Lecture 6: Query optimization, query tuning

Rasmus Pagh

Today's lecture

- Only one session (10-13)
- Query optimization:
  - Overview of query evaluation
  - Estimating sizes of intermediate results
  - A typical query optimizer
- Query tuning:
  - Providing good access paths
  - Rewriting queries

Basics of query evaluation

How to evaluate a query:

- Rewrite the query to (extended) relational algebra.
- Determine algorithms for computing intermediate results in the cheapest way.
- 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.
- Query optimizer should make an educated guess for a (near)optimal way of executing the query.

SQL Commands - Microsoft Internet Explorer

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

ORACLE Database Express Edition

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 (RG)

Schema:

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

Query:

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

Example, cont.

- Simple logical query plan:

$$\pi_{sname}(\sigma_{bid=100 \wedge rating>5}(Reserves \bowtie Sailors))$$

- Physical query plan:
  - Nested loop join.
  - Selection and projection ”on the fly” (pipelined).
- Cost: Around 500*1000 I/Os.

Example, cont.

- New logical query plan (*push* selects):

$$\pi_{sname}(\sigma_{bid=100}(Reserves) \bowtie \sigma_{rating>5}(Sailors))$$

- Physical query plan:
  - Full table scans of Reserves and Sailors.
  - Sort-merge join of selection results
- Cost:
  - 500+1000 I/Os plus sort-merge of selection results
  - Latter cost should be estimated!

Example, cont.

- Another logical query plan:

$$\pi_{sname}(\sigma_{rating>5}(\sigma_{bid=100}(Reserves) \bowtie Sailors))$$

- Assume there is an index on bid.
- Physical query plan:
  - Index scan of Reserves.
  - Index nested loop join with Sailors.
  - Final select and projection ”on the fly”.
- Cost:
  - Around 1 I/O per matching tuple of Reserves for index nested loop join.

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. $\sigma_C(E_1 \times E_2) = E_1 \bowtie_C E_2$.
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. $\pi_L(\sigma_C(E_1)) = \pi_L(\sigma_C(\pi_A(E_1)))$ 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: 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
  - Sorted 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).
- Details in RG.

Estimating sizes of relations

- 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.
- The total size can even be used as a crude estimate on the running time.

Classical approach: Heuristics

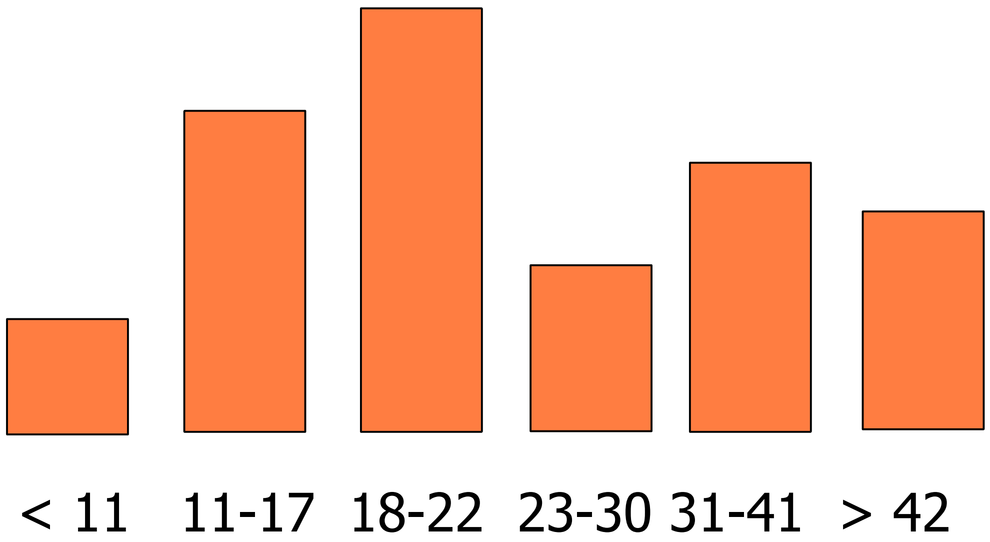
- In the book a number of heuristics for estimating sizes of intermediate results are presented.
- This classical approach works well in some cases, but is unreliable in general.
- The modern approach is based on maintaining suitable statistics summarizing the data. (Focus of lecture.)

Some possible types of statistics

- Random sample of, say 1% of the tuples. (NB. Should fit main memory.)
- The 1000 most frequent values of some attribute, with tuple counts.
- Histogram with number of values in different ranges.
- The *"skew"* of data values. (Not discussed in this lecture.)

Histogram

- Number of values/tuples in each of a number of intervals. Widely used.



- How to use a histogram to estimate selectivity?

On-line vs off-line statistics

- **Off-line**: Statistics only computed periodically, often operator-controlled (e.g. Oracle). Typically involves sorting data according to all attributes.
- **On-line**: Statistics maintained automatically at all times by the DBMS. Focus of this lecture.

Maintaining a random sample

- To get a sample of expected size 1% of full relation:
  - Add a new tuple to the sample with probability 1%.
  - If a sampled tuple is deleted or updated, remember to remove from or update in sample.

Estimating selects

- To estimate the size of a select statement $\sigma_C(R)$:
  - Compute $|\sigma_C(R')|$, where R′ is the random sample of R.
  - If the sample is 1% of R, the estimate is 100 $|\sigma_C(R')|$, etc.
  - The estimate is reliable if $|\sigma_C(R')|$ is not too small (the bigger, the better).

Estimating join sizes?

- Suppose you want to estimate the size of a join statement $R_1 \bowtie R_2$.
- You have random samples of 1% of each relation.
- **Question**: How do you do the estimation?

Estimating join sizes

- Compute $|R'_1 \bowtie R'_2|$, where R'1 and R'2 are samples of R1 and R2.
- If samples are 1% of the relations, estimate is

$$100^2 |R'_1 \bowtie R'_2|$$

Keeping a sample of bounded size

<u>Reservoir sampling (Vitter '85):</u>

- Initial sample consists of s tuples.
- A tuple inserted in R is stored in sample with probability s/(|R|+1).
- When storing a new tuple, it replaces a randomly chosen tuple in existing sample (unless sample has size < s due to a deletion).

Problem session

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.

Tuning

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

Key techniques:

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

Examples from SB

Query rewrite, example 1

- SELECT DISTINCT ssnum
  FROM Employee
  WHERE dept=’CLA’
- 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.

Exercises

- On Thursday morning, we will go through:
  - ADBT exam, June 2006, question 2
  - ADBT exam, June 2005, question 3.b+c