University of Edinburgh
School of Informatics
INFR11199 / INFR11217 - Advanced Database Systems
Spring 2025

Coursework Assignment
Due: **Thursday, 27 March 2025 at 12:00 noon**

IMPORTANT:

- **Plagiarism:** Every student has to work **individually** on this project assignment.

  All of the code for this project must be your own. You may not copy source code from other students or other sources that you find on the web. You may not share your code with other students. You may not host your code on a public code repository.

  Each submission will be checked using plagiarism detection software.

  Plagiarism will be reported to School and College Academic Misconduct Officers. See the University's page on Academic Misconduct for additional information.

- Start early and proceed in steps. Read the assignment description carefully before you start programming.

- The assignment is out of 100 points and counts for 40% of your final mark.

1 Goals and Important Points

In this project assignment, you will develop a lightweight database management system named BlazeDB. The goals of this assignment are threefold:

- to teach you how to translate SQL queries to relational algebra query plans,
- to familiarize you with the iterator model for relational operator evaluation, and
- to implement common operators (e.g., selection, projection, join, sort, aggregation).

The assignment consists of two tasks:

Task 1 (60%): Implementation of the iterator model and common operators.

Task 2 (30%): Optimisation of constructed query plans.

The remaining 10% of your mark is for code style and comments; more details will be provided later. Task 2 requires elements of independent thinking and creativity, which are necessary for distinction marks according to the Common Marking Scheme.

You will begin with a skeleton project that includes the main class `BlazeDB`, which defines the expected command-line interface and the evaluation loop, and the `Iterator` class, which defines the minimum set of functions that every operator must implement. You are free – and expected – to modify these classes, e.g., as they currently lack comments. However, any changes you make *must* preserve the command-line interface. The project is also pre-configured to use JSqlParser[1], so you do not have to write your own SQL parser unless you choose to do so. The main class includes an example of how to parse a SQL string into a Java object.

The rest of this document describes in detail how to complete the assignment. Take time to read it carefully. Note that some topics will be covered later in the course. Start working as early as possible. This is not a project that should be left to the last minute.

You may start by reviewing the course material on SQL. Also, brush up your knowledge of the basic principles of object-oriented programming (encapsulation, inheritance, polymorphism, abstraction) and design patterns; in particular, the singleton pattern and the visitor pattern might come in handy for this assignment.

[1] `https://github.com/JSQLParser/JSqlParser`

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 an input file containing one SQL query. The program will process and evaluate the SQL query on the database and write the query result in the specified output file.

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 include one or more `SUM` aggregates in the `SELECT`, a `DISTINCT`, an `ORDER BY`, a `GROUP BY`, or a combination of them. You do not need to support nested subqueries, set operators (e.g., `UNION`), other aggregate functions (e.g., `COUNT`, `AVG`), 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.

Here's an example of a valid query:

```
SELECT Student.name, Enrolled.grade FROM Student, Enrolled
WHERE Student.sid = Enrolled.sid;
```

- You may assume all valid queries follow correct SQL syntax and only refer to tables that exist in the database. If a query refers to a table column (e.g., `Student.age`), you may assume the column name (`age`) is valid for that table (`Student`). You may also assume that all column references will be fully qualified by the base table name, that is, in the form ⟨table⟩.⟨column⟩.
- You may assume there will be at least one table in the `FROM` clause. Additionally, tables in the `FROM` clause will *not* use aliases, such as `Student AS S` or `Student S`.
- Consequently, self-joins (i.e., joining a table with itself), which require the use of aliases, are not considered valid.
- The `WHERE` clause, if present, is a conjunction (`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, `Student.sid = Enrolled.sid`, `Student.sid < 3`, and `42 = 42` are all valid expressions for the `WHERE` cause. However, `Student.sid < Course.cid - 1` is not valid, even though it would be acceptable in standard SQL.
- The `SELECT` clause will either specify a subset of columns or have the form `SELECT *`. In both cases, it can optionally include one or more `SUM` aggregate functions. For

`SELECT *`, you should order the columns in your answer following the order of the tables in the `FROM` clause. For example, for `SELECT * FROM R, S`, each answer row has all the columns of `R` followed by all the columns of `S`. The order of columns in a table is defined by the table schema.

- The `SELECT` clause may optionally contain one or more `SUM` aggregate functions. We restrict the form of the `SUM` function to take as argument either one term (integer or column reference) or a product of terms. Thus, for the purpose of this assignment, `SUM(5)`, `SUM(A)`, `SUM(A*B)`, `SUM(C*C*C)` are valid expressions, while for example `SUM(A+B)` and `SUM(A/2)` are not valid expressions.

  You can assume that `SUM` aggregates, if present, are always at the end of the `SELECT` clause. It is possible that `SELECT` consists of just one `SUM`; e.g., `SELECT SUM(1) FROM R` is valid and returns one integer representing the number of tuples in R.

- There may be a `GROUP BY` clause that specifies a subset of columns for grouping. The `SELECT` list can only list group-by columns but not necessarily all of them, that is, some group-by columns can be omitted from the output; e.g., `SELECT A FROM R GROUP BY A, B` is a valid query. You may assume that `HAVING` will not be used.

  The `SUM` functions, if present, can reference any columns, including non-group-by columns; e.g., `SELECT A, SUM(A*B) FROM R GROUP BY A` is a valid query.

- There may be an `ORDER BY` clause that specifies a subset of columns for ordering. You may assume that we only want to sort in ascending order. If two tuples agree on all sort attributes, you can order them as you prefer. You may assume that no `ASC`, `DESC`, `OFFSET`, or `LIMIT` keywords will be used.

  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. This does not mean that every attribute in `ORDER BY` must be mentioned in the `SELECT` – a query like `SELECT * FROM Student ORDER BY Student.name` is valid.

- There may be a `DISTINCT` right after the `SELECT`, and it should be processed appropriately. Yes, `SELECT DISTINCT * FROM ...` is valid.

2.2 Data and Output Formats

We have provided you some sample data and some sample queries. Take a look at the `samples` directory. It has `db`, `input`, and `expected_output` as subdirectories.

- The `input` directory contains SQL files with some example queries. There is one SQL query per input file.

- The `db` directory contains `schema.txt`, which defines the database schema, and the `data` directory, which contains the actual data. The names `schema.txt` and `data`

are hard-coded and must exist in a valid database directory. Your program should support *any* schema defined in `schema.txt`, not just the schema of the sample data.

The `schema.txt` file contains one line per relation (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 with the added `.csv` extension. Every file contains zero or more tuples; a tuple is a line in the file with field (attribute) values separated by commas. *All attribute values are integers.* 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, but you do need to handle empty tables.

- The `expected_output` directory contains the expected output files for the queries we provided. For example, `query1.csv` contains the expected output for the query in `query1.sql`. The format for the output is the same as the format for the data.

2.3 Setting Up Local Development Environment

You are free to use any text editor or IDE to complete the project. We will use Maven to compile your project. We recommend setting up a local development environment by installing Java 8 or later and using an IDE such as IntelliJ or Eclipse. To import the project into IntelliJ or Eclipse, make sure that you import as a Maven project (select the `pom.xml` file when importing).

2.4 Compile and Run

Your SQL interpreter is a program that takes three arguments: the path to a database directory, the path to an input SQL file, and the path to an output file. The program then executes the SQL query from the input file on the given database and writes the result to the given output file. The `BlazeDB.java` file provides this command-line interface. Run the `BlazeDB` class from your IDE or command line, providing the required arguments.

We will compile your code from the command line, producing a runnable .jar file:

```
mvn clean compile assembly:single
```

This command will produce `target/blazedb-1.0.0-jar-with-dependencies.jar`. We can run this file as follows:

```
$ java -jar target/blazedb-1.0.0-jar-with-dependencies.jar
Usage: BlazeDB database_dir input_file output_file
```

The `BlazeDB` class requires passing three mandatory arguments.

```
$ java -jar target/blazedb-1.0.0-jar-with-dependencies.jar
    samples/db samples/input/query1.sql samples/output/query1.csv
SELECT items: [*]
WHERE expression: null
```

Your code should handle these arguments appropriately (i.e., do not hardcode any paths).

We will test your code using our own test queries and databases with potentially different schemas, including different table names. The database directory will have the same structure as described above, with files in the `data` directory named according to the database schema with `.csv` as the file extension. You may assume that prior to execution the given output file does not exist but the output directory does exist.

After we run your code, we will compare your output files with ours. For queries without an `ORDER BY`, it is acceptable 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 on sort attributes, while tied tuples may have a different order to ours. As you can imagine, it is important for you to respect the expected input and output format.

Note: We will test your code on a DICE machine with Ubuntu Linux. Remember that Linux/MacOS environments use '/' as path separator. The database directory will be provided with no final '/' symbol, as above. If you use Windows, make sure that when you form file paths, you use `File.separator` instead of '\' as path separator.

2.5 Operators and the Iterator Model

A key abstraction in this project 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*, *duplicate elimination*, and *group-by aggregation* operators, which are not part of the basic relational algebra but must be added to support `ORDER BY`, `DISTINCT`, `GROUP BY`, and `SUM`.
- 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 are provided an abstract class `Operator`, which all your operators must extend. 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()`, inherited from the 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()` returns 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., repeatedly scan the inner table in a join.

For each of the above operators, you will implement 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 materialising (writing to disk) intermediate results.

The bulk of this project involves implementing each of the aforementioned 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.

3 Task 1: Iterator Model Implementation

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

We also recommend (but do not require) you set up a test infrastructure 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 Setting up JSqlParser

For this project, you do not need to write your own SQL parser. We recommend using JSqlParser, which takes care of parsing your SQL and creating a Java object. JSqlParser is an open source project:

- The project page `https://github.com/JSQLParser/JSqlParser` contains a wiki with examples of how to get started with JSqlParser.
- The online documentation is available at `https://javadoc.io/doc/com.github.jsqlparser/jsqlparser/latest/index.html`.

The `pom.xml` file already has a JSqlParser dependency. You are not required to use JSqlParser, but you need to correctly parse all valid queries as defined in Section 2.1. In case you do use JSqlParser, you should play around with it 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 simple method in `BlazeDB.java` that uses JSqlParser to read a SQL query from a file and print it out. This method 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 `Statement`s we will work with are `Select`s, 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 the `distinct`,`orderByElements`, and `groupByElements` 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`, `LongValue`, `Column`, `Multiplication` (used in the `SUM` aggregate), `EqualsTo`, `NotEqualsTo`, `GreaterThan`, `GreaterThanEquals`, `MinorThan`, and `MinorThanEquals`. These capture the recursive structure of an expression. The last six of the aforementioned expression types 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 and an embedded `Table` object. Every `Table` object has a table name. `SUM` aggregates are expressions of type `Function`.

JSqlParser also provides a number of `Visitor` interfaces, which you may or may not choose to use. In particular, `ExpressionVisitor` and `ExpressionDeParser` are highly recommended to use once you get to the selection operator. Check out also the wiki page of JSqlParser on how to evaluate expressions.

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 Student`. To achieve this, you will need to implement the scan operator.

Implement a `ScanOperator` that extends your `Operator` abstract class. Every instance of `ScanOperator` knows which base table it is scanning. Upon initialisation, 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 modify the provided `Tuple` class for this purpose.

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.

Once you have written `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:

- parse the query from the input file
- construct a `ScanOperator` for the table in the `fromItem`
- execute the operator via the provided `execute` function; for testing purposes only, you may print the result to console.

3.3 Implement Selection

You will next implement single-table selection. You are aiming to support queries like `SELECT * FROM Student WHERE Student.sid = 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 does not 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 extends JSqlParser's `ExpressionDeParser`. 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 Student.sid FROM Student WHERE Student.age = 20`.

For implementing projection, you need a third `Operator` that is a `ProjectOperator`. Recall that bag projection does not eliminate duplicates. When `getNextTuple()` is called, it grabs the next tuple from its child, extracts only desired values into a new tuple, and returns that tuple. Note that the child (so far) 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 `SelectItem` objects, 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 produce different output results.

By this point your code should take an SQL query and produce a query plan with:

- 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 have not done so already. Thus, you should have a top-level interpreter class that reads the statement from the query file. You should also have a second class that knows how to construct a query plan for a `Statement`, and returns the query plan back to the interpreter, so the interpreter can `execute()` the plan and produce the results of the query plan.

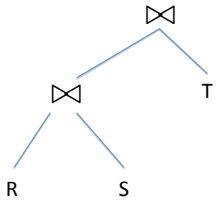
3.5 Implement Join

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` (or any other comparison operator, not just `=`), a conjunction (`AND`) of comparisons, or it could be `null` if the join is a cross product.

Implement the simple (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 part is not optimisation, you do not want to compute cross products unless you have to as this is grossly inefficient. 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.

3.6 Implement `ORDER BY`

Next is the `ORDER BY` operator. 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 numbers of tuples; for this project assignment, this is fine.

If your query has an `ORDER BY` clause, you should put the `SortOperator` as the root, followed by the rest of your plan. 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 "real" SQL. A query like `SELECT S.A FROM S ORDER BY S.B` is valid SQL, but we choose not to support it in this project.

3.7 Implement `DISTINCT`

To add support for duplicate elimination and `DISTINCT`, you will add a new operator `DuplicateEliminationOperator`. You are free to choose either a sorting-based or hashing-based implementation for this operator.

3.8 Implement `GROUP BY` with `SUM` Aggregation

The final operator to implement is group-by aggregation with the `SUM` function. The `SUM` functions may appear at the end of the `SELECT` clause, after all the selected attributes. Each `SUM` can take as argument one term (integer or column) or a product of terms.

You will implement one group-by aggregation operator, `SumOperator`, which reads all of the output from its child, extracts relevant values into tuples, organizes tuples into groups, and for each group computes aggregate values. The sum operator needs to see all of its input before producing any output (i.e., it is a *blocking* operator).

4 Task 2: Query Optimisation

The final task is to enrich BlazeDB with query optimisation. The goal of query optimisation is to transform query plans such that their operators process as few tuples as possible, thus reducing processing time and avoiding large intermediate results; yet such optimised query plans must still produce correct query results. To achieve this goal, you must keep the same join order but you are allowed to transform query plans, for example, swap operators or introduce new instances of the non-join operators discussed above. You should not implement any new relational algebra operator.

Properly optimised query plans should avoid processing large intermediate results whenever possible, thus allowing such plans to be executed under restricted memory budgets and time limits. We will evaluate the effectiveness of your optimisation rules by running a set of queries on large databases. Failing to properly optimise input queries will most likely lead to wrong results, out-of-memory exceptions, or timeouts.

Task 2 is not about improving the performance of individual relational algebra operators but about coming up with optimisation rules for the class of valid SQL aggregate queries, described in Section 2.1. Some optimisation rules will be covered in lectures, but some will require a bit of thinking on your side on how to reduce the size of intermediate results during query processing.

5 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. Do not hard-wire combinations of operators, e.g., projection should not assume selection as its child (i.e., enforce abstraction).
- 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 severe point deductions.

Next we give the grading breakdown.

5.1 Code Style and Comments (10 points)

Follow standard guidelines for writing clean and understandable code; e.g., use standard naming conventions for classes and methods, break up large monolithic blocks of code into smaller logical pieces, avoid code duplication if possible – the "rule of three" says if your code is copied more than twice, then it probably needs to be abstracted out.

You must provide comments for every method you implement. At minimum, the comment must include one sentence about the purpose 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.

5.2 Test Queries (90 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 database. The queries we provide with the assignment count for 24 out of the 90 points. You can expect that we will add

additional tables to the database; of course the schema of these tables will be mentioned in `schema.txt` and the data files will be found in the `data` directory.

Task 1 is worth 60 points, while Task 2 is worth 30 points. Any optimisation rules you implement for Task 2 must be correct, thus leaving them enabled for Task 1 is fine.

We will test with basic queries as well as 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 do not hardcode any functionality on the assumption that the queries will be run in any particular order.

If you cannot implement one or more operators, that is fine, although obviously you won't get full points. In that case, you must clearly tell us in the README what you have not been able to implement.

6 Submission Instructions

Double-check that your code compiles and runs as described in Section 2.2. If your code fails to compile or crashes during execution, we will invest a reasonable effort to fix the problem; this can cause (severe) point deduction.

You must keep the `BlazeDB` class as the top-level main class of your code.

Create a README text file containing the following information.

- For Task 1, 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.
- For Task 2, an explanation of your optimisation rules, reasons why they are correct, and how they can reduce the size of intermediate results during query evaluation.
- Any other information you want the grader to know, such as known bugs.

Create a .zip archive containing a README file and your project folder, so that we can compile and run your code from the command line. Do not include any `.class` files nor large `.csv` files. Upload the zip archive to Learn: Assessment → CW1 - Programming.

Ensure that all debugging statements are disabled before submitting your code to allow for testing with large datasets. Failure to do so may result in a point deduction.

Keep in mind that this is an individual assignment and not a group project. Your work will be reviewed for any signs of plagiarism.

Make sure you start early! Good luck!