Cost based Query Optimization in
(Distributed) Databases

- Cost based QO: parameters
- Cost functions
- Distributed joins:
  - searching the state space
  - brute force
  - hill climbing
- Joins using semi-joins

Example

Part of Geo-DB

Land (name, l_id, ...)
Berg (name, hoeh he, ...)

umfasst (land, kontinent, ...) /* land = l_id

sunLand (...) 
sunUmfasst (...) 
sungeo_Berg (...) 

A complex query: example
"for each country in America find highest mountain and height"

Example Query

SELECT DISTINCT l.name, b.name, b.hoehe, 'm'
FROM sunland l, geo.berg b, sunumfasst u,
geo.geo_berg gb 
WHERE u.kontinent = 'Amerika'
AND u.land = l.l_id
AND u.land = gb.l_id
AND gb.berg = b.name
AND b.hoehe = (SELECT max(b1.hoehe)
FROM geo.berg b1, sungeo_berg gb1
WHERE b1.name = gb1.berg
AND gb1.l_id = u.land)
ORDER BY 3 DESC

Cost based query optimization

- Optimization problems require
  - Search space
  - Search strategy
  - Optimization function

- Query optimization
  - Search space: Query execution plans (usually many for a query)
  - Search strategy: typical is dynamic programming newer technique: randomized
  - Cost function or response time

Cost

- Main influence: size of (intermediate) results
  ... as in centralized system
- Attribute length (length(a))
- number of distinct attribute (a) values of relation R |R.a|
- max(a), min(a): number of distinct values of attribute a
- cardinality of a |dom(a)|
- cardinality of the relation (or fragment) |R|

Important: Selectivity

- Join: sj(R,S) = |R | / |R | * |S|
- value: sj (a=val) = 1 / |R.a| ⇒ σa=val (R) = |R | / |R.a|
- value set S.A of a: sj(S.A) = sj (a e S.A) = |S.A | / |S.a|
Cost parameters

- **Projection**
  \[ |R \cdot a| = |R| \quad \text{if } a \text{ is key} \]

- **Cartesian Product**
  \[ |R \times S| = |R| \cdot |S| \]

- **Union**
  - upper bound: \[ |R \cup S| = |R| + |S| \]
  - lower bound: \[ |R \cup S| = \max\{ |R|, |S| \} \]

- **Set Difference**
  - upper bound: \[ |R - S| = |R| \]
  - lower bound: 0

Cost parameters

- **Join**
  - Special case: \( A \) is a key of \( R \) and \( B \) is a foreign key of \( S \):
    \[ |R \bowtie A = B \cdot S| = |S| \]
  - general:
    \[ |R \bowtie S| = |R| \cdot |S| \] (difficult to determine)

- **Semijoin**
  \[ |R \bowtie A \cdot S| = \text{sf semi} |S.A| \cdot |R| \]
  where
  \[ \text{sf semi} = \frac{|S.A|}{|\text{dom}(A)|} \]
  approximation, exact for \( A \) key in \( S \), foreign key in \( R \), \( \text{sf semi} = 1 \)

Cost parameters

... in practice

- DBS use object level statistics
  - e.g. \( |R| \) and number of blocks of \( R \).
  - # of B-tree levels if indexed.
  - column statistics for each column: min, max value, \(|R.a|\), column histograms

- System statistics
  - CPU speed
  - IO – performance, IO / CPU ratio

Statistics management

- Offline gathering
  - sampling: use only a small number of rows
  - monitoring: refresh if # updates > threshold
  - histogram selection for attribute a dependent on skew an frequency of use of an attribute
- Dynamic sampling during optimization

Histograms for attribute values

- **Histograms**
  Assumption "uniform distribution of data" mostly wrong
  Dynamically collected histograms allow for much better predictions

Different styles:

- width-balanced
- height balanced

Advantage of height balanced histogram for attribute: more exact approximation in ranges with many values
Disadvantage?

Cost functions in distributed DBS

**Cost functions**

Estimation of set (intermediate, result) size as in central, but…

- # IOs may not be best metric
  - **Transmission time** may dominate

work at site 1

Another reason why plain IOs not enough:

- **Parallelism**

Cost measure and parallelism

<table>
<thead>
<tr>
<th>Plan A</th>
<th>Plan B</th>
</tr>
</thead>
<tbody>
<tr>
<td>site 1</td>
<td>50 IOs</td>
</tr>
<tr>
<td>site 2</td>
<td>70 IOs</td>
</tr>
<tr>
<td>site 3</td>
<td>50 IOs</td>
</tr>
</tbody>
</table>

Influence on response time: max (I/O), not \( \sum \) I/O

Same holds for CPU time
Cost metrics in distributed environment

- Cost metrics
  - additive: total cost = total I/O + total CPU + total transmission

- Response time
  - much complexer:
    - task scheduling
    - dependencies
    - do not add!

Cost / Time in distributed (DB) systems

Parallel and distributed environment

- Start up costs (for parallel operation)
- Data distribution costs / time
- Contention
  - memory, disk, network, ...
- Assembling result

Example: Response time

```
Site 1
Site 2
Site 3
Site 4

Startup + Sending results + proc. + Final
```

Searching the search space

- Search space: all valid query execution plans
- Problem:
  - How to transform plans systematically?

Strategies

- Exhaustive (with pruning)
- Hill climbing (greedy)
- Query separation

Exhaustive

- systematic generation of all query plans
  - expensive!
  - unacceptable for ad hoc queries, ok for “canned” queries?
  - \( O(n!) \) different joins of \( n \) relations

Simple Join Ordering

Two relations

- \(|R| < |S|\)
- \(|S| < |R|\)

Multiple relations difficult, too many alternatives.

- Compute the cost of all alternatives and select the best one.
- Necessary to compute the size of intermediate relations which is difficult.
- Use heuristics
Join Ordering

Most important in DDBS: Joins
join n relations at n sites R1 R2 ... Rn
"Bushy" join tree
Linear join tree
if only linear join trees: O(2^n)

Exhaustive search: join ordering

Example: join

R  A  S  T
R->S  T->R

1 Prune because cross-product not necessary
2 Prune because larger relation first
e.g. S >> T has higher cost than T >> S

Hill climbing: join order

Goal: minimize data transmission!

<table>
<thead>
<tr>
<th>Rel</th>
<th>Site</th>
<th>Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>R</td>
<td>1</td>
<td>10</td>
</tr>
<tr>
<td>S</td>
<td>2</td>
<td>20</td>
</tr>
<tr>
<td>T</td>
<td>3</td>
<td>30</td>
</tr>
<tr>
<td>V</td>
<td>4</td>
<td>40</td>
</tr>
</tbody>
</table>

Initial plan: Send all relations to one site. Which one?
To site 1: cost=20+30+40=90
To site 2: cost=10+30+40=80
To site 3: cost=10+20+40=70
To site 4: cost=10+20+30=60

Hill climbing: join order

Goal: minimize data transmission!

Local transformation
Consider sending each relation to neighbor:
e.g.:
**Hill climbing: join order**

**Assume:**

- Size $R \Join S = 20$
- $S \Join T = 5$
- $T \Join V = 1$

Option (a)

- Cost in initial plan = 30
- Cost = 30

No win!

**Hill climbing: join order**

Option (b)

- Cost = 30
- Cost = 40

Worse off!

**Hill climbing: join order**

Option (c)

- Cost = 50
- Cost = 35

Win!

**Hill climbing: join order**

Option (d): Join $S$ with right neighbor in join graph

- Cost = 50
- Cost = 25

Bigger Win!

**Plan 1:**

- $P_{1a}: S (2 \rightarrow 3)$
- $\alpha = S \Join T$
- $P_{1b}: R (1 \rightarrow 4)$
- $\alpha (3 \rightarrow 4)$
- Compute answer at site 4

**Hill climbing: join order – repeated local search**

- Treat $\alpha = S \Join T$ as relation

**Hill climbing may miss best plan!**

**Example:**

- $R \Join S \Join T \Join V$
- Best plan could be

**Plan:**

- $P_e$: $T (3 \rightarrow 4)$
- $\beta = T \Join V$
- $\beta = 4 \rightarrow 2$
- $\beta = \beta \Join S$
- $\beta = \beta \Join R$
- $\beta = \beta (1 \rightarrow 4)$

**Compute answer**

- Costs could be low because is very selective

- 33 = total
Join in DDB: using filters

- Join with filters
  - Do not transfer relations
  - Transfer those attribute values of R needed to perform join with S
  - Use semi-join to find "join-partners"

Join by Semi joins

Use following identity:

\[ R \bowtie_A S = (R \bowtie_A S) \bowtie_A S \]
\[ = R \bowtie_A (S \bowtie_A R) \]
\[ = (R \bowtie_A S) \bowtie_A (S \bowtie_A R) \]

Criterion: \(|R| > |\bigcup_A S| + |R \bowtie_A S| \)

Philosophy: distributed join is hardest
join with join keys only,
pick up other attributes needed later

Conclusion

- Distributed Query Optimization is hard
- Cost based optimizers state of the art
- Huge number of parameters
  - concentrate on important ones, e.g. data transmission
- Heuristics inevitable
- Which goal?
  - Some systems allow to adjust optimizer for minimal response or minimal cost
  - Some systems allow "hints"
  - not only DB implementer must know about QO but also administrator / application programmer