CS-6360 Database Design

Programming Project #2: Database Files and Indexing
Instructor: Chris Irwin Davis
Due: August 1, 2018 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 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. 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 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 examples and support 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 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.
- `CREATE INDEX` – Create an index on a *single* column of an existing table.
- `DROP INDEX` – Remove an index.
- Note that you **do not** have to implement `ALTER TABLE` schema change commands.
- The database catalog (i.e. meta-data) shall be stored in two special tables that should exist by default: **`davisbase_tables`** and **`davisbase_columns`**.

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.

VDL
- "`SELECT-FROM-WHERE`" -style query
  - 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 nested queries.
  - You **do not** have to support `ORDER BY`, `GROUP BY`, `HAVING`, or `AS` alias.
- `EXIT` – Cleanly exits the program and saves all table and index information in non-volatile files to disk.

2.2.2. Supported Commands

The detailed syntax for the above commands is described below.

Show Tables

```
SHOW TABLES;
```

Displays a list of all table names in the database. 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] [PRIMARY KEY],
    column_name2 data_type2 [NOT NULL],
    ...
);
```

Create the table schema information for a new table. In other words, crate 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.

Note that unlike the official SQL specification, a **DavisBase** table **`PRIMARY KEY`** (if one is declared) must be (a) a single column, (b) the first column listed in the **`CREATE`** statement. This is to simplify your command parsing. Even though **`PRIMARY KEY`** is optional, your database engine should automatically create the internal key **`row_id`**, an auto-increment integer, on which the table file is organized. If a record is ever deleted, its **`row_id`** may never be reused.

Your table definition should support the data types in section 3.2, Table 2. 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 dedicated data type **`0`** (i.e.false), otherwise, it will be the dedicated data type **`1`**.

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—the table file itself as well as all indexes on the table and entries in the database catalog.

```
DROP TABLE table_name;
```

Insert Row Into Table

```
INSERT INTO TABLE [(column_list)] 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 (unquoted) in the values list as the special value NULL.

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

Delete one or more rows/records from a table given some condition.

Query Table

```
SELECT *
FROM table_name
WHERE condition;
```

DavisBase 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 (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.

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. You do not have to support, 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 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.

- Since all tables have a dedicated file, the SQLite Database Header (§1.2) is not needed.

- You may assume that the root page of the meta-data table files is page #1. User table files may either have the root page always be page #1 of the associated file, or you may add 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_2.tbl
   +-/table_name_3.tbl

etc.
```

3.1. B-tree Pages

Note that the term "Table B-tree" in this section refers to what your textbooks calls "B+ Trees", i.e. records are stored only in leaf pages. This terminology is similar to the SQLite documentation.

Page Header

Table 1 below replaces the "B-tree Pages Header Format" in SQLite File Format document §1.5. All DavisBase table pages have an 8-byte page header, immediately followed by a sequence of 2-byte integers that are the page offset location for each data cell.

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. • 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. • 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 1 gives the number of cells on the page.
`0x02`	2	The two-byte integer at offset 2 designates the start of the cell content area.
`0x04`	4	The four-byte integer page pointer at offset 4 has a different role depending on the page type: • Table/Index interior page - rightmost child page pointer • Table/Index leaf page - right sibling page pointer (0xfFFFFFFFF) if the node is the rigfhtmost leaf page.
`0x08`	2*n*	An array of 2-byte integers that indicate the page offset location of each data cell. The array size is 2*n*, where *n* is the number of cells on the page. The array is maintained in key-sorted order.

Table 1 - Page Header Format

B+-tree and B-tree page cells are added starting from the end of each page. Each new cell is added at the highest available memory address in the page.

Table File Cells

Table files are implemented as B+-trees. Table file page cell content is different for leaf pages versus interior pages. *Leaf page* cells contain the table's record data. *Interior page* data cells will contain only **`rowid`** keys and left child page pointers.

Index File Cells

Index files are implemented as B-trees. Index file page cell content is *almost* the same for leaf pages and interior pages. Both *Leaf page* cells and *Interior page* cells contain the index key (the column value on which the index file B-tree is organized) and **`rowid`** pointer back to the record in its associated table file. Each index file interior page cell additionally has a left child page pointer.

	Cell Header			Cell Content
Page Type	**Page # of Left Child**	**Payload Size**	**Rowid**	**Payload**
	4-byte Int	2-byte Int	4-byte Int	Variable
Table Leaf (0x0d)		✓	✓	✓
Table Inner (0x05)	✓		✓	
Index Leaf (0x0a)		✓		✓
Index Inner (0x02)	✓	✓		✓

Table 2 - B-tree Cell Format

3.2. Record Formats

1-byte Date Type Code	Data Type Name	Content Size (bytes)	Description
0x00	NULL	0	Value is NULL (content is zero size)
0x01	TINYINT	1	Value is a 1-byte twos-complement integer.
0x02	SMALLINT	2	Value is a big-endian 2-byte twos-complement integer.
0x03	INT	4	Value is a big-endian 4-byte twos-complement integer.
0x04	BIGINT	8	Value is an big-endian 8-byte twos-complement integer.
0x05	FLOAT	4	A big-endian single precision IEEE 754 floating point number
0x06	DOUBLE	8	A big-endian double precision IEEE 754 floating point number
0x07	DATETIME	8	A big-endian 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 with no spaces: YYYY-MM-DD_hh:mm:ss, e.g. `2016-03-23_13:52:23`.
0x08	DATE	8	A datetime whose time component is 00:00:00, but does not display.
0x08-0x09	—	—	Reserved for future use. These serial type codes will never appear in a well-formed database file.
0x0A + *n*	TEXT		Value is a string in ASCI encoding (value range 0x00-0x7F) of length *n*. The null string terminator (0x00) is not used to end a string.

Table 2 - Data Types and Their Implementation

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
   root_page 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
```