CPS 216 Spring 2003
Homework #2
Assigned: Wednesday, February 12
Due: Wednesday, February 26

Note: This homework is longer than the last one. Start early! If you have already taken CPS 196.3 or an equivalent undergraduate course in databases, you should complete Problems 3-8. Otherwise, you may complete Problems 1-5 and 8. Problem 9 is a substitute for Problem 8(c) or 8(d) if you are not familiar with C++ programming.

Problem 1.

Consider the table *Address* (*street_addr*, *city*, *state*, *zip*) with the following two functional dependencies:

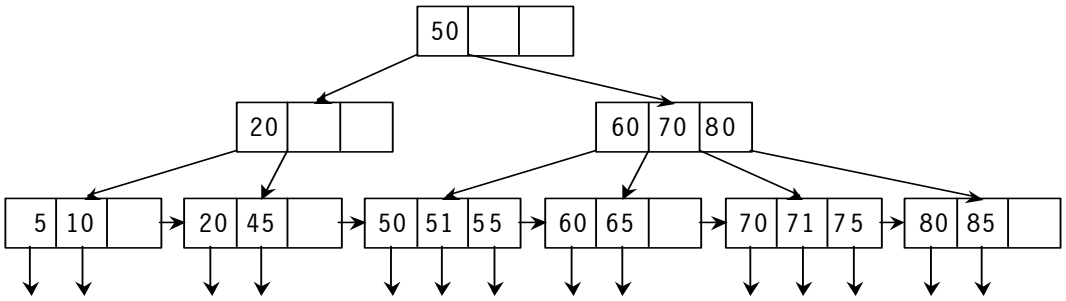
- *street_addr*, *city*, *state* → *zip*.
- *zip* → *city*, *state*.

For the following questions, keep all your SQL statements in a file named `hw2-1.sql`. You should use "`@`" instead of "`;`" as the statement termination character because of the trigger(s) that you are going to write in (b). When you are done, run "`db2 –td@ -vf hw2-1.sql > hw2-1.out`" on `rack40`. Then, print out the file `hw2-1.out` and turn it in together with the rest of the homework. You may find it helpful to put `DROP TRIGGER` and `DROP TABLE` statements in the beginning of `hw2-1.sql` so the ensuing `CREATE` statements can succeed. You may find it useful to consult the online document titled "DB2 SQL Programming Notes," found under the "Programming Notes" section of the course Web page.

(a) Decompose *Address* into BCNF. Write `CREATE TABLE` statements to create the decomposed tables, with appropriate key and foreign key declarations.
(b) The key and foreign key constraints on the decomposed tables are not enough to enforce the first functional dependency. Write trigger(s) to enforce it by rejecting any database modification (`INSERT/DELETE/UPDATE`) that violates this dependency. DB2's trigger syntax is discussed in "DB2 SQL Programming Notes."
(c) Write SQL modification statements to perform the following tasks in order. If your trigger works correctly, some of the statements will be automatically rejected.
    - Insert zip information: 27707 is in Durham, NC.
    - Insert address information: 123 University Ave., 27707.
    - Insert zip information: 27516 is in Chapel Hill, NC.
    - Insert address information: 123 University Ave., 27516.
    - Insert zip information: 27708 is in Durham, NC.
    - Insert address information: 123 University Ave., 27708.
    - Update zip information: 27516 is now in Durham, NC.

Problem 2.

Consider the following B^+-tree with a maximum fan-out of 4.



For all questions below, assume that you always start with the tree shown above—*not* the result tree you get for the previous question.

(a) Show the result tree after inserting 49.
(b) Show the result tree after deleting 50.
(c) Show the result tree after inserting 72.
(d) Show the result tree after deleting 5.
(e) What is the minimum number of keys you must delete for this tree to shrink down to two levels? Show the sequence of deletions.

Problem 3.

Suppose keys are hashed to 4-bit sequences, and each block can hold three records. We start with a hash table with two empty blocks (corresponding to 0 and 1), and insert 16 records with keys 0000, 0001, 0010, …, 1111, in order. Show the final state of the index:

(a) If the index is based on extensible hashing.
(b) If the index is based on linear hashing, with a capacity threshold of 100%. Here, we define capacity to be (actual number of records indexed) / (maximum number of records that can be held by primary buckets).

Problem 4.

Consider a secondary B^+-tree index. Each internal node contains m child pointers, and each leaf node contains m pointers to records. There are a total of N records indexed by the B+-tree. We wish to choose the value of m that will minimize search times on a particular disk drive, given the following information:

(1) For the disk holding the index, the time to read a given block into memory is given by $(30 + 0.01m)$ milliseconds. The 30 milliseconds represent the seek time and the rotational delay of the read; the 0.01m milliseconds is the transfer time.
(2) Once the block is in memory, a binary search is used to find the right pointer. The time to process a block in main memory is $(a + b \log_2 m)$ milliseconds, for some constants a and b.
(3) The main memory time constant a is negligible compared to the disk seek time and rotational delay of 30 milliseconds.
(4) You can assume that the index is completely full and perfectly balanced, and that N is conveniently a power of m.

Questions:

(a) What value of m minimizes the time to search for a given record? An approximate answer is okay.
(b) What happens as the disk latency (30 milliseconds) decreases? For instance, if this constant is cut in half, how does the optimal m value change?

Problem 5.

To build a hash index for a multi-attribute search key, we can use an approach called partitioned hashing. The partitioned hash function is really a list of hash functions, one for each attribute in the search key. Suppose that we wish to build a partitioned hash index on $R(A, B)$ with 2^n buckets numbered 0 to $2^n - 1$. In this case, the partitioned hash function consists of two hash functions h_A and h_B. Hash function h_A takes a value of A as input and produces a result with n_A bits, and h_B takes a value of B as input and produces a result with $(n - n_A)$ bits. The two results are concatenated together to produce the result of the partitioned hash function, which is then used to index the buckets. To locate records with $A = a$ and $B = b$, we simply go directly to the bucket numbered $h_A(a)\, h_B(b)$ (in binary).

(a) Which buckets do we have to examine in order to locate all records with $A = a$?
(b) Suppose we are given a query mix. Each query in this mix will either ask for records with a given value of A, or it will ask for records with a given value of B (but never both). With probability p, the value of A will be specified. Give a formula in terms of n, n_A, and p for the expected number of buckets that must be examined to answer a random query from the mix.
(c) Find the value of n_A, as a function of n and p, that minimizes the expected number of buckets examined per query.

Problem 6.

The following questions are based on the paper "Weaving Relations for Cache Performance," by Ailamaki et al.

(a) The paper does not directly address the performance of PAX versus NSM in handling point-based queries and updates. More specifically, a point-based query or update has a `WHERE` condition that specifies the exact value of the primary key, e.g., "`SELECT name FROM Student WHERE SID = 142;`". Such queries and updates are quite common in OLTP (On-Line Transaction Processing) workloads. Without any experiment results, can you guess how PAX performs in comparison to NSM?
(b) Consider a range query "`SELECT name FROM Student WHERE GPA > 3.0 AND GPA < 3.5;`". How would an index on *Student*(*GPA*) affect the performance comparison between PAX and NSM?

Problem 7.

The following question is based on the paper "Generalized Search Trees for Database Systems," by Hellerstein et al. Suppose we want to use GiST to index ranges of the form [*x*, *y*), where *x* and *y* are integers. These ranges may overlap with each other. The queries are of the form "find all ranges that overlaps with [*a*, *b*)." Discuss briefly how you would implement the six basic methods required by GiST, and whether you should set *IsOrdered* to true and define *Compare*. Highlight the differences between your implementation and the B^+-tree implementation described in the paper.

Problem 8.

For this problem, you may work either independently or in groups of two. You must complete (a) and (b), and one of (c) and (d). If you are not familiar with C++ programming, you may substitute Problem 9 for 8(c) and 8(d), but you still need to complete 8(a) and 8(b). Another alternative is to work in groups of three; however, in this case, you must complete the optional part of either 8(c) or 8(d).

In this problem, you are going to build a small piece of a database management system dubbed "DB216." To get started, copy the source files to you home directory using "`cp –r ~cps216/db216-hw2/ ~/`" on `rack40`. Go into directory `db216-hw2/` and type "`make clean`", and then "`make`". After compilation completes, you will find an executable file `db`. Typing "`./db`" creates a database file `test.db` (if one does not exist already) and runs a number of tests on it.

For this problem, turn in a pointer to the directory containing your documentation, source code, and executable. The directory should contain a `README` file documenting any known bugs/limitations, additional test cases or nifty features you have implemented, and instructions to compile and run your program.

(a) Study the existing code base. *Write down three things you do not like about the current class interface design, and discuss how you might fix them.*

(b) *Complete the implementation of* `bt_compare()` *in* `src/access/BDBBTree.cpp`.

DB216 implements tables with primary keys on top of Berkeley DB B^+-trees. Berkeley DB is a high-performance transactional storage manager, although we are not using its transaction support at this time. Documentation is available at `http://www.sleepycat.com/docs/`. Berkeley DB provides an interface for storing (*key*, *value*) pairs, where both *key* and *value* are uninterpreted sequences of bytes. Implemented on top of Berkeley DB B^+-trees, `BDBBTree` class (found in `src/access/BDBBTree.{h,cpp}`) provides a richer interface for manipulating database rows.

By default, Berkeley DB B^+-trees compare keys byte by byte. This default comparison method does not work for certain data types. For example, –1 should obviously be less than 1, but the byte representation of –1 (0xFFFFFFFF) is considered by Berkeley DB to be greater than that of 1 (0x00000001). Thus, in running `testTableWithPrimaryKey()` (in `src/main.cpp`), you might notice some rows are printed in the wrong order. By defining your own key comparison functions to

override the default, you should be able to get the natural order of the primary key.

You code should support the case when the primary key consists of multiple columns of different types. You will find the row/column type support provided by `src/metadata/{RowType,ColumnType}.{h,cpp}` useful.

(c) *Implement support for the variable-length column type* `VARCHAR`. Optional: *Implement support for* `NULL`*'s.*

Currently, `ColumnType` only supports three fixed-length SQL types `INTEGER`, `FLOAT`, and `CHAR`; `RowType` and `BDBBTree` assume that all columns (and hence rows) have fixed lengths and cannot be `NULL`. Add support for `VARCHAR`, and, optionally, `NULL`'s. You will also need to modify `src/main.cpp` (and `src/system/DatabaseManager.cpp`) to include additional tests. If you choose to tackle `NULL`'s, you will also need to change the `createTable()` interface and `src/metadata/TableMetadata.{h,cpp}` to allow specification of `NOT NULL` constraints.

(d) *Implement metadata support.* Optional: *Implement support for additional keys besides the primary key.*

A database management system stores *metadata*, or schema information (e.g., names and storage types of database tables, names and types of their columns, keys, indexes, etc.), in special system-managed tables called *catalogs*. Catalog tables can and should be manipulated just like any other table (which, taken to an extreme, means that any information about catalogs should be stored in catalogs as well).

When `DatabaseManager` (found under `src/system/`) receives a request to access a table *R*, it looks up *R* in the catalog tables and creates a `TableMetadata` object (found under `src/metadata/`) to hold all schema information about *R*. Information inside the `TableMetadata` object tells us, for example, where to locate *R* and how to interpret the bits representing *R*'s contents. Obviously, you will need some bootstrapping to be able to locate the catalog tables in the first place!

Before you start coding, you should design a good schema for catalog tables. Your job then is to remove the hard-wired code (used to get the tests to run) in `DatabaseManager` and implement real metadata support. For extra credit, you will also need to figure out how to enforce the other key constraints (hint: use `BDBBTree`'s to build secondary indexes to enforce uniqueness of key values). For simplicity, you do not have to worry about maintaining the secondary indexes automatically for now.

Problem 9.

For this problem, you may work either independently or in groups of two. If you are not familiar with C++ programming, you may substitute this problem for 8(c) and 8(d), but you still need to complete 8(a) and 8(b).

For this problem, you need to implement a JDBC application that acts a remote command-line client for accessing DB2. Once started, your application should enter a command-line loop that accepts and executes the following commands:

- `CONNECT` *machine dbname user password*
  Establish a JDBC connection to the specified remote database. Any existing connection should be closed.
- `LIST TABLES`
  List names of all tables in the remote database.
- `DESCRIBE` *table_name*
  List names and types of the columns of the given table, in order.
- `SELECT` ...
  Run a SQL query on the remote database and display its result. You may assume that the input line contains the entire SQL query string.
- `DISCONNECT`
  Close the connection to the remote database.
- `QUIT`
  Exit the client, and close any open connection.

Optionally, you may also support `INSERT`, `DELETE`, `UPDATE`, and `CREATE` statements. You may find it useful to consult the online document titled "Using JDBC with DB2," found under the "Programming Notes" section of the course Web page. You should test your application on a machine other than `rack40` to see if remote connection works correctly.

For this problem, turn in a pointer to the directory containing your documentation, source code, and executable. The directory should contain a `README` file documenting any known bugs/limitations, test cases or nifty features you have implemented, and instructions to compile and run your program.