difference between two versions in terms of *rid*

4.4.6 schema_parser.py

The file schema_parser.py is used in **init**, as discussed in section 3.4.3. It has only one static method **get_attribute_from_file** that return a list of string with column name and a list of string with column types. Exception will be thrown when the parsed column type is not valid PostgreSQL[1] type or the column name is reserved. In current version, it assumes file is in CSV format with out special column attribute such as **PRIMARY KEY**.

4.4.7 config.yaml

The file has only two fields: the host, the port number. Any changes to this file will be reflected immediately to OrpheusDB. Since this file is the first file OrpheusDB will read from in user interface, it should be placed under $ORPHEUS_HOME$ directory, otherwise a exception will be thrown.

4.4.8 .meta/config

The **.meta/config** file stores the OrpheusDB context. Before the subroutine of a command is invoked, the context will first inherit **host** and **port** fields from the **config.yaml** file, and read the context from last execution. After execution is finished, this file is over-written if needed. Apart from fields inherited, it has the following important fields.

- database, the database current OrpheusDB is talking to
- user, the current user name
- meta_info, the path to tracker file, default to **.meta/tracker**
- log_path, the path to log file where all the logs are stored
- orpheus_home, the path to OrpheusDB home directory

To change the **host** and **port**, user need to change the **config.yaml**.

4.4.9 .meta/tracker

This file is mainly used for command **checkout** and **commit**. It is dictionary object and has the following important fields.

- file_map, a dictionary that stores path to checked out file as key and CVD information as value
- table_map, a dictionary that stores table name as key and CVD information as value

All successful **checkout** will add its entry to the file. For **commit** command, OrpheusDB will look into the file and retrieve needed information.

CHAPTER 5

SUMMARY

5.1 Summary

OrpheusDB explores the possibility of data version control on existing relational database system, and presents a feasible implementation towards this effort.Based on PostgreSQL[1] storage layer, it provides command line interface allowing user to interact and query versioning information without much storage cost overhead. In addition to support the SQL syntax, also enriches the syntax to support declarative and complex query.

With in-depth discussions in design, manual, implementation details and future work, this thesis should serve as a reference to OrpheusDB at its current version.

5.2 Future work

The following are future directions that can be built on top of current implementation.

5.2.1 Secure user information

The security in OrpheusDB is kept at minimum as of current version. When commands **checkout**, **commit** and **drop** are invoked, OrpheusDB does not question the credential of the end user, which means there is nothing OrpheusDB can do to stop unauthorized party to manipulate data when system level security breach happens.

OrpheusDB can store encrypted password into PostgreSQL. In dong so, however, it would promote user to enter password frequently since each

command is executed in an individual process. Alternatively, OrpheusDB can stored the credential somewhere, **.meta** folder for example, so that all following commands (processes) can verify the identity of the user. Yet, it would expose credential to other parties who can easily access such destination. Neither approach is desired.

One suggestion would be use session, which grants the current user access upon successful login before expiration.

5.2.2 Secure access controller

OrpheusDB as of current version, does not handle access control. As is discussed in section , all CVD exists under **public** schema, meaning any role with valid access to PostgreSQL[1] can have access to them. It remains an open question how to store and use access information to best integrate with PostgreSQL built-in access control.

5.2.3 Multiple table attribute

Due to the table representation, OrpheusDB does not support data with special column attributes such as **PRIMARY KEY** to avoid confusion in datatable. A future implementation may add an additional table to CVD that stores such special column attributes.

5.2.4 Diverse version query

The version query supported in section 3.4.6 is read-only, i.e. **SELECT**. Key SQL queries such as **Update**, **INSERT** is not supported yet. A more diverse SQL parser and command should be implemented to replace the current one.

5.2.5 Concurrent control

Concurrent control is a issue with OrpheusDB when dealing with large dataset. PostgreSQL defines four level of transaction isolation[17], but they

are yet to be included in the current version. Future work would be to incorporate PostgreSQL level of concurrent control into OrpheusDB.

5.2.6 Support of multiple relational databases

For the current release, OrpheusDB supports only PostgreSQL[1] as the underlying data storage. In practice, end users may want to use other relational databases, many of which have subtle differences from PostgreSQL. To support those, a separate driver maybe needed.

REFERENCES

[1] "PostgreSQL," https://www.postgresql.org/.

[2] "Github," https://github.com/.

[3] "Subversion," https://subversion.apache.org/.

[4] S. Bhattacherjee, A. Chavan, S. Huang, A. Deshpande, and A. Parameswaran, "Principles of dataset versioning: Exploring the recreation/storage tradeoff," *Proceedings of the VLDB Endowment*, vol. 8, no. 12, pp. 1346–1357, 2015.

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

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

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

[8] M. Maddox, D. Goehring, A. J. Elmore, S. Madden, A. Parameswaran, and A. Deshpande, "Decibel: The relational dataset branching system," *Proceedings of the VLDB Endowment*, vol. 9, no. 9, pp. 624–635, 2016.

[9] "Liquibase," http://www.liquibase.org/.

[10] "dbv," https://dbv.vizuina.com/.

[11] 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.

[12] "Pip," https://pip.pypa.io/en/stable/installing/.

[13] "Click," http://click.pocoo.org/5/.

[14] "PostgreSQL copy syntax,"
https://www.postgresql.org/docs/9.5/static/sql-copy.html.

[15] "PostgreSQL Schema,"
https://www.postgresql.org/docs/9.5/static/ddl-schemas.html.

[16] "Psycopg2 package," http://initd.org/psycopg/docs/.

[17] "PostgreSQL transaction,"
https://www.postgresql.org/docs/9.5/static/ transaction-iso.html.