.

DM556 - Mandatory Project: Part 3
Query Evaluation

And now, the project you've all been waiting for! The one where everything comes together and runs for a long time without crashing, right? Well one thing is for certain: you should thoroughly understand sections 12.1.1, and 12.4 (pages 395 to 397, and 404 to 405) of the textbook before jumping into this project.

Due: Tuesday, May 9, 12:00 (Noon)

Note:

- This project will be carried out in groups of at most three persons. If you change group from the project part two, this must be emailed to the teaching assistant by Tue, April 18.
- Read the project description carefully, and START EARLY!

MiniSQL Parser

Relax; you don't have to write the parser! On the contrary, 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`. Requires external sorting, which isn't implemented.
- 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: SELECT

Your next task is to implement the Select class, allowing you to query actual data.

For this, 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). You must validate all query input. Please review (and call) the appropriate methods in the provided class QueryCheck to fulfill this requirement. The default 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),T2(c, d) and T3(e,f) and the following query:

```
EXPLAIN SELECT e,a, d FROM T1, T2,T3 WHERE a = c and b = d and d = c or a = 5;
```

The default (naive) execution plan is as follows:

```
Projection : {4}, {0}, {3}
  Selection : d = c OR a = 5
    Selection : b = d OR a = 5
      Selection : a = c OR a = 5
        SimpleJoin : (cross)
          SimpleJoin : (cross)
            FileScan : T1
            FileScan : T2
          FileScan : T3
```

Some things to note:

- Your implementation should be in the Select.java class.
- The class Optimizer distributes the queries, so each class should only take care of one thing.
- INSERT is already implemented, so you should not have a problem producing a test that checks your implementation.
- You should also take a look at the files CreateTable.java and DropTable.java. This is an implementation of the CREATE TABLE and DROP TABLE Msql queries.
- 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())

Part 2: Extensions

From the following list, you must choose two extensions to do and one of them must be chosen from 4 – 6 . More detailed descriptions can be found further below.

1. Print information from the System Catalog.
2. CREATE INDEX and DROP INDEX.

3. UPDATE and DELETE
4. Using catalog information
5. Predicate Matching
6. Pushing down predicates.

Regardless of which tasks you choose, your implementation must meet the following general requirements:

- You must validate all query input. 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.
- Execute the query using the other components of the DBMS.
- If the query affects the system catalogs (i.e. create/drop statements), then call the appropriate method(s) in Catalog to maintain them.
- Each query should print a one-line message at the end, such as "Index Created" – or anything else (appropriate) you would rather say.

Print information from the System Catalog

Using the keyword `TEST` in the `Msql.java` program, you should implement a detailed print of what the system catalog contains.

This should at least include table names and their schema, index's, etc. All of the information should be printed in an easy to read format.

You should start in the `Catalog.java` file

CREATE INDEX and DROP INDEX

This task is to implement the CreateIndex and DropIndex commands, allowing you to build and destroy hash indexes on tables. 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").

Your implementation should use the template files `CreateIndex.java` and `DropIndex.java`

UPDATE and DELETE

This task consist of implementing the two query types UPDATE and DELETE. As the case with INSERT, you should handle both the table and the index if one such exist.

- You should take a good look at Insert.java for inspiration.
- Tou must in your report include as an appendix a test showing the functionality of these two statements.

Your implementation should use the template files `Update.java` and `Delete.java`

Using catalog information

Although you could spend several years on this sort of work, one reasonable enhancement is to maintain catalog statistics (i.e. cardinality) and use this information to determine what order to join the tables. This task spans over different types of challenges, such as changing the catalog and the INSERT procedure.

- You must in your report include as an appendix a test showing the functionality using the EXPLAIN keyword.

Predicate Matching

Another good enhancement to the naive query optimization in part one is to use KeyScans (if you did extension 2) and/or HashJoins where possible. You should implement the use of HashJoin where possible (i.e. where two tables are joined by a predicate with equality).

- You must in your report include as an appendix a test showing the functionality using the EXPLAIN keyword.
- The result should also be correct ;)

Pushing down predicates

If possible selection predicates should be pushed down the tree (preferably below joins) to improve query performance. This is the content of this task if you choose to do this.

Getting Started

As usual you should download the code from the course homepage.

You should note that the DBMS creates the plan and executes it right after. Your constructors should therefore prepare everything (initialize variables and store information) for the execute() method.

What should I turn in (and where)

Please follow the same submission instructions as in the previous projects.