Programming Project #2: Database Files and Indexing

CS-6360 Database Design
Instructor: Chris Irwin Davis

1. Overview

The goal of this project is to implement a (very) rudimentary database engine that is based on a simplified file-per-table variation on the SQLite file format, which we call **DavisBase**. Your implementation should operate entirely from the command line and possibly API calls (no GUI).

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 your implementation must capable of supporting 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**.

DavisBase data is encoded in two different kinds of files—tables files and index files. Tables and Indices are both represented by a single DB file. Each DB file is stored as a file in the underlying OS.

Each DB file is comprised of one ore more *pages* (a virtual subdivision of the file). All pages of a file are the same size. For example, if the page size is set to 1024 bytes (1kb), then each DB file size is some multiple of 1024 bytes.

- Each page in a Table file is a node in a B+ tree, either interior or leaf.
- Each page in an Index file is a node in a B tree, either interior or leaf.

The location of each element in a page is referenced with a "page offset" value (i.e. the number of bytes from the beginning of the page that the element is located.

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 of Supported Commands

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.
  - `PRIMARY KEY.`
  - `AUTOINCREMENT.`
  - `UNIQUE.`
  - `DEFAULT.`
  -
- `DROP TABLE table_name;` – Remove a table schema, and all of its contained data.
- `CREATE [UNIQUE] INDEX ON table_name (column_list);` – Create an index table from table_name using column_list as a key.

  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 table_name [(column_list)] VALUES (value_list);`
  - Inserts a single record into a table.
- `DELETE FROM table_name [WHERE condition];`
  - Deletes one *or more* records from a a table.
- `UPDATE table_name SET column_name = value [WHERE condition];`
  - Modifies one *or more* records in a table.

VDL

- `SELECT column_list FROM table_name [WHERE condition];`
  - Note that 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 multiple WHERE conditions. The condition is a single *`column comparison_operator value`* clause.
  - 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 information to disk in non-volatile files.

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 (
    row_id 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 the official SQL specification, a **DavisBase** table PRIMARY KEY must be (a) a single column, (b) the first column listed in the CREATE statement, (c) named **`row_id`**, and (d) an INT data type. This requirement greatly simplifies the implementation. 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 usually called the **`rowid`** or **`row_id`**. In the DavisBase implementation, the row_id is *explicit* and required.

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 "**`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.

Drop Table

```
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 in the values list as the special value NULL.

Delete Row From Table

```
DELETE FROM TABLE table_name WHERE row_id = key_value;
```

Delete a single row/record from a table given the **`row_id`** primary key.

Query Table

```
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 (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.

- 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 are only two types of files, table files and index files. You *do not* have to 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" or "meta-data". This will eliminate the need to parse the SQL "CREATE TABLE" text every time you need to insert a row.
- Since each table has a dedicated file, the SQLite Database Header (§1.2) is not needed.

Store all your table files in a directory named **`data`**. It should have two sub-directories, catalog and user_

```
/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. data (records) are stored only in leaf pages. This terminology is consistent with the SQLite documentation. Each node in the B-tree is a *page* of the file.

Page size. The range of valid page sizes are 2^n bytes, where $8 \leq n \leq 16$. The default page size should be 4096 bytes.

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 two's complement signed integer at offset 1 gives the number of cells on the page. Thus, the maximum number of cells on a page is 127.
0x02	2	The two-byte two's complement signed integer at offset 2 designates the start of the cell content area. A zero value for this integer is interpreted as 65536.
0x04	4	The four-byte signed integer page pointer at offset 4 has a different role depending on the page type: • Table or Index interior page - rightmost child page number reference. • Table leaf page - right sibling page number reference. • Index leaf page - unused
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—i.e. **rowid** order for a table file and index order for an index file.

Table 1 - Page Header Format

Data Cells

Table data cells are located at the bottom of each page. Data cell content is different for leaf page versus interior pages. *Leaf page* data cells contain the table's records. *Interior page* data cells will contain only keys and left child page pointers.

The first data cell written to a page will be at the very end. The second data cell will be written immediately above it, etc. Note that the data cells may not be physically stored in key-sorted order—*however*, the offset location array in the page header will always be in key-sorted order.

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):

- The cell header (6-bytes)
  - A 2-byte **`SMALLINT`** which is the total number of bytes of payload, i.e. excluding the 6-byte header.
  - A 4-byte **`INT`** which is the integer key, a.k.a. "rowid"
- The payload (variable length)
  - 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. Unlike SQLite, NULL values occupy 1, 2, 4, or 8 bytes.

DavisBase does not support overflow payload, i.e. all cell payloads must fit on the same page.

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

- All Interior Cells have exactly 8-bytes
  - A 4-byte **`INT`** *page number* which is the left child pointer.
  - An **`INT`** which is the integer key delimiter between the left and right child nodes.

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. Note that there are different sizes of NULL values in DavisBase. This is to accommodate updating the field to a non-NULL value without increasing the size of the cell payload.

Serial TypeCode	Database Data Type Name	Content Size (bytes)	Description
0x00	NULL	1	Value is a 1-byte NULL (used for NULL TINYINT)
0x01	NULL	2	Value is a 2-byte NULL (used for NULL SMALLINT)
0x02	NULL	4	Value is a 4-byte NULL (used for NULL INT or REAL)
0x03	NULL	8	Value is a 8-byte NULL (used for NULL DOUBLE, DATETIME, or DATE
0x04	TINYINT	1	Value is a big-endian 1-byte twos-complement integer.
0x05	SMALLINT	2	Value is a big-endian 2-byte twos-complement integer.
0x06	INT	4	Value is a big-endian 4-byte twos-complement integer.
0x07	BIGINT	8	Value is an big-endian 8-byte twos-complement integer.
0x08	REAL	4	A big-endian single precision IEEE 754 floating point number
0x09	DOUBLE	8	A big-endian double precision IEEE 754 floating point number
0x0A	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 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.

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, -- optionally table_rowid INT
   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
```