Query Optimization: Join Optimization

1 Introduction

During this semester, we have talked a lot about the dierent components that comprise a DBMS, especially those that relate to query processing. In this project, you will have an opportunity to extend the functionality of a database to implement join algorithms you have seen inside of a toy DBMS called SimpleDB.

SimpleDB is database engine written in Java. This system supports Cartesian product, select, and project operations. As we know from class, we can then achieve joins through arranging these operations in a specic way. However, the code could run signicantly faster if we create a specic join operation. In this project, you will write and extend the query processor to support joins. The system is intended for pedagogical use only. The code is clean and compact. The APIs are straightforward. The learning curve is relatively small. Consequently, the system is intentionally bare-bones. It implements only a small fraction of SQL and JDBC, and does little or no error checking.

This part of the project focuses on query execution and has three parts

1. Writing two join operations
2. Computing costs for each join operation
3. Analyzing the effects of optimization

All modications will be made in the simpledb.query package.

2 Setting Up SimpleDB

In order to set up SimpleDB, download `/course/cs1270/pub/optimization/stencil.tgz` and extract the SimpleDB files.

3 Background on SimpleDB

SimpleDB is a pedagogical database managing system that can locally set up a server that hosts the database, modify the data, and perform queries on the data. For the purpose of this assignment, we will only be focusing on how to optimize SimpleDBs query execution. Lets first think about how SimpleDB goes from a user inputted SQL query to a result containing all tuples that correspond to the query. When SimpleDB parses SQL language, it reconstructs the query in tree form, where each node in the tree represents an operation. For example, for the simple query `SELECT id FROM student WHERE grade = A` produces the tree:

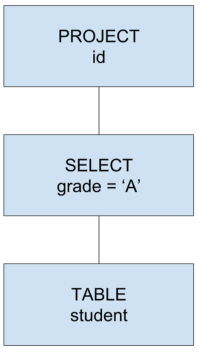

Figure 1: Simple query tree

For a more complex query, a much deeper tree is constructed. For example, `SELECT id FROM student JOIN enroll ON student.id = enroll.student_id WHERE enroll.semester = Fall` produces:

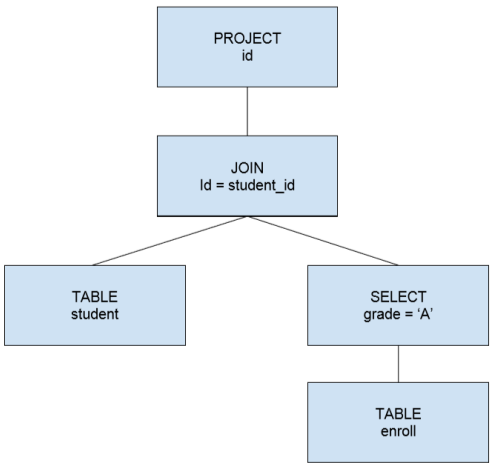

Figure 2: Simple query tree

As you can see above, every SQL operation (`SELECT`, `JOIN`, `AGGREGATE`, etc) can be represented by a node. In addition, every leaf node in the tree represents a table. To actually compute tuples from this tree, each node (or operation) is designed as a iterator (technically called a database cursor) that gets the next tuple that satises the operations condition. In particular, every node has a next() method that returns the next tuple. Thus, to get a particular tuple that satises the query, SimpleDB calls `next()` on the root node, which calls `next()` on its children, until execution reaches a leaf node (table). When it reaches a table, `next()` simply returns the next tuple in the relation.

Lets take a look at the query tree in Fig. 1 as an example. The root `PROJECT` node will call `next()`, which calls the `SELECT` nodes `next()`, which calls the student tables `next()` function until the selection condition is satised. In other words, the `SELECT` node repeatedly gets tuples one by one from the TABLE node until tuple.grade = A.

Each Plan class calculates three metrics for a operation:

1. of block accesses to construct the output
2. of records in the output
3. of distinct eld values in the output

4 The dataset

We have created for you a mini version of the Yelp database you used in your rst project. The schema is as follows:

```
business(id, name, city, state, reviews)
review(bid, uid, stars)
user(id, name, reviews, useful, funny, cool)
```

In review, `bid` corresponds to business ID and `uid` corresponds to user ID. The attribute `reviews` in both the `business` and `user` relations indicate the number of reviews and number of reviews written, respectively.

To import the database, first start the SimpleDB server with `ant server`. In a separate terminal on the same machine, run `ant createYelpDB`. This will create the relevant files in `/home/USERNAME/YelpSimpleDB`.

To make sure everything is working properly, try running the `SQLInterpreter` client demo with `ant sql`. Note that the server must be running at the same time. Enter some SQL statements and see what happens, but remember that SimpleDB only supports a subset of SQL. What happens when you try to execute an SQL statement that SimpleDB does

not support? For instance, just hitting Enter triggers an invalid syntax exception and a stack trace is printed to terminal. If you suspect something went wrong in setup or if youre getting a java.lang.OutOfMemoryError, simply delete `/YelpSimpleDB` and re-execute the import steps outlined above

5 Join Algorithms

In lecture, youve learned that there are different ways to join two tables. Specically, Stan talked about how different join algorithms can drastically improve in efficiency. Your goal is to implement these two join algorithms and analyze their effect on cost:

1. `NestedLoopJoinScan.java`
2. `BlockNestedLoopJoinScan.java`

These and the corresponding `Plan` classes can be found in `/src/simpledb/query`.

Refer to the lecture slides for the pseudocode for each join. Remember, `scan` classes simply call `next()`, so the trick is to think about how to apply each algorithm as an iterator.

Hints for `NestedLoopJoin`:

1. In the constructor, we can see that a call to `beforeFirst()` sets the pointers for both `scan1` and `scan2` before the rst tuple and then advances the pointer for `scan1` to the first tuple, meaning we now have the rst tuple for `scan1` but have not yet retrieved the rst tuple for `scan2`. How does this determine which should be the outer relation and which should be the inner?

Hints for `BlockNestedLoopJoin`:

1. Check out `getNextInnerJoin()` - we can see that after determining which blocks to pass in, we run a `NestedLoopJoinScan` on those blocks. What does that tell us about how the next() function should behave and under what conditions we should be calling `getNextInnerJoin()`?
2. In the support code for `BlockNestedLoopJoinPlan`, we have modeled it such that the bigger relation is always passed in as `TableInfo ti1` and the smaller relation is always passed in as `TableInfo ti2`. Which should be the outer relation and which should be the inner relation in order to minimize disk accesses?
3. `filesize` is represented in terms of blocks - how can you use this information to determine when to stop?
4. a `ChunkScan` is essentially just a block of tuples. No need to worry too much about this.

To test each join algorithm, you can run SQL queries using SimpleDBs `SQLInterpreter`. First, set the interpreter to use the right join: navigate to `SQLInterpreter.java` in the client package and modify the `JOIN TYPE` eld. 1 corresponds to nested loop join and 2 corresponds to block nested loop join. Then, run a SQL query containing a join in order to test your implementation.

NOTE: SimpleDBs SQL interpreter is very limited, so it does not support many SQL functions. Please run a join using the form: `SELECT <attributes> FROM <table1> JOIN <table2> ON <predicate>`.

6 Calculating Cost

Given multiple ways to evaluate an operation, the database system must be able to gure out which one is most optimal. Thus, each operation implementation must provide a cost. In class and in this project, we dene cost in terms of the number of disk block accesses.

For each operation you implemented above, ll out the corresponding `Plan` class to calculate the cost of that implementation. Each of these classes implement a `Plan.java` interface, which outlines these methods:

1. `int blocksAccessed()`
2. `int recordsOutput()`
3. `int distinctValues(String fldname)`

For each of the join algorithms you implemented, fill out the `int blocksAccessed()` method by returning the worst-case cost of that join algorithm. When you run either our test script or queries through the `SQLInterpreter`, the number of blocks accessed will print to the terminal window running your server.

7 Testing

Weve provided a tester to make sure your join algorithms are correct. You can run the tester by running ant test. Note that the server must also be running. This will run two joins using both your nested loop and block nested loop join implementations.

The tester checks to make sure the number of results are equal between our solution and your implementations results, and that the ordering of the tuples between our solution and your implementation results is the same. As per the hints and support code provided, we intend to guide you towards a straightforward solution. Any deviation from the algorithms outlined in the lecture slides or unconventional iteration through the scans may mean that your results will not match our solution.

8 Analysis

In your README, answer the following questions:

1. What are the differences between nested loop join and block nested loop join in terms of cost? Why is there such a noticeable difference?

2. Another way of optimizing queries is from reordering the query tree. Note in figure 2, the SELECT operator is close to the bottom of the tree. What would happen if we placed that operator closer to the root of the tree? To help you understand this, weve provided two different ways SimpleDB orders the query tree:

   `HeuristicQueryPlanner` and `SelectQueryPlanner`. The former creates the tree such that selection occurs at the bottom, whereas the latter places selection towards the root (you dont need to understand the specics of these classes). Try changing SimpleDBs query planning strategy by replacing line 105 in `server/SimpleDB.java`. Compare how the two strategies change the total cost of the query, and explain why we see this effect.

3. How could you integrate your B+Tree implementation into a third type of join? (Hint: read the lecture slides!) Describe qualitatively how this type of join would be implemented. In this case, is a B+Tree or B-Tree indexing scheme preferable? Which one is better and why?

9 Handin

Please hand in the following:

1. The entire simpledb subdirectory that contains all of the source code

2. A README le with answers to the analysis questions

To hand in the project, run the following command: `/course/cs127/bin/cs127_handin optimization`
Good luck!