Checkpoint 0: A Database Hello World

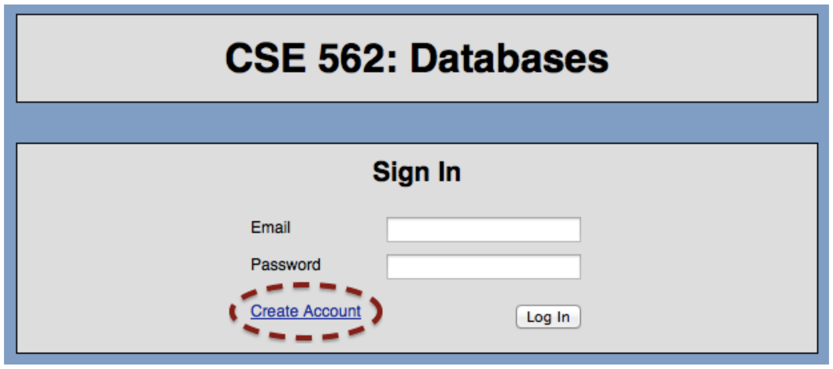
- **Overview**: Answer Trivial Queries
- **Deadline**: Feb 8
- **Grade**: 5% of Overall Grade

This project aims to get you familiarized with the course submission system, force you to set up your build and testing environment, and to introduce you to the basics of JSqlParser.

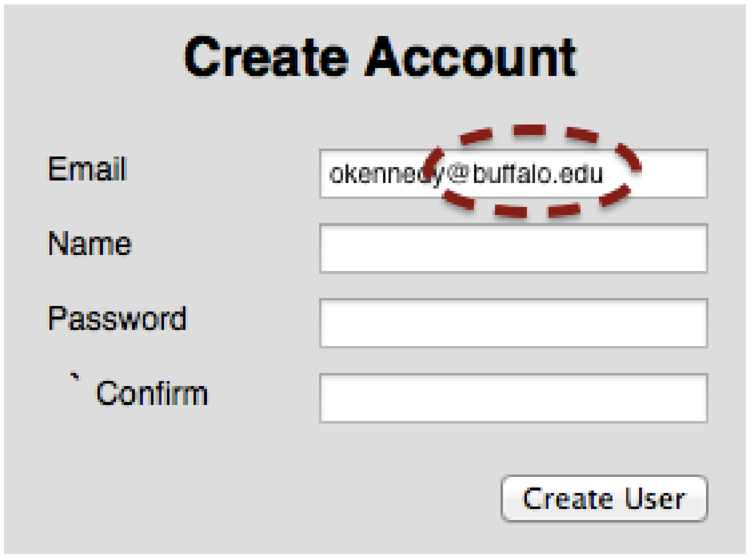
Meet The Submission System

Let's first get familiar with the submission system. It's located at: http://dubstep.odin.cse.buffalo.edu

After loading up the website, you will need to create an account.



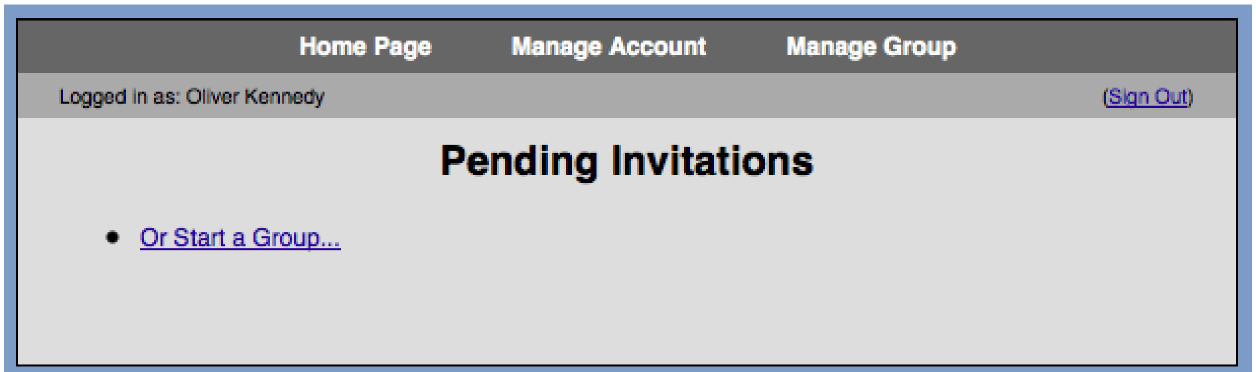
When creating an account, be sure to use your UB email address. If you don't have a UB email address, contact the teacher or a TA as soon as possible.



After you create an account, you'll receive an email with an activation token. Click on the link in the email, or copy it into your browser's location bar.

Forming Groups

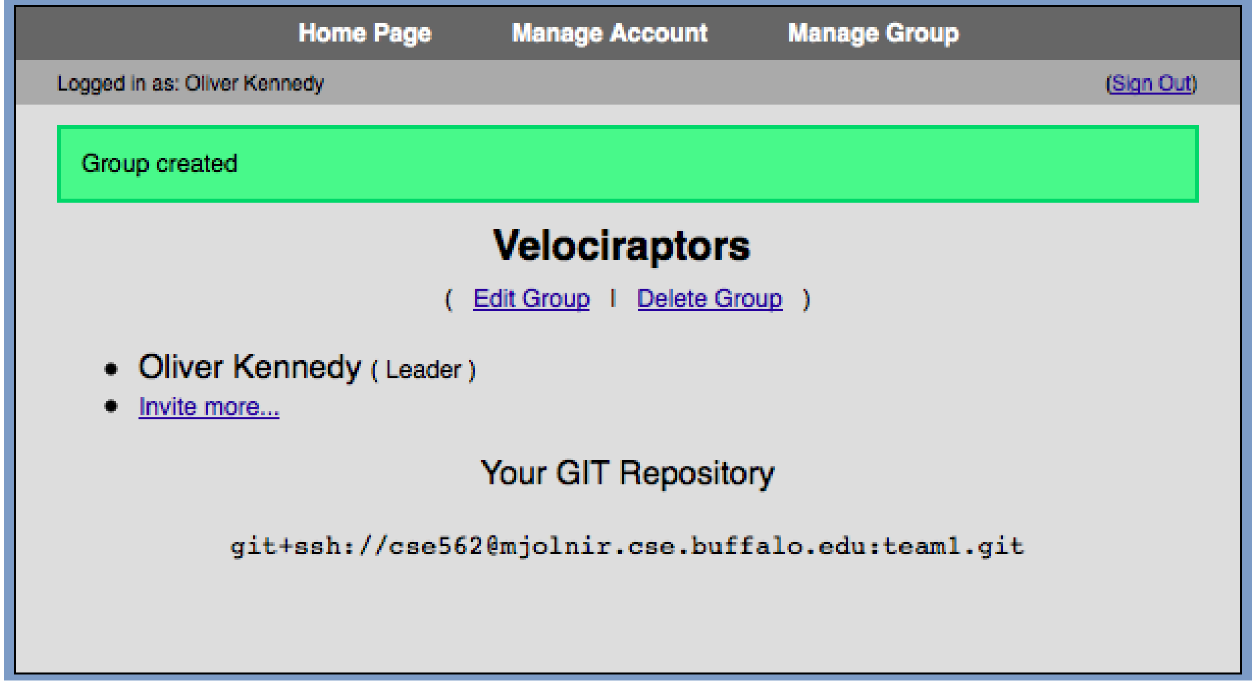
Find up to three other students in the class, and elect one member of your group to be the leader. You'll also need a group name. Be creative. This is how you'll show up on the leaderboards. The group leader should go to the **Manage Group** tab and click "Or Start a Group..."



Logged in as: Oliver Kennedy

Pending Invitations

- Or Start a Group...



Logged in as: Oliver Kennedy

Group created

Velociraptors

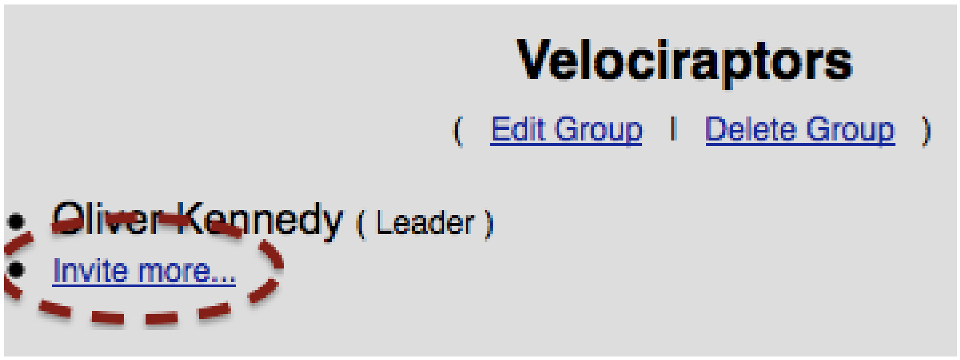
(Edit Group | Delete Group)

- Oliver Kennedy (Leader)
- Invite more...

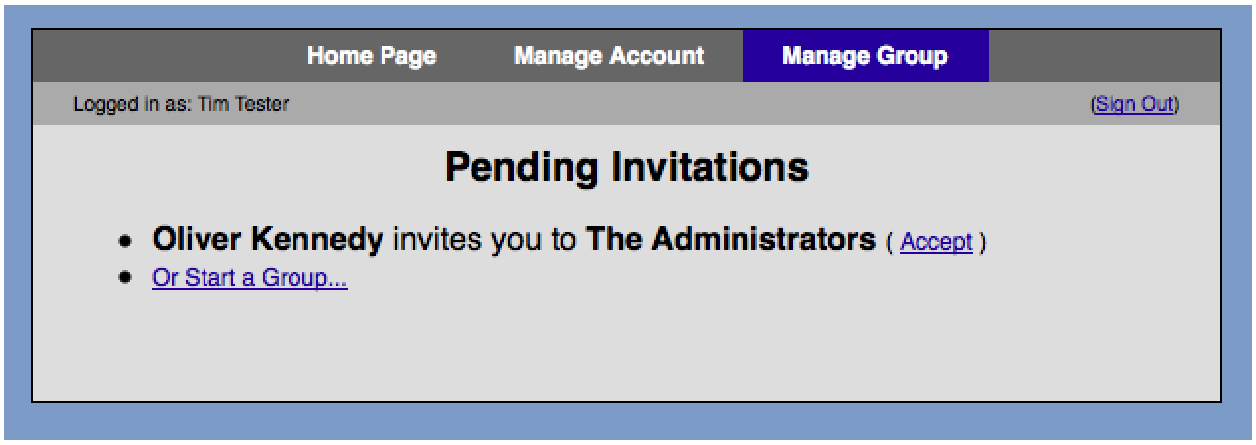
Your GIT Repository

```
git+ssh://cse562@mjolnir.cse.buffalo.edu:team1.git
```

After creating and naming your group, your leader should click "Invite more..." on the **Manage Group** tab and add all remaining group members by their email addresses.



All team members should now be able to accept their invitation by logging in and going to their **Manage Group** tab.



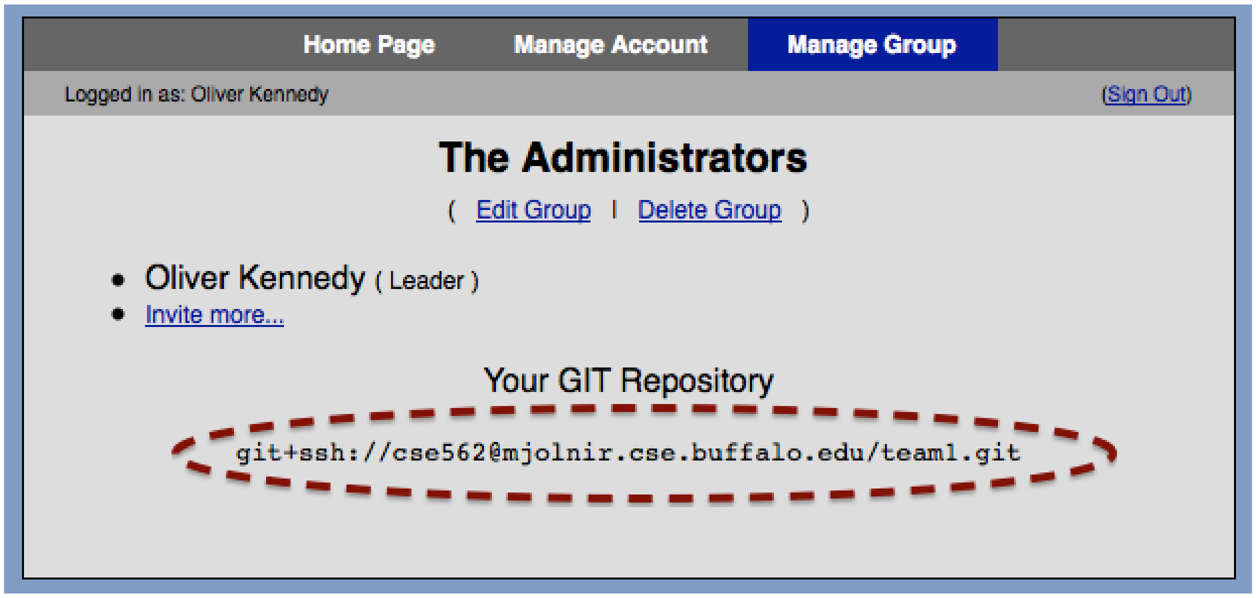
GIT and Source Code Management

For submissions, and for your group's convenience, DμBStep provides your group with a GIT repository. If you don't know how to use GIT, it's an important skill to have. Numerous tutorials and reference materials are available, including:

- http://git-scm.com/documentation
- http://git-scm.com/book/en

If you don't want to dive headfirst into GIT, a nice user-friendly front-end is SourceTree http://www.sourcetreeapp.com. Alternatively, read on below for a quick and dirty intro to the three GIT commands you can't live without.

The upstream URL of your team's GIT repository is available from the **Manage Group** tab.



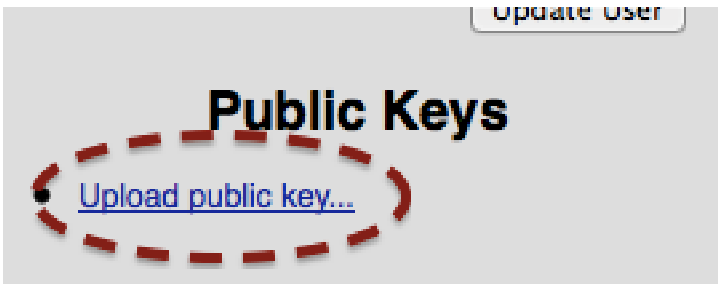
```
git+ssh://cse562@mjolnir.cse.buffalo.edu/team1.git
```

To access the repository, you'll first need to register your GIT public key. An overview of public key management can be found here. A public key should look something like this (with no line breaks):

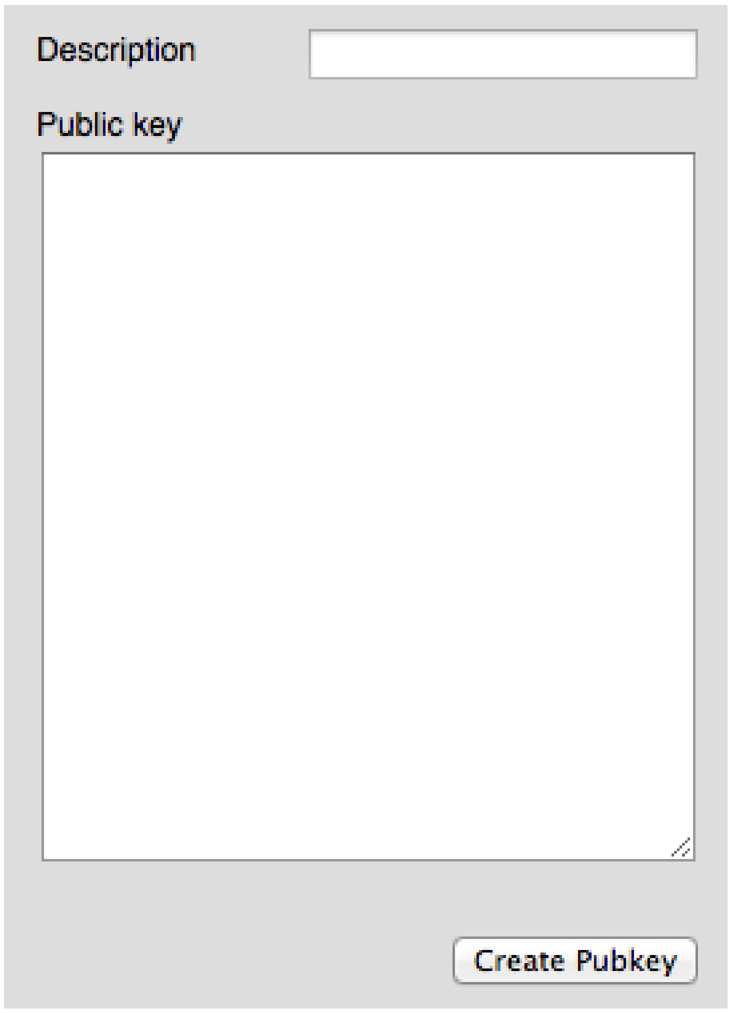
```
ssh-rsa AAAAB3NzaC1yc2EAAAADAQABAAABAQDgX8jMmapRQ7pIJ0JV9zfvkqef/OBV//y3t0ceV5KaZ4
DMIcn+xzonR/OR4cTuAyQRyQK3TlamleUATQe9JAieaI3dodnCfrN7C16RiqkB6iQorpCC+LdkdM7n3rVtle
IAY93Imoq6tJEf+boeLz7EtB6I7OJSZ+NgRv5Z4vvF2hlgJrXaCr+ofURm/ILOHB1AdcZiXVL8tPOVI/FG170/
i1fI+Y1eyQtko10XlHTHx4bGavYMsOKWoVjTBCruH8/VmiaUY7RBTn8Qg+yOQZPIOTrtWxRm0/Q373hK
n8Xt+Dh38tHL3Z8X2C4jup/JFRmoT+nH6m9pB79IcnBNYa7V okennedy@sif
```

Uploading Your Public Key

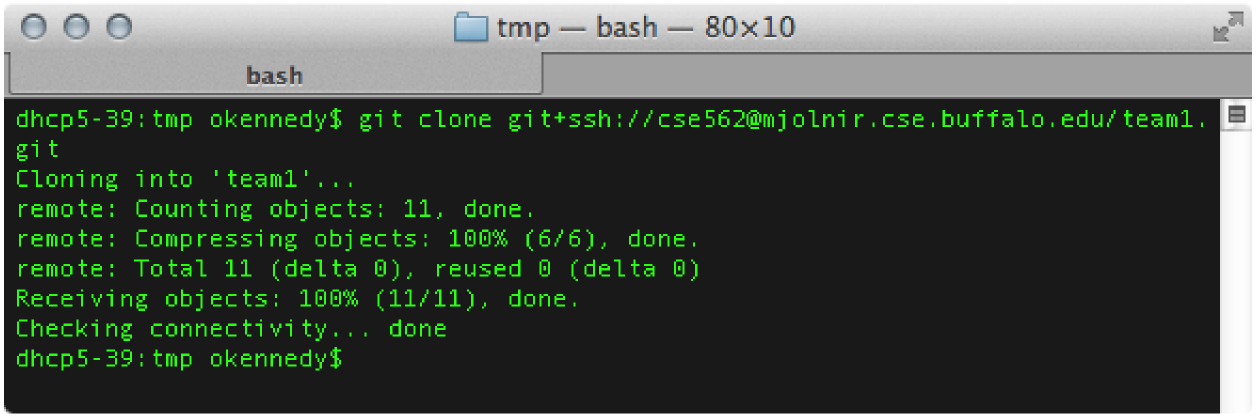
From the **Manage Account** tab, click on "Upload public key..."



Copy the entire public key into the field provided and add a short description (useful if you work on multiple computers).



You should now be able to clone your team's GIT repository:



```
dhcp5-39:tmp okennedy$ git clone git+ssh://cse562@mjolnir.cse.buffalo.edu/team1.
git
Cloning into 'team1'...
remote: Counting objects: 11, done.
remote: Compressing objects: 100% (6/6), done.
remote: Total 11 (delta 0), reused 0 (delta 0)
Receiving objects: 100% (11/11), done.
Checking connectivity... done
dhcp5-39:tmp okennedy$
```

A Quick and Dirty Intro to GIT

Once you have cloned a copy your repository (a directory called teamX, where X is your group ID), you'll need some organization. The grading script will attempt to compile all of the java files in your repository, but for your own sanity and ease of compilation it can be helpful to keep your repository organized. It's common to create a directory named `src` at the root of your git repository. Create that now.

```
> cd teamX

> mkdir src

> mkdir src/dubstep

> touch src/dubstep/Main.java
```

Now you need to make git aware of the file you just added.

```
> git add src/dubstep/Main.java
```

Next, you need to create a commit checkpoint -- a marker indicating that your local copy of the repository is in a stable state.

```
> git commit -a
```

The -a flag commits all files that have changed (you still need to manually add files that are new). You will be asked to provide a message that describes the changes that you've just made to the code. Finally, you need to send the changes to the central repository.

```
> git push
```

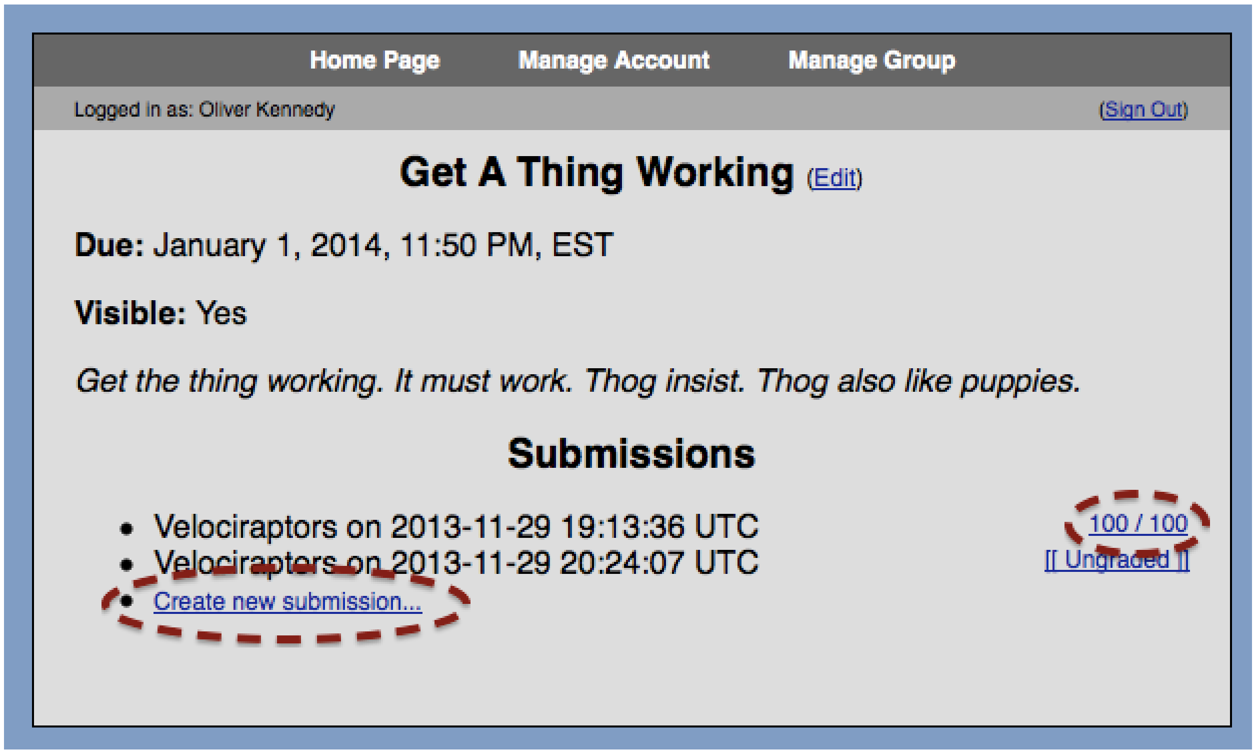
The files are now in your global repository. Your teammates can now receive your changes by pulling them from the central repository.

```
> git pull
```

If this works, you should be all set.

Submitting Code

To have your project graded, go to the **Home Page** tab, click "Show" for the project you want to submit, and click "Create new submission".



Logged in as: Oliver Kennedy

Get A Thing Working

Due: January 1, 2014, 11:50 PM, EST

Visible: Yes

Get the thing working. It must work. Thog insist. Thog also like puppies.

Submissions

- Velociraptors on 2013-11-29 19:13:36 UTC
- Velociraptors on 2013-11-29 20:24:07 UTC

A snapshot of your repository will be taken, and your entire group will receive an email notification once your project has been graded. You may only have one submission pending at any given time, but you may resubmit as many times as you like.

Grading Workflow

The grading scripts will do the following:

1. Retrieve your repository from GIT.
2. Recursively scan through the directory src/ at the root level of your submission for *.java files.
3. Compile all of these files (with all course-provided classes in the classpath). Compilation is performed using the Java 1.8 SDK. Do not use features from later versions of java or your submission will not compile!
4. Attempt to run your project as

```
java -cp {classpath} dubstep.Main {arguments}
```

5. Validate the output.

If these steps fail for any reason, your submission will receive a 0 and you will need to resubmit. A log of the testing process will be made available on the submission page so that you may correct any errors that occur.

Hello World!

Now that you're familiar with the submission system, you'll need to submit a simple test case to get yourselves familiar with JSQLParser. You'll need to be able to run the simplest possible SQL query:

```
SELECT * FROM <tablename>;
```

Let's start with the JavaDoc: http://doc.odin.cse.buffalo.edu/jsqlparser/
The first class to look at is CCJSQLParser. This class converts any SQL string into what's called an Abstract Syntax Tree or AST. An AST is a tree of objects representing the different parts of a query. There are a few different types of AST objects, or nodes. For example, Statements represent raw SQL queries. If you look closely, you'll see a Statement interface that quite a few classes inherit from: Select, Update, Insert, Delete and so forth.

Try out the following example:

```
StringReader input = new StringReader("SELECT * FROM R")

JSqlParser parser = new JSqlParser(input)



Statement query = parser.Statement()
```

A Statement alone isn't all that helpful. You need to find out what type it is in order to access the goodies inside. If you look at the Statement Javadoc you'll see a list of all known implementing subclasses. You can use Java's `instanceof` operator to check which type you just got. The one we'll be working with today is Select, so you'd do something like this:

```
if(query instanceof Select){

  Select select = (Select)



  /** Do something with the select operator **/

} else {

  throw new java.sql.SqlException("I can't understand "+query);

}
```

You'll see this pattern of `instanceof` followed by a cast repeated a bunch of times throughout the course projects.

If you look at the javadoc for Select, you'll see that it doesn't have a lot of methods. This is because Selects can appear in a lot of different contexts: Within `UNION`s, as part of a query (e.g., `SELECT (SELECT FOO FROM BAR) FROM BAZ`), or nested in a `FROM` clause. The Select object itself stores information that's only relevant to `SELECT` **statements**.

To get at the chewy center of the Select, we'll need to go one level deeper into the SelectBody. These can be one of two things:

```
SELECT ... FROM ... WHERE ...
```

represented as a PlainSelect. Or, it can be:

```
SELECT ... FROM ... WHERE ... UNION SELECT ... FROM ... WHERE ... UNION ...
```

represented as a Union, which is essentially a list of PlainSelects.
For this assignment, you can assume that you're getting just a PlainSelect and cast it as above (but be sure to check its type first). As you'll see in the JavaDoc, PlainSelect has a ton of methods. Don't be overwhelmed. Each method matches exactly one field of a SELECT statement.

For the purposes of this assignment, **ignore everything in PlainSelect except: getFromItem()**. Here, you'll need to go through one more level of unwrapping. As you'll see in the JavaDoc, a FromItem can be a Table, a nested query (i.e., SubSelect) or a `JOIN ... ON` expression (i.e., SubJoin). Today, we'll be working only with Tables.

Once you have a Table object, you can get the name of the table being referenced.

```
String tableName = Table.getName() /* Should be "R" in the example of "SELECT * FROM R" */
```

You'll find a file named `data/<tableName>.csv` (replacing `<tableName>` with the table name you get from the query).
Your goal for checkpoint 0 is to read a SQL query from standard in (`System.in`), identify the table being referenced in the `FROM` clause of the query, and print the corresponding csv file to standard out (`System.out`).

Checkpoint 1: Basic Queries

- **Overview**: Submit a simple SPJUA query evaluator.

- **Deadline**: Feb 25
- **Grade**: 10% of Project Component
  - 5% Correctness
  - 5% Efficiency

In this project, you will implement a simple SQL query evaluator with support for Select, Project, Join, and Bag Union 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://maven.mimirdb.info/info/mimirdb/jsqlparser/1.0.0/jsqlparser-1.0.0.jar

And documentation for the fork is available at

http://doc.odin.cse.buffalo.edu/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);

System.out.println("$> "); // print a prompt

Statement statement;

while((statement = parser.Statement()) != null){

  // `statement` now has one of the several

  // implementations of the Statement interface

            System.out.println("$> "); // print a prompt after executing each command

}

// 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

}
```

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://maven.mimirdb.info/info/mimirdb/evallib/1.0/evallib-1.0.jar

Documentation for the library is available at

https://github.com/UBOdin/evallib/blob/master/README.md

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 PrimitiveValue 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	**Equivalent `PrimitiveValue`**
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.csv' 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 non-aggregate queries with the following features. Keep in mind that this is only a minimum requirement; you're welcome to support more.

- 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.
    - 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
    - UNION ALL (but not UNION)

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. For this checkpoint, your code should support all of the TABLE*XX* and UNION*XX* queries.
Example NBA Benchmark Queries
    Some very simple queries to get you started. For this checkpoint, your code should support

- nba01.sql
- nba03.sql
- nba05.sql (but not nba05-short.sql)
- nba06.sql (but not nba06-short.sql)

Code Submission

As before, all .java and .scala files your GIT repository will be compiled. Also as before, the class `dubstep.Main` will be called and you will be expected to read SQL from `System.in`. Data will live in the `data` directory; Each `CREATE TABLE` indicates that there is a corresponding file `data/[tablename].csv`

Once your code is ready, it should print out a prompt:

```
$>
```

It should also print out this prompt after completing every SQL operation (After parsing each `CREATE TABLE` and after producing results for each `SELECT`). If you do not print out the prompt, the grader will assume your code is frozen, your trial will time out, and you will receive a 0 for the test.
As before, data will live in the `data` directory and For example, assuming the file `data/R.csv` contains

```
1|1|5

1|2|6

2|3|7
```

The resulting interaction should look like:

```
bash> java -cp your_code.jar:..other jars.. dubstep.Main

$> CREATE TABLE R(A int, B int, C int);

$> SELECT B, C FROM R WHERE A = 1;

1|5

2|6

$>
```

The testing environment is configured with the Sun JDK version 1.8. and the Scala 11.8 library.

Grading

Your code will be subjected to a sequence of test cases and evaluated for both correctness and performance. Incorrect results receive a grade of 0. Correct results receive a grade based on performance. The following thresholds are approximate, but as a rough guideline:

- **5/10 (C)**: Roughly within two orders of magnitude of the reference implementation.
- **7.5/10 (B)**: Roughly within one order of magnitude of the reference implementation.
- **10/10 (A)**: Comparable to the reference implementation.

Specific time thresholds and per-query scores will be reported as part of the execution log. Your overall project grade will be a weighted average of the individual components.
Additionally, there will be a per-query leader-board. To appear on the leader board, your group needs to submit an implementation that is approximately 50% faster than the reference implementation on that query.

Checkpoint 2

- **Overview**: New SQL features, Limited Memory, Faster Performance
- **Deadline**: March 16
- **Grade**: 10% of Project Component
    - 5% Correctness
    - 5% Efficiency

This project follows the same outline as Checkpoint 1. Your code gets SQL queries and is expected to answer them. There are a few key differences:

- Queries may now include a `ORDER BY` clause.
- Queries may now include a `LIMIT` clause.
- Queries may now include aggregate operators, a `GROUP BY` clause, and/or a `HAVING` clause.
- For part of the workload, your program will be re-launched with heavy restrictions on available heap space (see Java's `-XMx` option). You will most likely have insufficient memory for any task that requires O(N)-memory.

Sorting and Grouping Data

Sort is a blocking operator. Before it emits even one row, it needs to see the entire dataset. If you have enough memory to hold the entire input to be sorted, then you can just use Java's built-in Collections.sort method. However, for the memory-restricted part of the workflow, you will likely not have enough memory to keep everything available. In that case, a good option is to use the 2-pass sort algorithm that we discussed in class.

Join Ordering

The order in which you join tables together is **incredibly important**, and can change the runtime of your query by **multiple orders of magnitude**. Picking between different join orderings is incredibly important! However, to do so, you will need statistics about the data, something that won't really be feasible until the next project. Instead, here's a present for those of you paying attention. The tables in each FROM clause are ordered so that you will get our recommended join order by building a *left-deep plan* going in-order of the relation list (something that many of you are doing already), and (for hybrid hash joins) using the left-hand-side relation to build your hash table.

Query Rewriting

In Project 1, you were encouraged to parse SQL into a relational algebra tree. Project 2 is where that design choice begins to pay off. We've discussed expression equivalences in relational algebra, and identified several that are always good (e.g., pushing down selection operators). The reference implementation uses some simple recursion to identify patterns of expressions that can be optimized and rewrite them. For example, if I wanted to define a new HashJoin operator, I might go through and replace every qualifying Selection operator sitting on top of a CrossProduct operator with a HashJoin.

```
if(o instanceof Selection){

 Selection s = (Selection)o;

 if(s.getChild() instanceof CrossProduct){

  CrossProduct prod =

     (CrossProduct)s.getChild();

  Expression join_cond =

     // find a good join condition in

     // the predicate of s.

  Expression rest =

     // the remaining conditions

  return new Selection(

   rest,

   new HashJoin(

    join_cond,

    prod.getLHS(),

    prod.getRHS()

   )

  );

 }
}

return o;
```

The reference implementation has a function similar to this snippet of code, and applies the function to every node in the relational algebra tree.

Because selection can be decomposed, you may find it useful to have a piece of code that can split AndExpressions into a list of conjunctive terms:

```
List<Expression> splitAndClauses(Expression e)

{

  List<Expression> ret =

     new ArrayList<Expression();

  if(e instanceof AndExpression){

    AndExpression a = (AndExpression)e;

    ret.addAll(

      splitAndClauses(a.getLeftExpression())

    );

    ret.addAll(

      splitAndClauses(a.getRightExpression())

    );

  } else {

    ret.add(e);

  }

}
```

Grading Workflow

As before, the class `dubstep.Main` will be invoked and a stream of **semicolon-delimited** queries will be printed to System.in (one after after each time you print out a prompt)

All .java / .scala files in your repository will be compiled (and linked against JSQLParser). Your code will be subjected to a sequence of test cases and evaluated on speed and correctness. Note that unlike Project 1, you will neither receive a warning about, nor partial credit for out-of-order query results if the outermost query includes an ORDER BY clause. For this checkpoint, we will use predominantly queries chosen from the TPC-H benchmark workload.

Phase 1 (big queries) will be graded on a TPC-H SF 1 dataset (1 GB of raw text data). Phase 2 (limited memory) will be graded on either a TPC-H SF 1 or SF 0.2 (200 MB of raw text data). Grades are assigned based on per-query thresholds:

- **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 completes the test query workload within the timeout period, and produces the correct output.
- **7.5/10 (B)**: Your submission completes the test query workload notably slower than the reference implementation, and produces the correct output.
- **10/10 (A)**: Your submission runs the test query within a factor of 2 of the reference implementation, and produces the correct output.

Unlike before, your code will be given arguments. During the initial phase of the workload, your code will be launched with `--in-mem` as one of its arguments. During the memory-restricted phase of the workload, your code will be launched with `--on-disk` as one of its arguments. You may use the `data/` directory to store temporary files.

For example (red text is entered by the user/grader):

```
bash> ls data

R.dat

S.dat

T.dat

bash> cat data/R.dat

1|1|5

1|2|6

2|3|7
```

```
bash> cat data/S.dat

1|2|6

3|3|2

3|5|2

bash> find {code root directory} -name \*.java -print > compile.list

bash> javac -cp {libs location}/commons-csv-1.5.jar:{libs location}/evallib-1.0.jar:{libs location}/jsqlparser-1
.0.0.jar -d {compiled directory name} @compile.list

bash> java -cp {compiled directory name}/src/:{libs location}/commons-csv-1.5.jar:{libs location}/evallib-1.
0.jar:{libs location}/jsqlparser-1.0.0.jar edu.buffalo.www.cse4562.Main - --in-mem

$> CREATE TABLE R(A int, B int, C int);

$> CREATE TABLE S(D int, E int, F int);

$> SELECT B, C FROM R WHERE A = 1;

1|5

2|6

$> SELECT A, E FROM R, S WHERE R.A = S.D;

1|2

1|2
```

For this project, we will issue a sequence of queries to your program and time your performance. A randomly chosen subset of these queries will be checked for correctness. Producing an incorrect answer on any query will result in a 0.

- **Overview**: Add a pre-processing phase to your system.
- **Deadline**: May 3
- **Grade**: 15% of Project Component
  - 7% Correctness
  - 8% Efficiency

Once again, we will be tightening performance constraints. You will be expected to complete queries in seconds, rather than tens of seconds as before. This time however, you will be given a few minutes alone with the data before we start timing you.

Concretely, you will be given a period of up to 5 minutes that we'll call the Load Phase. During the load phase, you will have access to the data, as well as a database directory that will not be erased in between runs of your application. Example uses for this time include building indexes or gathering statistics about the data for use in cost-based estimation.

Additionally, CREATE TABLE statements will be annotated with PRIMARY KEY and INDEX attributes. You may also hardcode index selections for the TPC-H benchmark based on your own experimentation.

Interface

Your code will be evaluated in nearly the same way as Projects 1 and 2. Your code will be presented with a 1000MB (SF 1) TPC-H dataset. You will get a cumulative 5 minutes to process all of the CREATE TABLE statements; This time will not count towards your overall time. Taking more than 5 minutes will result in a 0 grade for the submission.

The first phase will occur immediately after processing CREATE TABLE statements. You will receive a series of random queries drawn from the TPC-H benchmark, and should produce responses as quickly as possible.

After processing queries for the first phase, your code will be terminated. Your code will be restarted **in the same directory**. In this phase, you will receive a similar series of random queries drawn from the TPC-H benchmark. However, unlike phase 1, there will no **no CREATE TABLE** statements at all. You are expected to preserve any and all state associated with the created tables on your own.

Grading

Your code will be subjected to a sequence of test cases and evaluated on speed and correctness.

- **0/15 (F)**: Your submission does not compile, does not produce correct output, or fails in some other way. Resubmission is highly encouraged.
- **6/15 (C)**: Your submission runs the test query workload, not taking more than 60 seconds per query, but taking more than 200 seconds to process the overall workload.
- **12/15 (B)**: Your submission runs the test query workload, taking between 20 and 200 seconds to process it.
- **15/15 (A)**: Your submission runs the test query, completing it in under 20 seconds.

Checkpoint 4

- **Overview**: Support lightweight updates.
- **Deadline**: May 20
- **Grade**: 10% of Project Component
    - 5% Correctness
    - 5% Efficiency

Short version: Support queries of the following three forms:

```
INSERT INTO R (A, B, C, ...) VALUES (1, 2, 3, 4, ...);

DELETE FROM R WHERE A < 3, ...

UPDATE R SET A = A+1, B = 2 WHERE C = 3;
```

Specifically, your code should support:

INSERT

You only need to support `INSERT ... VALUES (...)` style inserts. There will not be any queries of any other form (i.e., nothing like `INSERT ... SELECT ...` or `INSERT OR REPLACE ...`)

DELETE

Any selection predicate valid in a previous checkpoint is fair game for `DELETE`.

UPDATE

Any selection predicate valid in a previous checkpoint is fair game for `DELETE`. Update expressions **may** include non-primitive value expressions.

Updates **do not** need to be persisted across database reboots, but should be reflected in query results.

Interface

Your code will be evaluated in nearly the same way as Projects 1 - 3. Your code will be presented with a 1000MB (SF 1) TPC-H dataset. You will get a cumulative 10 minutes to process all of the CREATE TABLE statements; This time will not count towards your overall time. Taking more than 10 minutes will result in a 0 grade for the submission.

Queries will be interleaved with a sequence of randomly generated update commands. There will be 3 phases:

1. Insert and queries only (30% of grade).
2. Inserts, deletes and queries (30% of grade).
3. Inserts, deletes, updates, and queries (40% of grade).

	Total Time for Max (Half) Score	
Phase	Updates	Queries
1	0.5 s (2 s)	200 s (500 s)
2	0.5 s (2 s)	300 s (600 s)
3	1500 s (3000 s)	400 s (800 s)