

CMSC424: Programming Project

Due: November 12, 2009

This is a group project, and you should use the same groups as for the class project. There are two parts to the programming project. The first one involves generating and analyzing the query plans that Oracle generates. The second part asks you to implement some query processing operators in a Toy Relational Database Management System.

Part 1 (3 pts)

Oracle has following commands to help with analyzing the plans, and tuning the execution. Other database systems have their own set of commands.

- **set timing on;** : If this is set, the time taken to execute a query is output at the end.
- **set autotrace on;** : The plan used to execute the query is shown at the end of execution.
- **Oracle Hints:** SQLPlus also allows you to provide “hints” to the optimizer on what kinds of algorithms to use, what not to use etc. This can be used to change the plan to something else, if you think you know better than Oracle optimizer (not uncommon in practice). Link to detailed usage is on the course webpage. Here are a couple of examples you might want to try out:

- **Changing join method:** You can use hints to change the join method used by the optimizer.

```
select /*+ use_nl(f1, f2) */ count(*)
from friends f1, friends f2
where f1.userid2 = f2.userid1;
```

Without the *use_nl* there¹, this query would probably use hash join or merge join. *use_nl* forces it to use nested loops join for joining those two tables. Change *use_nl* to *use_merge* or *use_hash* to force Oracle to use other plans.

- **Enforcing join order:** By adding a clause called “ordered”, you can force the optimizer to use the same join order as specified in the from clauses.
- **Using index or not using index:** Similarly, say I had an index on results:

```
create index users_userid_idx on users (userid);
```

```
select /*+ index(users) */ *
from users
where userid = 'user0';
```

..forces the optimizer to use an index on the *results* table to execute this query, whereas:

```
select /*+ no_index(users) */ *
from users
where userid = 'user0';
```

..forces it to not use the index. By default, it would have probably used an index if one existed.

¹The syntax is very specific. Hints must appear as shown here.

Your tasks:

1. Oracle (as with all database systems) stores the “metadata” information in some special tables. For example, “all_tables” stores the names of all tables, and bunch of additional information. A second table “all_tab_columns” contains the information about all the table attributes, and is used to answer queries such as “describe groups;”. Write a query that returns the same answer as the “describe groups;” query using the second table. Your query must return similar answer, but not identical answer – that would be somewhat harder to manage. (Hint: Use “describe” to see the attributes of second table first).
2. Use the autotrace feature to find the plan for the following two queries and draw it (as shown in Figure 14.5). Note that the autotrace feature actually only prints out the operators; but the indentation levels, and also the order in which operators are printed out, can be used to deduce the query plan. Also, note down the time it took to execute the query (it will probably be too small to be measurable).

(a) **Query 1:** Draw the query plan for this query.

```
select u1.name, u2.name
from users u1, friends f, users u2
where u1.userid = f.userid1 and f.userid2 = u2.userid
      and extract(month from u2.birthdate) = 8
      and extract(day from u2.birthdate) between 16 and 30;
```

(b) **Query 2:** Draw the query plan for this query.

```
select * from users where userid NOT IN (select userid from status);
```

(c) **Query 3:** Draw the query plan for this (somewhat complicated) query, and explain it. More specifically, describe how the query is being executed, and what operators Oracle is using for answering this query.

```
with temp as (
    select groups.groupid, name, count(userid) as num_members
    from groups, members
    where groups.groupid = members.groupid
    group by groups.groupid, name
)
select name
from temp
where num_members = (select max(num_members) from temp);
```

3. Force **Query 1** from above to use a nested loops join instead of whichever join it uses, draw the plan and note down the time it took in this case (use the *use_nl* hint).
4. **Query 2** can be made more efficient through use of *merge_aj* (or *hash_aj*) hint. Briefly describe what the hint does, use it for the above query and draw that plan (NOTE: it is possible Oracle already does what this hint tries to tell Oracle to do, in which only explain the purpose of the hint).

It is fine if you are not successful in making Oracle do what you want as long as you used the hints correctly. Oracle may ignore the hints in some cases, and in other cases, there maybe requirements for using the hint that are not described.

Part 2 (7 pts)

The second part of the programming assignment requires you to write an operator in a toy Relational Database Management System, built by us.

Description of ToyRDBMS: The ToyRDBMS is written in Java, and uses BerkeleyDB as the underlying storage engine. BerkeleyDB does not provide any relational interface at all. In essence you can think of BerkeleyDB as a *persistent key-value store*. The Data in BerkeleyDB is stored in a set of “Databases”, each of which is a key-value store.

Our ToyRDBMS maps relations to BerkeleyDB Databases. So each Database stores the tuples corresponding to a single relation, indexed by the *primary key* (which must be specified in the CREATE TABLE command).

ToyRDBMS supports a very small subset of SQL with many restrictions. The supported commands are:

1. **CREATE TABLE:** Only two data types are supported: “integer” and “string”. The primary key must be identified, can only consist of a single attribute, and further must be the first attribute in the list.

Example: “create table R (i integer primary key, j integer, s string);”

2. **DROP TABLE.**

3. **INSERT VALUES:** The usual SQL syntax should work.

4. **SELECT:** ToyRDBMS supports a very small subset of the form:

```
select <list of attributes>
from <list of tables>
where <list of predicates>;
```

The Select Clause can either contain “*” or a list of attributes. No aliasing is allowed in either the select clause or the from clause. Only equality predicates are allowed. If a predicate of the type “R.a = 10” is used, “R.a” must be on the left hand side.

Finally, queries should not contain cycles or should not require Cartesian products.

Some further details about the implementation follow. Many of the commands listed below are present in the file “Makefile”. You can run them (on most UNIX-like machines) by: “make X” (where X is the shortcut). The make commands are listed below.

- You are provided with a zip file. On glue machines, unzip it to create a directory “ToyRDBMS”. All commands below must be executed from within this (ToyRDBMS) directory.
- ToyRDBMS is written in Java, with a handful of files. You can compile it using “javac”.

```
make all    javac -classpath ../javacc.jar:./je-3.3.82.jar *.java
```

The required libraries are present in the working directory.

- ToyRDBMS uses “javacc” for parsing SQL. The parser is specified in the file “SQLParser.jj”. “javacc” is used to convert the “.jj” file into a set of java files. This has already been done for you (you will see several files like: SQLParserConstants.java, SQLParserTokenManager.java etc., which were all automatically generated).

However if you modify the parser, you can recreate those files using:

```
make SQLParser.class   java -cp ./javacc.jar:. javacc SQLParser.jj
```

Ignore the warnings. The LOOKAHEAD is set so that the conflicts should not be a problem. Many of the limitations of the supported syntax are essentially limitations of this parser, and can be addressed quite easily.

- Some of key code files in ToyRDBMS are:
 - SQLParser.jj: As described above, this contains the SQL parser.
 - CommandLine.java: ToyRDBMS only has a commandline interface at this time. This class implements that interface by accepting user input from the standard input, and executing those statements.
 - Globals.java: This contains some of the Global definitions, and more importantly, much of the code that talks to BerkeleyDB, including creating databases, inserting tuples etc.
 - Operator.java: This is the abstract Operator class that defines the get_next() iterator interface.
 - ScanOperator.java: Implements Sequential Scan.
 - JoinOperator.java: Another abstract Operator class. All join operators should be its subclasses.
 - NestedLoopsJoinOperator.java: Implements the Nested Loops join operation.
 - Query.java: Contains the code for the analyzing a query, and creating a query plan for it. Currently the query plan is created in a fairly dumb fashion. Read the comments for details.
 - RelationSchema.java, Predicate.java, Tuple.java: All of these should be self-explanatory.

Running and Testing ToyRDBMS: As mentioned above, you can compile all the java files using the javac command (or you can import all of these into Eclipse if you like, but I won’t be able to help you with any compile or runtime issues).

ToyRDBMS stores the data in a directory which is a parameter to “CommandLine” command (see below). This directory must exist. The commands below assume that the directory is named “Data” and is in the current directory.

For testing purposes, two populate files are provided:

1. populate-sn.sql: Contains essentially the same data as the SQL Assignment, but modified to satisfy the restrictions of ToyRDBMS.
2. populate-bad.sql: This contains three very simple relations: R, S, T, each with just 1 attribute. Purpose of this one would be made clear below.

You can run these files as:

make populate-sn

```
cat populate-sn.sql | java -ea -classpath ../javacc.jar:./je-3.3.82.jar CommandLine Data
```

make populate-bad

```
cat populate-bad.sql | java -ea -classpath ../javacc.jar:./je-3.3.82.jar CommandLine Data
```

These will create the tables and insert tuples into it. As you can see, CommandLine takes in one argument, the data directory (which must be already created).

You can test it using:

make query2

```
echo "SELECT * FROM users, friends WHERE users.userid = friends.userid1 and users.userid = 'user10';" | java -ea -classpath ../javacc.jar:./je-3.3.82.jar CommandLine Data
```

Your Task: Your task is to implement a Hash Join operator to supplement the NestedLoopsJoinOperator.

Try the query:

make query1

```
echo "SELECT * FROM R, S, T WHERE R.a = S.a and S.a = T.a;" | java -ea -classpath ../javacc.jar:./je-3.3.82.jar CommandLine Data
```

If you look at the three relations, you will see that each of them contain 2002 tuples, but the join only contains two results. This forms a BAD case for the Nested Loops operator. When you run this query, it will take a long time.

You are to implement the HashJoinOperator so that such queries run (much) faster.

More Details on How: The code already contains a file: HashJoinOperator.java which defines the HashJoinOperator, but doesn't implement it.

You are to write appropriate init(), get_next(), and close() functions.

After you are done, change the "canBeUsed()" function to return "true", and the query planner will automatically start using your Hash Join Operator (see the code in JoinOperator.java to see how this happens).

Submission: You will be submitting just your HashJoinOperator.java file, and we will test it automatically against a set of input data files. We will essentially copy your file into our setup, compile it, and compare it against the NestedLoopsJoinOperator to see if the join results are correct, and also if the implementation is sufficiently fast. More details will be provided later.