Query optimization, query tuning

Rasmus Pagh

Today's lecture

- Query optimization:
  - Overview of query evaluation
  - A typical query optimizer
  - Left for next weeks:
    - Evaluation of relational operators
    - Estimating sizes of intermediate results
- Query tuning:
  - Providing good access paths
  - Rewriting queries

Basics of query evaluation

How to evaluate a query:

1. Rewrite the query to (extended) relational algebra.
2. Determine algorithms for computing intermediate results in the cheapest way.
3. Execute the algorithms and you have the result!

Complications, 1

"Rewrite the query to (extended) relational algebra."

- Can be done in many equivalent ways. Some may be "more equal than others"!
- Size of intermediate results of big importance.
- Queries with corellated subqueries do not really fit into relational algebra.

Complications, 2

"Determine algorithms for computing intermediate results in the cheapest way."

- Best algorithm depends on the data:
  - No access method (index, table scan,...) always wins.
  - No algorithm for join, grouping, etc. always wins.
  - There are dependencies, e.g. the form of an output from one operator influences the next.
- Query optimizer should make an educated guess for a (near)optimal way of executing the query.

SQL Commands - Microsoft Internet Explorer

File Edit View Favorites Tools Help

Address http://127.0.0.1:8080/apex/f?p=4500:1003:3337608012056264::NO:::

ORACLE Database Express Edition

Home Logout Help

User: HR

Home > SQL > SQL Commands

Autocommit Display 10 Save Run

```
SELECT AVG(SALARY)
FROM (EMPLOYEES NATURAL JOIN DEPARTMENTS NATURAL JOIN LOCATIONS NATURAL JOIN COUNTRIES)
WHERE COUNTRY_NAME='Denmark'
```

Results Explain Describe Saved SQL History

Query Plan

Operation	Options	Object	Rows	Time	Cost	Bytes	Filter Predicates *	Access Predicates
SELECT STATEMENT			1	1	10	26		
SORT	AGGREGATE		1			26		
MERGE JOIN	CARTESIAN		253	1	10	6.578		
TABLE ACCESS	BY INDEX ROWID	DEPARTMENTS	1	1	1	7	"EMPLOYEES""DEPARTMENT_ID" = """DEPARTMENT_ID"	
NESTED LOOPS			11	1	5	286		
MERGE JOIN	CARTESIAN		107	1	4	2.033		
INDEX	FULL SCAN	COUNTRY_C_ID_PK	1	1	1	8	"COUNTRIES"."COUNTRY_NAME" = 'Denmark'	
BUFFER	SORT		107	1	3	1.177		
TABLE ACCESS	FULL	EMPLOYEES	107	1	3	1.177		
INDEX	RANGE SCAN	DEPARTMENTS_IDX1	2	1	0		"DEPARTMENTS"."MANAGER_ID" IS NOT NULL	"EMPLOYEES"."MANAGER_ID" = "DEPARTMENTS"."MANAGER_ID"
BUFFER	SORT		23	1	9			
INDEX	FAST FULL SCAN	LOC_CITY_IX	23	1	0			

* Unindexed columns are shown in red

Done Internet

Motivating example

(derived from Lahdenmäki and Leach, 2005)

Schema:

- Customer(cno,name,country,type)
- Invoice(ino,cno,amount)

Query:

```
SELECT C.name,C.type,I.ino,I.amount
FROM Customer C, Invoice I
WHERE I.amount>10000 AND
      C.country="Sweden" AND
      C.cno=I.cno
ORDER BY amount DESC
```

Example, cont.

- Simple logical query plan (no `ORDER BY`):

$$\pi_{\text{name,type,ino,amount}}(\sigma_{\text{amount}>10000\wedge\text{country}=\text{"Sweden"}}(\text{Customer} \bowtie Invoice))$$

- Next week: How the join can be efficiently executed.
- For now assume that this takes a *linear* number of I/Os:
  - Time proportional to size of the intermediate result $\text{Customer} \bowtie Invoice$

Pipelining

- There is no need to *materialize* the two intermediate results, i.e. write them to disk:
  - As soon as a tuple of $\mathrm{Customer} \bowtie Invoice$ is known we can apply select and project.
- Performing several dependent relational operations ”in parallel” is known as *pipelining*.

Example, cont.

- Pushing selects:

$$\pi_{\text{name,type,ino,amount}}(\sigma_{\text{amount}>10000}(Invoice) \bowtie \sigma_{\text{country}=\text{"Sweden"}}(Customer))$$

- Reduces the amount of data in join
  - e.g. assume Invoice reduced to 0.1%
  - e.g. assume Customer reduced to 10%
- The two selects may make use of existing indexes.
  - Ideally covering indexes.

Pushing only one select

$$\pi_{\text{name,type,ino,amount}}(\sigma_{\text{amount}>10000}(\text{Invoice} \bowtie \sigma_{\text{country}=\text{"Sweden"}}(Customer))$$

- For this logical query plan the join may be implemented using an index.
  - "Index nested loop join".
  - Possibly faster than pushing both selects, depending on the size of $\sigma_{\text{country}=\text{"Sweden"}}(Customer)$

Pushing the other select

$$\pi_{\text{name,type,ino,amount}}(\sigma_{\text{country="Sweden"}}(\text{Customer} \bowtie \sigma_{\text{amount}>10000}(Invoice)))$$

- Again, the join may be implemented using an index.
- With an index on Invoice, we may get the results in decreasing order of amount.
  - This makes a last sorting step (`ORDER BY`) redundant.

Algebraic equivalences

- In the previous examples, we gave several equivalent queries.
- A systematic (and correct!) way of forming equivalent relational algebra expression is based on *algebraic rules*.
- Query optimizers consider a (possibly quite large) space of equivalent plans at run time before deciding how to execute a given query.

Problem session

For each of the following algebraic laws, consider whether it might be useful for rewriting an algebraic expression to have smaller computation time:

1. $\sigma_C(E_1 \cup E_2) = \sigma_C(E_1) \cup \sigma_C(E_2)$.
2. $\sigma_C(E_1 - E_2) = \sigma_C(E_1) - E_2$.
3. $\sigma_C(E_1 - E_2) = \sigma_C(E_1) - \sigma_C(E_2)$.
4. $\sigma_C(E_1 \times E_2) = \sigma_C(E_1) \times E_2$ if E_1 has all attributes in C.
5. $\sigma_C(E_1 \cap E_2) = \sigma_C(E_1) \cap \sigma_C(E_2)$.
6. $\pi_L(E_1 \bowtie E_2) = \pi_L(\pi_{L \cup (A_{E_2} \cap A_{E_1})}(E_1) \bowtie \pi_{L \cup (A_{E_1} \cap A_{E_2})}(E_2))$.
7. $\pi_L(\sigma_C(E_1)) = \pi_L(\sigma_C(\pi_{A \cup L}(E_1)))$ where $A =$ attributes mentioned in C.
8. $\delta(E_1 \bowtie E_2) = \delta(E_1) \bowtie \delta(E_2)$.

Simplification

- Core problem: $\sigma\pi\times$–expressions, consisting of equi-joins, selections, and a projection.
- Subqueries either:
  - Eliminated using rewriting, or
  - Handled using a separate $\sigma\pi\times$–expression.
- Grouping, aggregation, duplicate elimination, sorting: Handled in a final step.

Single relation access plans

- Example:

$$\pi_{rating,sname}(\sigma_{rating>5 \wedge age=20}(Sailors))$$

- Without an index: Full table scan. (Well, depends on the physical organization.)
- With index:
  - Single index access path
  - Multiple index access path
  - Index only access path (”covering index”)

Multi-relation access plans

- Similar principle, but now many more possibilities to consider.
- Common approach:
  - Consider subsets of the involved relations, and the conditions that apply to each subset.
  - Estimate the cost of evaluating the $\sigma\pi\times-$ expression restricted to this subset.
  - Need to distinguish between different forms of the output (sorted, unsorted).

Multi-relation access plans, cont.

- In general, cannot consider all possible plans. (Too many!) A common restriction is to consider only *left-deep* evaluation trees.
- If the DBMS does not consider a near-optimal plan, it is likely that this is because of bad *cost estimates*.
- The tuner can influence cost estimates and access plans, but not the optimization method.
- We refer to RG for more details.

Core problem: Size estimation

- The sizes of intermediate results are important for the choices made when planning query execution.
- Time for operations grow (at least) linearly with size of (largest) argument. (Note that we do not have indexes for intermediate results.)
- The total size can even be used as a crude estimate on the running time.

- *We will return to size estimation in two weeks.*

Lecture part 2: tuning

- Recall:
  - Query optimization is the DBMSs effort to make a query run as well as possible
  - Tuning is a ”manual” effort to make single queries (or a whole system) run well.

Before tuning: Identifying the problem

- A database *profiler* can identify activities on which the DBMS spends time. E.g.
  - Number of I/Os performed.
  - Lock queueing time.
  - Buffer cache hit ratio [don't count on it!]
  - "Slow log" of the longest running queries.
- The trace information needed may not be collected by default in a production environment.
  - However, it seems the performance overhead is rather low.

Tuning

What can be done to improve the performance of a query?

Key techniques:

- Denormalization
- Vertical/horizontal partitioning (in two weeks)
- Aggregate maintenance (OLAP lecture)
- Query rewriting (examples from SB p. 143-158, 195)
- Sometimes: Optimizer hints

Denormalization

- Denormalize: Introduce redundant data in a relation, that could alternatively be computed by a join.
- Advantages:
  - Saves the join cost.
  - New indexing possibilities.
- Disadvantages:
  - Higher space usage
  - Possibility of inconsistent data

Denormalizing the motivating example

- Customer(cno,name,country,type)
- Invoice(ino,cno,amount,country)

redundant attribute

$$\pi_{\text{name,type,ino,amount}}(Customer \bowtie \sigma_{\text{country}=\text{"Sweden"} \wedge \text{amount}>10000}(Invoice))$$

- Can make a covering index on Invoice(country,amount,cno,ino).

Query rewrite examples from SB

Query rewrite, example 1

- ```
  SELECT DISTINCT ssnum
  FROM Employee
  WHERE dept='Efficient Computation'
  ```
- Problem: ”DISTINCT” may force a sort operation.
- Solution: If ssnum is unique, DISTINCT can be omitted.
- (SB discusses some general cases in which there is no need for DISTINCT.)
  ```

Query rewrite, example 2

- ```
  SELECT ssnum
  FROM Employee
  WHERE dept IN
     (SELECT dept FROM ResearchDept)
  ```
- Problem: An index on Employee.dept may not be used.
- Alternative query:

  ```
  SELECT ssnum
  FROM Employee E, ResearchDept D
  WHERE E.dept=D.dept
  ```
  ```

Query rewrite, example 3

- The dark side of temporaries:

```
SELECT * INTO temp
FROM Employee
WHERE salary > 300000;
```

.

```
SELECT ssnum
FROM Temp
WHERE Temp.dept = 'study admin'
```

- Problems:
  - Forces the creation of a temporary
  - Does not use index on Employee.dept

Query rewrite, example 4

- ```
  SELECT ssnum
  FROM Employee E1
  WHERE salary =
  (SELECT max(salary)
  FROM Employee E2
  WHERE E1.dept=E2.dept)
  ```
- Problem: Subquery may be executed for each employee (or at least each department)

- Solution ("the light side of temporaries"):

```
SELECT dept,
    max(salary) as m
INTO temp
FROM Employee
GROUP BY dept;

SELECT ssnum
FROM Employee E, temp
WHERE salary=m AND
    E.dept=temp.dept
```
```

Query rewrite, example 5

- ```
  SELECT E.ssnum
  FROM Employee E, Student S
  WHERE E.name=S.name
  ```
- Better to use a more compact key:
  ```
  SELECT E.ssnum
  FROM Employee E, Student S
  WHERE E.ssnum=S.ssnum
  ```
  ```

Hints

- ”Using optimizer hints” in Oracle.
- Example: Forcing join order.

```
SELECT  /*+ORDERED */ *
FROM customers c, order_items l, orders o
WHERE c.cust_last_name = 'Smith' AND
      o.cust_id = c.cust_id AND
o.order_id = l.order_id;
```

- **Beware**: Best choice may vary depending on parameters of the query, or change over time! Should always prefer that optimizer makes choice.

Hint example

- ```
  SELECT bond.id
  FROM bond, deal
  WHERE bond.interestrate=5.6
      AND bond.dealid = deal.dealid
      AND deal.date = '7/7/1997'
  ```
- Clustered index on interestrate, nonclustered indexes on dealid, and nonclustered index on `date`.
- In absence of accurate statistics, optimizer might use the indexes on `interestrate` and `dealid`.
- Better to use the (very selective) index on `date`. May use force if necessary!
  ```

Conclusion

- The database tuner should
  - Be aware of the range of possibilities the DBMS has in evaluating a query.
  - Consider the possibilities for providing more efficient access paths to be chosen by the optimizer.
  - Know ways of circumventing shortcomings of query optimizers.
- Important mainly for DBMS implementers:
  - How to parse, translate, etc.
  - How the space of query plans is searched.

Exercise 1

A mystery. Suppose you are the database administrator of a large company. One day your boss comes to you complaining that the following query takes several hours to run, even though the end result is quite small.

```
SELECT Sales.amount, Events.type
FROM Sales, Events, Goods, Suppliers
WHERE Sales.date=Events.date
      AND Sales.partno=Goods.partno AND Suppliers.sid=Goods.sid
      AND Goods.category='engine' AND Suppliers.country='DK'
```

The query plan looks reasonable:

$$(\sigma_{\text{category='engine'}}(\text{Goods}) \bowtie \sigma_{\text{country='DK'}}(\text{Suppliers})) \bowtie (\text{Sales} \bowtie \text{Events})$$

What do you do? Propose queries on the relations that could help shed light on what the problem is? (Feedback on proposals, tests, etc. from teacher in class.) Propose possible cures.

Exercise 2

- Sailors(<u>sid</u>, sname, rating, age)
  - 40 bytes/tuple, 100 tuples/page, 1000 pages
- Reserves(<u>sid, bid, day</u>, rname)
  - 50 bytes/tuple, 80 tuples/page, 500 pages

```
SELECT S.sname
FROM (Reserves NATURAL JOIN Sailors)
WHERE bid=100 AND rating>5
```

Consider possible query plans. Assume the conditions have selectivity 1% and 10% respectively – what query plan has the smallest intermediate results?

Exercise 3

```
4. SELECT *
   FROM Ships, Classes, Outcomes
   WHERE (Outcomes.ship = Ships.name) AND
         (Classes.class = Ships.class);

5. SELECT *
   FROM Movie
   WHERE studioName LIKE 'D%' AND year>1980 AND year<1990;

6. SELECT *
   FROM Movie
   WHERE NOT EXISTS (SELECT *
                     FROM Movie M
                     WHERE M.year > Movie.year);
```