Project report - CS3223

Yoshiaki Nishimura - Julien Heitmann - Tiago Kieliger

April 5, 2018

1 Dynamic programming optimizer

The optimizer we implemented is based on the dynamic programming approach which performs an (almost) exhaustive search over the plan space with the assumption of optimal substructure. This yields a better running time than a truly exhaustive search algorithm. We restricted ourselves to left-deep trees, which are trees that contain only base tables on the right children. In such trees, the number of states $= \mathcal{O}(2^K)$. In addition, we avoid Cartesian products, meaning that we only consider a join of two tables if they have an explicit condition that joins on a common attribute.

2 Limitation of the provided SPJ query engine

2.1 Real values in random database generator

The random database generator is supposed to allow data types of either INTEGER, STRING or REAL (and TIME, see next subsection). But we noticed that the REAL type did not work. Indeed, when choosing the REAL type, the corresponding attribute would simply not be generated. This was due to a bug in the code of *RandomDB.java*. **We addressed it** and REAL values can now be generated.

2.2 Time data type

In the instructions we are asked to use TIME data type for some of the attributes. However, neither the random database generator nor the PSJ support such an attribute so **we addressed that problem** and modified the code in consequence. The type used to display the time in the HH:MM:SS format is java.sql.Time but in the database.tbl file, the time is represented as a long value. An example query would be :

```
SELECT *
FROM flights
WHERE flights.departs="12:45:00"
```

Moreover, one can use RandomDB to generate a database with time attribute. In the .det file, the type 'TIME' should be used, the size in memory is 8 bytes (stored as a long) and the range should be given as an integer and corresponds to the range in hours. The ConvertTxtToTbl.java file has been updated accordingly. An exeample .det file would be :

```
3
32
id INTEGER 1000 PK 4
name STRING 10 NK 20
start_time TIME 20 NK 8
```

Here, the range is set to 20, this means that random times from 00:00:00 to 20:00:00 will be generated, for example, a possible .txt file would be :

```
11 fddjjskwoo 18:36:01
524 dkimvzucjb 09:50:49
460 atpqgtvtgv 12:45:50
...
```

2.3 Lack of support for Cartesian products (not addressed)

The SPJ query engine takes it for granted that there exists a common attribute between the two relations we are joining. Therefore, it is not able to compute the Cartesian products which lacks a join condition.

2.4 *next()* returns an empty page

The call to *next()* of an operator that represents a raw table (saved as txt-file) might return an empty batch before returning null, which is not necessary.

3 Experiment 1

In this experiment, we perform three two-relation joins involving the following relations, each under block-nested join and sort-merge join, and compare the performance of the two plans.

```
aircrafts.det - 10'000 tuples generated
3
28
aid INTEGER 50000 PK 4
aname STRING 10 NK 20
cruisingrandge INTEGER 20000 NK 4

certified.det - 13'000 tuples generated
2
8
eid INTEGER 50000 PK 4
aid INTEGER 10000 FK 4
```

```
employees.det - 11'000 tuples generated
3
28
eid INTEGER 35000 PK 4
ename STRING 10 NK 20
salary INTEGER 150000 NK 4

flights.det - 12'000 tuples generated
6
56
flno INTEGER 40000 PK 4
from STRING 8 NK 16
to STRING 8 NK 16
distance INTEGER 20000 NK 4
departs TIME 24 NK 8
arrives TIME 24 NK 8

schedule.det - 10'500 tuples generated
2
8
flno INTEGER 20000 PK 4
aid INTEGER 10000 FK 4
```

We have written *Experiment.java* class that runs each join in a JUnit test environment. The execution time we calculate is the time since *root.open()* is called and till all tuples are written into a temporary file. Therefore, the time for the optimizer to compute the plan is NOT included. We have set batch size = 3,000 bytes and number of buffers = 30 for each test. The average execution time for each plan over 5 samples for each join is shown in the following tables.

3.1 Join 1: Employees and Certified (via eid)

number of tuples generated: 2864	
Join method	Execution time (s. 4dp)
block nested	23.7660
sort merge	2.0630

3.2 Join 2: Flights and Schedule (via flno)

number of tuples generated: 3138	
Join method	Execution time (s. 4dp)
block nested	21.7080
sort merge	2.4720

3.3 Join 3: Schedule and Aircrafts (via aid)

number of tuples generated: 2045	
Join method	Execution time (s. 4dp)
block nested	17.7400
sort merge	1.6880

4 Experiment 2

In this experiment, we perform a three-relation join indicated by the following query.

```
SELECT employees.ename
FROM employees,certified,schedule
WHERE employees.eid=certified.eid,certified.aid=schedule.aid,schedule.flno="x"
```

where x denotes the flight number of our interest. We are going to execute this query under three different plans (by restricting the join methods supported by the optimizer) as specified by the project description; under page-nested join, block-nested join, and sort-merge join. The operator tree for each plan looks as follows:

```
Project(NestedJoin(NestedJoin(certified [certified.eid == employees.eid] \\
employees) [certified.aid == schedule.aid] schedule))

Project(BlockNested(BlockNested(certified [certified.eid == employees.eid] \\
employees)[certified.aid == schedule.aid] schedule))

Project(SortMerge(SortMerge(employees [employees.eid == certified.eid] \\
certified)[certified.aid == schedule.aid] schedule))
```

We have set batch size = 3,000 bytes and number of buffers = 30 for the experiment. The average execution time for each plan over 5 samples is shown in the following table.

num tuples generated: 2999	
Join method	Execution time (s. 4dp)
page nested	52.8120
block nested	12.5600
sort merge	5.5840

5 Interpretation of the experiment results

As covered throughout the Relational Operators and Query Optimizer section of CS3223, the results from the experiment demonstrate the effects of the choice of join methods and execution plan on the execution time of a query. To recap,

$$\text{Cost block-nested join} = R + S \cdot \lceil \tfrac{R}{B-2} \rceil$$

$$\text{Cost sort-merge join} = L \cdot [2 + \lceil \log_{B-1}(\lceil \tfrac{L}{B} \rceil) \rceil] + R \cdot [2 + \lceil \log_{B-1}(\lceil \tfrac{R}{B} \rceil) \rceil] \text{ (BC)}$$

$$\text{Cost page-nested join} = \text{R} + \text{R*S}$$

where R = number of pages of outer table and S = number of pages of inner table, and B is the number of buffers available for this join.

We can see from the results from Experiment 1 that sort-merge join will outperform block-nested join in I/O cost under normal conditions. In experiment 1, on average sort-merge join only required approximately 10% of the time required for block-nested join.

In Experiment 2, we compared the execution time for three different join methods. The worst performance resulted from page-nested join as expected, since it does not leverage on the buffers available and only reads one page of R for each iteration of join step. The block-nested join that does leverage on B-2 buffers available performed the join under approximately 24% of time required by page-nested join. The best performance came from sort-merge join as expected. This further reduced the execution time, requiring less than 42% of time for block-nested join.

These results reinforce the idea that the join methods heavily affects the execution time of a join operation in databases. In addition, as we can see from the formula for page-nested and block-nested joins, the performance of some join operators are also affected by the ordering of the tables (R and S). A real-world database which contains relations with much larger number of tuples than our example therefore must optimize the join operations in order to make applications running on the database usable.

5.1 Limitations in the experiments

A limitation of our methodology in the experiments is that we are only respecting the buffer contraints on the program-level through the use of *BufferManager* class and not on the OS-level. This could have overstimated the performance of our query engine overall.

Another limitation is that we only use the time to execute the code as the estimator for the query costs, since it is difficult to separately calculate CPU cost and I/O cost. This could have been biased towards some join operators due to the fact that there will be different overhead in preparing the objects and classes required to perform different joins as well as the fact that there exists garbage collection by Java.