Checkpoint1

- **Overview**: Submit a simple SPJUA query evaluator.
- **Deadline**: Feb 23
- **Grade**: 15% of Project Component
  - 5% Correctness
  - 5% Efficiency
  - 5% Code Review

In this project, you will implement a simple SQL query evaluator with support for Select, Project, Join, Bag Union, and Aggregate operations. You will receive a set of data files, schema information, and be expected to evaluate multiple SELECT queries over those data files.

Your code is expected to evaluate the SELECT statements on provided data, and produce output in a standardized form. Your code will be evaluated for both correctness and performance (in comparison to a naive evaluator based on iterators and nested-loop joins).

Parsing SQL

A parser converts a human-readable string into a structured representation of the program (or query) that the string describes. A fork of the JSQLParser open-source SQL parser (JSQLParser) will be provided for your use. The JAR may be downloaded from

http://odin.cse.buffalo.edu/resources/jsqlparser/jsqlparser.jar

And documentation for the fork is available at

http://odin.cse.buffalo.edu/resources/jsqlparser

You are not required to use this parser (i.e., you may write your own if you like). However, we will be testing your code on SQL that is guaranteed to parse with JSqlParser.

Basic use of the parser requires a `java.io.Reader` or`java.io.InputStream` from which the file data to be parsed (For example, a `java.io.FileReader`). Let's assume you've created one already (of either type) and called it `inputFile`.

```
CCJSqlParser parser = new CCJSqlParser(inputFile);
Statement statement;
while((statement = parser.Statement()) != null){
  // `statement` now has one of the several
  // implementations of the Statement interface
}
// End-of-file.  Exit!
```

At this point, you'll need to figure out what kind of statement you're dealing with. For this project, we'll be working with `Select` and`CreateTable`. There are two ways to do this. JSqlParser defines a Visitor style interface that you can use if you're familiar with the pattern. However, my preference is for the simpler and lighter-weight `instanceof` relation:

```
if(statement instanceof Select) {
  Select selectStatement = (Select)statement;
  // handle the select
} else if(statement instanceof CreateTable) {
  // and so forth
}
```

Example

Project 1 Demo : Invoking JSqlParser

https://www.youtube.com/watch?v=U4TyaHTJ3Zg

Project 1 Demo : Building a Relational Plan

https://www.youtube.com/watch?v=KeSKi4EtZEg

Project 1 Demo : Concepts to Implementation

https://www.youtube.com/watch?v=i1ipGRFlx2U

Expressions

JSQLParser includes an object called Expression that represents a primitive-valued expression parse tree. In addition to the parser, we are providing a collection of classes for manipulating and evaluating Expressions. The JAR may be downloaded from

http://odin.cse.buffalo.edu/resources/expressionlib/expression.jar

Documentation for the library is available at

http://odin.cse.buffalo.edu/resources/expressionlib

To use the `Eval` class, you will need to define a method for dereferencing `Column` objects. For example, if I have a `Map` called`tupleSchema` that contains my tuple schema, and an `ArrayList`called `tuple` that contains the tuple I am currently evaluating, I might write:

```
public void LeafValue eval(Column x){
  int colID = tupleSchema.get(x.getName());
  return tuple.get(colID);
}
```

After doing this, you can use Eval.eval() to evaluate any expression in the context of tuple.

Source Data

Because you are implementing a query evaluator and not a full database engine, there will not be any tables — at least not in the traditional sense of persistent objects that can be updated and modified. Instead, you will be given a **Table Schema** and a **CSV File**with the instance in it. To keep things simple, we will use the `CREATE TABLE` statement to define a relation's schema. You do not need to allocate any resources for the table in reaction to a `CREATE TABLE`statement — Simply save the schema that you are given for later use. Sql types (and their corresponding java types) that will be used in this project are as follows:

SQL TYPE	**JAVA EQUIVALENT**
string	StringValue
varchar	StringValue
char	StringValue
int	LongValue
decimal	DoubleValue
date	DateValue

In addition to the schema, you will be given a data directory containing multiple data files who's names correspond to the table names given in the `CREATE TABLE` statements. For example, let's say that you see the following statement in your query file:

```
CREATE TABLE R(A int, B int, C int);
```

That means that the data directory contains a data file called 'R.dat' that might look like this:

```
1|1|5

1|2|6

2|3|7
```

Each line of text (see `java.io.BufferedReader.readLine()`) corresponds to one row of data. Each record is delimited by a vertical pipe '|' character. Integers and floats are stored in a form recognized by Java's Long.parseLong() and Double.parseDouble() methods. Dates are stored in YYYY-MM-DD form, where YYYY is the 4-digit year, MM is the 2-digit month number, and DD is the 2-digit date. Strings are stored unescaped and unquoted and are guaranteed to contain no vertical pipe characters.

Queries

Your code is expected to support both aggregate and non-aggregate queries with the following features. Keep in mind that this is only a minimum requirement.

- Non-Aggregate Queries
    - SelectItems may include:
        - **SelectExpressionItem**: Any expression that ExpressionLib can evaluate. Note that Column expressions may or may not include an appropriate source. Where relevant, column aliases will be given, unless the SelectExpressionItem's expression is a Column (in which case the Column's name attribute should be used as an alias)
        - **AllTableColumns**: For any aliased term in the from clause
        - **AllColumns**: If present, this will be the only SelectItem in a given PlainSelect.
- Aggregate Queries
    - **SelectItems** may include:
        - **SelectExpressionItem**s where the Expression is one of:
            - A Function with the (case-insensitive) name: SUM, COUNT, AVG, MIN or MAX. The Function's argument(s) may be any expression(s) that can be evaluated by ExpressionLib.
            - A Single Column that also occurs in the GroupBy list.
        - **AllTableColumns**: If all of the table's columns also occur in the GroupBy list
        - **AllColumns**: If all of the source's columns also occur in the GroupBy list.
    - GroupBy column references are all Columns.
- Both Types of Queries
    - From/Joins may include:
        - **Join**: All joins will be simple joins
        - **Table**: Tables may or may not be aliased. Non-Aliased tables should be treated as being aliased to the table's name.
        - **SubSelect**: SubSelects may be aggregate or non-aggregate queries, as here.
    - The Where/Having clauses may include:
        - Any expression that ExpressionLib will evaluate to an instance of BooleanValue
    - Allowable Select Options include
        - SELECT DISTINCT (but not SELECT DISTINCT ON)
        - UNION ALL (but not UNION)

- Order By: The OrderByItem expressions may include any expression that can be evaluated by ExpressionLib. Columns in the OrderByItem expressions will refer only to aliases defined in the SelectItems (i.e., the output schema of the query's projection. See TPC-H Benchmark Query 5 for an example of this)
- Limit: RowCount limits (e.g., LIMIT 5), but not Offset limits (e.g., LIMIT 5 OFFSET 10) or JDBC parameter limits.

Output

Your code is expected output query results in the same format as the input data:

- One output row per ('\n'-delimited) line. If there is no ORDER BY clause, you may emit the rows in any order.
- One output value per ('|'-delimited) field. Columns should appear in the same order that they appear in the query. Table Wildcards should be resolved in the same order that the columns appear in the CREATE TABLE statement. Global Wildcards should be resolved as Table Wildcards with the tables in the same order that they appear in the FROM clause.
- A trailing newline as the last character of the file.
- You should not output any header information or other formatting.

Example Queries and Data

These are only examples. Your code will be expected to handle these queries, as well as others.

Sanity Check Examples: A thorough suite of test cases covering most simple query features.
Example NBA Benchmark Queries: Some very simple queries to get you started.
The TPC-H Benchmark: This benchmark consists of two parts: DBGen (generates the data) and a specification document (defines the queries). A nice summary of the TPC-H queries can be found here.
The SQL implementation used by TPC-H differs in a few subtle ways from the implementation used by JSqlParser. Minor structural rewrites to the queries in the specification document will be required:

- The date format used by TPC-H differs from the date format used by SqlParser. You will need to replace all instances of date 'YYYY-MM-DD' with DATE('YYYY-MM-DD') or {d'YYYY-MM-DD'}
- Many queries in TPC-H use INTERVALs, which the project does not require support for. However, these are all added to hard-coded parameters. You will need to manually add the interval to the parameter (e.g., DATE '1982-01-01' + INTERVAL '1 YEAR' becomes DATE('1983-01-01'))

Queries that conform to the specifications for this project include: Q1, Q3, Q5, Q6, Q8*, Q9, Q10, Q12*, Q14*, Q15*, Q19* (Asterisks mean that the query doesn't meet the spec as written, but can easily be rewritten into one that does)

- Q2 requires SubSelect expressions.
- Q4 requires EXISTS and SubSelect expressions.
- Q7 requires an implementation of the EXTRACT function.
- Q8 violates the restriction on simple select items in aggregate queries. It can be rewritten into a compliant form with FROM-nested Selects.
- Q11 violates the simple select item restriction, and requires SubSelect expressions.
- Q12 requires IN expressions, but may be rewritten into a compliant form.
- Q13 requires Outer Joins.
- Q14 violates the simple select item restriction, but may be rewritten into a compliant form.
- Q15 uses views, but may be rewritten into a compliant form
- Q16 uses IN and NOT IN expressions as well as SubSelects
- Q17 uses SubSelect expressions and violates the simple select item restriction
- Q18 uses IN and violates the simple select item restriction
- Q19 uses IN but may be rewritten into a compliant form
- Q20 uses IN and SubSelects
- Q21 uses EXISTS, NOT EXISTS and SubSelects
- Q22 requires an implementation of the SUBSTRING function, IN, NOT EXISTS and SubSelects

Code Submission

As before, all .java files in the src directory at the root of your repository will be compiled (and linked against JSQLParser). Also as before, the class

```
   edu.buffalo.cse562.Main
```

will be invoked with the following arguments:

- –data data directory: A path to a directory containing the .dat data files for this test.
- sql file: one or more sql files for you to parse evaluate.

For example:

```
$> ls data

R.dat

S.dat

T.dat

$> cat R.dat

1|1|5

1|2|6

2|3|7

$> cat query.sql

CREATE TABLE R(A int, B int, C int)

SELECT B, C FROM R WHERE A = 1

$> java -cp build:jsqlparser.jar edu.buffalo.cse562.Main --
data data query.sql

1|5

2|6
```

Once again, the data directory contains files named table name.dat where table name is the name used in a CREATE TABLE statement. Notice the effect of CREATE TABLE statements is not to create a new file, but simply to link the given schema to an existing .dat file. These files use vertical-pipe (’|’) as a field delimiter, and newlines (’\n’) as record delimiters.

The testing environment is configured with the Sun JDK version 1.8.

Grading

Your code will be subjected to a sequence of test cases, most of which are provided in the project code (though different data will be used). Two evaluation phases will be performed. Phase 1 will be performed on small datasets (< 100 rows per input table) and each run will be graded on a per-test-case basis as follows:

- **0/10 (F)**: Your submission does not compile, does not produce correct output, or fails in some other way. Resubmission is highly encouraged.
- **5/10 (C)**: Your submission runs the test query in under 30 seconds on the test machine, and produces properly formatted output.
- **7.5/10 (B)**: Your submission runs the test query in under 15 seconds on the test machine, and produces the correct output.
- **10/10 (A)**: Your submission runs the test query in under 5 seconds on the test machine, and prduces the correct output.

Phase 2 will evaluate your code on more complex queries that create large intermediate states (100+ MB). Queries for which your submission does not produce correct output, or which your submission takes over 1 minute to process will receive an F. Otherwise, your submission will be graded on the runtime of each test as follows

- **5/10 (C)**: Produce correct output in 1 minute or less.
- **7.5/10 (B)**: Produce correct output in 30 seconds or less.
- **10/10 (A)**: Produce correct output in 15 seconds or less
- **12/10 (A+)**: Produce correct output in 8 seconds or less

Your overall project grade will be a weighted average of the individual components. It will be possible to earn extra credit by beating the reference implementation.

Additionally, there will be a per-query leader-board for all groups who manage to beat the reference implementation.