CS-6360 Database Design

Programming Project: Relational Database Internals

Due: Aug 3, 2020 9:00AM

1. Overview

The goal of this project is to implement a (very) rudimentary database engine that is loosely based on a hybrid between MySQL and SQLite, which I call **DavisBase**. Your implementation should operate entirely from the command line and API calls (no GUI).

Your database will only need to support actions on a single table at a time, no joins or nested query functionality is required. Like MySQL's InnoDB data engine (SDL), your program will use *file-per-table* approach to physical storage. Each database table will be physically stored as a separate single file. Each table file will be subdivided into logical sections of fixed equal size call *pages*. Therefore, each table file size will be exact increments of the global `page_size` attribute, i.e. all data files must share the same `page_size` attribute. The test scenarios for grading will be based on a page_size of 512B. You may make `page_size` be a configurable attribute of other sizes, but you must support a page size of 512 Bytes. Once a database is initialized, your are *not* required to support a reformat change to its `page_size` (but you may implement such a feature if you choose).

You may use any programming language you like, but all coding examples will be in **Java**.

2. Requirements

2.1. Prompt

Upon launch, your engine should present a prompt similar to the MySQL **`mysql>`** prompt or SQLite **`sqlite>`** prompt, where interactive commands may be entered. Your prompt text may be hardcoded string or user configurable. It should appear something like:

```
davisql>
```

2.2. Required Supported Commands

2.2.1. Summary

Your database engine must support the following DDL, DML, and QDL commands. All commands should be terminated by a semicolon (;). Each one of these commands will be tested during grading.

DDL

- `SHOW TABLES` – Displays a list of all tables in DavisBase.
- `CREATE TABLE` – Creates a new table schema, i.e. a new empty table.
- `DROP TABLE` – Remove a table schema, and all of its contained data.

DML

- `INSERT INTO` – Inserts a single record into a table.
- `DELETE FROM` – Delete one *or more* recrods from a table.
- `UPDATE` – Modifies one *or more* records in a table.

QDL

- "`SELECT-FROM-WHERE`" -style query
  - You must support up to two `WHERE` conditions connected by a Boolean
  - You **do not** have to implement query `JOIN` commands or nested queries. All queries will be single table queries.
  - You **do not** have to support `ORDER BY`, `GROUP BY`, `HAVING`, or `AS` alias.

System Commands

- `EXIT` – Cleanly exits the program and saves all table system information in non-volatile files to disk.

3. Data Definition Language (DDL) Commands

The detailed syntax for the commands summarized in section 2.2.1 (DDL) is described in this section.

Show Tables

```
SHOW TABLES;
```

Displays a list of all table names in the database. The result set should appear on the command line as plain text in a single column

```
table_name
----------------
Employees
Departments
Dogs
```

Note: this is equivalent to the query:

```
SELECT table_name FROM davisbase_tables;
```

Create Table

Create a table schema

```
CREATE TABLE table_name (
    column_name1 data_type1 [NOT NULL] [UNIQUE] [PRIMARY KEY],
    column_name2 data_type2 [NOT NULL] [UNIQUE],
    ...
);
```

Create the table schema information for a new table. In other words, create a new file whose name is ***table_name*.tbl**. Initially, the file will be one page in size and have no record entries. Additionally, add appropriate entries to the database catalog tables **davisbase_tables** and **davisbase_columns** that define meta-data for the new table. The **ordinal_position** of a column should its order in which the column appeared in the table's create statement, the first column in the create statement has **ordinal_position = 1**, the second has **ordinal_position = 2**, etc.

Reject the command if the table already exists.

Note that unlike the official SQL specification, a **DavisBase** table **PRIMARY KEY** (if one is declared) must be (a) a single column, and (b) the first column listed in the **CREATE TABLE** statement. This is to simplify your command parsing.

The only column constraints that you are required to support are **PRIMARY KEY**, **UNIQUE** , and **NOT NULL** (to indicate that NULL values are not permitted for a particular column). If the first column If a column is defined as **NOT NULL**, then its **davisbase_columns.IS_NULLABLE** attribute will be False, otherwise, it will be True.

Even though a user specified **`PRIMARY KEY`** is optional, your database engine shall include the internal key **`rowid`**, an auto-increment integer, on which the table file is organized. If a record is ever deleted, its **`rowid`** may never be reused.

Your table column definitions should support the data types in section 3.2, Table 2.

You are *not* required to support any type of **`FOREIGN KEY`** constraint, since multi-table queries (i.e. Joins) are not supported in DavisBase.

Drop Table

Removes all table definition information from the database catalog—(a) delete the table file itself, and (b) remove meta-data for the table in the database catalog files (`davisbase_tables.tbl` and `davisbase_columns.tbl`).

```
DROP TABLE table_name;
```

Use your application to remove **`table_name.tbl`** from the file system. If your meta-data is stored in `davisbase_tables.tbl` and `davisbase_columns.tbl`, then remove the associated meta-data by using delete commands.

```
DELETE FROM davisbase_tables WHERE table_name = table_name;
DELETE FROM davisbase_columns WHERE table_name = table_name;
```

4. Data Manipulation Language (DML) Commands

The detailed syntax for the commands summarized in section 2.2.1 (DML) is described in this section.

Insert Row Into a Table

Insert a new record into the indicated table.

```
INSERT INTO table_name (column_1,column_2,...) VALUES (value_1,value_2, ...);
```

String values must be quoted. Numeric values should not be quoted. Use the keyword **`NULL`** (without quotes) to represent the null value.

Any column not included in the column list shall be inserted with a value of **`NULL`**.

If no column list is provided, then *n* values that are supplied will be mapped onto the first *n* columns in **`ordinal_position`** order.

Prohibit inserts that do not include the primary key column (if it exists) or do not include a NOT NULL column. For columns that allow NULL values, INSERT INTO TABLE should parse the keyword NULL (unquoted) in the values list as the special value NULL.

Reject the insert if:

- A non-unique value is supplied for a column defined as **`UNIQUE`**. This includes any primary key.
- A non-null **`NOT NULL`** column has an explicit **`NULL`** value or no value is provided. This includes any primary key.

Delete Row From a Table

```
DELETE FROM table_name WHERE condition;
```

Delete one or more rows/records from a table given some condition. Note that, like queries, ***`condition`*** may contain up to two comparisons connected by a Boolean.

Deletion of a record may be accomplished in two ways: (a) Remove the row/record location(s) from the page header array. It is not necessary to "zero out" the bytes of record data in the page cells. Nor is it necessary to "defragment" the page to recover newly free bytes, orf (b) Add an extra Boolean "deletion" byte to each record who value indicates whether the record is active or not.

You do *not* have to restructure any B-trees due to the deletion of a cell. Simply, unreference the cell by removing it's offset location from the page header or set the record's deletion byte to True.

Update Row(s) in a Table

```
UPDATE table_name SET column_name = value WHERE condition;
```

Modify a column value in one or more rows/records from a table given some condition. For every record that matches ***condition***, change its value of ***column_name*** to be ***value***.

Note that, like queries, ***condition*** may contain up to two comparisons connected by a Boolean.

5. Query Definition Language (QDL) Commands

The detailed syntax for the commands summarized in section 2.2.1 (QDL) is described in this section.

Query Table

```
SELECT [column_list | * ]
FROM table_name
WHERE condition;
```

DavisBase query syntax is similar to formal SQL. Query output may be displayed to the terminal (stdout) in either SQLite-style column mode or MySQL-style column mode. The differences between DavisBase query syntax and standard SQL query syntax is described below.

If **SELECT** has the * wildcard, it will display all columns in **ORDINAL_POSITION** order.

The **WHERE** ***condition*** should be able to accomodate up to two sub-conditions that are connected with a Boolean operator, e.g. **WHERE** ***condition_1*** **[AND|OR]** ***condition_2***;

6. Storage Definition Language (SDL)

Table data must be saved to files so that your database state is preserved after you exit the database. When you re-launch **DavisBase**, your database engine should be capable of loading the previous state from table files.

Detailed SDL directory and file information can be found in the supplement document "*DavisBase File Format Guide: Storage Definition Language (SDL)*". A summary follows below.

The file formats shall be based on the documented format of SQLite (https://www.sqlite.org/fileformat2.html), with the following simplifications.

- ~~There are only two types of files, table files and index files~~. [Update: There is only *one* type of file, **TABLE FILES**.] You do not have to support index files, lock-byte files, freelist files, payload overflow files, or pointer map files.
- Unlike SQLite, instead of storing all data structures in one large file, DavisBase stores each table and each index as a separate file (i.e. *file-per-table* strategy, like MySQL).
- Instead of the database catalog being stored in the single **`sqlite_master`** table (whose root page is the first block of every SQLite database file), there should be *two system tables* created by default **`davisbase_tables`** and **`davisbase_columns`**, which encode the database schema information, i.e. the "database catalog". This will eliminate the need to parse the SQL "CREATE TABLE" text every time you need to insert a row. You may choose to store meta-data in your own file format, although it is strongly encouraged to use **`davisbase_tables`** and **`davisbase_columns`**.
- Since each table has a dedicated file, unlike SQLite, an overall global Database Header is not needed.
- The root page of a table file may be determined dynamically by tracing the parent page of page 0 upward to the root. You may alternatively include a column (**`root_page`**) to the **`davisbase_tables`** table to indicate which page is the root page.

Store all your files in a directory structure whose top-level name is **`data`**. Store all your table ~~and index~~ files in **`data/user_data`**. Store the two database catalog files in **`data/catalog`**.

```
data/
    |
    +---- catalog/
    |             |
    |             +-- davisbase_tables.tbl
    |             +-- davisbase_columns.tbl
    |
    +-- user_data/
                  |
                  +-- table_name_1.tbl
                  +-- table_name_1.column_name_1.ndx
                  +-- table_name_1.column_name_2.ndx
                  +-- table_name_2.tbl
                  +-- table_name_2.column_name_1.ndx
                  +-- table_name_3.tbl

etc.
```