CS 4321/5321 Project 2

Spring 2017

Due October 3rd, 2017 at 11:59 pm

This project is out of 90 points and counts for 22% of your grade.

1 Goals and important points

This is the first project where you actually develop some database functionality. The goals are:

- to teach you how to translate from SQL queries to a relational algebra query plan
- to familiarize you with the iterator model for relational operator evaluation, as well as naïve implementations for the most common operators (selection, projection, join, sort)
- to provide you with a codebase that you will build on in future projects.

You will be starting from an empty directory for this project, i.e. no skeleton code is provided. We are, however, providing a .jar with a parser so you do not have to write your own (unless you want to).

The reference implementation of this project is about 1100 lines of code, not including comments. Whether or not you consider this a lot, it is not a project that should be left to the last minute.

No solution code will be given out. The code you implement for Project 2 will be the code you reuse/refactor for Projects 3, 4 and 5. Subsequent projects will assume you have implemented all the functionality required for Project 2.

2 Overview

In this project, you will implement a simple interpreter for SQL statements. That is, you will build a program that takes in a database (a set of files with data) and a file containing several SQL queries. It will process and evaluate each SQL query on the database.

2.1 Supported language features

Your interpreter will not support all of SQL, but it will handle a lot of relatively complex queries. Here we give information about the queries you must support.

Your interpreter will process `SELECT-FROM-WHERE` queries, which may optionally also have a `DISTINCT`, an `ORDER BY`, or both. You do not need to support nested subqueries, set operators (`UNION` etc.), `GROUP BY`, aggregates like `COUNT`, or any other features. In addition, we make a few simplifying assumptions as below. When we say a query is *valid*, we mean it is a permitted input to your interpreter which you should be able to handle. When we talk about a *base table*, we mean a real table that exists in the database.

- You may assume all valid queries follow correct SQL syntax and that they only refer to tables that exist in the database. Also, when a query refers to a table column such as `Sailors.name`, you may assume the column name is valid for that table.
- You may assume there will be at least one table in the `FROM` clause.
- Valid queries may use aliases (range variables) such as `Sailors S` or they may just use the names of base tables. If a query does not use aliases, all column references are fully qualified by the base table name. If a query does use aliases, all tables use aliases and all column references are qualified by alias. Here are two examples of valid queries, the first one without aliases and the second with aliases:
  - `SELECT Sailors.name, Reservations.date FROM Sailors, Reservations WHERE Sailors.id = Reservations.sid;`
  - `SELECT S.name, R.date FROM Sailors S, Reservations R WHERE S.id = R.sid;`

  You may assume that any string used as an alias will *not* also be the name of a base table.
- Self-joins, i.e. joining a table with itself, are valid and must be supported (and require the use of aliases)
- The `WHERE` clause, if present, is a conjunction (i.e. an `AND`) of expressions of the form A *op* B, where *op* is one of $=, !=, <, >, <=, >=$ and A and B are either integers or column references. Thus `Sailors.id = Reservations.sid`, `Sailors.id < 3` and `42 = 42` are all valid expressions for the `WHERE` cause, while for example `Sailors.id < Boats.id - 1` is not a valid expression even though it would be ok in "real SQL".
- The `SELECT` clause will either specify a subset of columns or have the form `SELECT *`. For `SELECT *`, order the columns in your answer following the `FROM` clause. Thus for `SELECT * FROM R, S` each answer row has all the columns of `R` followed by all the columns of `S`.
- There may be an `ORDER BY` clause which specifies a subset of columns for ordering. You may assume that we only want to sort in ascending order. You may also assume that the attributes mentioned in the `ORDER BY` are a subset of those retained by the `SELECT`. This allows you to do the sorting last, after projection. Note that this does not mean that every attribute in `ORDER BY` must be mentioned in the `SELECT` - a query like `SELECT * FROM Sailors S ORDER BY S.name` is valid.
- There may be a `DISTINCT` right after the `SELECT`, and it should be processed appropriately. Yes, `SELECT DISTINCT * FROM ...` is valid.

We require a particular behavior for `ORDER BY` for consistency in grading. You must order the tuples first by the attributes specified in the `ORDER BY`, and then by any remaining attributes in the order in which they appear in the tuples that are input to the sort. Thus suppose a relation `R` has attributes `A, B, C, D`. If processing the query `SELECT * FROM R ORDER BY R.C, R.A` we expect you to sort by `C, A, B, D`, in that order. (If two tuples agree on C, break ties based on A, if they also agree on A, break ties based on B, and so on).

2.2 Data and output formats

We have provided you some sample data and some sample queries. Take a look at the `samples` directory. There is an `input` and an `expected_output` subdirectory.

In the `input` directory you will find a `queries.sql` file containing some example queries. You will also find a `db` subdirectory. This contains a `schema.txt` file specifying the schema for your database as well as a `data` subdirectory, where the data itself is stored.

The `schema.txt` file contains one line per table in the database. Every line contains several strings separated by spaces. The first string on each line is the table name and all the remaining ones are attribute (column) names, in the order in which they appear in the table.

The `data` subdirectory contains one file per database table, and the name of the file is the same as the name of the database table. Every file contains zero or more tuples; a tuple is a line in the file with fields (attributes) separated by commas. All attributes are integers, as they will be for the remainder of the course. Using integer attributes simplifies your job and allows you to focus on implementing "interesting" functionality rather than boilerplate code to handle different data types. Also, you do not have to handle null values in this or any other 4321 project (but you do need to handle empty relations).

In the `expected_output` directory, you will find the expected output files for the queries we provided. File `query1`, for example, contains the expected output for the first query in the `queries.sql` file. The format for the output is the same as the format for the data.

Your SQL interpreter should read from the `db` directory and from the `queries.sql` file, and write output to a suitable `output` directory. We expect you to follow the same scheme for naming output files as we do, i.e., put the answer to each query in its own file and number it with the number of the query (starting at 1).

When we run your code, we will use the command line. Thus you need to submit a runnable .jar of your code. To export a runnable .jar in Eclipse, use File → Export → Runnable Jar File using the run configuration for your top level class. Make sure to test your runnable .jar to be sure it works.

Your runnable .jar should run on the command line as follows:

```
java -jar cs4321_p2.jar inputdir outputdir
```

In our testing, `inputdir` will have the same structure as described above, but it will contain our own test queries and data, and `outputdir` will be an empty directory. This means your top-level Java class has to accept `inputdir` and `outputdir` as command-line arguments and handle them appropriately.

Both `inputdir` and `outputdir` will be given as absolute paths to the appropriate directory with no final / symbol. If you are concerned about Windows vs *nix file separator issues, consider using `File.separator` and/or making a note in your README specifying where file separators appear in your code, so that the grader can modify those portions of your code if needed.

After we run your code, the grading harness will compare your `outputdir` directory to a directory containing expected outputs for each query and check if the files with the same names match. For queries without an `ORDER BY`, it is ok if your answer file has the answer tuples in a different order to ours; for queries with an `ORDER BY`, your ordering must match our ordering.

As you can imagine, it is very important for you to respect the expected input and output format. **Also, make sure that if your interpreter throws an error on some query, it will still process the subsequent queries in the file and place their answers in appropriately-named files.**

2.3 Operators and the iterator model

A key abstraction in this assignment will be the iterator model for relational operators. You will implement several operators:

- the bag relational algebra *select*, *project* and (tuple nested loop) *join*.
- *sort* and *duplicate elimination* operators, which are not part of the basic relational algebra you saw in 4320 but must be added to support `ORDER BY` and `DISTINCT`.
- a *scan* operator which is the leaf operator for any query plan. This is really a physical operator rather than something you would add to the relational algebra, but for now we will put it in the same category as the above.

The standard way to implement all relational operators is to use an *iterator* API. You should create an abstract class `Operator`, and all your operators will extend that. Certain operators may have one or two child operators. A scan operator has no children, a join has two children, and the remaining operators have one child. Your end goal is to build a query plan that is a tree of operators.

Every operator must implement the methods `getNextTuple()` and `reset()` (put these in your abstract `Operator` class). The idea is that once you create a relational operator, you can call `getNextTuple()` repeatedly to get the next tuple of the operator's output. This is sometimes called "pulling tuples" from the operator. If the operator still has some available output, it will return the next tuple, otherwise it should return null.

The `reset()` method tells the operator to reset its state and start returning its output again from the beginning; that is, after calling `reset()` on an operator, a subsequent call to `getNextTuple()` will return the *first* tuple in that operator's output, even though the tuple may have been returned before. This functionality is useful if you need to process an operator's output multiple times, e.g. for scanning the inner relation multiple times during a join.

Implementing each of the above operators will involve implementing both `getNextTuple()` and `reset()`. Remember that if your operator has a child operator, the `getNextTuple()` of your operator can - and probably will - call `getNextTuple()` on the child operator and do something useful with the output it receives from the child.

A big advantage of the iterator model, and one of the reasons it is popular, is that it supports *pipelined* evaluation of multi-operator plans, i.e., evaluation without materializing (writing to disk) intermediate results.

The bulk of this project involves implementing each of the above six operators, as well as writing code to translate an SQL query - i.e. a line of text - to a query plan - i.e. a suitable tree of operators. Once you have the query plan, you can actually compute the answer to the query by repeatedly calling `getNextTuple()` on the root operator and putting the tuples somewhere as they come out.

We suggest you add a `dump()` method to your abstract `Operator` class. This method repeatedly calls `getNextTuple()` until the next tuple is null (no more output) and writes each tuple to a suitable `PrintStream`. That way you can `dump()` the results of any operator - including the root of your query plan - to your favorite `PrintStream`, whether it leads to a file or whether it is `System.out`.

3 Implementation instructions

We recommend that you implement and test one feature at a time. Our instructions below are given in suggested implementation order.

We also recommend you set up a test infrastructure/harness early on. You should do two kinds of testing - unit tests for individual components and end-to-end tests where you run your interpreter on queries and look at the output files produced to see if they match a set of expected output files. As you add more features, rerun all your tests to check that you didn't introduce bugs that affect earlier functionality.

After you implement and test each feature, make a copy of your code and save it so if you mess up later you still have a version that works (and that you can submit for partial credit if all else fails!).

3.1 Set up JSqlParser

For this project, you no longer need to write your own parser. We recommend using JSqlParser, which takes care of parsing your SQL and creating a Java object. There are several versions of JSqlParser available online, although there are issues with many of them (some are poorly documented, others are very old, etc). We have provided a `.jar` and Javadocs for a fork of JSqlParser which we recommend. You are not required to use this version, or indeed to use JSqlParser at all, but you need to correctly parse all valid queries as defined in Section 2.1.

The documentation is a little bare-bones but it will be sufficient for our purposes. You should expect to play around with JSqlParser on your own and read the documentation to understand the structure of the objects that it outputs.

To get you started, we have provided a sample `ParserExample.java` file that uses JSqlParser to read SQL queries from a file and print them out. This file illustrates the use of JSqlParser and some methods to access fields of the `Statement` object, such as `getSelectBody()` if the `Statement` is a `Select`.

You may assume all the `Statements` we will work with are `Selects`, and have a `PlainSelect` as the `selectBody`. Take a look at the `PlainSelect` Javadocs; the first table in the `FROM` clause will be in the `fromItem`, the remaining ones in `joins`. Also of relevance to you is the `where` field for the `WHERE` clause, and eventually the `distinct` and `orderByElements` fields. Write some SQL queries and write code to access and print out all these objects/fields for your queries to get an idea of "what goes where".

The `where` field of a `PlainSelect` contains an `Expression`; take a look at the docs for that. For this project, you only need to worry about `AndExpression, Column, LongValue, EqualsTo, NotEqualsTo, GreaterThan, GreaterThanEquals, MinorThan` and `MinorThanEquals`. These capture the recursive structure of an expression, which should bring back memories of Project 1. The last six of the expression types mentioned are comparison expressions, the `AndExpression` is a conjunction of two other `Expressions`, the `LongValue` is a numeric literal, and `Column` is a column reference (such as the `S.id` in `S.id < 5`). Every `Column` object has a column name, as well as an embedded `Table` object. Every `Table` object has a name and an alias (if aliases are used).

JSqlParser also provides a number of `Visitor` interfaces, which you may or may not choose to use. In particular, `ExpressionVisitor` is highly recommended to use one you get to the selection operator.

The above should be enough to get you started, but you should expect to do further explorations on your own as you implement more and more SQL features.

3.2 Implement scan

Your first goal is to support queries that are full table scans, e.g. `SELECT * FROM Sailors` (for now assume the queries do not use aliases). To achieve this, you will need to implement your first operator - the scan operator.

Implement a `ScanOperator` that extends your `Operator` abstract class. Every `ScanOperator` knows which base table it is scanning. Upon initialization, it opens a file scan on the appropriate data file; when `getNextTuple()` is called, it reads the next line from the file and returns the next tuple. You probably want to have a `Tuple` class to handle the tuples as objects.

The `ScanOperator` needs to know where to find the data file for its table. It is recommended to handle this by implementing a *database catalog* in a separate class. The catalog can keep track of information such as where a file for a given table is located, what the schema of different tables is, and so on. Because the catalog is a global entity that various components of your system may want to access, you should consider using the singleton pattern for the catalog (see Wikpedia and many other online references on this).

Once you have written your `ScanOperator`, test it thoroughly to be sure `getNextTuple()` and `reset()` both work as expected. Then, hook up your `ScanOperator` to your interpreter. Assuming that all your queries are of the from `SELECT * FROM MyTable`, write code that grabs `MyTable` from the `fromItem` and constructs a `ScanOperator` from it.

In summary the top-level structure of your code at this point should be:

- while more queries remain, parse the next query in the queries file
- construct a `ScanOperator` for the table in the `fromItem`
- call `dump()` on your `ScanOperator` to send the results somewhere helpful, like a file or your console.

3.3 Implement selection

The next order of business is single-table selection, still with fully specified table names (no aliases). That is, you are aiming to support queries like `SELECT * FROM Boats WHERE Boats.id = 4`.

This means you need to implement a second `Operator`, which is a `SelectOperator`. Your query plan will now have two operators - the `SelectOperator` as the root and the `ScanOperator` as its child. During evaluation, the `SelectOperator`'s `getNextTuple()` method will grab the next tuple from its child (i.e. from the scan), check if that tuple passes the selection condition, and if so output it. If the tuple doesn't pass the selection condition, the selection operator will continue pulling tuples from the scan until either it finds one that passes or it receives `null` (i.e. the scan runs out of output).

The tricky part will be implementing the logic to check if a tuple passes the selection condition. The selection condition is an `Expression` which you will find in the `WHERE` clause of your query. The `SelectOperator` needs to know that `Expression`.

You will need to write a class to test whether a `Expression` holds on a given tuple. For example, if you have a table `R` with fields `A, B` and `C`, you may encounter a tuple $1, 42, 4$ and an expression `R.A < R.C AND R.B = 42`, and you need to determine whether the expression is true or false on this tuple.

This is best achieved using a visitor on the `Expression`. You should have a class that implements the JSqlParser `ExpressionVisitor`. The class will take as input a tuple and recursively walk the expression to

evaluate it to `true` or `false` on that tuple. The expression may contain column references - in our example `R.A < R.C AND R.B = 42` refers to all three columns of `R`. The visitor class needs some way to resolve the references; i.e., if our input tuple is $1, 42, 4$, it needs a way to determine that `R.A` is 1, etc. So, your visitor class also needs to take in some *schema* information. It is up to you how you structure your schema information, but obviously it must allow mapping from column references like `R.A` to indexes into the tuple.

Once you have written your visitor class, unit-test it thoroughly. Start with simple expressions that have no column references, like `1 < 2 AND 3 = 17`. Then test it with column references until you are 100% sure it works. Once your expression evaluation logic is solid, you can plug it into the `getNextTuple()` method of your `SelectOperator`.

3.4 Implement projection

Your next task is to implement projection, i.e. you will be able to handle queries of the form `SELECT Sailors.id FROM Sailors WHERE Sailors.age = 20`. We still assume that the queries do not use aliases.

In comparison with selection, implementing projection is relatively easy. You need a third `Operator` that is a `ProjectOperator`. When `getNextTuple()` is called, it grabs the next tuple from its child. It extracts only the desired attributes, makes them into a new tuple and returns that. Note that the child could be either a `SelectOperator` or a `ScanOperator`, depending on whether your SQL query has a `WHERE` clause.

You get the projection columns from the `selectItems` field of your `PlainSelect`. `selectItems` is a list of `SelectItems`, where each one is either `AllColumns` (for a `SELECT *`) or a `SelectExpressionItem`. You may assume the `Expression` in a `SelectExpressionItem` will always be a `Column`. Once you grab these `Columns` you need to translate that information into something useful to the `ProjectOperator`.

Note that the attribute order in the `SELECT` does not have to match the attribute order in the table. The queries `SELECT R.A, R.B FROM R` and `SELECT R.B, R.A FROM R` are both valid, and they are different queries with different output.

By this point you should have code takes in a SQL query and produces a query plan containing:

- an optional projection operator, having as a child
- an optional selection operator, having as a child
- a non-optional scan operator.

Thus your query plan could have one, two or three operators. Make sure you are supporting all possibilities; try queries with/without a projection/selection. If the query is `SELECT *`, do not create a projection operator, and if the query has no `WHERE` clause, do not create a selection operator.

You are now producing relatively complex query plans; however, things are about to get much more exciting and messy as we add joins. This is a good time to pull out the logic for constructing the query plan into its own class, if you haven't done so already. Thus, you should have a top-level interpreter/harness class that reads the statements from the queries file. You should also have a second class that knows how to construct a query plan for each `Statement`, and returns the query plan back to the interpreter so the interpreter can `dump()` the results of the query plan somewhere.

3.5 Implement join

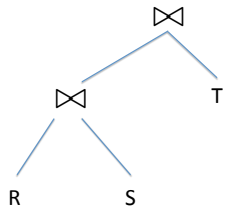
Next up, the star of the show: joins. Assume that there are still no table aliases, so you don't have to worry about self-joins for now.

You need a `JoinOperator` that has both a `left` and `right` child `Operator`. It also has an `Expression` which captures the join condition. This `Expression` could be a single comparison such as `R.A = S.B`, a conjunction (`AND`) of comparisons, or it could be null if the join is a cross product.

Implement the tuple nested loop join algorithm: the join should scan the left (outer) child once, and for each tuple in the outer child, it should scan the inner child completely (finally a use for the `reset()` method!). Once the operator has obtained a tuple from the outer and a tuple from the inner, it glues them together. If there is a non-null join condition, the tuple is only returned if it matches the join condition (so you will be reusing your expression visitor class from Section 3.3). If the join is a cross product, all pairs of tuples are returned.

Once you have implemented and unit-tested your `JoinOperator`, you need to figure out how to translate an SQL query to a plan that includes joins.

For this project, we require that you construct a left-deep join tree that follows the order in the `FROM` clause. That is, a query whose `FROM` clause is `FROM R,S,T` produces a plan with the structure shown below:

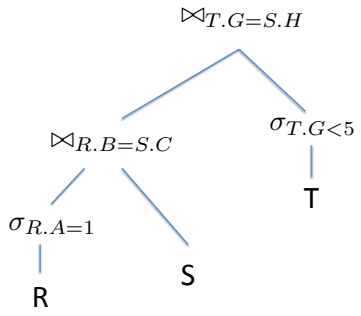


The tricky part will be processing the `WHERE` clause to extract join conditions. The `WHERE` clause may contain both selections on a single table as well as join conditions linking multiple tables. For example `WHERE R.A < R.B AND S.C = 1 AND R.D = S.G` contains a selection expression on `R`, a selection expression on `S`, and a join condition on both `R` and `S` together. Obviously it is most efficient to evaluate the selections as early as possible and to evaluate `R.D = S.G` during computation of the join, rather than computing a cross product and having a selection later.

While the focus of this homework is not optimization, you do not want to compute cross products unless you have to as this is grossly inefficient. Also, you will not be optimizing your query plan until Project 5, so you don't want to be carrying a cross-product based implementation through Projects 3 and 4. Therefore, we **require** that you have some strategy for extracting join conditions from the `WHERE` clause and evaluating them as part of the join. You do not need to be very clever about this, but you may not simply compute the cross products (unless of course the query actually asks for a cross product). You must explain your strategy in comments in your code and in the README that you submit with your code.

A suggested way to do this is as follows. Write another class that implements `ExpressionVisitor` and processes the `WHERE` clause. For every conjunct, the visitor determines which tables are referenced, and adds the conjunct to an appropriate `Expression`. If there are k tables being joined, there are $2k - 1$ running `Expressions`: $k - 1$ join conditions and k selection conditions on the individual tables. Once the whole `WHERE` clause is processed, the $2k - 1$ `Expressions` can be integrated into the appropriate selection and join operators in the query plan. Of course some of these `Expressions` may turn out to be null, depending on the query.

For example, if we have `SELECT * FROM R, S, T WHERE R.A = 1 AND R.B = S.C AND T.G < 5 AND T.G = S.H`, the above approach would result in the following query plan:



You don't have to follow exactly this strategy, but it is recommended as following it will make your life easier in Projects 3 and 4. You don't need to worry about pushing projections past the joins, you may have one big projection at the root of your query plan.

3.6 Implement aliases

The next step is to implement aliases (range variables), which allow you to support queries like `SELECT R.A FROM Reservations R`. These are handy in any case, but they are essential to support self-joins.

The aliases themselves come from the `FROM` clause, and you can extract them from the `fromItem` and the `joins` of your `PlainSelect`. Unfortunately, when you reference columns in the `SELECT` and `WHERE` clauses, JSqlParser is not smart enough to know whether you are using aliases or references. Thus, if you have a `SELECT R.A`, the `R.A` is returned as a `Column`, and the embedded `Table` object has `R` as the table `name` whether or not `R` is a base table name or an alias. Thus you will need to keep track of these things yourself. You need to determine, when you start building your query plan, whether or not the query uses aliases. If yes, you need to figure out a way to keep track of all the aliases used in the query so you can resolve the column references throughout your code.

Implementing aliases is not conceptually difficult, but you may find it a bit fiddly. It is a great test of how clean and modular your code is; if you have been structuring it well, you will have to modify relatively little code. You may need to modify classes that implement `ExpressionVisitor`, as well as the class that builds your query plan from the `PlainSelect`.

It may be useful to start with supporting aliases only on single-table queries, then move on to joins. Be sure to test your code on lots of queries including self-joins, as these can bring out bugs which are not apparent when each base table is used only once. Also be sure that you can still correctly handle all the "old" queries you tested (which do not use aliases).

3.7 Implement `ORDER BY`

Next is the `ORDER BY` operator. Make sure to review Section 2.1 for important assumptions and expected semantics. You will implement `ORDER BY` by adding a `SortOperator`. This is going to read all of the output from its child, place it into an internal buffer, sort it, and then return individual tuples when requested. You can use `Collections.sort()`; you will want a custom `Comparator` to specify the different sort orders.

You may be alarmed by the above description. Yes, sort is a *blocking* operator, which means it really needs to see all of its input before producing any output (think about it - what if the tuple that comes first in the desired sort order is the last one that the child operator is going to spit out?). As you imagine, buffering all the tuples in memory will not work for very large relations; we will address this issue in Project 3 with a more general sort algorithm.

If your query has a `ORDER BY` clause, you should put the `SortOperator` as the root, followed by the rest of your plan. While we are not getting fancy with optimization yet, it is good to delay sorting as late as possible, in particular to do it after the projection(s), because there will be less data to sort that way. We are making life easier for ourselves by assuming it's always safe to defer the `ORDER BY` after the projections; this is not always the case in full "real" SQL. A query like `SELECT S.A FROM S ORDER BY S.B` is valid SQL in the real world, we just choose not to support them in 4321.

3.8 Implement `DISTINCT`

Given that you have just implemented sorting, it is easy to add support for duplicate elimination and `DISTINCT`. If the query doesn't already have an `ORDER BY`, add a `SortOperator`. Then, add a new `DuplicateEliminationOperator`. This operator assumes the input from its child is in sorted order; it reads the tuples from the child and only outputs non-duplicates.

4 Grading

We strongly suggest that you follow the architecture we described. However, we will not penalize you for making different architectural decisions, with a few exceptions:

- you must have relational `Operator`s that implement `getNextTuple()` and `reset()` methods as outlined above. This is the standard relational algebra evaluation model and you need to learn it.
- you must construct a tree of `Operators` and then evaluate it by repeatedly calling `getNextTuple()` on the root operator.
- as explained in Section 3.5, you must build a left-deep join tree that follows the ordering of the tables in the `FROM` clause. Also, you must have a strategy to identify join conditions and evaluate them as part of the join rather than doing a selection after a cross product.

Disregarding any of the above three requirements will result in arbitrarily severe point deductions. It will also make it difficult/impossible for you to reuse your own code for subsequent Projects.

Next we give the grading breakdown.

4.1 Code style and comments (10 points)

As in Project 1, 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. 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.

4.2 Automated tests (80 points)

Be sure to read Section 2.2 carefully for information on expected input and output format.

We will run your code on our own queries and data and award you 2 points for every query that returns the correct output. The queries we provide with the assignment count for 16 out of the 80 points. You can expect that we will add additional tables to the database; of course the schema of these tables will be mentioned in the schema file and the data files will be found in the data directory.

We may test with arbitrarily complex queries that include any/all of the features you are to implement. We may also reorder the queries we gave you and/or intersperse them with our own, so don't hardcode any functionality on the assumption that the queries will be run in any particular order.

5 Submission instructions

Double-check that your code behaves as described in Section 2.2. If you are unclear about anything in Section 2.2, please clarify with a member of the course staff.

Create a README text file containing the following information.

- a line stating which is the top-level class of your code (the interpreter/harness that reads the input and produces output).
- an explanation of your logic for extracting join conditions from the WHERE clause. If this logic is fully explained in comments in your code, your README does not need to repeat that; however, it must mention exactly where in the code/comments the description is, so the grader can find it easily.
- any other information you want the grader to know, such as known bugs.

Submit via CMS a .zip archive containing:

- a `cs4321_p2.jar` file of your project that we can run on the command line as described in Section 2.2
- your Eclipse project folder (yes, we still need this even though you are submitting a .jar as well)
- your README file
- an acknowledgments file if you consulted any external sources, as required under the academic integrity policy

6 Credits/acknowledgments

O. Kennedy, D. Darde, S. Jain.