Homework Assignment 2

15-415 Database Applications
Carnegie Mellon University

February 10, 2003
Due: March 3, 2003 (8pm)

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 values* grouping columns, 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. Make sure your code runs smoothly on the Itanium machines prior to submission, since *all* grading wilbe done on those. The instructions provided are designed to work smoothly on these machines, which are the only ones we will officially support.

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 aggregation implementation based on sorting and hashing that is 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 `/usr0/dbclass/hw2/hw2-pkg.tar.gz` on all Itanium machines. To begin, copy this package to your own directory and untar it with: `tar -zxvf hw2-pkg.tar.gz`. You will see the following two subdirectories:

1. `postgresql-7.2.2/`: Source code of a version of PostgreSQL based on the 7.2.2 release. This distribution includes a *backport* of the 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.

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/pgsql-hashagg/bin` directory. Note that your `PATH` should be 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`or 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, 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.

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 !

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)
   - 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 valus 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. Starting off in ExecEndAgg(), it's 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 subsection 4.3).

3.3 Memory 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(...)
while (...)  {
    newNode = MemoryContextAlloc(treeCxt, sizeof(TreeNode))
    ...
}
MemoryContextDelete(treeCxt)
```

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)
```

is exactly equivalent to

```
oldCxt = MemoryContextSwitchTo(cxt)
ptr = palloc(n)
MemoryContextSwitchTo(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 !*

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 above information is more than sufficient for what you need to do in this assignment.

3.4 Tuple table slots

Examine the PostgreSQL implementation of the *iterator* model. A good place to start in the executor component is the module `execProcnode.c` that contains dispatch functions that call the appropriate operator-specific methods. Observe that the function `ExecProcNode` and indeed, all the functions it calls such as `ExecAgg` and `ExecSort` return pointers to `TupleTableSlot` structures. This is defined in the file `src/include/executor/tuptable.h`. PostgreSQL tuples (of type `HeapTuple`) are exchanged via `TupleTableSlot` objects. Each `TupleTableSlot` contains a `val` field which points to a `HeapTuple`.

Heap files are just an unordered (hence the name) collection of tuples. The heap manager is built on top of the buffer manager, which you saw in the first assignment. The buffer manager knows only of pages; it is the heap manager's job to keep track individual tuples within pages. Tuples that are read from disk reside in the very buffer slots you saw in the previous assignment!

However, during query execution, tuples are passed around executor nodes and copies are made in memory locations outside the buffer pool. It is necessary to keep a slew of "bookkeeping metadata" (such as if the tuple is in a buffer slot, is the page pinned in the buffer, and so on). The *tuple table* is a structure that allows the executor to store all this information in a central place, instead of explicitly passing it around all the time, wasting time and space. A tuple table *slot* is an entry in the tuple table that points to the actual tuple data (through the `val` field), as well as all the necessary "bookkeeping metadata".

Thus, the tuple table makes your life much easier! For this assignment, you should not have to deal with this metadata. All you need to do is remember to pass tuple table slots around and access the actual tuples via the `val` field of a slot.

4 Implement hybrid hashing

Here is a a brief sketch of the algorithm you have to implement:

```
// Initial pass
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 (group-key, 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 (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
```

Much of this is already in the implementation we have given you—you have to modify the code to implement spilling in the first pass and then processes each partition (which isn't essentially different from the way tuples are processed during the first pass, except for the fact that they are fetched from disk rather than produced by the outer plan). In the following sections we describe various aspects that you should pay attention to while doing the actual implementation.

4.1 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 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. *The only thing you have to be careful about is the memory context in which `BufFile` structures are allocated.* 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 the size of `long`, 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 write `size` number of bytes from `ptr` into `file`.
- `BufFileWrite(BufFile *file, void *ptr, size_t size)`: This is the equivalent of `fread()` and will read `size` bytes into `ptr`. As with the C standard library, you have to make sure you wont overrun the space pointed to by `ptr`.

These last two functions are used by functions which we already provide for you:

- `hash_write_tuple()`: Write a `HeapTuple` into a file.
- `hash_read_tuple()`: Read a tuple from a file into a `TupleTableSlot`.

These are all you should need to use (see also subsection 4.2).

4.2 Batch tuple slot

In order to make things easier for you, we have allocated a tuple table slot into which you can read tuples from the temporary files. This slot is pointed to by the `batchSlot` field of the aggregate node's state structure. All you need to do is make sure to pass this slot to `hash_read_tuple()`. For convenience, this function returns a pointer to this slot, after it has properly read the tuple from the temporary file.

4.3 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. As explained in subsection 3.3, this is one of the primary reasons for using memory contexts. 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 runs in this context (such as the `BufFile` structures for your temporary files)!

4.4 Setting the initial hash table size

The fourth argument (`nentries`) to `BuildTupleHashTable()` is the size of the array of pointers to bucket chains. This should be proportional to the number of entries that the hash table is expected to have; a good rule of the thumb is to set that to twice the expected hash size in order to avoid excessively long bucket chains.

In `build_hash_table()`, this parameter is set to the product of `HASH_FUDGE` and the expected number of unique group-by values. The latter is estimated by the planner simply as 10% of the outer plan's `plan_rows`. You should define the former as some *reasonable* value. It is entirely possible that, with this initial size, the hash table will exceed `HashMem` Kbytes immediately after its creation. Thus, you need to ensure that the initial size of the hashtable is such that it will fit in the allowed space. You can do this by choosing to reduce the estimate of unique group-by values so that the initial hash table fits in `HashMem` Kbytes.

4.5 Estimating number of partitions

When the hash table grows large enough so you have to begin spilling, you need to decide upon the number of partitions. One way to come up with an estimate for this number is to assume that the distribution of unique values will not change. In this case, you can set the number of partitions to the number of tuples you expect to see divided by the number of tuples you've seen up to the point you begin spilling. You can keep track of the latter number easily. You are only missing the overall number of tuples you expect to see. An *estimate* of this is stored in the `plan_rows` field of each plan node. You can examine this field in the outer plan of your aggregate node implementation. Thus, assume that in your `exec_hashed_agg(Agg *aggNode)` implementation, you keep a counter of tuples seen so far, which you increment each time you fetch a new tuple from the subplan:

```
Plan *outerPlan = outerPlan(aggNode);
int   tuples_so_far = 0;
[...]
outerslot = ExecProcNode(outerPlan, (Plan *) aggNode);
++tuples_so_far;
```

Then, when you need to decide on the number of partitions, you can use something like:

```
int numBatches = FUDGE_FACTOR * (outerPlan->plan_rows / tuples_so_far);
```

As with `HASH_FUDGE`, choose a reasonable value for `FUDGE_FACTOR`. Remember, however, that `outerPlan->planRows` is only an *estimate*. If the input stream of the aggregate operator is a table on disk, then the estimate will be the correct number of tuples, (provided the ANALYZE utility has been run, as explained in subsection 4.6). In general, the input stream from the subplan may be the result of several other operators (such as a join), in which case it is difficult (if not impossible) to know the number of tuples that it will produce.

Therefore, you should check that the number of partitions you estimate in this way actually makes sense (e.g., if it is less than 1, you probably want to set it to a reasonable number!). Conversely, if the estimate is too large, you probably want to reduce it to something reasonable (you don't want to start creating thousands of temporary files!).

4.6 Maintaining statistics

For the optimizer to produce "good quality" plans, it needs to have estimates of the data in the various tables in the system. This would include the number of rows in a table, the most common values for each column etc. In PostgreSQL the `ANALYZE` command is used to update these estimates at administrator controlled intervals. You can use `ANALYZE` to update table statistics so that the estimates in `plan_rows` are not horribly off. You can do this from within `psql`:

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

The test scripts we have provided should do this for you, but 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.

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.

4.8 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`irectory 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.

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
```

The two datasets are represented by tables `R` and `S` that each have 1 million records. The number of distinct values of `col1` are however very different. Use the following procedure after changing your directory to `hw2/execnd` enter your results in the tables below: I/Os in Figure 1 and elapsed time in seconds in Figure 2. You will want to use at least the following scripts:

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

Please 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!

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

Figure 1: Experimental Results: I/Os

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

Figure 2: Experimental Results: Total Time (seconds)

In addition, answer the following questions:

1. Explain the behavior of your performance experiments
2. Under what circumstance would you expect sorting to be a better option *overall*.

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 Submitting your solution

You should submit a tarball with the filename `hw2-solution.tar.gz` containing the following:

- A subdirectory `backend/executor` which at the very least should contain the files `aggHash.c` and `nodeHash.c` (even if you haven't modified all of them).
- A subdirectory `include/executor` which at the very least should containt the files `aggHash.h` and `nodeAgg.h` (again, even if you do not modify all of them).
- A subdirectory `include/nodes` which at the very least should containt the file `execnodes.h`.
- A textfile `GROUP` as in the first assignment, with group member Andrew IDs and your grading of each group partner. Also, if you implement recursive paritioning, please make sure to state it clearly. At your option, you may also *briefly* comment on what works and what doesn't.
- A textfile `PERFORMANCE` with your results from section 5, i.e., the output from `results.sh`, along with your answers to the questions.
- *Optionally* a context diff `recursive.diff` of your recursive partitioning implementation. This is only if you need to modify files outside the directories allowed by the above scheme (see below). Follow the instructions in `src/tools/make_diff/README` on how to create the context diff from our distribution.

Thus, at the very least, your tarball contents should look something like:

```
$ tar -ztvf hw2-solution.tar.gz
GROUP
PERFORMANCE
backend/executor/aggHash.c
backend/executor/nodeAgg.c
include/executor/aggHash.h
include/executor/nodeAgg.h
include/nodes/execnodes.h
```

You can create this tarfile by going into `postgresql-7.2.2/src`, placing your textfiles in there and running `mkhandin.sh` (examine it to see exactly what it does—this is only for your convenience and *you* will need to make sure you include all the files). However, we allow you to modify and/or create more than these files, if you wish to (this *may* be useful if you decide to do the bonus section). The contents of the `backend/executor` subdirectory will be copied into `postgresql-7.2.2/src/backend/executor` and the contents of your `include/{executor,nodes}` subdirectories will be copied into `postgresql-7.2.2/src/include/{executor,nodes}` of a clean source tree during our testing. Thus, these are the only two places in the PostgreSQL source tree that you are allowed to make modifications and/or additions.

Make sure that you include all modified or added files (including Makefiles, if you have added source files) that are necessary for us to properly compile your solution! We will not accept solutions that do not compile properly!

If you submit on time, then the person submitting (only *one* group member has to submit) should place the tarball in the AFS directory `/afs/cs.cmu.edu/academic/class/15415/submit/`*`AndrewID`*`/hw2`. Your write permissions will be revoked when the deadline has passed. If you need to use slip days, you should email your tarball to the TAs instead.

8 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 `cmu.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.

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.