Experiments for a Database Course: Access Path Selection by the Oracle Query Optimizer

Jamal Alsabbagh
School of Computing and Information Systems
Grand Valley State University, Allendale, Michigan, USA

Abstract – Standard database textbooks describe physical data organization and may present analytical cost models in order to give an appreciation of the impact of physical design on SQL queries performance. The main goal of this paper is to present hands-on experiments so that students can ‘see’ rather than just ‘hear’ about such impact. Another goal is to show students that while optimizers can be very sophisticated, they can miss obvious opportunities for optimization making it incumbent upon the database programmer to write queries in a form that their system will be more likely to optimize. While the paper intentionally uses only a subset of system-specific monitoring tools, it motivates the reader to deploy more sophisticated tools for further probing into the query optimizer’s characteristics.

Keywords: Database. Query optimization. Teaching tools.

1 Introduction

SQL and aspects of the physical organization of data (such as heap organization, indexing, and hashing) are two standard topics in database courses. The purpose of covering the latter topic in a course is to give students an appreciation of the impact of physical organization on query performance. Coverage of the topic in typical database textbooks such as [2, 4, 7, 8] is, however, confined to being descriptive and may, at best, also include the presentation of analytical cost models that aim to estimate query execution cost in terms of the number of disk blocks that have to be retrieved in order to process a query. Invariably, the coverage lacks actual examples of how industrial-strength query optimizers behave. The main goal of this paper is to bridge the gap in teaching the two topics of SQL and physical organization by presenting students with hands-on experiments in order for them to see rather than just hear about the relationship between the two topics. Another goal is to show students that while optimizers can be very sophisticated, they can miss what may seem to be obvious opportunities making it incumbent upon the database programmer to write queries in a form that their system will be more likely to optimize. The paper also motivates the need for exploring more sophisticated system-specific tools in order to tune query performance in real-world large database systems. The experiments reported in this paper were conducted using Oracle 12c. However, it should be relatively easy to map them to other database systems by using the comparable tools in such systems.

Obviously, one of the challenges in conducting experiments on industrial-strength systems like Oracle in an academic course is that such systems are very large thus making the learning curve of the intricacies of their monitoring tools quite steep and may therefore require devoting an unacceptably large amount of precious class time. In order to address that issue and make the material easily accessible to students, the work reported in this paper requires learning only a few system-specific features that are explained in the paper and may not even require digging into the system manuals. As a prerequisite, however, students are assumed to know SQL and data organization in order to benefit from the experiments.

In the remainder of this paper, section 2 describes the setup for the experiments and how to obtain and interpret query execution plans in Oracle. Section 3 presents the experiments and discusses the results. Section 4 gives an example to show why it is important for the programmer to have some knowledge about the behavior of their query optimizer in order to avoid optimization pitfalls. Section 5 explains how the material presented in this paper can act as a stepping stone to conducting further experiments using more advanced performance monitoring tool. Finally, section 6 concludes the paper.

2 Experiments setup

The experiments reported in this paper were conducted using Oracle 12c. This section describes the data used to conduct the experiments and also introduces the Oracle-specific features needed for obtaining and inspecting the query execution plans that Oracle’s cost-based optimizer (CBO) generates.
2.1 The database

Since the experiments concentrate on access path selection only, the database consists of one table called T250 that has 250,000 rows. The table was generated as part of a 9-table database using the Bucky database generator [1]. The table T250 has, however, been customized for the purposes of this paper by adding/dropping/renaming columns and by creating indexes. Details of the customization are not necessary for reading this paper. (The data and the scripts needed to load the table and its indexes are available from the author upon request.) The statistics about the table, value distributions, and the indexes are depicted in Figure 1.

2.2 How to inspect the query plans

The following analyze command must be run after the creation of the database, the addition/dropping of any indexes, or any update on the table. The command instructs the database to collect and update the database catalog that is needed by the CBO when estimating the relative costs of alternative execution plans for a query.

```sql
analyze table T250 compute statistics;
```

- **Table T250 Schema**
  
  T250 (Id, name, state, college, major, loan, notes)

- **The commands to create the indexes**
  
  ```sql
  alter table T250 add constraint T250_id primary key (id) enable validate;
  create index T250_state_ix on T250(state);
  create index T250_major_ix on T250(major);
  ```

- **The commands to inspect the data definitions and distributions in the table T250**
  
  ```sql
  Select min(id), max(id), count(distinct id) from T250 ;
  -- <Repeat for every column>
  ```

- **The commands to retrieve row length and size of the table T250**
  
  ```sql
  Select table_name, num_rows, avg_row_len, blocks from user_tables
  where table_name='T250';
  ```

- **The commands to retrieve information about the indexes**
  
  ```sql
  select index_name, table_name, leaf_blocks, distinct_keys from user_indexes
  where table_name = 'T250' order by index_name;
  ```

<table>
<thead>
<tr>
<th><strong>col name</strong></th>
<th><strong>Index name</strong></th>
<th><strong>Min val</strong></th>
<th><strong>Max val</strong></th>
<th><strong>distinct vals</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>Id</td>
<td>INTEGER</td>
<td>75,001</td>
<td>325,000</td>
<td>250,000</td>
</tr>
<tr>
<td>name</td>
<td>CHAR(20)</td>
<td>Name100000</td>
<td>Name99999</td>
<td>50,000</td>
</tr>
<tr>
<td>state</td>
<td>CHAR(20)</td>
<td>Alabama</td>
<td>Wyoming</td>
<td>50</td>
</tr>
<tr>
<td>college</td>
<td>INTEGER</td>
<td>0</td>
<td>1,999</td>
<td>2,000</td>
</tr>
<tr>
<td>major</td>
<td>INTEGER</td>
<td>1</td>
<td>250</td>
<td>250</td>
</tr>
<tr>
<td>loan</td>
<td>INTEGER</td>
<td>101</td>
<td>74899</td>
<td>21,617</td>
</tr>
<tr>
<td>notes</td>
<td>CHAR(250)</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th><strong>TABLE NAME</strong></th>
<th><strong>NUM ROWS</strong></th>
<th><strong>AVG ROW LEN</strong></th>
<th><strong>BLOCKS</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>T250</td>
<td>250000</td>
<td>318</td>
<td>11926</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th><strong>INDEX_NAME</strong></th>
<th><strong>TABLE_NAME</strong></th>
<th><strong>LEAF BLOCKS</strong></th>
<th><strong>DISTINCT KEYS</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>T250 ID</td>
<td>T250</td>
<td>523</td>
<td>250,000</td>
</tr>
<tr>
<td>T250 MAJOR IX</td>
<td>T250</td>
<td>510</td>
<td>250</td>
</tr>
<tr>
<td>T250 STATE IX</td>
<td>T250</td>
<td>1,117</td>
<td>50</td>
</tr>
</tbody>
</table>

**Figure 1:** Table T250 and its Indexes characteristics
In order to inspect the plan that the CBO generates for a query, one needs to run the following two commands. The first command stores the selected plan for the query into a plan table that is internal to Oracle. The second command retrieves the plan from the plan table and displays it.

```
explain plan for <SQL query>;
select * from TABLE(DBMS_XPLAN.DISPLAY());
```

Figure 2 below depicts three examples that illustrate the sample output from the above commands for three queries. (Some parts of the output have been deleted due to space limitations.)

Briefly, a plan output should be read from the bottom up with the most deeply indented steps executed first rather than in the order shown in the 1st column in the plan output. The details about the plan table and a description of the available access paths in Oracle are outlined in [3] and described in details in [5].

Example-1 in figure 2 shows that the plan requires a 'Full Table Access' (2nd column) in order to retrieve all 250,000 records (4th column), at an estimated cost of 3,309 (6th column). As stated in [5], cost “does not have any particular unit of measurement; it is merely a weighted value used to compare costs of execution plans. The value of this column is a function of the CPU_COST and IO_COST columns.”

Example-2 shows that the plan performs an 'Index Fast Full Scan' at a cost of 144 on the index T250_ID (which indexes the primary key) in order to count the number of index entries. In other words, it does not require scanning the table T250 itself which would have cost 3,309 as example-1 above showed.

```
Example-1
explain plan for select * from T250;
select * from TABLE(DBMS_XPLAN.DISPLAY());
```

```
Example-2
explain plan for select count(*) from T250;
select * from TABLE(DBMS_XPLAN.DISPLAY());
```

```
Example-3
explain plan for select * from T250 where major=100;
select * from TABLE(DBMS_XPLAN.DISPLAY());
```

Figure 2: Three examples of query plans
Finally, Example-3 shows that an ‘Index Range Scan’ on the index T250_major_ix is performed for the range ‘major=100’ and then the rowIDs (i.e. record addresses) found in the index are used to retrieve the relevant records from the data table at a total cost of 1,615. In the example, the CBO estimated that there will be 1,000 records having major=100 since, as per figure 1, there are 250 different majors for the 250,000 students. In other word, the CBO assumes a uniform distribution of values – more on the issue of value distribution in section 4.

3 Experiments and Discussion

This section demonstrates through several queries the types of plans that the CBO generates. As will be seen shortly, some plans conform closely to the programmer’s expectations based upon basic knowledge about physical organization. In contrast, some plans are, to say the least, interesting and should motivate the need to actually look at them rather than assume how the CBO might work. In order to streamline the discussion, every query will be stated together with its estimated plan cost and then followed by highlights of its plan and a discussion relating its plan to those of other queries whenever applicable. The actual plans (as shown in figure 2) will not be reproduced in this section due to space limitations.

3.1 Baseline queries

The following two queries, Q-10 and Q-11) show the relative costs of performing a full table scan vs. an index only scan.

Q-10: (Cost: = 3,309)

```sql
select * from T250;
```

As expected, the query required a full table scan of T250 in order to retrieve all 250,000 rows in table T250 whose size is 11,926 disk blocks as per figure 1.

Q-11: (Cost: = 144)

```sql
select count(*) from T250;
```

As expected, the query required a full scan of the unique primary key index T250_id (whose size is 523 disk blocks as per figure 1) in order to count the number of entries in it.

It is interesting to note that in queries Q-10 and Q-11 above, the ratio between their costs is 23:1 which is also the ratio between the respective sizes of the files that had to be scanned. The implication here is that for simple queries, the cost is almost entirely based upon the number of disk blocks that have to be retrieved.

3.2 Selectivity and index usage

The selectivity factor of a predicate in a query is the ratio between the number of rows that satisfy the predicate and the total number of rows and is, therefore, in the range 0 and 1. Systems tend to use indexes only when a predicate is very selective (meaning that the selectivity factor is low) since random access to a table to retrieve a large number of rows that satisfy a weak predicate can be worse than a full scan of the table due to the random nature of access to disk. Queries Q-12 through Q-18 below illustrate that point.

Q-12: (cost = 3,309)

```sql
select *
from T250
where state='Michigan';
```

In Q-12, the estimated selectivity factor of the predicate (state='Michigan') is 5,000/250,000 (= 0.02) since there are 50 states each of which is estimated to have 5,000 students. The CBO deemed the index on state not to be useful and opted for a full scan of the table at a cost of 3,309.

Q-13: (cost = 6,042)

```sql
Select
/*+ index(T250 T250_state_ix) */ *
from T250
where state='Michigan';
```

In order to see why the CBO ignored the index in query Q-12, it was rewritten as query Q-13 with the hint /*+ index(T250 T250_state_ix) */. The hint forces the CBO to use the index on state. When it did, using an index range scan on ‘Michigan’ to access the table T250, the estimated cost shot up to 6,042 which is more than 82% higher than the full table scan required for Q-12.

Q-14: (cost = 1,615)

```sql
select *
from T250
where major = 100;
```

In Q-14, the estimated selectivity factor of the predicate (major=100) is 1,000/250,000 (0.004) since there are 250 different majors each of which is estimated to have 1,000 students. Unlike query Q-12, the CBO now deemed the index on major to be useful and opted for performing a range scan on the index and then retrieving the records from the table. Compared to a full table scan
(as in Q-12), the index was useful and dropped the cost by nearly 50%.

Q-15: (Cost = 3,230)

```
select *
from T250
where major in (100, 101);
```

The predicate (`major in (100, 101)`) is weaker than the one in Q-13 with an estimated selectivity factor of 2,000/250,000 (= 0.008). As in Q-14 above, the CBO still deemed the index on `major` to be useful and opted for performing a range scan on the index and then retrieving the records. As expected, the cost now is twice the cost of Q-14 but is still less than the 3,309 cost of a full table scan in Q-12.

Q-16: (cost = 3,309)

```
select *
from T250
where major in (100, 101, 102);
```

In order to probe further into the selectivity issue, Q-16 is designed to have an even weaker condition than query Q-15 whereby the selectivity factor of the predicate (`major in (100, 101, 102)`) is 3,000/250,000 = 0.012. Unlike queries Q-14 and Q-15 above, the CBO now deemed the index on `major` to be no longer useful and opted for performing a full table scan.

Q-17: (cost = 4,843)

```
select /*+ index(T250 T250_major_ix) */ *
from T250
where major in (100, 101, 102);
```

In order to demonstrate why the optimizer didn’t utilize the index in query Q-16, it was rewritten, as Q-17, with the hint `/*+ index(T250 T250_major_ix)*/` in order to force it to use the index on `major`. When it did, the estimated cost became 4,843 which is 46% higher than a full table scan since the table T250 had to be probed for too many records through the index.

Q-18: (cost = 98)

```
select *
from T250
where major in (100, 101, 102)
and state='Michigan';
```

Query Q-18 is yet another demonstration of the effect of selectivity. Its predicate (`major in (100, 101, 102) and state='Michigan'`) is the conjunction of the two predicates of queries Q-12 and Q-16 neither of which used their relevant indexes. The estimated selectivity factor for the conjunctive predicate in Q-18 is now 60/250,000 (= 0.00024) since only 1/50th of the 3,000 students majoring in majors 100, 101, or 102 are estimated to be from Michigan. Here, the CBO opted to perform two index range scans (one on `major` and one on `state`), intersect the results in order to get the addresses of the records that satisfy both predicates, and only then did it probe into the table. The estimated plan cost now dropped dramatically to just 98.

### 3.3 The possible high cost of sorting

Q-19: (cost = 19,602)

```
select *
from T250
order by state;
```

Query Q-19 is similar to Q-12 in section 3.1 above except that it asks for the results to be sorted by state. Its purpose is to demonstrate that sorting a large amount of data can be quite high since it may require an external sort that involves staging data to/from disk and/or in-memory data movement. As in Q-12, the CBO generated a plan that required a full table scan at a cost of 3,309 and then performed the sort for an additional cost of 16,293 resulting in an estimated total cost of 19,602.

Q-20: (cost = 4,225)

```
select state, college, count(*)
from T250
group by state, college
order by state;
```

Query Q-20 demonstrates that sorting in a query should not always be pre-judged to be detrimental to cost as was the case in Q-19. Query Q-20 asks for the number of students from every state by every college which requires an implicit sort on `<state, college>` in order to process the `GROUP BY` clause and additionally present the results ordered by state. Consequently one might jump to the conclusion that it will be an expensive query based upon what Q-19 revealed about the high cost of sorting. However, inspection of the plan for Q-20 revealed that although it required a full table scan (costing 3,309), the sort added only 916 to the total cost, rather than the 16,293 addition in Q-19. The reason for the low additional cost here is that, unlike Q-19, the sorting in Q-20 involved only short records (which is usually the case in `GROUP BY` queries) requiring less movement of data than was the case in Q-19.

### 4 The need to know the optimizer’s behavior

Although query optimizers are very sophisticated and do transformations (such as join re-ordering and query de-correlation) on submitted SQL queries in order to find efficient forms for execution, it is still incumbent upon the programmer to have some basic knowledge about their optimizer’s behavior. In particular, one
should not assume that the optimizer will do the right transformation even in simple situations. One can come up with several examples, but due to space limitations, here is one simple example to illustrate the point.

Q-22: (cost = 7,147)  
```
select * from T250
where major in (100, 101, 102)
intersect
select * from T250
where state='Michigan';
```

Query Q-22 is logically equivalent to query Q-18. However the CBO generated a plan that required two full scans of the table; one to process the predicate on major (see query Q-16) and one to process the predicate on state (see query Q-12). It then intersected the results and estimated the plan cost to be 7,147 – quite a dramatic difference from the cost of just 98 for Q-18 that used index intersection.

5 Probing further

Using the explain plan tool in Oracle in the above experiments reveals the plan that the CBO estimates to be the most efficient. However, the actual cost of executing a plan can be significantly different from the estimate mainly due to the possibility that the actual data distributions may be skewed. For example, the plans discussed in section 3 were based upon the CBO’s assumptions that there is an equal number of students from each state. But if 20% of students were from Michigan while only 0.1% were from Wyoming then a query similar to Q-13 in section 3.1 that asks, using an index, for Wyoming students rather than Michigan students will have a much lower actual execution cost than Q-12 although its estimated plan cost will be similar to the one for Michigan. The lesson here is that if the programmer knew the actual data distributions then it will be appropriate to use a hint to force the use of an index.

In order to account for data skew (especially in multi-table join queries where the estimated sizes of intermediate results is crucial for accurate cost estimate) Oracle, for example, facilitates the collection of statistics about data distributions in the form of histograms. Furthermore, Oracle provides tools such as Autotrace for tracing the cost of actual execution of queries thereby allowing one to investigate the actual, rather than the estimated, cost [5].

6 Conclusions

This paper presented a set of experiments that gives students insight and hands-on experience in order to establish a connection between the two topics of SQL and physical data organization that get covered in a typical database course. The tools used to investigate the optimizer-generated plans were intentionally kept to a very small subset of available tools in order to make it both accessible to students and to minimize the amount of time needed to cover the topic in a course. The paper motivated the need for programmers to acquaint themselves with the capabilities and limitations of whatever optimizer their system has in order to avoid costly formulations of their queries. Finally, the paper gave pointers to other tools that one needs to be familiar with in an actual development environment.

References