Project Stage 1.0: Paginated Data Access

Due date: Wednesady, May 11, 11:59pm.

Overview

Stage 1 of the project incorporates the work you have done during **Lab 2** and extends it to build:

- Complete **buffer manager** layer.
- Functionality for **paginated disk access** for heap, sequential and hashed DB files.

FLOPPY: Stage 1.0

This document describes the lower tiers of FLOPPY architecture:

- Buffer Manager
- Page Read/Write layer

Buffer Management

FLOPPY's buffer manager largely follows the design you have built in your lab assignments. The buffer consists of two spaces with M slots allocated for the buffer proper - connected to the persistent storage, and N slots allocated for the volatile cache. Each slot, either in the persistent part or the volatile part of the buffer has the size of a single disk page. The supporting data structures for the buffer have been described in Lab 2 and Lab 3. The full Buffer Manager API has been described in Lab 3 and is repeated here for the sake of completeness.

function declaration	brief description
`int commence(char * Database, Buffer * buf, int nBufferBlocks, int nCacheBlocks);`	initialize the buffer
`int squash(Buffer * buf);`	graciously end the work of the buffer manager
`int readPage(Buffer * buf, DiskAddress diskPage);`	read access to the disk page
`int writePage(Buffer *buf, DiskAddress diskPage);`	write access to the disk page
`int flushPage(Buffer *buf, DiskAddress diskPage);`	flush the page from buffer to disk
`int pinPage(Buffer *buf, DiskAddress diskPage);`	pin the page
`int unPinPage(Buffer *buf, DiskAddress diskPage);`	unpin the page
`int newPage(Buffer *buf, fileDescriptor FD, DiskAddress * diskPage);`	add a new disk page
`int allocateCachePage(Buffer *buf, DiskAddress diskPage);`	allocate an empty slot in the cache to store a page with `diskPage` virtual address
`int removeCachePage(Buffer *buf, DiskAddress diskPage);`	clear the volatile slot that contains a given page

Data Files

FLOPPY shall store information in the following types of files:

1. Fixed-record heap files. Heap files store data on first-come, first-serve basis. Fixed-record heap files store records of pre-set fixed witdth. While the structure of most data tables used by the Peer-to-Peer system which FLOPPY will support is fixed, you shall implement fixed record heap file storage for database tables with arbitrary record structures. We discuss record structure descriptors below.

2. Fixed-record sequential files. You will use sequential files primarily to develop secondary index structures for faster access to the data. Most of the FLOPPY queries will result in retrieval of objects from tables by their unique id or other easily indexable property. The primary index structure you will be maintaining on top of fixed-record sequential files is the B+trees, although for some indexing you may get away with a simple index structure. (From the point of view of software development, it might be cheaper to represent all index stuctures as B+trees though).

3. Fixed-record hashed sequential files. These may be needed for storing some volatile storage data.

The structure of the index files is fixed, but the data files representing the data tables must store different types of records. We discuss what you will support below.

FLOPPY Data Types

FLOPPY shall support the data of the following types:

1. `INTEGER` or `INT` type. An `INT` value is 4 bytes long.

2. `FLOAT` type. A floating point value is 8 bytes long and is represented as a linux 32-bit floating point number.

3. `VARCHAR(n)` type. A string value type represented as an array of $n + 1$ bytes. The length of a `VARCHAR` value is the actual length of the string contained in it, which can be less than n. You will use null termination for representing strings.

4. `DATETIME` type. A type for representing date/time values. In C, this is a `time_t` structure. It is 8 bytes long. You need to be able to decode the data represented in a `time_t` value into a standard Linux timestamp format (e.g., the result of running the Linux `date` command that returns a DayOfWeek Month Day Hour Minute Second TimeZone Year format. You can decide whether you want to store `time_t` values verbatim (convenient, compact, divisible by 8), or whether you want to convert them to string representations (human readable, verbose) and store them that way. Either way, you must store the `DATETIME` values the same way throughout.

5. `BOOLEAN` type. True or False. Can be stored in a single byte, or if you want to prevent padding issues, use a four-byte representation, which essentially makes it a synonym for an `INT`.

CREATE TABLE Statements

FLOPPY shall support a simple standard SQL `CREATE TABLE` statement for creating relational tables. The syntax of the `CREATE TABLE` statement is:

```
CREATE TABLE <Table> [VOLATILE] (
  <Attribute>  <Type>,
  ...
  <Attribute> <Type>,
  PRIMARY KEY (<Attribute>[,<Attribute>]*)
  [, FOREIGN KEY (<Attribute> [,<Attribute>]*) REFERENCES <TableName>]
);
```

Here,

`<Table>`	name of a relational table (a FLOPPY identifier)
`<Attribute>`	name(s) of the table attributes (a FLOPPY identifier)
`<Type>`	FLOPPY data type. One of `INT`, `FLOAT`, `VARCHAR(n)`, `DATETIME` , or `BOOLEAN`
`<TableName>`	A name or another table (where the foreign key constraint is pointing)

Specifically, the `CREATE TABLE` statement names the table being created, lists its attributes, identifies the primary key (this is mandatory), and specifies, as necessary, one or more foreign key constraints.

For now, other constraints allowed in the SQL `CREATE TABLE` statements will not be required.

A `CREATE TABLE` statement that has an optional `VOLATILE` qualifier shall result in a creation of a table that is stored in the volatile part of the buffer. If the `VOLATILE` qualifier is not included in the statement, `CREATE TABLE` statement shall result in a data file representing the table created in the persisten storage (tinyFS).

Processing CREATE TABLE Statements

A `CREATE TABLE` statement passed to FLOPPY shall be processed as follows

1. Parsing. The parser shall identify the statement as a `CREATE TABLE` statement, and shall extract from it the following information:

   - Table name
   - Table storage presecription: volatile or persistent.
   - List of attributes with their types
   - Primary key
   - Foreign keys, if any.

   Note: All foreign keys must reference tables that already exist in the database, otherwise an error must be returned.

2. Preparation. The next step is to create a representation of the database table records. Create a `struct tableDescription` following the instructions below, and for each `CREATE TABLE` statement generate an instance of such `struct` (in C++, feel free to create the appropriate class). This instance shall be passed as a parameter to the actual table creation function.

3. Table Creation. There are two table creation functions that you will implement: `createPersistentTable()` and `createVolatileTable()`. Each function, given the description of the table stored ina `tableDescription` instance, shall create the appropriate data file - either in persistent storage or in volatile storage.

4. Index Creation. You must create the following secondary index structures for all persistent tables:

   (a) Index on primary key. This one must always be created.

   (b) Index on each of the foreign keys.

   The index creation step can be postponed until after the B+trees are implemented, but it has to be implemented.

5. Maintenance. Finally, you shall update your internal data structures - the list of database tables, etc., to reflect the new state of the database.

Parsing. Largely left to individual teams. You can postpone writing of the parser until later in the class and concentrate on steps 2–5 of the process for now. However it might also make sense to actually write the parser for the `CREATE TABLE` statement early to be able to debug your code conveniently.

Preparation. The `tableDescription struct` can be envisioned roughly as follows:

```
 typedef struct Attributes{
   char * attName;       /* attribute name */
   int    attType;       /* attribute type. a better choice is to use enums */
   Attribute * next;          /* pointer to next attribute */
  } Attribute;

\begin{verbatim}
 typedef struct FK{
   char * tableName;    /* table the foreign key references */
   Attribute * key;     /* foreign key itself */
   foreignKeys * next;  /* next foreign key on the list.
  } foreignKeys;


 typedef struct Tables {
    char * tableName;           /* name of the table */
    Attribute * attributeList;  /* list of attributes */
    Attribute * pKey;           /* primary key */
    foreignKeys * fKeys;        /* list of foreign keys */
 } tableDescription;
```

`createPersistentTable()` . This function shall create a tinyFS file representing a described table, and set up all the infrastructure in the buffer manager to support writes to and reads from this data file. The function description is:

```
int createPersistentTable(Buffer buf, tableDescription table);
```

The first argument is the buffer the FLOPPY uses, and the second argument is the table description constructed prior to calling `createPersistentTable()` function.

This function shll behave as follows:

1. Check to see if a table with a given name already exists in the database (check the table list stored in the `Buffer` object. If it does, exit with a `Table Exists` error code.

2. Determine the structure of the record for the new database file. Create a descriptor of this structure.
3. Create a new tinyFS data file for storing the given table.
4. Read the file header page for the new file into the buffer.
5. Place all necessary information on the header page. This includes the descriptor of the record structure that you have constructed on Step 2.
6. Force-write the header page to disk.

Index Creation. Will be discussed at later stages.

Completion. At a minimum, you shall maintain a list of database tables together with their persistent properties: the name of the tinyFS file, their status (persistent, volatile), and so on. This list of tables may later be used by other parts of your system to store some useful information (access timestamps, locks, and so on), but for now, just treat is as a directory listing of all objects created in the database. This will include the index structures you created for the new data file.

Good Luck!