PostgreSQL Query Processor: Homework Assignment 2

CS186 Introduction to Database Systems
UC Berkeley
October 31, 2006

1. Overview: Hybrid Hashing for Grouped Aggregates

In Project 1, you implemented prefix key compression on a B+-tree index. In this project you will move to a higher level in the system and add functionality to the PostgreSQL query executor.
We will restrict our focus to grouped aggregates. This project will be considerably more complex than Project 1, especially in terms of the amount of existing code you need to understand. The major parts of the project are:

1. A "big picture" understanding of a query executor
2. An understanding of different aggregation strategies
3. Examining, understanding, and enhancing existing code

In lecture, will see how grouped aggregates can be implemented with sorting as well as hashing. We are providing you with a PostgreSQL version that supports sorting and in-memory hashing. Your job is to understand hashing grouped aggregate code that we provide, and enhance it to deal with insufficient memory by spilling data to disk.

1.1 Administrivia

For this assignment, we will ask that you move your homework 1 work off your student class account (saving it somewhere in case there are grading problems), and remove your pgsource directory. For this assignment, it will be necessary to store your pgdata directory (because the home directory quota will be exceeded when you run the tests on the larger data sets) in your tmp directory, so if you haven't done so, run the following to create a tmp directory for your account. (Please note that this script has not been set up to run on the ilinux* machines, so please use another machine to run the script and log back onto the ilinux* machines to do the project.)

```
/share/b/bin/mkhometmpdir
```

Now a directory will be created for you in /home/tmp/cs186-xx. Note that this directory is not regularly backed up like your class home directory, so it is your responsibility to keep a backup of your working code.

Make sure your code runs smoothly on the specified ***Linux*** EECS instructional machines (either ilinux1 or ilinux2) prior to submission since ALL grading will be done on those. You should also run your performance experiments on the instructional machines, as you may see different performance due to different system libraries, etc. Note that the instructions provided are designed to work with the EECS instructional machines and may not work as smoothly elsewhere. The TAs and Professor will only support these machines.

1.2 Tasks

The following lists the various tasks you will have to complete while working on this assignment.

1. Compile and test our version of PostgreSQL
2. Study and understand the hash based aggregation implementation provided
3. Enhance hashed aggregation to spill to disk when required
4. Compare the performance of sorted and hashed aggregation

This project will take a fair amount of time. Spend some time to read this document completely before beginning work. Start early and avoid a last-minute scramble!

Since understanding the code is an important part of this project, the TAs and Professor will not assist you in understanding the existing code beyond what is discussed here. This handout is as follows:

- Section 2 the hashing grouped aggregation algorithm and the disk spilling algorithm that you need to implement.
- Section 3 gives a brief walkthrough of the code you have.
- Section 4 describes the performance study you have to complete.
- Section 5 gives some additional resources that you might find interesting.
- Section 6 tells you what (and how) you need to submit.

2. Hash Grouped Aggregate Description

In this section, we first describe the hash grouped aggregate algorithm in the source we give you. We then describe the algorithm that you need to implement, which is a hash algorithm that spills to disk when the hash table memory size is exceeded. These algorithms will be covered in class sometime in the next week, and the notes should soon be available.

```
Initialize the hash table
while (tuple ← get_next() ≠ NULL) do
        (hashKey, exists) ← lookup_hash_table(tuple)
        if exists then
                update trans-value in hash
        else
                initialize new trans-value
                insert (hashKey, trans-value) in hash
        endif
endwhile
for each (group-key, trans-value) in hash do
        append finalize(trans-value) to output stream
endfor
```

The pseudocode for the hashing algorithm provided in the code is shown above (we detail later how this is implemented in the actual source). The aggregate algorithm runs in an operator that takes tuples from the operator below it in the query plan using the get_next() call. For each input tuple, it performs a hash table lookup using the tuple's group by field(s). If an entry exists, this entry is a transition value, which is a set of information about the aggregates of the tuples seen so far for a given group. For a maximum aggregate, for instance, the transition value would keep the maximum value of the tuples seen so far for a given group. If such a transition value exists for the tuple, it is updated with the tuple. Otherwise, a new transition value is created and inserted into the hashtable. Once all tuples are processed, the algorithm finalizes each transition value in the hashtable and outputs it. This works fine as long as the hash table is small enough to fit in the memory allocated to it (call it HashMem). If not, the system must use another algorithm that uses the disks. We show such an algorithm (which you have to implement) on the next page.

```
// Initial pass
Initialize the hash table
while (tuple ← get_next() ≠ NULL) do
        (hashKey, exists) ← lookup_hash_table(tuple)
        if exists then
                update trans-value in hash
        else if hash table size ≤ HashMem then
                initialize new trans-value
                insert (hashKey, trans-value) in hash
        else
                if necessary, initialize temporary files
                partition ← partition_hash_index(hashKey)
                write_tuple(tempFile[partition], tuple)
        endif
endwhile
for each (group-key, trans-value) in hash do
        append finalize(trans-value) to output stream
endfor

// Processing of spilled tuples
for each partition i do
        re-initialize hash table
        while (tuple ← read_tuple(tempFile[i]) do
                (hashKey, exists) ← lookup_hash_table(tuple)
                if exists then
                        update trans-value in hash
                else
                        initialize new transition value
                        insert (hashKey, trans-value) in hash
                endif
        endwhile
        for each (group-key, trans-value) in hash do
                append finalize(trans-value) to output stream
        endfor
endfor
```

Above is a two pass algorithm for dealing with a number of transition values that do not fit in memory. (NOTE: pay attention to the above pseudocode, because it exactly describes the algorithm that you have to implement!) The first phase operates just as the previous algorithm does when the size of the hash table is below HashMem. When the size exceeds HashMem, the algorithm writes any tuples for which it does not have a transition value to a temporary file that represents a partition. If a transition value already exists for a tuple, note that the algorithm always updates the value instead of sending the tuple to a partition. The algorithm does this because updating existing transition value for the aggregates we support (avg, sum, min, max) does not require any more space in the hash table.

The second phase processes each partition that was written out to disk in the first phase. For each partition, each tuple from the partition is read in and a new hash table is built with transition values. In phase one, note that you should not use the same function to map the hash keys to both the hash table buckets and partitions. If you do, all the tuples from a partition in phase 2 will hash to the same bucket!

Thus, partition hash_index() needs to be different than the function that maps hash keys to hash table buckets. (*hint:* in implementation, you can use the hash value to define {partition number, bucket number} pairs).

The second phase of this algorithm differs in two ways from the second phase of the spilling hash algorithm in the notes and lecture. First, note that the second phase never checks HashMem. It is possible that the hash table built from a given partition exceeds HashMem, and more processing would have to be done, such as using overflow pages or recursively creating more partitions.

When we set HashMem in PostgreSQL it is fortunately a soft limit on the hash table size. That is, PostgreSQL allows us to allocate much more memory for a hash table than HashMem. To keep the assignment simple, the algorithm you implement should NOT check HashMem on the second pass, and it should just let the hash tables created in the second pass grow larger than HashMem if need be. Even with this simplification, this spilling algorithm will support much larger relations than the in-memory algorithm.

The second difference also has to do with memory allocation. In the notes, we learned that B, the amount of buffers available for the operator limits the size of the in-memory hash table and the output buffer for each temporary file. However, in our PostgreSQL implementation we only care about ensuring that the in-memory hash table in the first pass fits in HashMem. Accordingly, you should assume that the buffers used for the temporary files are not accounted for as part of the HashMem. As we will see later, we do this because of the way PostgreSQL allocates memory.

3. Code Walkthrough

We now describe the code that implements the main memory hashing aggregation, and some additional code that you will need to complete your assignment. Section 3.1 describes how to get and compile the code. Section 3.2 gives a brief walkthrough of the code, and Sections 3.3, 3.4, 3.5, and 3.6 discuss some other PostgreSQL functions that you will need to do your assignment.

3.1 Obtain, Compile, and Test PostgreSQL

Make sure you backup your hw1 code and then proceed to remove each of the postgres directories by running the following (be careful that you are ***absolutely sure*** there is nothing you want to keep within these directories before running this):

```
\rm -rf ~/pgsource ~/pgdata ~/pgsql
```

Then, run the following command to configure and copy over your postgres source tree and the scripts provided to test the performance.

```
~cs186/fa06/hw2/ConfigurePostgres.sh
```

You will see the following two subdirectories in your home directory:

- **pgsource/:** Similar to hw1, this is the directory for storing your postgres code. Source code of a version of PostgreSQL based on the 8.1.4 release. This distribution (relatively) new in-memory hashing strategy for grouped aggregates that has just been developed by the PostgreSQL Global Development Group. By implementing the disk spilling algorithm, you will be finishing the job started by the PostgreSQL developers!

- **scripts/:** Some scripts you can use to compile and run experiments. The scripts that we provide in scripts for running experiments can be used to selectively enable and parameterize the different strategies for grouped aggregation in PostgreSQL. We will detail these later in Section 4, but it is it is useful to know the parameters that they hand to PostgreSQL so you can understand

the code. By default, our version of PostgreSQL will always pick a hashing strategy, unless prohibited on startup with the -fg option. The hashing and sorting strategies can be tuned with the configuration parameter work_mem. This parameter affects the memory that is used by the two strategies. The easiest way to set these parameters is using the -S options to the postmaster. In the code, the value of work_mem can be found in HashMem. Note that HashMem is not used in the implementation provided — however, you will use it in your implementation!

Note that PostgreSQL is a little sloppy in accounting for memory usage. Unlike what was explained in the notes work_mem does not include memory for input and output buffers used with each run. As mentioned in Section 2, you will use the same convention in this assignment.

3.2 Code Walkthrough

The code that you have to modify (and most of the code that you have to understand) is in the following files:

- src/backend/executor/nodeAgg.c
- src/include/nodes/execnodes.h

In fact, the hw2 distribution only allows you to change these files. The rest of the source files which are used by the Makefile (and, eventually by the autograder) are read-only copies of the source in ~cs186.

The modifications you make will primarily be to the Agg operator. The current Agg operator combines the implementation of the sorting and in-memory hashing based strategies for grouped aggregates. Your job is to modify the hashing implementation so that it runs the spilling algorithm as described in the last section.

The state of an Agg operator is stored in the Agg structure. To add fields to the operator, you will add fields to Agg.aggstate, which is a struct of type AggState. AggState can be found in execnodes.h. You do NOT want to define global variables that are used by the Agg operator (why?). Agg nodes are operated on by the query processor by four functions, found in nodeAgg.c. It would be helpful if you read the code in tandem with this walkthrough.

- **ExecInitAgg():** the function that PostgreSQL calls to initialize an Agg node in a query plan. Most of this you do not have to (and should not) touch or understand, but you do need to deal with some of the code. The code you need to worry about is commented with “CS186” comments.

  First, ExecInitAgg() allocates the memory context aggstate->aggcontext. We will discuss why this is important in Section 3.6. Also, EcecInitAgg() builds the hash table with build_hash_table(), and sets aggstate->table_filled to false. Note that ExecInitAgg() takes advantage of node->aggstrategy being set. node->aggstrategy is set by postgres according to whether you chose sort or hashed aggregates when you started postgres.

- **ExecAgg():** The system calls ExecAgg() with the Agg node when the node’s parent in the query plan requires another input tuple. The function returns a pointer to a TupleTableSlot, which contains the result. If you think in terms of the iterator model, the system calls ExecAgg() every time the node’s parent calls getNext().

  Thus, the system calls ExecAgg() with the Agg node once for each group it outputs, plus one extra call. This call will return NULL, and the parent node will know that the Agg node has no more tuples to output.

  This function calls one of agg_retrieve_direct() or agg_retrieve_hash_table() based on the aggstate->aggstrategy.

While you need not read the former, make sure that you understand agg_retrieve_hash_table (), as this is the code path that you will modify. Next, we detail agg_retrieve_hash_table (), as well as agg_fill_hash table().

- ***agg_fill_hash_table():*** As stated above, agg_fill_hash_table() is the call that processes all input tuples and builds the complete hash table. ExecProcNode() reads in input tuples from this node's child, returning a TupleTableSlot pointer. lookup_hash_entry() either finds or builds and initializes a hashtable entry for a given tuple's group. You'll want to modify this call so that you have the option of checking for a hash entry without inserting a new one if one doesn't exist. You can use the call LookupTupleHashEntry() inside lookup hash entry() to do this. After the call to lookup_hash_entry() in agg_fill_hash_table(), the aggregate is advanced for each such entry. After all tuples are processed, the table is marked filled and the iterator is reset. Again, there is some setup code in the loop that is somewhat complicated and unintuitive. Mainly, you need to make sure that tmpcontext->ecxt scantuple = outerslot is before the lookup_hash_entry() and advance aggregates() calls, but after you have a non-NULL value for outerslot. Also, you must call ResetExprContext(tmpcontext) at the end of the loop. If you write a similar loop for the second stage of your algorithm, you must follow these conventions as well.

  When modifying the code, you might find the following useful. The call GetTupleHashKey() may be useful in getting the hash key for finding the partition to send a tuple to. Also, the TupleHashSize macro defined in nodeAgg.c is helpful in determining the size of the hash table. When comparing output from TupleHashSize to HashMem, you need to multiply HashMem by 1024L because HashMem is expressed in kilobytes and we're dealing with longs.

- ***agg_retrieve_hash_table():*** basically iterates over the transition values in the hash table and outputs either the first one, or the first one matching the having predicate if a having clause exists. The code in the while loop takes an entry from thes hash table, copies the entry into a tuple slot, finalizes each entry, and projects out any needed values. The ExecQual() in the while statement applies any predicates from any having clauses, so that agg_retrieve_hash_table() only returns a tuple that satisfies these predicates. Note that there's a lot of setup code in the while loop that doesn't make a lot of intuitive sense. Thus, it would not be recommended touching the code in the while loop after the point where you know you have a non-NULL transition value from the hash table. You should also not change any of the declarations or assignments at the beginning of the function unless explicitly needed.

- **ExecEndAgg():** destructor for an Agg node. Note that this is where aggcontext is deleted. Again, we cover memory contexts in detail in Section 3.6. Make sure any data structures *you* create for a node are destroyed here.

- **ExecReScanAgg():** This resets a current Agg node so it can be reused if need be. Note that the code resets aggcontext (this is also discussed in Section 3.6). It also creates (but does not populate) a new hash table with build_hash_table(), and sets table filled to false. Most of the other code is not of interest to you, and should be left as is.

In general, there's a lot going on in this code, and you should change as little as possible to avoid accidentally introducing bugs into a very complicated system. Although we gave warnings above about certain portions of the code that are particularly complicated, your success will probably depend on changing as little as possible throughout the code in order to get the job done.

3.3 Hashtable Initialization

One of your tasks for the spilling algorithm is to initialize the hash table appropriately. In the code, this is done in build_hash_table(), where the optimizer's estimate of the number of distinct groups (node->numGroups) is used as the number of buckets of the hash table, and passed as the fourth argument to BuildTupleHashTable(). When this estimate is very high, the hash table becomes too large. In your implementation, you must instead use the HashMem global variable to limit the size of the hashtable. To avoid long bucket chains, fix the number of buckets to be half the maximum number of entries. Given this information, you must calculate the maximum number of entries (max entries) that can be accommodated in the hash table. You can then use (0.5 * max entries) as the number of buckets that is passed to BuildTupleHashTable() (a call made from within build_hash_table()).

The macro TupleHashSize defined in nodeAgg.c shows how to compute the size of a hash table given the number of buckets and entries. You can use the equation expressed in the macro to help design your computation for max entries. Remember from above that when using HashMem with the TupleHashSize macro, HashMem must be multiplied by 1024L.

3.4 Determining the Number of Partitions

When the hash table grows large enough, the code you implement should begin spilling. This condition can be checked by using the TupleHashSize operator. When you begin spilling, you need to decide upon the number of partitions to spill non-matching tuples into.

To do this, we can use the number of input tuples seen so far (call this tuples_so_far). The expression outerPlan(node)->plan_rows can be used for the total number of records of the input table. Now you can use outerPlan(node)–>plan_rows / tuples_so_far to estimate the number of partitions (NOTE: I believe you have to cast outerPlan(node)->plan rows to an int before this division will work). With this estimate, we assume that the distribution of group values does not change in the rest of the input table.

You can use the gendata program in the scripts directory to generate different tables with different cardinalities and different numbers of distinct values in the group by column. Using this data, run some queries to figure out which estimate's assumptions are most reasonable, to guide your choice of weights.

Beware of corner cases: Since the is not exact, you should sanity check your estimated number of partitions. If it is less than one, you can either issue an error, or set it to some reasonable value.

3.5 Temporary files

To spill your tuples to disk, you will need to be able to open, read, write and close files. Rather than using the standard C file I/O library, we require you to use functions within PostgreSQL. Specifically, the BufFile interfaces that are declared in postgres/src/include/storage/buffile.h. These functions are attractive because you don't have to deal with file names and you get buffered I/O for free.

> **Note that you will not be able to generate performance numbers if you do not use the BufFile interfaces. Worse still, our autograder will not give you any credit either.**

The interface is almost identical to the standard C library functions (i.e., fopen() and tmpfile(), fclose(), fseek()). Instead of FILE * you will work with BufFile * pointers, but everything else remains essentially the same. More details on memory contexts are given in Section 3.6. You should only need to use the following buffer file functions:

- **BufFileCreateTemp(bool interXact):** This is the equivalent of tmpfile() of the C standard library. It will return a pointer to the associated file structure, which is allocated in the current memory

context. If interXact is true, the caller had better be calling us in a memory context that will survive across transaction boundaries. For this project, interXact will generally be false.

- **BufFileSeek(BufFile *file, int fileno, long offset, int whence):** This is the equivalent of fseek(). The only extra argument is fileno. The operating system limits the maximum size of a single file (typically to 2^{31} bytes, i.e. 2 Gbytes for 32-bit architectures). However, a database file can grow larger than the maximum physical file size, so PostgreSQL implements BufFiles as a collection of physical disk files. So file offsets are specified by a fileno, offset pair (where offset refers within the fileno-th physical file). In any case, you will need to seek to the beginning of the file before you start reading from it, which you can do (for BufFile* fp) with BufFileSeek(fp, 0, 0L, SEEK_SET).

- **BufFileClose(BufFile *file):** The equivalent of fclose(), will close the file and de-allocate the space occupied by the file structure.

In addition, BufFileRead(BufFile *file, void *ptr, size t size) and BufFileWrite(BufFile *file, void *ptr, size t size) read and write, respectively, from a file. However, we provide two functions, hash_read_tuple and hash_write_tuple, which are easier to use. These functions can read tuples from and write tuples into, respectively, a BufFile. In the PostgreSQL executor, tuples are accessed from TupleTableSlot structures, pointers to which are passed between between operators as part of the iterator model. hash_read_tuple and hash_write_tuple use a TupleTableSlot as the unit of I/O, and should be used instead of BufFileRead and BufFileWrite.

- **hash_write_tuple(BufFile *, TupleTableSlot *):** write tuple in TupleTableSlot into file

- **hash_read_tuple(BufFile *,TupleTableSlot *):** read tuple from file into TupleTableSlot

Note that tuples from the input operator are stored in the outerSlot variable in the agg_fill_hash table function. However, you will need another slot for reading tuples from temporary files. We have created and allocated the batchSlot field of the AggState structure for this purpose.

Note: You have to be careful about the “memory context” in which BufFile structures are allocated. We discuss memory contexts in more detail next.

3.6 Managing Memory with Contexts

PostgreSQL uses its own memory manager, which performs functions similar to malloc and free that you are familiar with. However, instead of allocating memory from a single global heap, the PostgreSQL allocators work off an abstraction called memory contexts for convenience and performance. Essentially, each allocated chunk of memory belongs to a particular context and all chunks in a context can be deallocated in one function call.

Imagine that at some point you allocate a relatively complex data structure (e.g., a linked list, or a tree, or a hashtable!). Normally, in order to free the memory it uses, you would have to traverse the structure and free each node individually. This traversal is both tedious to write and expensive to perform. Instead, PostgreSQL allows you to do the following:

```
/* Creates & assigns new context to treeCxt */
treeCxt = AllocSetContextCreate(...);
while (...)
{
      /* Allocate sizeof(TreeNode) bytes from treeCxt */
      newNode = MemoryContextAlloc(treeCxt, sizeof(TreeNode));
      ...
}
/* Free the entire treeCxt context */
```

```
MemoryContextDelete(treeCxt);
```

In this code, you can use newNode like any other piece of allocated memory, until you call MemoryContextDelete(treeCxt). Instead of MemoryContextDelete(), you can use MemoryContextReset() if you want to free all the memory chunks allocated in a context, but still use the context to allocate new memory. For convenience, PostgreSQL provides the functions palloc and pfree which operate on a default memory context (called the current context and pointed to by the CurrentContext global variable). Thus

```
/* Alloc n bytes from the cxt memory context */
ptr = MemoryContextAlloc(cxt, n);
```

is exactly equivalent to

```
/* Current context:save in oldCxt,switch to cxt */
oldCxt = MemoryContextSwitchTo(cxt) ;

/* Allocate n bytes from current (cxt) context */
ptr = palloc(n);

/* Restore current contextto oldCxt */
MemoryContextSwitchTo(oldCxt);
```

While this approach can save you typing and is used in the code that you need to understand, you have to be very careful if you use this approach. It is your code's responsibility to properly manage (i.e., set and restore) the current memory context. Be careful to avoid insidious bugs, in which you forget to reset the memory context to what it was, and confuse another piece of the code!

Finally, memory contexts are organized in a hierarchy. When creating a new context with AllocSetContextCreate(), the first argument is the parent context. Deleting a context also deletes all its child contexts at the same time, though you shouldn't need to exercise this feature. The file postgres/src/backend/util/mmgr/README contains a detailed description of the PostgreSQL memory manager, although the information above should suffice.

3.6.1 Allocating a per-run context

You will need to be able to efficiently deallocate all memory occupied by the hash table after you are done processing one batch. The hash table may grow fairly large and traversing each individual bucket to deallocate the space it occupies is tedious and time consuming. Therefore, you should create a separate memory context for the partition files and anything other data that you keep between runs. This way, you can reset aggstate->aggcontext whenever you start a new run.

4. Performance study: Sorting vs Hashing

In this part of the project, you will conduct a performance study with a single query and different data sets to understand the relative costs of the implementations of the two strategies (sort and hash). We will provide you with the data sets and the query. You must run the experiments using our scripts and interpret the results for us.

The query is:

```
SELECT avg(col3), sum(col2), col1, max(col4), min(col5)
FROM <table>
GROUP BY col1;
```

The two datasets are represented by tables R and S that have the same size. However, the number of distinct values of col1 in each table are very different. You can use the scripts/initexp.sh script to create these two tables and insert them into the database. To use scripts/initexp.sh, cd to the scripts directory, and run initexp.sh with the following arguments: <DATADIR> refers to a PostgreSQL directory that initdb will create and <DBNAME> is the database name you want to use for the tests.

After the data is loaded with scripts/initexp.sh, you can run scripts/runall.sh, which runs the query with different aggregation strategies and different HashMem and SortMem values. scripts/runall.sh takes the same arguments as scripts/initexp.sh, as well as <LOGBASE>, which is the common prefix for the logfile for each run of the experiment. You can run scripts/results.sh on the resulting log files with common prefix <LOGBASE> to extract the performance information.
Note that you have to run results.sh in the same directory that the log files reside in for the script to work properly. The easiest way to deal with this (though not the most elegant) is to just produce the log files in the scripts directory.

Also, you should use the instructional machines to run your experiments, as there will be differences in the data your machine and these machines generate due to different random number generators.

Again, you should only run your code on the specified ***Linux*** EECS instructional machines (either ilinux1 or ilinux2). Also, remember to be your data directory in your tmp directory or the larger experiments will cause your disk quota to be exceeeded. Make sure you are consistent and use the same <DATADIR>, <DBNAME> and <LOGFILEBASE> in each script that follows. You should be able to read off your results from the output of the results.sh script.

1. **Initialize setup:** ./initexp.sh <DATADIR> <DBNAME>
2. **Run experiments:** ./runall.sh <DATADIR> <LOGBASE> <DBNAME>
3. **Produce results:** ./results.sh <LOGBASE>

You should do a back of the envelope sanity check to determine if the I/Os you obtain are reasonable. To do this back of the envelope calculation, remember that the system counts the last hash read tuple call for each partition (which returns NULL) as an I/O. Also, a disk page in PostgreSQLis 8K. You should also keep in mind that gendata uses a random number generator, so it may not produce tables with as many distinct groups as you asked for. Of course, your back of the envelope calculation is an estimate. If your I/Os are within a factor of 2 or 3 of your estimate and they seem reasonable in other ways (e.g. smaller HashMem yields more I/Os), you're probably on the right track.

Also, remember to turn off DEBUG log messages when running your experiments to measure time. Writing too many messages to the logfile may cause a serious performance hit! In addition, answer the following questions. Some of these may require running more experiments than those described above. You can easily run more experiments by using (and slightly modifying) the programs and scripts used by initexp.sh and runall.sh. Specifically, you should look at init.sh, run.sh, and gendata. (Hint: running these experiments is a great way to test your code with cases similar to those that the TAs might use to grade your code!)

Questions:

1. Which of the tables (R or S) has more distinct values for col1?
2. When were the I/Os non-zero and why?
3. Is sorting better than hashing? When would you expect sorting to be a better approach and why?
4. How would you change your implementation to facilitate a "HAVING" clause in your query? What about a "SORT BY" clause?

5. Resources

In addition to lecture notes, you might find some of the following material interesting, though unnecessary to complete this project:

1. Textbook [2], Section 14.6, pp 469–471

2. Survey paper [1], Sections 2,4.2,4.3,4.4.

If you are in the *berkeley.edu* domain you can download the survey paper from the ACM Digital Library website: http://www.acm.org/dl. Contact us if you need help getting these materials.

6. What to submit

You must submit 4 files (even if they are incomplete or unchanged):

- README - the members of your group (names and class accounts), what works, and what doesn't.

- Files for your spilled aggregate implementation:
  - nodeAgg.c
  - execnodes.h

- perf.txt - TEXT ONLY! Performance comparison, table, answers to the questions.

Each group only needs to submit once. If there are multiple submissions, the last submission by any group member will be used for grading and the calculation of slip days.

7. References

[1] Goetz Graefe. Query evaluation techniques for large databases.
ACM Computing Surveys, 25(2):73–170, June 1993.

[2] Raghu Ramakrishnan and Johannes Gehrke. Database Management Systems.
McGraw Hill, 3rd edition, 2003. 10