Project 3: Joins

For this assignment, you are expected to implement three standard join algorithms and evaluate their relative performance. Their description is given in the lecture notes on the evaluation of relational operators and can be found in any database systems textbook (e.g., Gehrke and Ramakrishnan's book).

Background

Let us quickly review some of the Minibase components you will need for this assignment. In Minibase, a relation is implemented as a heap file, which is a collection of records. Records can be inserted or deleted from a heap file, and each record is uniquely identified by a record id. A scan is the interface used to access records in a heap file, one by one.

An index provides fast access to records in the heap file, and currently Minibase only supports B+ tree indices. Each entry in the index is a (key, record id) pair. Entries can be inserted or deleted from an index. An index search scan provides an interface for accessing the records in the index. In this assignment, you will need the methods provided in the following four classes:

- **HeapFile**: implements a heap file
- **Scan**: scan interface to a heap file
- **BtreeFile**: implements a B+ tree index
- **BtreeFileScan**: implements the scan interface to the B+ tree

The Join Algorithms

1. **Tuple-At-A-Time Nested Loop Join**

Start with this one, it is the simplest.

2. **Block Nested Loop Join**

Since Minibase does not provide page-by-page access into a relation stored in a heap file (i.e., you can only retrieve records one at a time), you will simulate a “block" with an array storing the records. The parameter B represents the size (in BYTES) of the array storing the outer relation. In *main.cpp*, to perform block nested loop join, set B to the following value before calling the **blockJoin** method:

(MINIBASE_BM->GetNumOfBuffers()-3*3)*MINIBASE_PAGESIZE.

Note that the (-3 * 3) term is in this expression because each heap file scan object (one each for the two input and one output buffers) will not pin more than 3 pages in the buffer pool (at most two directory pages and the page storing the current record in the scan).

The pseudocode for this join is:

```
For each block b in R
        For each tuple s in S
                For each tuple r in b
                        Match r with s
                        if Match then
                                Insert (r,s) into the result relation
```

Compare the performance of the "Block nested loop" join for various block sizes.

3. Index Nested Loop Join (Optional)

Implement this algorithm by creating an unclustered B+-tree index on the inner relation. Determine whether it is beneficial to build a B+-tree index for the purpose of performing a single join operation by comparing the cost of this join with that of other join methods.

Simplifications

- A MAJOR simplification: You can assume that you are joining on a foreign key from R to S, e.g., no two records in relation S will join with the same record in R, but several records in R could join with the same record in S.
- You can assume that all records are of fixed length.
- You can assume that we only perform joins on integer fields.

Performance comparison

- Compare the relative performance of the three join algorithms. Record the times taken for each algorithm to run, and the number of page misses.
- Study the effect of the buffer pool size on the algorithms by changing the buffer pool size (NUM_OF_BUF_PAGES in *main.cpp*).
- Study the effect of the relation size on the algorithms by changing the number of records in the given Employee and Project relations (NUM_OF_REC_IN_R and NUM_OF_ REC_IN_S in *join.h*).
- Submit a report containing documentation, tables and graphs of these statistics, together with an analysis of the results. Note that your analysis must be thorough, and cover a wide range of buffer pool sizes and relation sizes.

Collecting Statistics

You should let your algorithm run several times and report the average (wall-clock, not user) running time for each result reported. For every configuration and every join algorithm, print out the average running time and the number of buffer pool page misses.

Source Code

You are given an archive which contains the following files:

- *join.cpp*, *join.h*: utility functions useful for writing join algorithms.
- *relation.cpp*, *relation.h*: functions to create test relations.
- *blockjoin.cpp, indexjoin.cpp, tuplejoin.cpp*: each file contains a skeleton for implementing a particular join method.
- *main.cpp*: main program.
- *include*: subdirectory of all .h files needed.

You should write your code in *blockjoin.cpp, indexjoin.cpp, tuplejoin.cpp,* and *main.cpp*. The functions in *join.cpp* and *relation.cpp* will be useful for writing your join methods and for debugging. The method **SortFile** in *join.cpp* is particularly useful as an example of how to use **HeapFile**, **Scan**, **BTreeFile** and **BTreeFileScan**.

Hints

- Study the **Scan** class. Learn how to access records in a heap file. Study the **SortFile** method in *join.cpp*.
- Use a small numbers of records (by changing the constants in *join.h*) for testing.
- A rough estimate: each join algorithm should be around 100-150 lines of code.
- You are free to modify anything in the main function, but do not change the functions that create relations so that we can compare your results with our own.

Compile and run the code

Differently from previous practicals, there are not test cases to be passed this time. You are required to modify the *main.cpp* file in order to invoke the three different join algorithms, collect statistics during each run and print these statistics at the end of each run. You are required to run each algorithm several times and report the average of all the running times. The instructions to compile and run the code are the same of the previous practicals. Assuming your work directory is ~/, copy the file *Joins.zip* with the skeleton code into your working directory and unzip it using the command *unzip Joins.zip*. This will result in a folder *~/Joins/* containing the source code and CMake build files. Create a directory *~/Joins-bin/* , change to this directory and run *cmake ~/Joins/* in order to create the makefiles for the project. Now run *make* to compile the source code. (To recompile the source code after having modified parts of the code, it suffices to run *make*; *cmake* does not need to be run again.) The resulting executable file is *./minibase-joins*. You should be able to compile and run the code, but the execution of the code will only create

random records for two relations without executing the join (you have to implement the joins and invoke them in the main class).

What to turn in

The successful completion of tasks 1 and 2 alone suffice for a mark S. The successful completion of the optional task 3 (in addition to the mandatory tasks 1 and 2) brings an S+.

You should submit the same set of files given to you at the beginning of this assignment, plus any additional header and source files you have created for this assignment. These files should be zipped up into a file named *Joins-<FirstName-LastName>.zip* (using the command *zip –r Joins-First-Last.zip ~/Joins*) or *Joins-<FirstName-LastName>*.tgz (using the command *tar -cvzf Joins-First-Last.tgz ~/Joins*). These files should be organized in the same directory structure as supplied. **Upload your solution (ie zip/tgz file) via the course website.** In addition to this submission, you will have to present your solution to the demonstrator in one of the sessions.

Deadline: End of your practical session in Week 8. Good Luck!