# CS44800, Spring 2016 - Project 4
# SQL Parsing and Query Optimization

- Due: Monday 18 April, 2016, 11:59PM. Submit using Blackboard.
- (There will be a 10% penalty for each late day. After 4 late days, the project will not be accepted.)

---

**Notes:**

- This project should be carried out in groups of two. Choose a partner as soon as possible. Each group will get the same grade, so make sure that every member backs up his/her teammate (no excuses will be accepted).
  If you submit individually, you have to have an approval from the TA (Spencer Pearson) before the 7th of April 2016 and you will be responsible for the entire project (i.e., no optional parts), however, a full credit will be from 80 instead of 100 for those who have acceptable excuses to work individually.

## MiniSQL Parser

In this project, you do not have to write the parser. Instead, a complete parser and type checker for the *MiniSQL* language is provided with the skeleton code. This small subset of SQL includes the basic commands CREATE, DROP, INSERT, SELECT, UPDATE, DELETE, DESCRIBE, and EXPLAIN. Supported data types include INTEGER, FLOAT, and STRING (notice the slight deviations from the SQL standard). The following (incomplete) list illustrates what is not included in the language:

- Support for NULL values
- Complex expressions / parentheses, for example, WHERE a = b + 1
- Aliasing (i.e. FROM Employees E), this means column names should be unique
- DISTINCT and ORDER BY
- Aggregates, GROUP BY, HAVING, etc.

Once a MiniSQL statement is parsed, the *abstract syntax tree* (AST) is passed to the Optimizer, which in turn dispatches the query to the corresponding class implementing the Plan interface. In this project, you will implement several of these plan classes, using the provided parser and system Catalog.

---

# PART 1: CREATE and DROP INDEX

Your first task is to implement the CreateIndex and DropIndex commands, allowing you to build and destroy hash indexes on tables. Your implementation of these and the remaining Plan classes must meet the following general requirements:

1. You must **validate all query input**.
2. For example, you should not create an index if the file name already exists. Please review (and call) the appropriate methods in the provided class QueryCheck to fulfill this requirement.
3. **Execute the query** using the components we have developed throughout the semester.
4. If the query **affects the system catalogs** (i.e. create/drop statements), then call the appropriate method(s) in Catalog to maintain them.
5. Each query should **print a one-line message** at the end, such as "Table Created" or "1 row inserted" -- or anything else you would find appropriate.

Some useful hints and tips:

- You may want to use CreateTable and DropTable as a reference.
- (i.e. these are provided to demonstrate how to use the parser)
- Don't forget that CREATE INDEX should actually build the hash index!
- (i.e. don't just rename the word "table" in CreateTable to "index")

---

# Part 2: INSERT, UPDATE, DELETE, and SELECT

Your next task is to implement the Insert and Select classes, allowing you to create and query actual data. Remember to fulfill the general requirements listed in part one. In addition, you need to implement the Update, Delete.

For Select, you will implement a basic query optimizer. The parser will give you an array of table names to select from, an array of (unique) column names to project, and an array of selection predicates (in **conjunctive normal form**). The basic plan is to use FileScans and SimpleJoins for all the tables, add a Selection for each conjunct, and have one Projection at the root of the Iterator tree.
For example:
*Given the tables T1(a, b) and T2(c, d) and the following query*:

```
EXPLAIN SELECT d, a FROM T1, T2 WHERE a = c and b = d or a = 5;
```

*The default (naive) execution plan is as follows*:

```
Projection : {3}, {0}
 Selection : b = d OR a = 5
  Selection : a = c OR a = 5
```

```
SimpleJoin : (cross)
  FileScan : T2
  FileScan : T1
```

(*Note how the conditions of the WHERE clause change into the predicates*)

You need to implement the following optimizations:

1. **Pushing Selections:** If the predicates of a selection involve only the attributes of one table, you should execute the selection before the join.
2. **Join Ordering:** You should maintain catalog statistics (i.e. record counts) and use this information to determine what order to join the tables.

Some more useful hints and tips:

- Think carefully about any side effects of INSERT and DELETE statements, and how they affect the catalog counts. Also, what if the table you're inserting/deleting into/from has indexes?
- The main goal of Select's constructor is to create an Iterator query tree. (i.e. all you need to do in execute() is call iter.explain() or iter.execute())

For testing purpose, your execution plan will be printed before the returned results.

---

# Getting Started

Here is Project documentation, and Project skeleton code.
Note that this code is a complete starting point for the project, i.e. the bufmgr, heap, index, and relop packages are provided for you (in jar files). The framework classes may be slightly different than the ones used in previous projects.
Running and testing this project will be quite different from the others. Instead of using an automated test driver, you will run the provided command-line utility, Msql, which resembles the behavior of SQL*Plus. Several test queries are provided with the skeleton code, but more queries will be tested at the time of grading. Use the STATS command to view performance counters. The Msql program can receive input from the command line, or from a file. In the latter case, you need to provide the file name as an input parameter. For example:

```
java -classpath bin global.Msql mytest.sql
```

If you use an IDE, feel free to set up your run configuration to accept that input parameter.

---

# Turnin

Only one of the group members should submit the work via Blackboard. Include with your code the Makefile, Readme (which lists group members, how you do the optimization, new features that you want us to know, and roles of each member, i.e. who did what). All files need to be zipped in a file named: **your_career_login1_your_career_login2_qe.zip**.
We should be able to compile/run your program using make on a CS department Unix machine. The directory structure of your zip file should be identical to the directory structure of the provided zip file (i.e., having the directory src, the Makefile, ...), except the top-level name (should be your career login above). Your grade may be deduced 5% off if you don't follow this.