15-721 Assignment 3:
Join Processing

Out: October 9th — Due: October 24th

1 Introduction

In this assignment, you will implement two join algorithms: index nested loop and sort-merge joins. The main contact person for this assignment is `spapadim@cs.cmu.edu`.

2 Where to Find Makefiles, Code, etc.

Copy all the files from `/afs/cs/academic/class/15721-f01/mini_hwk/assign/Join/src` into your own local directory. The files of interest to you are:

- `Makefile`: A makefile to compile your project.
- `indexNestedLoop.h`: Specifications for the class index_nested_loop. You have to implement these specifications as part of the assignment.
- `sortMerge.h`: Specifications for the class sortMerge. You have to implement these specifications as part of the assignment.
- `joinErr.[Ch]`: Error code and message definitions (for both join algorithms, which are in the `JOINS` error layer).
- `INLJTester.C` and `SMJTester.C`: test drivers for index nested loop and sort-merge joins, respectively.

You can find the interfaces for `HeapFile`, `Scan`, `BTree`, `BTreeFileScan` and `Sort` classes in the directory `/afs/cs/academic/class/15721-f01/mini_hwk/assign/Join/include`.

As in previous assignments, the makefile has two targets: a default target which produces an executable named `join` with your code linked in and a target `solution` which produces an executable with the reference solution. The driver program accepts exactly one argument, which should be either `inl` or `sm` depending on which join algorithm is to be tested. Also, the makefile has the usual `handin`, `uuencode`, etc. targets. Finally, do not forget to `make depend`!

3 Index Nested Loop Join

```
class indexNestedLoop {
public:
indexNestedLoop(
    char *       filename1,        // Name of heapfile for relation R.
    int          len_in1,          // # of columns in R.
    AttrType     in1[],            // Array containing field types of R.
    short        t1_str_sizes[],   // Array containing size of columns in R.
```

```
    int         join_col_in1,      // The join column number of R.
    char *      filename2,         // Name of heapfile for relation S
    int         len_in2,           // # of columns in S.
    AttrType    in2[],             // Array containing field types of S.
    short       t2_str_sizes[],    // Array containing size of columns in S.
    int         join_col_in2,      // The join column number of S.
    char *      filename3,         // Name of heapfile for joined results
    IndexType   intype,            //Index type
    Status &    s                  // Status of constructor
);
~indexNestedLoop();
};
```

The `indexNestedLoop` constructor joins two relations R and S, represented by the heapfiles `filename1` and `filename2`, respectively, using the index nested loops join algorithm. Note that the columns for relation R (S) are numbered from 0 to len_in1 $- 1$ (len_in2 $- 1$). You are to concatenate each matching pair of records and write it into the heapfile `filename3`. The error layer for the indexNestedLoop class is `JOINS`.

You will need to use the following classes which are given: `HeapFile`, `Scan`, `BTreeFile` and `BTreeFileScan`. You will call `BTreeFile` methods to build up a B-tree index for the inner heapfile S. Then you scan the outer heapfile R, for each tuple in the outer heapfile R, retrieve the matching S tuples by scaning the B-tree index on the inner heapfile S.

4 Sort-Merge Join

```
class sortMerge {
public:
sortMerge(
    char *      filename1,         // Name of heapfile for relation R.
    int         len_in1,           // # of columns in R.
    AttrType    in1[],             // Array containing field types of R.
    short       t1_str_sizes[],    // Array containing size of columns in R.
    int         join_col_in1,      // The join column number of R.
    char *      filename2,         // Name of heapfile for relation S
    int         len_in2,           // # of columns in S.
    AttrType    in2[],             // Array containing field types of S.
    short       t2_str_sizes[],    // Array containing size of columns in S.
    int         join_col_in2,      // The join column number of S.
    char *      filename3,         // Name of heapfile for merged results
    int         amt_of_mem,        // Number of pages available for sorting
    TupleOrder  order,             // Sorting order:  Ascending or Descending
    Status &    s                  // Status of constructor
);
~sortMerge();
};
```

The `sortMerge` constructor joins two relations R and S, represented by the heapfiles `filename1` and `filename2`, respectively, using the sort-merge join algorithm. Note that the columns for relation R (S) are numbered from 0 to len_in1 $- 1$ (len_in2 $- 1$). You are to concatenate each matching pair of records and write it into the heapfile `filename3`. The error layer for the sortMerge class is `JOINS`.

You will need to use the following classes which are given: `Sort`, `HeapFile`, and `Scan`. You will call the `Sort` constructor to sort a heapfile. To compare the join columns of two tuples, you will call the function `tupleCmp` (declared in `sort.h`). Once a scan is opened on a heapfile, the scan cursor can be positioned to any record within the heapfile calling the `Scan` method `position()` with a RID argument. The next call to the `Scan` method `getNext()` will proceed from the new "cursor" position.

5 Error Protocol

An explanation of the error protocol follows. First of all, you have to define an array of error messages and its corresponding array of error codes. Here is an example for the buffer manager component:

```
enum bufErrCodes { HASHTBLERROR, HASHNOTFOUND, BUFFEREXCEEDED, ... };

static const char* bufErrMsgs[] = {
        "hash table error",
        "hash entry not found",
        "buffer pool full",
        ...
};
```

Note that the error message array is declared as `static`. To register the error message array for the buffer manager component, you would just need to create a static `error_string_table` object in the same file as the error message array as follows:

```
static error_string_table bufTable(BUFMGR, bufErrMsgs);
```

There are two macros defined that make it easy to report errors when you discover them. These macros also add the name of the file and the line number where the error happens, as a debugging aid.

To add a "first" error, use the `MINIBASE_FIRST_ERROR` macro. For example, if the buffer manager detects that the buffer pool is too full to complete an operation, it posts its `BUFFEREXCEEDED` error like this:

```
MINIBASE_FIRST_ERROR(BUFMGR, BUFFEREXCEEDED);
```

To add a "chained" error, use the `MINIBASE_CHAIN_ERROR` macro. For example, if the buffer manager calls on the database manager to write a page to disk, and that operation fails, the buffer manager adds an error that records the fact that the execution path that failed went through it:

```
status = MINIBASE_DB->write_page(...);
if (status != OK)
    return MINIBASE_CHAIN_ERROR(BUFMGR, status);
```

6 What to Turn In, and When

You should turn in copies of your code (ie. files `indexNestedLoop.[Ch]` and `sortMerge.[Ch]` at least and possibly `joinErr.[Ch]` and `Makefile` if you need to modify them). Optionally, you may want to turn in copies of the output produced by running the provided tests.
Note: The actual text output produced by these tests is actually needed to verify the correctness of your solutions (as opposed to just the exit status). Please make sure that your output is readable (ie. you have not forgotten debugging output, etc.).

The assignment is due at midnight on October 24th. The solution will be made public after that time, and solutions turned in after that time will not receive any credit. So be sure to turn in whatever you have, even if it is not working fully, at that time.