CS 4321/5321 Project 3 (Complete Instructions)

Fall 2016

Checkpoint due Monday October 17th, 11:59 pm
Project due Tuesday November 1st, 11:59 pm

This project counts for 24% of your grade, of which 6% is for the checkpoint and 18% for the main submission.

1 Goals and important points

In Project 2, you developed a lot of functionality. However, the priority was end-to-end evaluation of SQL rather than efficiency. So, you used line-at-a-time I/O. You also used a naïve implementation for joins (tuple-nested loop join, TNLJ) and for sorting (in-memory sort). Your sort implementation was particularly problematic because it kept *unbounded state* – the amount of memory required by the operator depended on the size of the input rather than being bounded by a constant.

For Project 3, you will address the above shortcomings:

- you will move from using line-at-a-time I/O to faster page-at-a-time I/O
- you will refactor your code to support multiple different physical implementations for each relational algebra operator
- you will implement Block Nested Loop Join (BNLJ), External Sort and Sort Merge Join (SMJ)
- you will ensure that you have at least one implementation for each operator that does not keep unbounded state
- you will do some performance benchmarking of your join implementations against each other

You will still support the same subset of SQL as for Project 2, and you should still construct the join tree in the same manner as in Project 2, following the query's `FROM` clause.

This is a challenging Project, including significant refactoring and nontrivial algorithms to implement. You have a substantial time window to do it, but this window also includes Fall Break and the 4320 prelim. Thus you need to budget your time wisely.

To encourage you to get started early, we are requiring a *checkpoint* submission on October 17th The checkpoint requirements are given in Section 5. Note that completing the checkpoint does not mean you have completed half the project - there is much more work to do after the checkpoint.

2 Input and output formats

2.1 Top-level inputs and outputs to your project

As in Project 2, we will run your code from the command line by exporting a runnable JAR and typing:

```
java -jar cs4321_p3.jar inputdir outputdir tempdir
```

This means your top-level class has to accept `inputdir`, `outputdir` and `tempdir` as command-line arguments and handle them appropriately.

`inputdir` is the directory where relevant inputs to your program will be found. This directory will contain:

- a `queries.sql` file as in Project 2.
- a `db` subdirectory. This will contain a `data` subdirectory and and a `schema.txt` file. The `data` directory contains files for the relations, with one file per relation as in Project 2, except that the files are now in a binary format described in Section 2.2. The `schema.txt` file is as in Project 2.
- a file `plan_builder_config.txt` which is the configuration file for the `PhysicalPlanBuilder` as described in Section 2.3.

`outputdir` is the directory where your program should write output. You may assume this directory will exist. As in Project 2, after running your code, the answer to the ith query (starting at 1) should be found in file `outputdir/queryi`.

`tempdir` is the temporary directory where your external sort operators should write their "scratch" files, a process described more fully in Section 3.4. Again, you may assume this directory exists.

2.2 Binary File Format

This describes the file format for input and output data in Project 3.

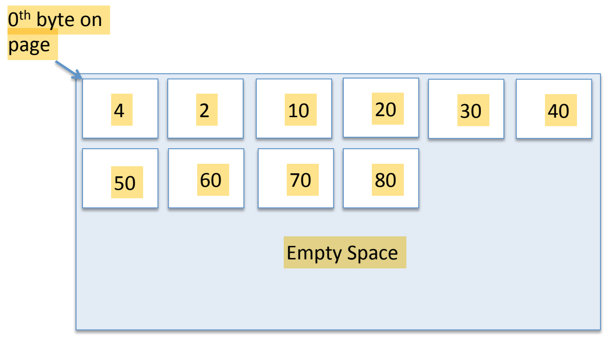
The motivation for moving to a new file format is that doing line-at-a-time file I/O is inefficient. See `http://www.idryman.org/blog/2013/09/28/java-fast-io-using-java-nio-api/` for an interesting performance comparison of various ways to do I/O in Java. In this Project, you will use Java NIO to do page-at-a-time I/O. A "page" for our purposes is a buffer, specifically a `ByteBuffer`. We will read the file in fixed-sized "chunks". This means we need a file format where we can read data in fixed-sized chunks rather than in variable-sized lines. (Lines are variable-sized in that each relation potentially has tuples of a different size/length.)

Your textbook in Chapters 9.5-9.7 contains an extensive discussion of file formats for databases. In this project, we can work with a fairly simple format as we are not supporting updates.

Every relation, whether it's a table in the database or a query answer, is stored in a single data file. Each data file is a sequence of pages. Every page is 4096 bytes in size. Thus, the first 4096 bytes of the file are the first page, the next 4096 bytes are the second page, and so on. This also means that every data file is a multiple of 4096 bytes in size - there are no "half pages."

Every page contains two pieces of metadata. The first is the number of attributes of the tuples stored on the page - we assume all tuples stored on the page have the same number of attributes. The second piece of metadata is the number of tuples on the page; this is particularly useful information in case the page is not full. The two pieces of metadata, as well as the tuples themselves, are stored as (four-byte) integers. You may assume all integer values will fall into the int domain (32 bits). You may also assume that all tuples are sufficiently small that you can fit at least one tuple per page.

The overall layout of metadata and data on a page is best illustrated by example. Suppose we have a relation which contains just two four-attribute tuples: (10, 20, 30, 40) and (50, 60, 70, 80). Then it fits in a single page, set out as in the image below (each "cell" represents a four-byte integer). **Any remaining empty space at the end of any page should be filled with zeroes.**



There are no "half-tuples" possible on a page. If the last tuple does not fit on the page (e.g. the tuple has 3 attributes so we need 12 bytes but the page only has 10 bytes left) it will be placed on the next page. If a page does not have space for any more tuples, it is *full*. Any relation that requires multiple pages is stored so that all pages except possibly the last one are full.

Your program must accept data in the binary format and produce outputs (answers) in the binary format. In the sample input and output we have provided, we give you each file in both binary format and in human-readable Project-2-style format as a courtesy to aid with debugging, but we will be testing with binary format files only.

When working with binary files, be aware that you can open, view and manipulate such files in programs called hex editors (or binary editors). This could be handy for debugging.

2.3 Format of the configuration file for `PhysicalPlanBuilder`

This section explains the format for the configuration file for your `PhysicalPlanBuilder`. If this is your first time reading the document, we recommend skipping it and returning to it once you have read Section 3 (and actually understand what the `PhysicalPlanBuilder` is!!). We have placed this section here because once you are familiar with the project, it will be easier to have all format related info in one place.

You may assume for this project that the `PhysicalPlanBuilder` will use the same join implementation for every logical join operator in the logical plan, and the same sort implementation for every logical sort operator in the logical plan. That is, you do not need the ability to generate physical plans that have a mix of TNLJ, SMJ and BNLJ physical operators, or a mix of BNLJ operators each with a different number of buffer pages, etc.

The configuration file has two lines; the first specfies the join method, the second the sort method. The join method is either 0 for TNLJ, 1 for BNLJ, or 2 for SMJ. If the join method is BNLJ, there is a second integer on the same line specifying the number of buffer pages to be used. The sort method is either 0 for in-memory sort, or 1 for external sort. If the sort method is external sort, there is a second integer on the same line specifying the number of buffer pages to be used in the sort. This will always be at least 3.

For example the file:

```
0
0
```

Specifies that you want TNLJ and in-memory sort, i.e. the Project 2 naïve implementation. The file

```
1 5
1 4
```

Specifies you want BNLJ with 5-pages outer relation buffers, and external sort with 4-page sort buffers.

3 Implementation instructions

First, some general remarks to consider as you begin the implementation.

You will begin with substantial refactoring of your Project 2 code; save snapshots of your code at intermediate stages in case you mess up. After each refactoring, be sure to test your code to verify you haven't broken any "old" functionality.

In this Project, your codebase will expand considerably. Do not skimp on comments and on setting up a good debugging infrastructure as you go; it will be particularly important to stay on top of your codebase and communicate with your partner(s) about APIs.

You will be working with bigger relations than in Project 2, so if you are relying on console output for debugging, you may want to move to a setup where your debugging statements get logged to a file instead. This avoids the problem where you have so many things getting output to the console that the first lines get truncated. One simple way to do file-based logging is to create a `Logger` class that uses the singleton pattern, that way every component in your code can call the logger as needed.

You should consider writing yourself a random data generator to test your code. This doesn't have to be fancy; it can create tuples by generating each attribute as a random integer in a specified range. The range you allow for attributes will dictate how many tuples "match" in joins and selections, so you should probably have that as a configurable setting in your data generator. For example, if you generate two relations with 5000 tuples each, and each attribute has values in the range 0 to 100, probably a lot of tuples will match if you try to join these two relations. If you generate the attribute values in the range from 0 to 10000, far fewer tuples will match.

Finally, be aware that once you are done with the two refactoring steps (Sections 3.1 and 3.2), the remaining three tasks can be completed in parallel if you are looking to split up work between partners. BNLJ and external sort can definitely be developed independently; SMJ requires a sorting implementation, but you can use your in-memory sort from Project 2 until your external sort is ready. Also, completing Sections 3.1 and 3.2 constitutes the requirements for the project checkpoint. All of these are great reasons to complete this portion of the implementation ASAP.

3.1 Refactor to use the new file format

Your first task is to refactor your Project 2 code to use the new binary file format for input and output. You are required to use Java NIO for this. If you are unfamiliar with Java NIO, start by reading this tutorial: `http://www.ibm.com/developerworks/java/tutorials/j-nio/j-nio.html` and looking at the documentation. We recommend using `ByteBuffer`s. They provide handy `getInt` and `putInt` methods that relieve you of the need to write code for serializing an integer into bytes.

We recommend doing this refactoring by setting up a layer of abstraction so that most of your code does not need to know about file I/O, or even what file format is being used. For example, you can create `TupleReader` and `TupleWriter` interfaces. `TupleReader` has a method to read the next tuple (and probably some bookkeeping methods such as close, reset etc). `TupleWriter` has a method to write a tuple. Then the rest of your code can just create TupleReaders/TupleWriters as needed; for example, your file scan operator can create a TupleReader and grab tuples from that instead of from the file directly. The logic for getting tuples from a page or writing them to a page is then encapsulated in concrete implementations of `TupleReader` and `TupleWriter`.

For the new file format, the `TupleReader` you implement will read the file one "page" at a time – i.e. it will have a running `ByteBuffer` which it will fill using `read` calls to an appropriate `FileChannel`. Then it can extract specific tuples from the page as needed when someone requests the next tuple. The writer's behavior will be symmetric - it can buffer the tuples in memory until it has a full page and then flush (write) the page.

It is a good idea to implement TupleReaders and TupleWriters for both the new binary format and the Project 2 human-readable format. That also allows you to write some helpful utilities for debugging, such as a converter between the two formats.

Be aware of a couple of quirks of Java NIO we discovered last semester:

First, some students reported strange behavior when using the relative get/put methods for `ByteBuffer`, i.e. `getInt()` and `putInt(int value)`. If you encounter issues, try the absolute methods `getInt(int index)` and `putInt(int index, int value)`, which explicitly specify the desired location in the buffer.

Second, be aware that `Buffer.clear()` does not do what you might think it does. The official documentation at `http://docs.oracle.com/javase/8/docs/api/java/nio/Buffer.html#clear--`, tells us the surprising fact that: "This method does not actually erase the data in the buffer, but it is named as if it did because it will most often be used in situations in which that might as well be the case."

The above affects you because it could mess up your padding with zeroes at the end of each page. Specifically, if you reuse the same `ByteBuffer` for each page when writing, even if you `clear()` it between pages, this does not refill the buffer with zeroes. If your second-to-last page is full and your last page is not full, and you only do a `clear()` between pages, the "empty" final portion of the last page you write will actually contain some "garbage" values that are left over from the second-to-last page. The moral of the story is that you have to do something more than `clear()` to ensure correct padding with zeroes.

Once you have sorted out your I/O, this is a great time to write a random data generator.

You may also want to write a sorting utility that takes in a file (in either format), sorts the tuples in memory using `Collections.sort` and writes out a sorted file. This will be very handy once you start implementing different join algorithms - when run on the same data, they may output tuples in different ordrers. If you have a sorting utility, you can sort the outputs afterwards and compare the files to make sure your SMJ is actually outputting the same results as your BNLJ.

Finally, this is as good a time as any to introduce timing functionality into your code. When your top-level class has a query plan and is ready to call `dump()` to write the results, call `System.currentTimeMillis()` before and after the `dump()` to get a measure of the elapsed time. You will need this functionality for the performance benchmarking (Section 3.6) and it is handy to implement it now. Generate yourself some relations with, say, 5000 or 10000 tuples and see how long it takes to run your queries, as a ballpark figure.

3.2 Refactor to use both logical and physical query plans

Your Project 2 code generated the query plan directly from the `Statement` objects that JSqlParser produced. You will now refactor this into a two-stage process: the generation of a *logical query plan* from the `Statement`, and the generation of a *physical query plan* from the logical query plan. The logical query plan will just be a relational algebra tree; the physical query plan will contain concrete implementations for each operator, such as SMJ or BNLJ.

Thus, your overall workflow in your top-level class will be:

- read a query from a file and parse it using JSqlParser
- convert the `Statement` you obtained into a logical query plan
- convert the logical query plan into a physical query plan
- evaluate, i.e., call `dump()` on the physical query plan, including timing the evaluation

This refactoring is necessary to separate two different concerns:

- building a relational algebra tree that represents the query at a mathematical level, and
- translating the relational algebra tree into some code that will actually run and spit out tuples.

This separation makes it easier to add functionality that pertains to only one of these - for example, in Project 5 you will be optimizing the logical query plan. If you are pushing selections/projections past joins, a lot of that can and should be done before you choose a concrete join implementation.

This separation also allows you flexibility in modifying or enhancing the logical plan as you build the physical plan. Notably, if you want to use SMJ, you can insert sort operators on both inputs to the join while constructing the physical query plan. That way the join operator itself only has to perform the merge. Of course if you are *not* using SMJ, it doesn't make sense to insert these extra sort operators, so you want to defer the decision until you know which join implementation you are using.

To refactor, start by creating separate packages with classes for logical and physical operators. Your old Project 2 operators will basically become your physical operators, and you will add a new package with logical operators. A logical operator does not need to store a lot of info; depending on the type of operator, it may need to know things such as the join condition, selection condition, sort order, etc. Basically, think about writing your query in relational algebra by hand as you have done in 4320 homeworks; if the information appears in your relational algebra translation, it probably needs to be in the logical operators. On the other hand, the logical operators do not need `getNextTuple()` or `reset()` methods since they will never actually "run". That logic definitely belongs in physical operators.

Once you have both physical and logical operators, you need code to translate a logical plan to a physical plan. Fortunately, you have extensive experience with the visitor pattern, so this should not be difficult conceptually. Write a visitor class that recursively walks your logical plan and builds up a corresponding physical plan. In the remainder of these instructions we will call this visitor the `PhysicalPlanBuilder`, but you may name it what you like. For now the `PhysicalPlanBuilder` code will not be very "clever", but this will soon change. For example, once you have a BNLJ implementation available, you can set your `PhysicalPlanBuilder` to convert the logical join operator into either a tuple-nested loop join or BLNJ physical operator. Once you have a SMJ implementation, your `PhysicalPlanBuilder` will actually insert physical sort operators as well as a physical SMJ operator.

3.3 Implement BNLJ

After substantial refactoring, you are now ready to implement a new join algorithm: block-nested loop join.

3.3.1 Implement the BNLJ physical operator

You need to implement a new physical operator to compute joins using the BNLJ algorithm. This is described at a high level in your textbook, pp. 455-456. The basic idea is to read the outer relation one block at a time into a buffer, and execute the following logic:

```
procedure BNLJ(outer $R$, inner $S$)
    for each block $B$ of R do
        for each tuple $s$ in $S$ do
            for each tuple $r$ in $B$ do
                if $r$ and $s$ satisfy join condition then
                    add new tuple formed from $r$ and $s$ to result
```

Of course this is the logic to compute the entire result, whereas in the iterator model you need to output one tuple at a time. You will therefore have to restructure the above triple-nested-loop structure for the `getNextTuple()` method, and your operator will need to keep some state between invocations so it knows where to resume. You already had to something like this for the tuple nested loop join, but it will be a little more involved here due to three nested loops.

You will notice the description in your textbook talks about building a hashtable for each block of the outer. This is a good refinement for equijoins, but requires careful handling if the join condition is *not* equality. Yes, most real-world joins are equijoins, but your BNLJ algorithm should support all join conditions as specified in the Project 2 description. Thus if you choose to implement a hashtable, you need to do something to handle non-equality join conditions as well. Alternately you may omit the hashtable and iterate over the entire buffer for each tuple of the inner relation as suggested by the above pseudocode.

As regards implementing the buffer, we are going to "cheat" a bit. Our buffer pages will not directly correspond to the file pages from Section 2.2. This is because we are not building a full-fledged buffer manager, so there is no reason to have uniform treatment for the buffers in your `TupleReader`/`TupleWriter` and for the buffers in the BNLJ. The buffer in your BNLJ can just be a data structure that contains `Tuple`s.

However, the size of this buffer should be configurable and this should be something that can be passed in into the operator's constructor. To maintain some residual consistency across the implementation, we require the ability to set the size of the buffer in pages, where a page is 4096 bytes. Your BNLJ constructor can calculate how many tuples of the outer relation will fit in a page, assuming each tuple has size $4*$ the number of attributes it contains. (Note the discrepancy with respect to the page format from Section 2.2 where eight bytes are used up for metadata.) The BNLJ operator can then internally use a `Tuple` buffer with the appropriate maximum size in tuples.

Note that your textbook talks about reserving two of the pages for input and output of the inner relation; we do not need to worry about those here because these two buffers are maintained in the `TupleReader` and `TupleWriter`. Thus, the buffer size parameter for your BNLJ should just be the number of "pages" to devote to each block of the outer relation.

To read a block of the outer relation, the BNLJ operator should repeatedly call `getNextTuple()` on the outer child until the buffer is filled. You may wonder if this means that we are regressing to tuple-at-a-time

I/O. However, note that if the outer child is a base table and you are using a page-at-a-time `TupleReader`, you are actually doing page-at-a-time file I/O even though it "looks like" the BNLJ is pulling the tuples one at a time. Of course if the outer child is not a base table, but another operator, the BNLJ needs to pull tuples one at a time anyway.

3.3.2 Integrate your BNLJ operator with the rest of your code

Your `PhysicalPlanBuilder` should set the number of pages to be used for the BNLJ buffer when it creates a BNLJ physical operator. This means you need a way to specify:

- which physical implementation should be used for the logical join operator (TNLJ or BNLJ)
- if the physical implementation desired is BNLJ, how many buffer pages to use.

You should specify the join algorithm desired and the number of buffer pages in a *configuration file*. When your `PhysicalPlanBuilder` is constructed, it should read this configuration file and set appropriate fields internally so it can do the desired thing during plan construction. The format for the configuration file is described in Section 2.3.

Note that the above implies you should hang on to your Project 2 TNLJ implementation. You will be comparing your other join implementations against it, both to ascertain correctness and to benchmark running times as explained in Section 3.6.

In fact, this is a good time to run some queries and compare the execution times with BNLJ versus TNLJ, for various buffer sizes in the BNLJ. The results of the queries should obviously be the same, although they may be ordered differently.

3.4 Implement external sort

Next, implement a physical external sort operator to replace your in-memory sort from Project 2. This is described in Section 13.3 of your textbook.

The sort operator needs to create intermediate files as it runs, to keep the partial sorted runs which are pending a merge in the next pass. Thus you need a dedicated directory to keep temporary "scratch" files. For grading purposes, we require the ability to set the location of this directory as a command-line argument (See Section 2.1).

A couple of tips about the temp directory management:

- If your plan contains more than one external sort operator, say you have one on each input to a SMJ, make sure these operators do not get confused and read each other's "scratch" files. You may choose to have each operator create its own subdirectory within the temp directory. Or you may resolve this in some other way with clever file naming, but you need a solution to this issue.
- Make sure you have some way to clean up the temp directory between queries, as we will be grading by using the same temp directory for all the test queries.

Once the sort has been performed and the operator knows the location of the temporary file with the full sorted relation, `getNextTuple()` calls can be handled by simply opening a `TupleReader` on that file and returning tuples from the `TupleReader`. In case the sort operator is `reset()`, obviously you should not delete the sorted file, but instead reset the `TupleReader`.

For the “scratch” files, you may use either file format - the binary format from this Project or the human-readable format from Project 2. We recommend implementing it so you can use either one, with the choice being made by setting a flag in the code. The human-readable format is useful for debugging, but you will probably find that using the binary format makes your sort run faster.

As with BNLJ, the sort operator should allow you to set a buffer size in pages. Also as with BLNJ, we will make some simplifications which mean your buffer will function a little differently from what is described in the textbook but (hopefully) make your life easier.

Suppose your buffer is B pages. This affects your code in two ways.

First, it dictates the number of tuples you can read and sort in-memory in the initial (0-th) pass of the sort. You should handle this as in BNLJ - compute how many tuples you can hold in memory based on the fact that a page is 4096 bytes and based on how wide your tuples are (i.e. how many attributes they have). When you are done with sorting these tuples, write them to a temporary file using a `TupleWriter` without worrying about how many pages the resulting file will have. As explained in the BNLJ case, that file may theoretically contain more pages due to the extra bytes needed in the binary format for metadata, but that’s fine for our purposes.

Second, it determines the fan-in of your merge in the merge passes (pass 1 and any following). You will be doing a $B-1$-way merge, unless of course you are in the edge case where you have fewer than $B-1$ runs remaining to merge.

You may assume we will always set $B >= 3$, so you are doing at least a two-way merge. You should allow B to be passed into the constructor of the sort operator, and it should be set by the `PhysicalPlanBuilder` just like for BNLJ. B will be specified in the `PhysicalPlanBuilder` configuration file - see Section 2.3.

A word on the merge passes: because we are using `TupleReaders` that hide the page-wise file I/O from higher layers of your code, you do not need in your sort operator to read and buffer a page’s worth of tuples from each of the $B-1$ runs being merged. In your sort operator, it is sufficient to buffer *one* tuple from each of the $B-1$ runs. The underlying `TupleReader` is already buffering a page’s worth of tuples for you, so this is not “cheating” in any way. It just means that your $B-1$ buffers are spread out across $B-1$ `TupleReaders`, rather than being centrally managed by the sort operator itself.

3.5 Implement SMJ

At this point you have all the ingredients in place to support sort-merge join. Unlike TNLJ and BNLJ, SMJ is not universally applicable. In particular, it is really only good for equijoins. It is theoretically possible to adapt the algorithm on p.460 of your textbook to other conditions such as $<$ or $<=$, but realistically at that point you’d be better off using BNLJ rather than paying the overhead of doing a sort. Inequality and other “exotic” joins are likely to contain many tuples and be pretty close to the cross product anyway. You may assume that all queries we use to test your SMJ will contain equijoins only.

Furthermore, you may assume that if we are joining more than two tables, the join tree will contain at least one equality condition in every join node, assuming you construct it in the manner described in Project 2 - i.e. that the join order follows the order in the `FROM` clause and that you evaluate join conditions as early as possible.

For example, the query `SELECT * FROM R, S, T WHERE R.A = S.B AND R.C = T.K` is a valid query, because it should be translated to a join between R and S with condition `R.A = S.B`, and the tuples from that join will be joined with T with condition `R.C = T.K`. Thus every join node has a nontrivial join condition and gives you something to sort the left and right inputs on. On the other hand the query `SELECT * FROM R, S, T WHERE R.A = T.B AND S.C = T.K` is not a query we will test with, because if you follow the join order in the `FROM` clause the first join is a cross product of `R` and `S`.

Before you actually write the sort-merge join physical operator, you need to do a bit of setting up. In particular, when you build the physical plan, you need to insert two sort operators so that the inputs into the SMJ operator will be already sorted on the required fields. That means your SMJ operator only needs to implement the merge logic, and allows you to reuse the sort logic you just wrote in the sort operator.

You will quickly realize you need to determine how to sort each of the inputs to the join. This will require processing the join condition (using another `ExpressionVisitor`, most likely), to extract the attributes for sorting the left child and for sorting the right child. It will also be really helpful here to push selections, if you haven't done so already. If you have a query like `SELECT * FROM R, S, T WHERE R.A = S.B AND R.C = T.K AND S.M = 5`, it will really make your life much easier to get rid of the `S.M = 5` early and only worry about the `R.A = S.B AND R.C = T.K` portion when you are trying to set up the sort merge join.

Once you have determined what to sort by, you are ready to implement the SMJ operator itself. This needs to know what the left and right children are sorted by. It basically implements the merge logic from p.460.

One important point is that you will have to write a custom function to compare the tuples from the left and right child, to see whether they are equal on the join attributes, or whether the left tuple is smaller or larger than the right one. This complexity is somewhat masked in the pseudocode on p.460 because they give the very simplest case, where there is only one attribute from each relation in the join condition, so checking for a condition like $T_{r_i} < G_{s_j}$ is trivial. In general, there may be more than one attribute in the join condition - `SELECT * FROM R, S WHERE R.A = S.B AND R.C = S.D` is a valid query. You will need suitable notions of "less than" or "greater than" for such queries.

Also note that you will need a way to reset the right (inner) relation to the start of the current partition (see the line labeled "reset S partition scan" on p.460). It is NOT ok to do this by buffering the tuples in the S partition in your SMJ operator, as there may be very many of them - think of the degenerate case where all the tuples from R match all the tuples from S. Thus, buffering the whole partition in memory would require unbounded state. So, you need a way to reset the inner operator to a particular tuple or tuple index.

The recommended way to implement this is by adding a `reset(int index)` method to your abstract `PhysicalOperator` class. You don't need to implement it for every physical operator, since you will only be calling it on a sort operator. Thus you just need to implement it in your physical sort operators (in-memory sort and external sort).

For the external sort, it is probably easiest to push the implementation into a `reset(int index)` method on your `TupleReader`, since the external sort operator produces output tuples by reading them from its sorted file. If your `TupleReader` is reading the binary file format, you will want to implement `reset(int index)` by computing the number of the page that tuple will be on and fetching that page, rather than by resetting the `TupleReader` and calling `getNextTuple` an `index` number of times. Note that you can reset the position of your `FileChannel` to the start of the desired page within the file using the `position(long newPosition)` method of `FileChannel`.

3.6 Performance benchmarking

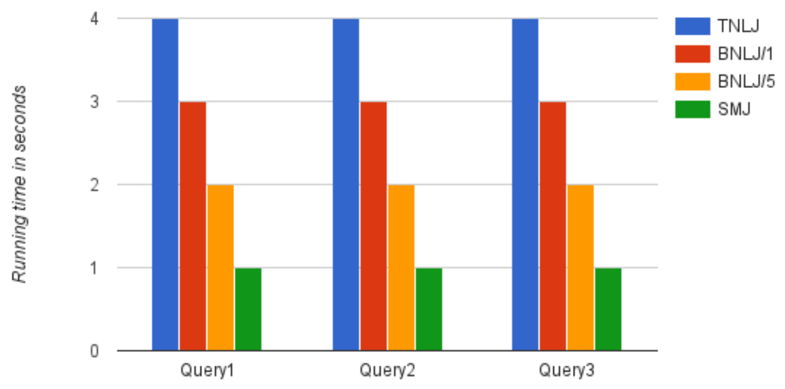
Now that you have implemented two new join algorithms, you should compare how they measure up against each other and against TNLJ. Generate some data (at least 5000 tuples per relation) and choose at least three queries which are valid for all three algorithms (i.e. they meet the restrictions we specified in Section 3.5). At least one of your queries must include more than two relations.

For each of the three queries, run it using TNLJ, BNLJ with a one-page buffer, BNLJ with a five-page buffer, and SMJ with external sort (buffer for sort set to a value of your choice). Generate bar graphs to show the comparison of running times.

You will not be graded on how fast your implementation is or how much improvement BNLJ/SMJ give you over TNLJ. We just want you to run these experiments to gain an understanding of how your own implementation does.

For this part, you must create a pdf file called `Experiments.pdf`. This file must contain:

- the three queries you ran
- a description of the data you used - i.e the schema of each relation, how many tuples per relation, and how these tuples were generated (e.g. "each attribute value was chosen uniformly at random in the range 0 to 1000")
- the number of pages you used as buffer size for the sort in the SMJ case
- a bar graph comparing the running times for the three queries and the four different algorithms. An example graph with some fictitious running times is given below.



If you have been unable to implement any of the above algorithms, still provide results for the algorithms that you have, substituting as appropriate (e.g. if you have SMJ but not external sort, show the results for SMJ with in-memory sort).

3.7 Revisit `DISTINCT`

Check that your implementation of `DISTINCT` does not use unbounded state. If you implemented `DISTINCT` in Project 2 using a sorting approach as suggested in the instructions, this is already true for your implementation. If you have some other implementation of `DISTINCT`, you need to revisit and potentially refactor it so doesn't keep unbounded state. The simplest way to go is still sort-based; if you don't want to do it that way, you are free to use another implementation as long as it doesn't keep unbounded state.

4 Requirements

As usual, you are free to make your own architectural decisions except that you must satisfy the following requirements. Disregarding any of these will result in arbitrarily severe point deductions.

- you must use Java NIO to read/write the binary-format data files (you can use any API you like for the human-readable files).
- you must refactor your code to use logical and physical operators and a `PhysicalPlanBuilder` (you don't need to call it that name, but it must use the visitor pattern).
- no operator (except your old Project 2 in-memory sort) may keep unbounded state. In particular:
  - the SMJ implementation must not buffer tuples from the right relation's partition in memory; you need proper `reset` logic as explained in Section 3.5.
  - your handling of `DISTINCT` must not keep unbounded state, as explained in Section 3.7.

  It is understood that if the configuration file requires you to use in-memory sort, there will be unbounded state and that is ok.
- you must have good-faith implementations of BNLJ, external sort, and SMJ, or tell us clearly in the README if you have not been able to implement some of those operators. Also, your configuration file for the `PhysicalPlanBuiler` must function as described in Section 2.3. For more on this, see below.

If you cannot implement one or more of BNLJ, external sort, or SMJ, that is fine, although obviously you won't get full points for the project. In that case, you must clearly tell us in the README what you have not been able to implement. Note that if you cannot implement external sort, you can still try to implement SMJ using the in-memory sort algorithm.

What you may *not* do is try to "fool us" into believing that you have implemented one of these algorithms when you haven't. For example, suppose you take your TNLJ implementation, rename it `BlockNestedLoopJoin`, obfuscate the code so the logic is hard to follow, and submit it as a BNLJ implementation in the hope we won't notice. Or suppose you submit a project that ignores the config file for the `PhysicalPlanBuiler` and only creates plans with a TNLJ - but you don't tell us this, in the hope that we won't realize your buggy BNLJ/SMJ implementations are never called.

You can philosophize to what extent that counts as cheating, but it certainly counts as lying and it is not OK. If you do this and if it is clear this was deliberate (not a good-faith bug in the parsing of your config file, say) **you will get a very severe deduction, potentially zero points for the entire project**. If your grader has any doubt that you have submitted a good-faith implementation of BNLJ, SMJ and/or external sort, and that your config file sets the options correctly for the `PhysicalPlanBuilder`, they may ask you to come to office hours and explain your code's logic.

5 Checkpoint

For the project checkpoint, which is due on March 21st, you need to complete the work specified in Sections 3.1 and 3.2. That is, you need to implement reading/writing the binary file format, and you need to refactor your code to have a `PhysicalPlanBuilder`. Your `PhysicalPlanBuilder` is not yet required to parse the config file as described in Section 3.3.2. This means you are allowed to ignore the config file for now and hardcode TNLJ and in-memory sort as your physical implementations of join and sort respectively. However, you must be able to run all the queries we have provided in the samples directory.

We will test your submission on the tests in the samples directory. There are 15 queries for 15 points; thus, you can actually know your score without waiting for the graders, by running the sample queries and comparing to the sample answers. Remember that queries can return answer tuples in any order and still be correct, unless the query contains an `ORDER BY` in which case a specific ordering is required like in Project 2.

In addition, the graders will check that you have refactored your code as specified in Section 3.2; this is worth 10 points.

Thus the whole checkpoint is out of 25 points and counts for 6% of your final grade.

For the checkpoint, you need to submit via CMS a .zip archive containing:

- a runnable `cs4321_p3.jar` file of your project
- your Eclipse project folder
- a README file that explains:
  - which is the top-level class of your code (the interpreter/harness that reads the input and produces output)
  - where your grader can see the logical operators, the physical operators, and the `PhysicalPlanBuilder`
- an acknowledgments file if you consulted any external sources, as required under the academic integrity policy

6 Main project grading

The main project submission (after the checkpoint) is graded out of 65 points and counts for 18% of your final grade. These points are allocated as follows:

6.1 Code style and comments (10 points)

As in previous projects, you must provide comments for every method you implement. At minimum, the comment must include one sentence about the purpose/logic of the method, and `@params`/`@return` annotations for every argument/return value respectively. In addition, every class must have a comment describing the class and the logic of any algorithm used in the class. As in previous projects, if you follow the above rules and write reasonably clean code that follows our overall architecture, you are likely to get the full 10 points for code style.

6.2 Benchmarking (10 points)

You will not be graded on the running times of your code; you will be graded on whether your `Experiments.pdf` file contains all the info we ask for in Section 3.6. So these should be an easy 10 points.

6.3 Automated tests (45 points)

We will run your code on our own queries and our own data to test your BNLJ, SMJ and external sort operators, with approximately 15 points for each of these. The testing will be designed to test the operators as independently as possible, but it will assume that your implementations of TNLJ and in-memory sort are correct. We may test your implementation on relations up to 10000 (ten thousand) tuples in size.

Remember that queries can return answer tuples in any order and still be correct, unless the query contains an `ORDER BY` in which case a specific ordering is required like in Project 2.

7 Main project submission instructions

Carefully reread everything in Section 2 to be sure your code will not break automation during testing. In particular be sure that you have some way of clearing the temp directory for sort between queries.

Create a README text file containing the following information. **Submissions without a README will receive zero points.**

- a line stating which is the top-level class of your code (the interpreter/harness that reads the input and produces output).
- a description of your logic for handling the partition reset during SMJ and for handling `DISTINCT`; we are looking for a clear explanation why your implementations of SMJ and `DISTINCT` don't keep unbounded state.
- any other information you want the grader to know, such as known bugs.

Submit via CMS a .zip archive containing:

- a runnable `cs4321_p3.jar` file of your project
- your Eclipse project folder
- your README file
- the `Experiments.pdf` file showing the results of your benchmarking, as discussed in Section 3.6
- an acknowledgments file if you consulted any external sources, as required under the academic integrity policy

8 Credits/acknowledgments

O. Kennedy, L. Kot, D. Darde, S. Jain., W. White