miniDB Documentation (v4 - 2022)

The miniDB project is a minimal and easy to expand and develop for RMDBS tool, written exclusively in Python 3. MiniDB's main goal is to provide the user with as much functionality as possible while being easy to understand and even easier to expand. Thus, miniDB's primary market are students and researchers that want to work with a tool that they can understand through and through, while being able to implement additional features as quickly as possible.

The miniDB project consists of multiple files, with the ones that encapsulate the main ideas and functionality being the following:

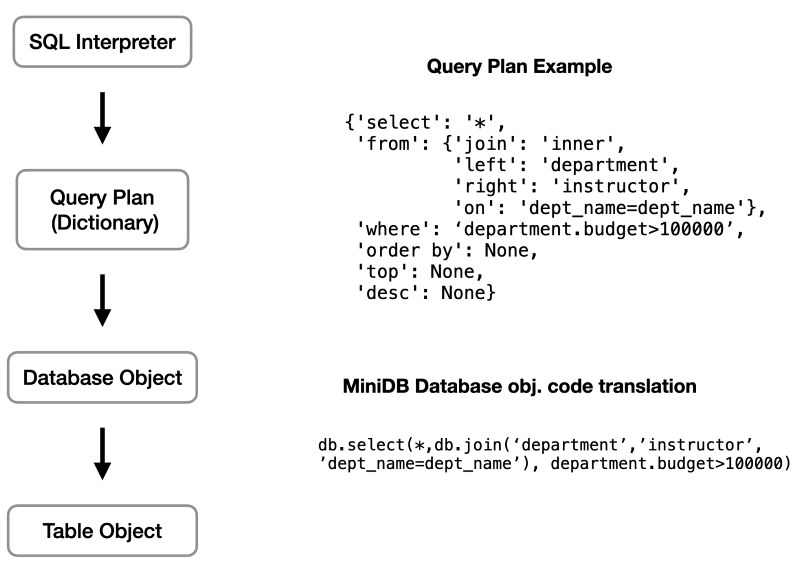
- mdb.py
- miniDB/database.py
- miniDB/table.py

Other files, including the following will be presented later:

- miniDB/btree.py
- miniDB/joins.py
- miniDB/dashboard.py
- miniDB/misc.py

Introduction

The workflow that corresponds to the way data is stored and served using miniDB is presented in the following figure.



MiniDB’s three main interfaces are:

1. The SQL interpreter that is implemented in the mdb.py file.
2. The database interface that is implemented in the miniDB/database.py file and acts as an intermediary between interfaces 1 and 3.
3. The table interface that handles all the per table functionality.

mSQL (miniDB SQL)

Like every other RDBMS tool, miniDB supports the use of a slightly altered SQL implementation. This implementation (let’s call it mSQL) has many things in common with PostgreSQL. The following table is a side by side comparison between PostgreSQL and mSQL.
Most of the table related SQL commands are identical, with slight changes regarding JOIN queries. Database related commands are separate, being closer to the psql commands

Action	PostgreSQL	miniDB code (python3)
Databases		
Create DB	CREATE DATABASE name	cdb name # load if exists, create if not # saves are automatic
Save DB	SAVE DATABASE name	
Connect to DB	\c name	
Delete DB	DROP DATABASE name	rmdb name
List all DBs	\l	lsdb
List all tables in DB	\dt	lstb
Tables		
Create table	CREATE TABLE name (column_name1 column_type1, …)	CREATE TABLE name (column_name1 column_type1, …)
Delete table	DROP TABLE name	DROP TABLE name
Cast column to type	ALTER TABLE name ALTER COLUMN column TYPE INTEGER(int)	CAST column FROM name TO int
Import table from .csv file	CREATE TABLE name (column_name1 column_type1, …); COPY name FROM file.csv;	IMPORT TABLE name FROM file.csv
Export table to .csv file	EXPORT TABLE name TO csvname.csv;	EXPORT TABLE name TO csvname.csv

<table>
<tr><td>Insert into table</td><td>INSERT INTO table VALUES
(row);</td><td>INSERT INTO table VALUES
(row);</td></tr>
<tr><td>Delete from table</td><td>DELETE FROM table WHERE
condition;</td><td>DELETE FROM table WHERE
condition;</td></tr>
<tr><td>Update values of
table</td><td>UPDATE table SET column=value
WHERE condition;</td><td>UPDATE table SET column=value
WHERE condition;</td></tr>
<tr><td>Select from where
...</td><td>SELECT [DISTINCT] columns/*
FROM table WHERE condition
[limit k] [ORDER BY column
[ASC|DESC]]
[limit K]</td><td>SELECT [DISTINCT] columns/*
FROM table WHERE condition
[limit k] [ORDER BY column
[ASC|DESC]] [limit K]</td></tr>
<tr><td>Select from join
where</td><td>SELECT * FROM table1 INNER
JOIN table2 ON condition;</td><td>SELECT * FROM table1 INNER
JOIN table2 ON condition;</td></tr>
<tr><td>Join with multiple
tables (*)</td><td>SELECT columns FROM T1
INNER JOIN (SELECT * FROM T2
INNER JOIN T3 ON col=col) ON
col=col</td><td>SELECT columns FROM T1
INNER JOIN (SELECT * FROM T2
INNER JOIN T3 ON col=col) ON
col=T2.col</td></tr>
<tr><td colspan="3">Locks</td></tr>
<tr><td>Lock and unlock
table</td><td>BEGIN WORK;
LOCK TABLE name IN
EXCLUSIVE MODE;
...
COMMIT WORK;</td><td>LOCK TABLE name;
...
UNLOCK TABLE name;</td></tr>
<tr><td colspan="3">Indexes</td></tr>
<tr><td>Create index</td><td>CREATE INDEX name ON
table(column) using BTREE;</td><td>CREATE INDEX name ON
table(column) using BTREE;</td></tr>
<tr><td>Delete index</td><td>DROP INDEX name;</td><td>DROP INDEX name;</td></tr>
</table>

(*) Note about joins: Multiple types of joins have been implemented, including “inner”, “outer”, “left”, “right” and algorithms like “smj” (sort merge join) and “inlj” (index nested loop join). In miniDB you can force the usage of a specific algorithm by using the keyword in front of the “join” part (“sm join” and “inl join” - like “inner join”).

Codebase

mdb.py - mSQL Interpreter

The mdb.py file is the file that is run in order to create a connection with an existing miniDB database or to create a new one. This file is also the one that contains all the logic with regards to interpreting mSQL commands. To start a miniDB instance, run:

```
DB=DB_NAME python3.9 mdb.py
```

You need to specify a name for the database that you wish to connect to. If a database with that name does not exist, it will be created. The REPL window will look something like this:

```
            _       _ _____  ____
           (_)     (_)  __ \|  _ \
  _ __ ___  _ _ __  _| |  | | |_) |
 | '_ ` _ \| | '_ \| | |  | |  _ <
 | | | | | | | | | | | |__| | |_) |
 |_| |_| |_|_|_| |_|_|_____/|____/   202x - vX.X

(DB_NAME)>
```

At this point, the user can write any mSQL command that is supported and get an output.
The mSQL interpreter works in a very specific and detached way with respect to the rest of the miniDB code. Its one and only job is to create a query plan in the form of a dictionary. In order to inspect the query plan of a specific query, the user need to prefix the mSQL query with the EXPLAIN keyword. The result will look something like this:

```
(smdb)> explain select * from instructor inner join department on
dept_name=dept_name

{'select': '*',
 'from': {'join': 'inner',
            'left': 'instructor',
            'right': 'department',
            'on': 'dept_name=dept_name'},
 'where': None,
 'order by': None,
 'limit': None,
 'desc': None}
```

The returning dictionary is the query plan, meaning that it accurately presents the steps that will be taken by the miniDB code in order to generate the resulting table.

Query Plan - Recursiveness

A very important concept that needs to be understood is the recursive nature of the Query Plan. As it is presented above, a query plan is a dictionary (a key-value store). Each key specifies an argument that will be supplied to the underlying python code in order for the mSQL command to be executed. However, if an argument is a new dictionary (like in the example above where the JOIN clause is a separate dictionary that is the value of the 'FROM' key), this nested dictionary is evaluated separately. When it produces a result (a Table object - will be discussed later on), the result takes the place of the nested dictionary. Then, the "parent" dictionary (in this case the 'SELECT' command) is evaluated and run, with its 'FROM' key having a Table object as value. This interim step can be thought of as a new dictionary that looks something like this:

```
{'select': '*',
 'from': TABLE_OBJECT,
 'where': None,
 'order by': None,
 'limit': None,
 'desc': None}
```

This way, the interpreter can interpret infinitely nested commands, providing advanced functionality for a myriad of use-cases (multijoins, interim tables "view-like" tables etc.).

miniDB/database.py - Database class

The Database class is the backbone of the miniDB pipeline. It is responsible for orchestrating all the commands that will be run in a specific database, while updating all the metadata that is needed to keep miniDB consistent and usable.

Each mSQL action (ex. 'SELECT', 'INSERT INTO' etc.) maps to a function of the Database class with the same name (If the action keyword is spaced - like 'INSERT INTO', the space is replaced with an underscore - mSQL 'INSERT INTO' maps to Database.insert_into). This is crucial, since it allows us to avoid writing a lengthy wrapper that maps each possible action to a separate but not consistently named Database function.

Managing the underlying metadata is the other important task that the Database class is responsible for. Currently, only 4 meta_tables are present. These are:

- meta_length -> keeps track of the number of rows of each table in the database (meta_tables are excluded)
- meta_locks -> keeps track of all the active locks in the database (meta_tables are excluded)

- meta_insert_stack -> Keeps track of all the empty indexes in a table, in order to fill them in case a new row is inserted (if no indices then the new row is appended at the end of the table) (meta_tables are excluded)
- meta_indexes -> Keeps track of all the available indexes, as well as their name, the table and the column that they index, as well as their type (only B-Tree is currently supported).

miniDB/table.py - Table class

The Table class is the class that is responsible for executing each operation that is specified by the Database class. It contains functions that do not follow strict naming rules, since they are not mapped in the same way that the Database functions are mapped to the mSQL interpreter. Each Table function executes a specific command in a specific way, meaning that extending the functionality of the miniDB project (when that involves creating new features) starts by creating a new Table function that implements the new feature. For example, unlike the unified DB.select function, the Table class implements multiple select functions (Table._select_where and Table._select_where_with_btree). These functions implement different things. In this case, the Table._select_where function implements a linear search for a given condition and the Table._select_where_with_btree implements a B-Tree based search.