# **CS44800 – Project 3 Query Optimization**

* Due: November 21, 11:50PM. Submit using Blackboard.

**MiniSQL Parser**

In this project, you will be implementing the Query Optimization component of a real, functioning database management system. This system, *Minibase*, is intended primarily for educational purposes and therefore has relatively limited functionality and features compared to commercial database systems, but otherwise functions as a real Database Management System does. The design architecture of Minibase is a layered approach, with different layers of abstraction providing the functionality to manage files on disk and in-memory, perform low-level operations on data, execute relational operators, and parse, optimize, and execute SQL queries. We will be focusing on this last layer for this project – specifically, the optimization section. Full Javadoc for Minibase is provided [here](http://www.cs.purdue.edu/homes/pearso29/MinibaseJavadoc).

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   
  *e.g. WHERE a = b + 1*
* Aliasing (e.g. 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](https://www.cs.purdue.edu/homes/pearso29/MinibaseJavadoc/query/Optimizer.html), which in turn dispatches the query to the corresponding class implementing the [Plan](https://www.cs.purdue.edu/homes/pearso29/MinibaseJavadoc/query/Plan.html) interface. In this project, you will implement the [Select](http://www.cs.purdue.edu/homes/pearso29/MinibaseJavadoc/query/Select.html) plan, using the provided [parser](https://www.cs.purdue.edu/homes/pearso29/MinibaseJavadoc/parser/package-summary.html) and system [Catalog](https://www.cs.purdue.edu/homes/pearso29/MinibaseJavadoc/query/Catalog.html). This is similar to how you built queries in the previous project, although here you will have to handle more complex queries and be able to optimize them based on some known information.

**Tasks**

You have been provided with a mostly complete implementation of the Minibase DBMS – Create/Delete Table, Create/Delete Index, Insert, Update, Delete, and Describe have been provided for you. You only have to complete the [Select](https://www.cs.purdue.edu/homes/pearso29/MinibaseJavadoc/query/Select.html) class (in the query package of the provided code), allowing you to query actual data.

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**). As in the previous project, CNF for predicates implies that every Selection operator will OR the predicates it is evaluating, and implementing AND statements requires separate Selection operators. The parsed [AST](http://www.cs.purdue.edu/homes/pearso29/MinibaseJavadoc/parser/AST_Select.html) you are provided has handled interpreting the input predicates from an SQL query into the correct set of predicates to use in your Selection operators.

The first step is to validate the query. This can be done by calling the methods of the [QueryCheck](http://www.cs.purdue.edu/homes/pearso29/MinibaseJavadoc/query/QueryCheck.html) class. All the methods of QueryCheck will throw an exception if the validation fails. You should not attempt to handle this exception within the Select constructor, simply throw this exception and Minibase will handle it appropriately. You need to validate that the tables provided exist, and that the columns and predicates are valid. Study the documentation for the QueryCheck class to see which methods you should use – the source code for QueryCheck is also given to you.

Next, you must construct the actual Query Plan. This will be done in a similar fashion to the previous project. Both of these steps – validation and the creation of the plan tree – should be done in the constructor for the Select class.

The basic plan is to use [FileScan](https://www.cs.purdue.edu/homes/pearso29/MinibaseJavadoc/relop/FileScan.html)s and [SimpleJoin](https://www.cs.purdue.edu/homes/pearso29/MinibaseJavadoc/relop/SimpleJoin.html)s for all the tables, add a Selection for each conjunct, and have one [Projection](https://www.cs.purdue.edu/homes/pearso29/MinibaseJavadoc/relop/Projection.html) at the root of the [Iterator](https://www.cs.purdue.edu/homes/pearso29/MinibaseJavadoc/relop/Iterator.html) tree. Study the provided documentation for information on how to use these relational operators (Note: To perform a cross product, create a SimpleJoin object and provide an empty Predicate array for the ‘preds’ parameter).

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*)

Although the naïve plan will produce the correct output, it is not necessarily efficient for all queries. You need to implement the following optimizations as you build your plan:

1. **Pushing Selections:** If the predicates of a selection involve only the attributes of one relation, you should execute the selection before any additional joins.
2. **Use Joins instead of Cross Products where possible**:For every join, if predicates exist that are suitable to use for the join, you should pick one of those suitable predicates and use it for the join instead of using a cross product.
3. **Join Ordering:** You should use catalog statistics (e.g. record counts) and use this information to determine what order to join the tables.

You should use the available information to determine the optimal ordering. Metadata used for estimating query costs and decided on an optimal ordering is stored in the *catalog* of a relational DBMS. For Minibase, this catalog only tracks the number of records in each relation. A catalog is implemented as actual relational database tables, which can be accessed as you would any other table. Study the [Catalog](http://www.cs.purdue.edu/homes/pearso29/MinibaseJavadoc/query/Catalog.html) documentation and source code to determine which catalog tables you need to access to get the requested information.

To actually access the data in a table in Minibase, you should make use of the [HeapFile](http://www.cs.purdue.edu/homes/pearso29/MinibaseJavadoc/heap/HeapFile.html) object (an in-memory Java object used to access and manipulate on-disk tables) corresponding to the table you wish to read. Each row in a table in Minibase is identified by an [RID](http://www.cs.purdue.edu/homes/pearso29/MinibaseJavadoc/global/RID.html), specifying where in the table the row is actually stored. HeapFile.selectRecord(RID rid) will give you the byte[] representation of the row of the table that corresponds to that RID. You can then construct an instance of the [Tuple](http://www.cs.purdue.edu/homes/pearso29/MinibaseJavadoc/relop/Tuple.html) class using this byte[] and the [Schema](http://www.cs.purdue.edu/homes/pearso29/MinibaseJavadoc/relop/Schema.html) of the catalog table you’re reading from in order to create a Tuple object that you can work with and read the data in specific fields.

Once you have used the Catalog table to retrieve information about the tables involved in the query and have constructed a set of FileScan objects as the base operators in your query tree, then you can start the process of optimization. The general algorithm for optimizing queries is as follows:

First, push *all* WHERE Predicate clauses that can be evaluated under a currently available Schema in the plan tree. For example, for the query σa=b(σa<5(R ⨝R.a = S.b S)), we would start with two separate Iterator plan trees consisting of FileScans for R and S. However, the selection σa<5 can be evaluated on R – it does not need to wait to be processed after the join. Therefore, we should push the selection down on to the R tree, leaving us with the tree with the FileScan on S and another tree with the Selection followed by the FileScan on R, and one so-far unprocessed Predicate σa=b. You may use the QueryCheck class methods to determine whether a Predicate is valid on a given Schema. Remember, since these are provided to you in CNF you might have multiple Predicates in one clause. You cannot separate a clause, so all Predicates in that clause must be valid under the Schema you want to push them to.

Second, you need to determine what the first join you perform in your query should be. If the only information we know about relations is how many records are in them, then the most efficient way to order joins is by joining the smallest relations first. Use the information from the Catalog to determine which join is most efficient to perform first.

Finally, check the remaining set of Predicate clauses that have not been used in your query already. If there is any Predicate clause that is valid on the joined Schema and contains Predicates that are suitable to use in a join (e.g. R.a = S.b where R and S are the relations involved in the join – other operators are suitable as well besides ‘=’), then you should use that clause as a Predicate for the SimpleJoin Iterator you will construct. If there are no suitable Predicates clauses, then you should simply do a Cross Product. Join the two plan trees together into one plan tree with a SimpleJoin operator at its root. For the purposes of estimating further join orderings, you can consider this plan tree to have a number of tuples equal to the product of its joined relations.

Repeat this process until all relations have been joined and all Predicates have been used. Finally, if necessary, add the Projection Iterator to the top of your tree.

At the end of these optimizations you should have a plan that uses joins instead of cross products wherever possible, all selections are pushed immediately after the minimal joined relation that they can be processed on, and some reasonable attempt at join ordering using the information available to you has been made. Remember that your implementation must be capable of handling an arbitrary number of joins and Predicates, not just two or three joins.

Given the above example, the optimized plan should look as follows:

Projection : {3}, {0}   
 Selection : b = d OR a = 5  
 SimpleJoin : (a = c OR a = 5)   
 FileScan : T2   
 FileScan : T1

Some more useful hints and tips:

* 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.execute() or, if the EXPLAIN keyword was used, iter.explain()**. If you attempt to create and optimize the plan in Select.execute() you will lose points).
* Some predicates cannot be pushed immediately above the base table scans. However, they can be pushed below some intermediate joins. You should make sure that your optimization handles these cases.

To implement the execute() method in the Select class, all you need to do is to determine whether this query has been designated an EXPLAIN query (print the plan tree without executing it) or if it should be executed. The parsed Select AST contains the information as to whether or not the query is a normal execution query or is just explaining the plan.

If the query is an EXPLAIN query, you should simply call explain(0) on the Iterator you created in the constructor. If the query is not an EXPLAIN query, you should call execute() on the iterator instead..

**Running Queries**

You will use the provided command-line utility, [Msql](https://www.cs.purdue.edu/homes/pearso29/MinibaseJavadoc/global/Msql.html), 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.

Although different data will be used, the queries for testing will be loaded from a file with the same name as the test query file we provide to you. The provided sample makefile should work during grading – you should only have to edit it if you use a different directory structure than expected.

**Turnin**

Submit your Select.java file on Blackboard. You should not submit any other files, as we will only be using your Select.java file in grading. You should not rely on any changes you’ve made in other files for your implementation, as they will not be available during grading.

We should be able to compile/run your program using make on a CS department Unix machine.