PostgreSQL Query Processor: Project Assignment 2

CS186 Introduction to Database Systems
UC Berkeley

February 14, 2003
Due: March 7, 2003

1 Overview: Hybrid Hashing for Grouped Aggregates

In Project 1, you studied how to change the page replacement policy of the PostgreSQL buffer manager. 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, both in terms of the amount of coding involved and in understanding existing code. 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 and understanding existing code
4. Enhancing an implementation to be more "real world"

In lecture 7, you saw 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. When there are many *distinct* group values, the in-memory hashing implementation performs poorly. Your job is to understand the code that we provide, and enhance it to deal with insufficient memory by *spilling* data to disk.

1.1 Administrivia

As with Project 1, we will provide you a leaner PostgreSQL source tree. If you still run out of space, use the `mkhometmpdir` command. Make sure your code runs smoothly on EECS instructional machines (rhombus and pentagon) prior to submission since ALL grading will done on those. As before, you may find it easier to develop your code at home or on other machines you have access to. However, 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 EECS instructional machines. If you were able to get Project 1 working on your home machine, you should not face any problems this time around. We will make a version of the code available for this purpose. Watch out for an announcement.

1.2 Tasks

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

Task	Name	Credit
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.	75%
4	Compare the performance of sorted and hashed aggregation	25%
5	Understand and implement recursive partitioning for **very large** data sets.	30% *Bonus*

Please remember that you only have 3 weeks. 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.

2 Compile and Test PostgreSQL

Everything needed for this project is available at `~cs186/sp03/Hw2/hw2_pkg.tar.gz` on all instructional machines. To begin, copy this package to your own directory and untar it with: `gtar -zxvf hw2_pkg.tar.gz`[1].

You will see the following two subdirectories in your `Hw2/` directory:

1. `postgresql-7.2.2/`: Source code of a version of PostgreSQL based on the 7.2.2 release. This distribution is a special cs186 variant of the new in-memory hashing strategy for grouped aggregates that has just been developed by the PostgreSQL Global Development Group. You will not find this particular version at any other location, so make sure you use it. You will be finishing the job started by the PostgreSQL developers !
2. `exec/`: Some scripts you need to run experiments.

To compile and run PostgreSQL you just have to `cd` to the `Hw2/postgresql-7.2.2` directory and execute `./compile.sh`. There is no need to run the "configure" utility. After running `compile.sh` the PostgreSQL binaries will be installed in your `$HOME/bin/pgsql-7.2.2-hw2` directory. Note that your PATH is set to pick up this local PostgreSQL installation first. Test this by running `which postgres` to confirm that you are accessing executables from this directory.

3 Examine code for aggregation: Sorting and Hashing

As stated earlier, the implementation of PostgreSQL that we are providing is capable of using either a *sort* or a *hash* based strategy for grouped aggregation. You will find the details in the `Group` and `Agg` operators in source files `nodeGroup.c`, `nodeAgg.c` and `aggHash.c`. These modules are all part of the PostgreSQL query executor and can be found in the `postgresql-7.2.2/src/backend/executor` directory. In this section we will provide you with a short tour of the modules and routines that you should study.

3.1 Enabling a specific strategy

The scripts that we provide in `Hw2/exec` for running experiments can be used to selectively enable and parametrize the appropriate strategy for grouped aggregation in PostgreSQL. However, for you to understand and debug the code it is useful to know what's going on.

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 parameteres `sort_mem` and `hash_mem`. These parameters affect the private memory that is used by the two strategies. The easiest way to set these parameters is using the `-S` and `-H` options to the postmaster.

Note that PostgreSQL is a little sloppy in accounting for memory usage. Unlike what was explained in lecture, the `sort_mem` does *not* include the input and output buffers used with each run. You will follow the same measurement convention in your hash implementation.

The parameter `hash_mem` is actually available in the code as the global variable `HashMem`. This is the amount of memory available for the in-memory hash table in Phase 1 of hybrid hashing. Note that this variable is not used in the implementation provided—however, *you* will use it in your implementation !

[1] Remember to remove the hw2_pkg.tar.gz file if you are out of disk space.

3.2 An Agg overview

The Agg operator *combines* the implementation of the sorting and hashing based strategies for grouped aggregates. The choice is made by the optimizer and fixed in the aggstrategy field of the Agg node. The entry point to the Agg node is the function ExecAgg in the file nodeAgg.c – this function calls one of exec_sorted_agg or exec_hashed_agg based on the aggstrategy. While you need not read the former, make sure that you understand every detail in exec_hashed_agg, as this is the code path that you will modify.

Note that you might *find it necessary to change the interfaces of some of the functions we have provided.* In particular you should carefully examine the following:

1. Building, using and freeing hash tables

   - build_hash_table() and BuildTupleHashTable() (pay attention to the memory context explained in subsection 4.6).
   - GetTupleHashKey() – note that this can also be used for your partitioning hash function.
   - lookup_hash_entry() – when you are in Phase 2 you will need to check for existence without inserting into the hash table.
   - TupleHashSize – a macro that keeps track of memory usage in the hash table.

2. Transition values (building and maintenance): The AggStatePerAggData structure contains all the information for each aggregate *function* that you need to evaluate. This structure contains all the information necessary about how to initialize and advance transition values. The AggStatePerGroupData structure holds the aggregate function transition *values* for each distinct group-by value. The following functions should be fairly self-explanatory:

   - initialize_aggregates(): Initializes a per-group structure based on the information of the per-agg structure.
   - advance_aggregates(): Advances the transition value in the per-group structure, based on the new tuple values and the aggregate function information in the per-agg structure.
   - finalize_aggregates(): Computes the final aggregate function value from the transition value information in the per-group state and the aggregate function described by the per-agg structure.

   For the most part, these functions are called in the appropriate places already. However, when spilling to disk, you will need to move things around in the code, so you should be aware of what these functions do.

3. Cleanup. It starts off in ExecEndAgg() and is useful for similar work you might want to do in cleaning up an existing in-memory hash table. You may also want to see the code in nodeHash.c and nodeHashjoin.c (see also subsubsection 4.6.1).

4 Implement hybrid hashing

Your task is to extend the hash-based implementation of grouped aggregates to efficiently support both small and large hash tables via the hybrid hashing scheme presented in class. The code that we provide assumes that the number of distinct groups is small, so that the hash table used to store group values and their associated aggregate state information, fits in HashMem KB of memory.

4.1 Where to make changes

You should not change anything apart from the following files.

1. src/backend/executor/nodeAgg.c

2. `src/backend/executor/aggHash.c`
3. `src/include/executor/aggHash.h`
4. `src/include/nodes/execnodes.h`

The `Hw2/exec` directory contains scripts that can aid you in producing smaller test cases for debugging. You are encouraged to look at `exec/init.sh` – see how `initexp.sh` calls `exec/init.sh` for more details.

4.2 What to implement

Here is a brief sketch of the algorithm you have to implement. Much of this is already in the implementation provided (we'll call this "the code" from now on). You have to modify the code to implement spilling in the first pass and then process the tuples in each spilled partition. This is not very different from the way tuples are processed during the first pass, except that they are fetched from the spilled partition on disk, as opposed to from the input iterator over the heap table.

```
// Initial pass
Initialize the hash table
while (tuple $\leftarrow$ get_next()) $\neq$ NULL do
    (hashKey, exists) $\leftarrow$ lookup_hash_table(tuple)
    if exists then
        update trans-value in hash
    else if hash_table_size $\leq$ HashMem then
        initialize new trans-value
        insert (group-key, trans-value) in hash
    else
        if necessary, initialize temporary files
        partition $\leftarrow$ 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 $\leftarrow$ read_tuple(tempFile[i]) do
        (hashKey, exists) $\leftarrow$ lookup_hash_table(tuple)
        if exists then
            update trans-value in hash
        else
            initialize new transition value
            insert (group-key, trans-value) in hash
        endif
    endwhile
    for each (group-key, trans-value) in hash do
        append finalize(trans-value) to output stream
    endfor
endfor
```

This is essentially what was covered in lecture, with one important difference: In lecture, we learnt that `B`, the amount of buffers available for the HashAgg 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 and its contents fits in `HashMem`. Accordingly, you should assume that the buffers used for the temporary files are not accounted for as part of the `HashMem` parameter.

We will now go over some of these steps in more detail. The main ideas are: the size of hash tables (subsection 4.3), the number of partitions (subsection 4.4), managing temporary files (subsection 4.5) and managing memory (subsection 4.6).

Pay careful attention to estimating the size of hash tables and the number of partitions as they are key to getting the project right. From now on, will use `node` to refer to the `Agg` node. Note that you might have to change header files. In particular you might want to change the `AggState` structure defined in the `execnodes.h` header file.

4.3 Initializing the hash table

Your first task is to initialize the hash table appropriately. In the code, this is done in `build_hash_table`, where an 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 maximum number of entries to be twice the number of buckets. Given this information, you must calculate the maximum number of entries (`max_entries`) that can be accomodated in the hash table. You can then use `(0.5 * max_entries)` as the number of buckets that is passed to `BuildTupleHashTable`.

The `TupleHashSize` macro defined in `aggHash.h` 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 for your computation. Remember that `HashMem` is expressed in kilobytes.

4.4 Estimating number of partitions

When the hash table grows large enough you have to begin spilling. This condition can be checked by using the `TupleHashSize` macro. At this point you need to decide upon the number of partitions to spill non-matching tuples into. Let `tuples_so_far` represent the number of tuples seen so far. The expression `outerPlan(node)->plan_rows` can be used for the total number of records of the input table. Let `input_size` represent this value.

Assume that the distribution of group values does not change in the rest of the input table. Now you can use `input_size / tuples_so_far` to estimate the number of partitions.

Note that `input_size` is only an estimate of the number of records in the input table. To ensure that this estimate is accurate, you must run the `ANALYZE` command after you create your tables. While our test scripts will do this, you may want to do this yourself if necessary (i.e., if you create or modify a table for testing purposes) while debugging your code. This can be done from within `psql`:

```
test=# ANALYZE table;
ANALYZE
```

Beware of corner cases: Since this value is only an estimate, 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.

4.5 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 `postgresql-7.2.2/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. *Note:* You have to be careful about "memory context" in which `BufFile` structures are allocated. More details in subsection 4.6. You should only need to use the following functions:

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

- `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 only have to seek to the beginning of the file, which you can do 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.

Also, for completeness, here are two more file-related functions:

- `BufFileRead(BufFile *file, void *ptr, size_t size)`: This is the equivalent of `fread()` and will read `size` number of bytes from `file` into `ptr`. As with the C standard library, you have to make sure that you do not overrun the space pointed to by `ptr`.

- `BufFileWrite(BufFile *file, void *ptr, size_t size)`: This is the equivalent of `fwrite()` and will write `size` bytes from `ptr` into `file`.

In the PostgreSQL executor tuples are accessed from `TupleTableSlot` structures, pointers to which are passed between between operators as part of the iterator model. We provide two functions, `hash_read_tuple` and `hash_write_tuple` to write tuples into and read tuples from a `BufFile`.

- `hash_write_tuple(BufFile *, TupleTableSlot *)`: Write a tuple in the `TupleTableSlot` into a file.

- `hash_read_tuple(BufFile *,TupleTableSlot *)`: Read a tuple from a file into a `TupleTableSlot`.

Note that tuples from the input table are stored in the `outerSlot` local variable in the `agg_fill_hash_table` function. You will however need another slot for reading tuples from temporary files. We have created and allocated the `batchSlot` field of the `aggState` structure for this purpose.

4.6 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 can be deallocated *in one shot.* 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:

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

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

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

is exactly equivalent to

```
oldCxt = MemoryContextSwitchTo(cxt)       /* Current context:save in oldCxt,switch to cxt */
ptr = palloc(n)                           /* Allocate n bytes from current (cxt) context */
MemoryContextSwitchTo(oldCxt)             /* Restore current contextto oldCxt */
```

If you are performing several allocations, this can save you some typing. You can change the current context using `MemoryContextSwitchTo(cxt)`. *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.

The file `postgresql-7.2.2/src/backend/util/mmgr/README` contains a detailed description of the PostgreSQL memory manager, should you need it, although the information above should suffice.

4.6.1 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 hash table (which is the `hashcxt` argument to `BuildTupleHashTable()`).

You may want to look at how `hashCxt` is used in `nodeHash.c` and `nodeHashjoin.c` for exactly the same purpose. Be careful not to allocate anything that should persist across contexts, such as the `BufFile` structures for your temporary files.

4.7 Query plans and `explain`

In the process of debugging your implementation, you may want to see how the query planner has chosen to use your operator and where it fits into the overall query plan. To this end, you can use `explain` in `psql`. If you prefix a query with `explain`, the DBMS will complete all the stages up to and including query planning. However, instead of executing the plan and returning the results, it will stop at that point and print out the plan. For example, if you have not enabled the hashed aggregate mode:

```
test=# EXPLAIN SELECT a, AVG(b) FROM table GROUP BY a;
NOTICE:  QUERY PLAN:

Sorted Aggregate  (cost=1.14..1.17 rows=1 width=8)
  ->  Group  (cost=1.14..1.15 rows=6 width=8)
        ->  Sort  (cost=1.14..1.14 rows=6 width=8)
              ->  Seq Scan on table  (cost=0.00..1.06 rows=6 width=8)

EXPLAIN
test=# EXPLAIN SELECT a, AVG(b) FROM table GROUP BY a ORDER BY a;
NOTICE:  QUERY PLAN:

Sorted Aggregate  (cost=1.14..1.17 rows=1 width=8)
  ->  Group  (cost=1.14..1.15 rows=6 width=8)
        ->  Sort  (cost=1.14..1.14 rows=6 width=8)
              ->  Seq Scan on table  (cost=0.00..1.06 rows=6 width=8)

EXPLAIN
```

At this stage, you can ignore the information contained in the parentheses. This tells you that your query would be executed by scanning the entire `table` relation (operator `Seq Scan`), sorting it (operator `Sort`) then feeding the sorted result into the `Group` operator (which adds NULL delimeters between groups) and finally executing the aggregate operator in sorted mode.

When you enable hashed aggregates, you should see something like:

```
test=# EXPLAIN SELECT a, AVG(b) FROM table GROUP BY a;
NOTICE:  QUERY PLAN:

Hashed Aggregate  (cost=1.14..1.17 rows=1 width=8)
  ->  Seq Scan on table  (cost=0.00..1.06 rows=6 width=8)

EXPLAIN
test=# EXPLAIN SELECT a, AVG(b) FROM table GROUP BY a ORDER BY a;
NOTICE:  QUERY PLAN:

Sort  (cost=1.18..1.18 rows=1 width=8)
  ->  Hashed Aggregate  (cost=1.14..1.17 rows=1 width=8)
        ->  Seq Scan on table  (cost=0.00..1.06 rows=6 width=8)

EXPLAIN
```

Note that in this case, since the hashed aggregate produces its results in a random order, its output needs to be explicitly sorted (`Sort` operator at the top) when you add an `ORDER BY` clause.

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

The two datasets are represented by tables `R` and `S` that have the same size. The number of distinct values of `col1` are however very different. Use the following procedure after changing your directory to `hw2/exec`. In the scripts `<DATADIR>` refers to a PostgreSQL directory that `initdb` will create, `<DBNAME>` is the database name you want to use for the tests, and `<LOGFILEBASE>` is the common prefix for the logfile for each run of the experiment. Make sure you are consistent and use the same `<DATDIR>`, `<DBNAME>` and `LOGFILEBASE` in each script.

Enter your results in the tables below: I/Os in Figure 1 and elapsed time in seconds in Figure 2. 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>`

In addition, answer the following questions:

1. Which table has more distinct values for `col1`
2. When were the I/Os non-zero and why ?
3. Is sorting better than hashing ?
4. The elapsed time might not reflect the number of I/Os. Why might this be so ?

	32KB				128KB				1024KB			
	R		S		R		S		R		S	
Strategy	Read	Write	Read	Write	Read	Write	Read	Write	Read	Write	Read	Write
Sort												
Hash												
Hash Spill												

Figure 1: Experimental Results (I/Os): Blocks read and written

	32KB		128KB		1024KB	
Strategy	R	S	R	S	R	S
Sort						
Hash						
Hash Spill						

Figure 2: Experimental Results (Total Time): Seconds

6 Implement recursive partitioning (30% Bonus)

The version of spilling you have to implement has one drawback. During the initial pass over the input tuple stream, the hash table is guaranteed not to exceed HashThreshold. However, when subsequently loading the hash table with each partition, it is conceivable that the hash table may exceed HashThreshold. If the hash function has done a good job, then distinct values *of the group-by key* should be evenly distributed among runs (*regardless* of how many times each of these values occurs in the tuple stream—why?). Therefore, HashThreshold should not be exceeded by much; you can think of it as a *soft threshold* that will be exceeded with low probability. However unlikely, the possibility remains that the table will grow excessively large.

One simple approach to enforce a hard threshold on the hash table size is the following: First, fill the hashtable with up to HashThreshold unique values and spill the tuples that didn't make it into a single disk file. Then, repeat this process iteratively, treating the tuples in the disk file as the original input and creating another temporary file for those that again didn't make it. When all tuples have made it, we're done.

The main problem with this approach is inefficiency: If a tuple makes it during iteration $i + 1$, it will have been written to disk *exactly* i times. Reading and writing tuples to disk multiple times is inefficient. As explained, we expect the threshold to be exceeded infrequently and we want to incur the cost of extra writes *only when absolutely necessary.*

Another approach would be to store hash buckets instead of tuples on disk. However, with this approach, you again have to access the disk to update the transition value for *each tuple* that does not fit in memory during the original pass through the data.

This suggest a recursive partitioning approach (detailes in [2], section 2.2, pp 92-93). Tuples are split into partitions. When loading a partition, it is recursively split *only if* the hash table size is exceeded. Since small variations in the distribution of unique group-by values is expected, it is a good idea to use a hard threshold that is slightly larger than HashThreshold. This should avoid unnecessary re-partitioning that will lead to sub-partitions with very few distinct group-by key values.

In order to implement this algorithm, you will have to manage a nested structure (and the associated memory contexts for each nesting level) instead of a plain array of partitions.

7 Resources

In addition to lecture notes, you might find some of the following material useful:

1. Textbook [4], Section 14.6, pp 469–471

2. Survey paper [2], Sections 2,4.2,4.3,4.4.
3. Unary Hashing paper [1]
4. Hybrid Cache paper [3]

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

8 What to submit

You must submit at least 6 files (even if they are incomplete or unchanged):

1. README - the members of your group (names and class accounts), what works, and what doesn't. If you have implemented recursive partitioning make sure you state it.
2. Files for your spilled hashAgg implementation
   (a) `nodeAgg.c`
   (b) `aggHash.c`
   (c) `aggHash.h`
   (d) `execnodes.h`
3. perf.txt - TEXT ONLY! Performance comparison, tables, answer to the questions.
4. *Optional:* `hashaggrp.diff`: Your recursive partitioning implementation. Follow the instructions for generating a diff from our distribution in: `src/tools/make_diff/README` and call your file `hashaggrp.diff`

Make sure all the files above are in a directory called `~/Project2` in one of your group member's class account. `cd` to that directory, and type `submit Project2` to submit the files. 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.

References

[1] Kjell Bratbergsengen. Hashing methods and relational algebra operations. In *Proceedings of 10th International Conference on Very Large Data Bases*, pages 323–333, August 1984.

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

[3] Joseph M. Hellerstein and Jeffrey F. Naughton. Query execution techniques for caching expensive methods. In *Proceedings of ACM SIGMOD International Conference on Management of Data*, pages 423–464, 1996.

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