CS-6360 Database Design

Programming Project #2: Database Files and Indexing

Instructor: Chris Irwin Davis

Due: August 3, 2016 11:59PM

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 queries. Like MySQL's InnoDB data engine (SDL), this project will use *file-per-table* approach to physical storage. Each database table will be physcially stored as a separate 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. You may make page_size be a configurable attribute, but you must support a page size of **512 Bytes**. The test scenarios for grading will be based on a page_size of 512B. Once initialized, a database is *not* required to support a reformat change to its page_size.

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

2. Requirements

2.1. Prompt

Upon launch, your engine should present a prompt similar to the **`mysql>`** 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. Supported Commands

2.2.1. Overview

Your database engine must support the following DDL, DML, and VDL 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.
- Note that you **do not** have to implement `ALTER TABLE` schema change commands.

DML

- `INSERT INTO TABLE` – Inserts a single record into a table.
- `UPDATE` – Modifies one *or more* records in a table.

VDL

- "`SELECT-FROM-WHERE`" -style query
- `EXIT` – Cleanly exits the program and saves all table information in non-volatile files.
- Note that you **do not** have to implement query `JOIN` commands. All queries will be single table queries.

2.2.2. Supported Commands

The detailed syntax for the above commands is described below.

```
SHOW TABLES;
```

Displays a list of all table names in the database. Note: this is equivalent to the query:
SELECT

```
CREATE TABLE table_name (
    column_name1 INT PRIMARY KEY,
    column_name2 data_type2 [NOT NULL],
    column_name3 data_type3 [NOT NULL],
    ...
);
```

Create the table schema information for a new table. In other words, add appropriate entries to the system **davisbase_tables** and **davisbase_columns** tables that define the described **CREATE TABLE** and create the associated **.tbl** data file.

Note that unlike official the SQL specification, a **DavisBase** table PRIMARY KEY must be (a) a single column, (b) the first column listed in the CREATE statement, and (c) an INT data type. This requirement greatly simplifies the implementaiton. In most commercial databases a unique key, which is an INT data type, is automatically created in the background if you do not explicitly create the PRIMARY KEY as a single column INT. In commercial databases this "default" key is called the **rowid** or **row_id**.

Your table definition should support the following data types. All numbers should be represented as binary byte sequences in Big Endian order.

The only table constraints that you are required to support are PRIMARY KEY and NOT NULL (to indicate that NULL values are not permitted for a particular column). If a column is a primary key, its **davisbase_columns.COLUMN_KEY** attribute will be "**PRI**", otherwise, it will be NULL. If a column is defined as NOT NULL, then its **davisbase_columns.IS_NULLABLE** attribute will be "**NO**", otherwise, it will be "**YES**".

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

```
INSERT INTO TABLE table_name VALUES (value1,value2,value3,…);
```

Insert a new record into the indicated table.

If *n* values are supplied, they will be mapped onto the first *n* columns. Prohibit inserts that do not include the primary key column or do not include a NOT NULL column. For columns that allow NULL values, INSERT INTO TABLE should parse the keyword NULL in the values list as the special value NULL.

```
SELECT *
FROM table_name
WHERE column_name operator value;
```

Query syntax is similar to formal SQL. The result set should display to stdout (the terminal) formatted like a typical SQL query. The differences between DavisBase query syntax and SQL query syntax is described below.

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

Query output may be displayed in either SQLite-style column mode or MySQL-style column mode.

3. File Formats

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.

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

- Instead of storing all data structures in one large file, DavisBase stores each table as a separate file (i.e. file-per-table strategy, like MySQL).
- There is only one type of file, table pages. You do not have to support index files, lock-byte files, freelist files, payload overflow files, or pointer map files.
- 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.
- Since all tables have a dedicated file, the SQLite Database Header (§1.2) is not needed.

Store all your table files in a directory named **`data`**.

```
/data
|
+-/davisbase_tables.tbl
|
+-/davisbase_columns.tbl
|
+-/user_table_name1.tbl
|
+-/user_table_name2.tbl
|

etc.
```

3.1. B-tree Pages

The following tables replaces the “B-tree Pages Header Format” in SQLite File Format document §1.5.

Offset from beginning of page	Content Size (bytes)	Description
0x00	1	The one-byte flag at offset 0 indicating the b-tree page type. • A value of 2 (0x02) means the page is an interior index b-tree page (*not used*). • A value of 5 (0x05) means the page is an interior table b-tree page. • A value of 10 (0x0a) means the page is a leaf index b-tree page (*not used*). • A value of 13 (0x0d) means the page is a leaf table b-tree page. Any other value for the b-tree page type is an error.
0x01	1	The one-byte integer at offset 3 gives the number of cells on the page.
0x02	2	The two-byte integer at offset 5 designates the start of the cell content area. A zero value for this integer is interpreted as 65536.
0x04	4	The four-byte page number at offset 8 is the right-most pointer. This value appears in the header of interior b-tree pages only and is omitted from all other pages.

The following description and table replaces the “B-tree Cell Format” in SQLite File Format document §1.5.

The format of a cell depends on which kind of b-tree page the cell appears on. The following info shows the elements of a cell, in order of appearance, for the various b-tree page types.

Table B-Tree Leaf Cell (in pages whose page type header is 0x0D):

- A 2-byte **`SMALLINT`** which is the total number of bytes of payload
- A 4-byte **`INT`** which is the integer key, a.k.a. "rowid"
- The payload.
  - 1-byte TINYINT that indicates the number of columns *n*.
  - *n*-bytes which are Serial Type Codes, one for each of *n* columns
  - binary column data
- DavisBase does not support overflow payload.

Table B-Tree Interior Cell (in pages whose page type header is 0x05):

- A 4-byte **`INT`** page number which is the left child pointer.
- An **`INT`** which is the integer key

3.2. Record Formats

Replace the “Serial Type Codes Of The Record Format” table in SQLite document with the following table. The **`VARINT`** data type does not have to be supported.

Serial TypeCode	Database Data Type Name	Content Size (bytes)	Description
0x00	NULL	1	Value is a 1-byte NULL
0x01	NULL	2	Value is a 2-byte NULL
0x02	NULL	4	Value is a 4-byte NULL
0x03	NULL	8	Value is a 8-byte NULL
0x04	TINYINT	1	Value is a 1-byte twos-complement integer.
0x05	SMALLINT	2	Value is a 2-byte twos-complement integer.
0x06	INT	4	Value is a 4-byte twos-complement integer.
0x07	BIGINT	8	Value is an 8-byte twos-complement integer.
0x08	REAL	4	A single precision IEEE 754 floating point number
0x09	DOUBLE	8	A double precision IEEE 754 floating point number
0x0A	DATETIME	8	An unsigned LONG integer that represents the specified number of milliseconds since the standard base time known as "the epoch”. It should display as a formatted string string: YYYY-MM-DD_hh:mm:ss, e.g. `2016-03-23_13:52:23`.
0x0B	DATE	8	A datetime whose time component is 00:00:00, but does not display.
0x0C + *n*	TEXT		Value is a string in ASCI encoding (range 0x00-0x7F) of length *n*. For the purposes of this database you may consider that the empty string is a NULL value, i.e. empty strings do not exist. The null terminator is not stored.

3.3. Database Catalog (meta-data)

The DavisBase Catalog consists of two tables containing meta-data about each of the user table. You may optionally choose to include meta-data about the two catalog files in the catalog itself. These two tables (and their assocated implmentation files) have the following table schema, as if they had been created via the normal **CREATE** command.

```
CREATE davisbase_tables (
   rowid INT,
   table_name TEXT,
   record_count INT,   -- optional field, may help your implementation
   avg_length SMALLINT -- optional field, may help your implementation
);
```

```
CREATE davisbase_columns (
   rowid            INT,
   table_name       TEXT,
   column_name      TEXT,
   data_type        TEXT,
   ordinal_position TINYINT,
   is_nullable      TEXT
);
```

If you choose to include these two tables in the catalog itself, their content would intially be:

```
SELECT * FROM davisbase_tables;

rowid   table_name
--------------------------
1       davisbase_tables
2       davisbase_columns
```

```
SELECT * FROM davisbase_columns;

rowid   table_name        column_name     data_type  ordinal_position is_nullable
-------------------------------------------------------------------------------
1       davisbase_tables  rowid            INT        1                NO
2       davisbase_tables  table_name       TEXT       2                NO
3       davisbase_columns rowid            INT        1                NO
4       davisbase_columns table_name       TEXT       2                NO
5       davisbase_columns column_name      TEXT       3                NO
6       davisbase_columns data_type        TEXT       4                NO
7       davisbase_columns ordinal_position TINYINT    5                NO
8       davisbase_columns is_nullable      TEXT       6                NO
```