

Query tuning

Rasmus Pagh



Today's lecture

- Round-off on B-trees.
- Part I: Query plans
 - Case studies, focus on joins
 - Key part: Size estimation
- Part II: Query tuning techniques

Round-off on B-trees

- Building a B-tree:
 - Repeated insertion is very slow.
 - “Bulk-loading” sorts the keys, and inserts in order – much more I/O efficient.
 - Sorting normally costs 2 scans over data (reads and writes all data in each scan).
- B⁺-tree locking and next key locking
 - Lock a subset of rows (satisfying a predicate) without locking adjacent rows, by locking B-tree pages.
 - Requires that all access goes via the index.



Query evaluation in a nutshell

How to evaluate a query:

1. Rewrite the query to (extended) relational algebra*, or similar. (Often many possible ways!)
2. Determine algorithms for computing intermediate results in the cheapest way. (There may be several ways!)
3. Execute the algorithms and you have the result!

* Queries with correlated subqueries do not necessarily translate to relational algebra. Can be handled as iterated queries.



Case study

- Relation StarsIn(movie, actor).
Sorted by actor name.

- Find all co-actors:

```
SELECT DISTINCT s1.name, s2.name  
FROM StarsIn s1, StarsIn s2  
WHERE s1.movie=s2.movie
```

- Assume:
 - a clustered index on (movie,actor) exists,
 - there are 10 million rows in StarsIn, and
 - 1 million movies (similar in size to IMDB).

Index nested loop join

- If there is an index that matches the join condition, the following join algorithm can be considered:
 - For each tuple in R_1 , use the index to locate matching tuples in R_2 .
- In general, the cost is at least 1 I/O (hash index) for each tuple in R_1 .
 - Fast only if $|R_1|$ is small (interesting at all?)
 - What can we do for large relations?
- If many tuples may match each tuple, a clustered index is preferable.



Back to case study

- Index nested loop join will create 10 million index lookups:
 - 100,000 seconds at 100 lookups/second
 - And this does not include the time for eliminating duplicates...
- *Buffer manager* may help, depending on lookup order.
- Small test:
 - 500,000 tuples of starsIn (8,000 movies)
 - MySQL used an index on movie
 - Stopped without finishing after 7 hours...



```

db2expln -t -database dbt10 -q "\
select distinct Employee.Emp_No, Employee.name \
from Employee, Booked_On, Assigned_To \
where Employee.name = 'Peters' and Employee.Name = Booked_On.Name and \
Assigned_To.Emp_No = Employee.Emp_No and Assigned_To.Dep_date = Booked_On.Dep_Date \
\
"

Selected rows of query plan:

Access Table Name = ROOT.BOOKED_ON ID = 2,12
| Index Scan: Name = SYSIBM.SQL100217110004260 ID = 1
| | Regular Index (Not Clustered)
| | Index Columns:
| | | 1: NAME (Ascending)
| | | 2: DEP_DATE (Ascending)
| | | 3: NUM (Ascending)
| | Start Key: Inclusive Value
| | | 1: 'Peters'
| | Stop Key: Inclusive Value
| | | 1: 'Peters'
| | Index-Only Access
Hash Join
| Early Out: Single Match Per Outer Row
| Estimated Build Size: 4000
| Access Table Name = ROOT.EMPLOYEE ID = 2,6
| | Index Scan: Name = SYSIBM.SQL100217110003180 ID = 1
| | | Regular Index (Not Clustered)
| | | Index Columns:
| | | 1: NAME (Ascending)
| | Start Key: Inclusive Value
| | | 1: 'Peters'
| | Stop Key: Inclusive Value
| | | 1: 'Peters'
| Nested Loop Join
| | Access Table Name = ROOT.ASSIGNED_TO ID = 2,13
| | | Index Scan: Name = SYSIBM.SQL100217110004360 ID = 1
| | | | Regular Index (Not Clustered)
| | | | Index Columns:
| | | | 1: EMP_NO (Ascending)
| | | | 2: DEP_DATE (Ascending)
| | | | 3: NUM (Ascending)
| | | Start Key: Inclusive Value
| | | | 1: ?
| | | Stop Key: Inclusive Value
| | | | 1: ?
| | | Index-Only Access

```



```

db2expln -t -database dbt10 -q "\
select distinct Employee.Emp_No, Employee.name \
from Employee, Booked_On, Assigned_To, Pilot \
where Employee.Name = Booked_On.Name and Employee.Emp_No = Pilot.Emp_No and \
Assigned_To.Emp_No = Employee.Emp_No and Assigned_To.Dep_date = Booked_On.Dep_Date\
"

Selected rows of query plan:

Access Table Name = ROOT.PILOT ID = 2,7
| May participate in Scan Sharing structures
Hash Join
| Early Out: Single Match Per Outer Row
| Estimated Build Size: 0
| Estimated Probe Size: 0
| Access Table Name = ROOT.ASSIGNED_TO ID = 2,13
| | Index Scan: Name = SYSIBM.SQL100217110004360 ID = 1
| | | Regular Index (Not Clustered)
| | | Index Columns:
| | | | 1: EMP_NO (Ascending)
| | | | 2: DEP_DATE (Ascending)
| | | | 3: NUM (Ascending)
| | | Index-Only Access
Hash Join
| Estimated Build Size: 0
| Estimated Probe Size: 0
| Access Table Name = ROOT.BOOKED_ON ID = 2,12
| | Index Scan: Name = SYSIBM.SQL100217110004260 ID = 1
| | | Regular Index (Not Clustered)
| | | Index Columns:
| | | | 1: NAME (Ascending)
| | | | 2: DEP_DATE (Ascending)
| | | | 3: NUM (Ascending)
| | | Index-Only Access
Hash Join
| Early Out: Single Match Per Outer Row
| Estimated Build Size: 0
| Estimated Probe Size: 0
| Access Table Name = ROOT.EMPLOYEE ID = 2,6
| | Index Scan: Name = SYSIBM.SQL100217110003180 ID = 1
| | | Regular Index (Not Clustered)
| | | Index Columns:
| | | | 1: NAME (Ascending)
Return Data to Application

```



A simple cost estimate

- A reasonably accurate estimate of the cost of a given query plan is that it is proportional to:
 - The size of all intermediate results, plus
 - The cost of accessing the base tables, indexes, and/or materialized views to retrieve the required data.
- The proportionality factor depends on many factors, and is hard to predict.
 - But usually enough to be able to *compare* query plans.

Another 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):

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

- Time proportional to size of the intermediate result $\text{Customer} \bowtie \text{Invoice}$

Example, cont.

- Pushing selects:

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

- 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

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

- 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 $\sigma_{\text{country} = \text{"Sweden"}} (\text{Customer})$

Pushing the other select

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

- 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.



Core problem: Size estimation

- The sizes of intermediate results are important for the choices made when planning query execution.
- Time for operations grows (at least) linearly with size of their inputs.
 - Note that we do not have indexes for intermediate results.
- We will focus on statistical approaches to size estimation.

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.
- Last 10-12 years: "Sketches".

- DB2 command:

```
RUNSTATS ON TABLE R WITH DISTRIBUTION
```

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).



Sampling using a hash function

- Basic idea:
 - Want sampling decision based on the value of a particular attribute A.
 - Use a hash function $h: U \rightarrow (0;1]$ and sample the tuples with hash value at most 0.01 on the value of A (samples 1%).
 - Use same hash function for all relations – the sampling is *dependent*.
- App: Estimating #distinct A-values:
 - $1/v$, where v is the smallest hash value.

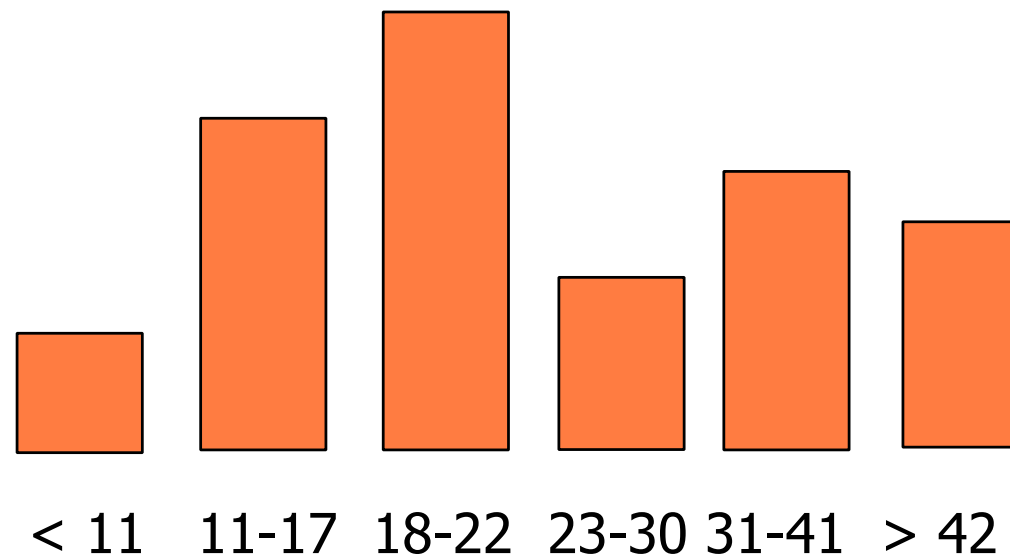
Estimating join size

- Easy for foreign-key joins. (Why?)
- Compute $|R'_1 \bowtie R'_2|$, where R'_1 and R'_2 are *hashing based* samples of R_1 and R_2 on the join attribute.
- If samples are 1% of the relations, estimate is $100|R'_1 \bowtie R'_2|$
 - Observe: *Variance* may be large.



Histogram

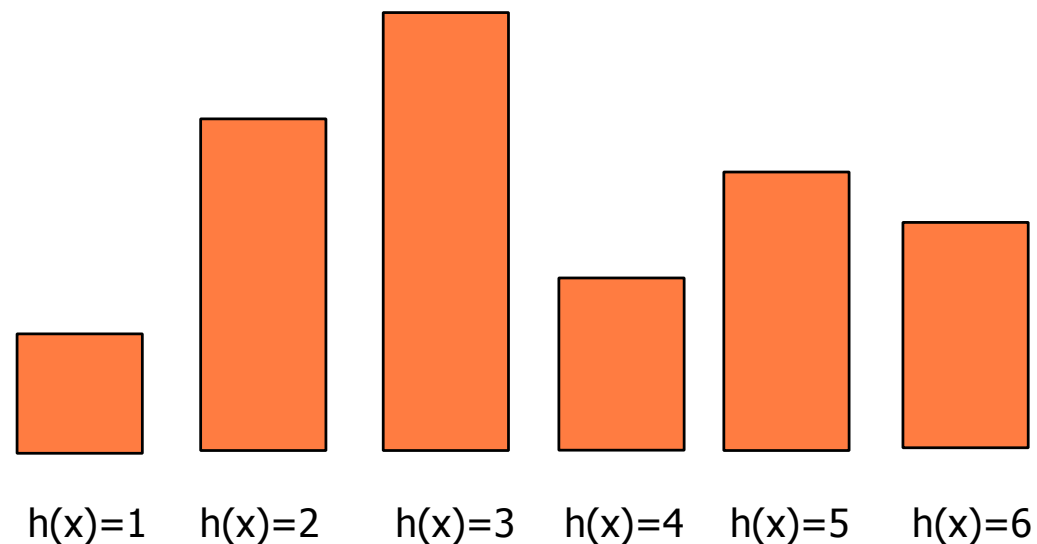
- Number of values/tuples in each of a number of intervals. Widely used.



- Question: How do you use a histogram to estimate selectivity?

Random histograms

- A basic technique used in “sketching” of information is a histogram where the column for each attribute value is chosen at random, using a hash function.



Count-min estimation

Cormode and Muthukrishnan, 2004

- Store k random histograms $X_i[1..n]$, using hash functions h_1, \dots, h_k .
- Estimate of $|\sigma_{A=y}(R)|$ is $\min_i(X_i[h_i(y)])$
 - Never less than true value.
 - The estimate is not too far off if most of the tuples have values from a small set (size n , say) - i.e. there is high *skew*.



Fast-Count estimation

Thorup and Zhang, 2004

- Suppose we want an estimate whose expected value is $|\sigma_{A=y}(R)|$.
- Again, sketch is k random histograms.
- Fast-Count estimation:
 - Subtract the expected "noise" from $X_i[h_i(y)]$.
 - To reduce error, take mean for $i=1, \dots, k$.
- Can also be used for join size estimation.



Lecture part 2: tuning

- *Query optimization* is the DBMSs effort to make a query run as well as possible
- *Query tuning* is a "manual" effort to make single queries (or a whole system) run well.



1st step: Identifying the problem

- A database *profiler* can identify activities on which the DBMS spends time.
- Examples:
 - Number of I/Os performed.
 - Lock queueing time.
 - Buffer cache hit ratio [don't count on it!]
 - "Slow log" of the longest running queries.



Query tuning

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

Key techniques:

- Indexing ✓
- Vertical/horizontal partitioning ✓
- Aggregate maintenance (mat. view) ✓
- Denormalization
- Query rewriting
- 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 example

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

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

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

Query rewrite, example 1

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



Query rewrite, example 2

- `SELECT snum
FROM Employee
WHERE dept IN
 (SELECT dept FROM ResearchDept)`
- Problem: An index on `Employee.dept` may not be used.
- Alternative query:
`SELECT snum
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 ssnnum
FROM Temp
WHERE Temp.dept = 'study admin'
```

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



Query rewrite, example 4

- ```
SELECT snum
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)

- Rewritten query:

```
SELECT snum
FROM Employee E,
 (SELECT dept,
 max(salary) as m
 FROM Employee
 GROUP BY dept) maxsal
WHERE salary=m AND
 E.dept=maxsal.dept
```

# Query rewrite, example 5

- Get top salaries:

```
SELECT EMPNO,SALARY
FROM EMP E1
WHERE 20<=
 (SELECT COUNT(*) FROM EMP E2
 WHERE E2.salary>=E1.salary)
```

- Better to use special "top-k" function (example for DB2):

```
SELECT EMPNO,SALARY
FROM EMP
ORDER BY SALARY DESC
OPTIMIZE FOR 20 ROWS;
```



# Hints

- **Example: Forcing join order (Oracle).**

```
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;
```

- **DB2:**

- Put the plan you like in PLAN\_TABLE
- SET CURRENT OPTIMIZATION HINT = 'planname'

- **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.interestrates=5.6  
      AND bond.dealid = deal.dealid  
      AND deal.date = '7/7/1997'
```
- Clustered index on `interestrates`, nonclustered indexes on `dealid`, and nonclustered index on `date`.
- In absence of accurate statistics, optimizer might use the indexes on `interestrates` 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.



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(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

```
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

Consider tuning of the following query types:

4. `SELECT *`

```
FROM Ships, Classes, Outcomes
WHERE (Outcomes.ship = Ships.name) AND
      (Classes.class = Ships.class) AND
      (Classes.weight > 1000) AND
      (Outcome.date = '1942-08-01');
```

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);
```