ADB Lecture 4

Query Optimization

- Goal is to select the best available strategy for executing the query
  - Evaluation Plan = query tree + algorithm for each operator + access methods for each relation

* Involves cost-based & heuristic-based approaches

• Heuristic-based Query Optimization

→ apply heuristic rules to modify internal representation of query – to improve its expected performance.

- High-level steps
  - → query parser generates an initial internal representation.
  - → Apply heuristic rules to optimize the internal representation
  - → Based on the access paths (e.g. index) available on the files involved in the query, generate a query evaluation plan.

- Main Heuristic Rule
  - → apply first the the operations that reduce the size of intermediate results.
    (Push down selections, projections to delay cartesian products)

- General Transformation Rules

1. $\sigma_{C_1 \wedge C_2 \wedge \ldots \wedge C_n}(R) = \sigma_{C_1}(\sigma_2(\ldots(\sigma_{C_n}(R))))$ • Cascading Selections

2. $\sigma_{C_1}(\sigma_{C_2}(R)) = \sigma_{C_2}(\sigma_{C_1}(R))$ • Commutativity of Selections

3. $\pi_{a_1}(R) = \pi_{a_1}(\pi_{a_2}(\ldots(\pi_{a_n}(R))))$ • Cascading Projections

   • Requires $a_i \subseteq a_{i+1}$ $\forall i$

4. $\pi_a(\sigma_c(R)) = \sigma_c(\pi_a(R))$ • Selection Projection Comm.

   • Requires that every attribute in c is in a.

5. $R \bowtie_c S = S \bowtie_c R$ • Join and × are Comm. (also, ∪ and ∩)

6. $\sigma_c(R \bowtie_c S) = (\sigma_c(R)) \bowtie_c S$ • Combining σ with ⋈

   • Requires that c uses only attributes from R, else

   $\sigma_c(R \bowtie_c S) = \sigma_{c_1}(R) \bowtie_c \sigma_{c_2}(S)$

   • c_1 uses attrib. only from R, c_2 only from S (not always possible)

7. $\pi_L(R \bowtie_c S) = \pi_{A_1,\ldots,A_N}(R) \bowtie_c \pi_{B_1,\ldots,B_M}(S)$

- If $L = \{A_1,\ldots,A_N, B_1,\ldots,B_M\}$ and A_i only in R, B_i only in S
- if the join condition contains additional attributes A_j, B_j not in L then they must be added to each Projection.

} no Problem if X instead of $\bowtie_c$

8. $R \bowtie (S \bowtie T) = (R \bowtie S) \bowtie T$

- associativity of $\bowtie, \times, \cup, \cap$

9. $\sigma_c(R \times S) = R \bowtie_c S$

- Combine $(\times, \sigma)$ into $\bowtie$

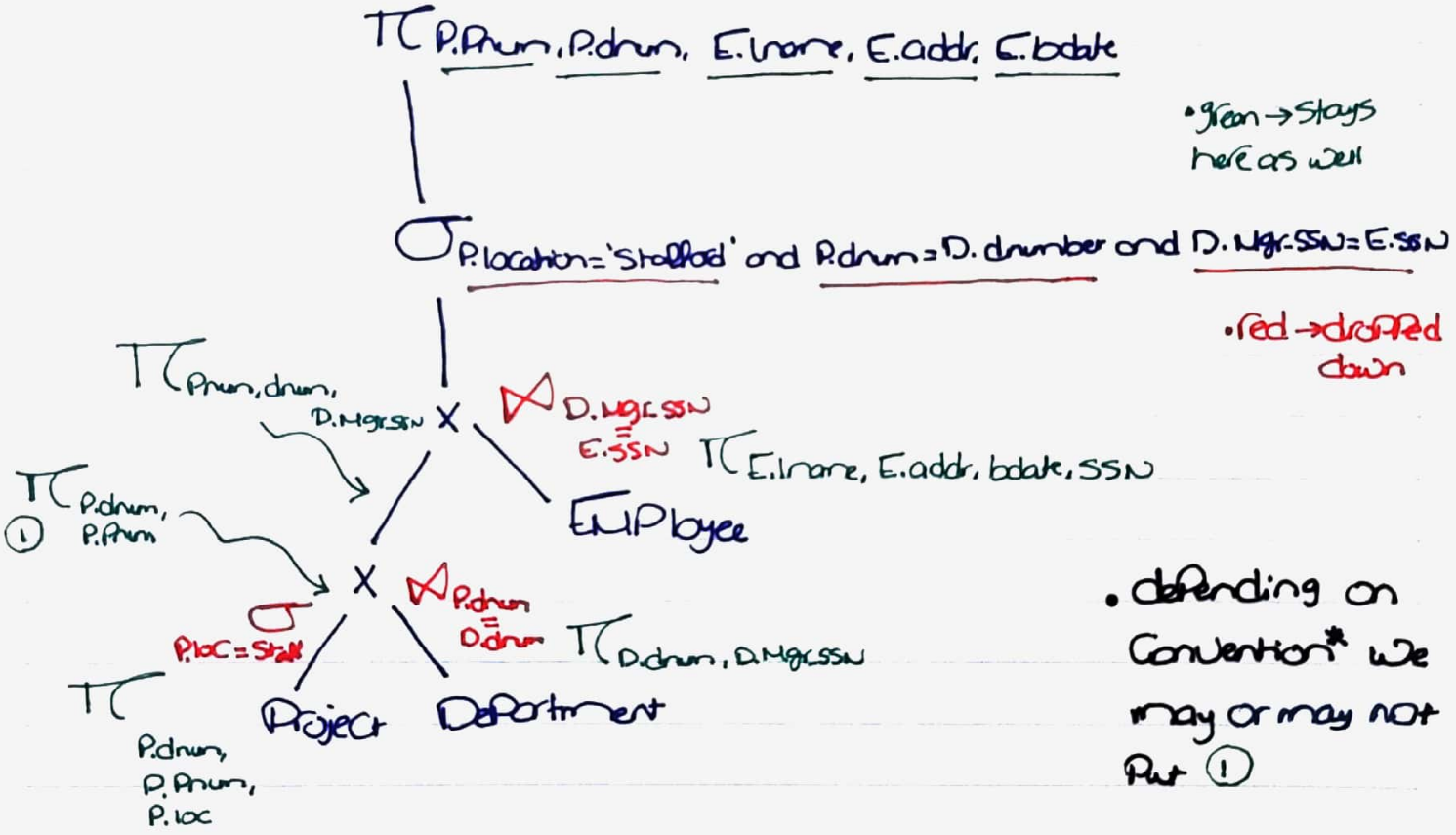
- Heuristic Algebraic Optimization Using Equiv.

1. Use Rule 1 to break up selects
2. Use Rules 2, 4, 6 to move each select operation as far down
3. Use Rules 5, 8 to rearrange tree leaf nodes so that more restrictive selects are executed first.
4. Use Rule 9 to form joins
5. Use Rules 3, 4, 7 to break up and move down Projections (may need to create new π in the process) as far as possible
6. Identify subtrees that represent groups of operations that can be executed by a single algo.

Example

```
Select  P.Pnum, P.dnum, E.Lname, E.addr, E.bdate
From    Project P, department D, employee E
Where   P.dnum = D.dnumber and D.Mgr_SSN = E.SSN
        and P.location = 'Stafford'
```

. initial (canonical query tree) → heuristic optimization



• depending on convention* we may or may not put ①

* Query Execution Plans

(Execution = Evaluation)

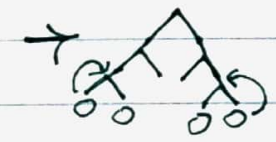
Materialized Evaluation

• the result of an operation is stored as a temporary relation on disk

Pipelined Evaluation

• one result of an operation is produced it's forward to the next operator.

better & faster



} bushy plans

deep plans { → requires a left/right most tree

only write to disk here

- A query optimizer doesn't rely only on heuristics, it also uses

- Cost-based query optimization

1. Estimate the costs of executing a query using different execution strategies
2. Compare and choose strategy with lowest cost estimate

- more suitable for compiled queries (store the strategy along side the query rather than having to figure it out during runtime)

→ Issues
- designing a cost function to estimate the actual cost of executing the query.
- No. of execution strategies to be considered. (usually non-polynomial)

- Estimating the cost of a plan ← query tree +..
  1. For each node estimate the cost of performing the corresponding operation
  2. For each node estimate the size of the result and its properties (eg, sorted)
     → important as result is input to next operator

- (1) is possible because for each access method/ operation we know the cost given input size (cardinality) // depends on (2)

→ An estimate of the entire plan can hence be found by combining estimates at each node

But how do we estimate output size (cardinality)?

• Selectivity Factor

→ Given a query (#tuples)
- Maximum Cardinality of result = Product of Cardinalities of Participating relations (in the worst case where-clause always true and its just a cross Product)

- But the where clause matters
  → Suppose relations A, B, C have cardinalities P_A, P_B, P_C (maximum cardinality of result is $P_A P_B P_C$)

  → then if the Selectivity of each Condition (e.g. 3 of them) in where clause is S_1, S_2, S_3, the total result size can be estimated at
  $$(P_A P_B P_C)(S_1 S_2 S_3)$$
  assumes independence

- Recall, the Selectivity of a Predicate (Condition) in the where clause is the ratio of expected result size to maximum result size

- Selectivity Estimates

  ⇒ Condition is Column = Value

$$S_c = \frac{1}{\#\text{Keys}(\text{Column})}$$

(assumes uniform dist.)

- i.e., if Column takes one of 10 unique values and they're all equally likely then $S_c = P(\text{Col.} = \text{Value}) = \frac{1}{10}$

⇒ Condition is Column1 = Column2

$$S_c = \frac{1}{\max(\#Keys(Col1), \#Keys(Col2))}$$

→ assume that Col1 has more unique values and that each value in Col2 will match with Col1. In this case, each value in Col2 matches with a Probability $\frac{1}{\#Keys(Col1)}$

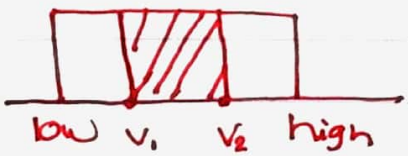
⇒ Condition is Column > Value

$$S_c = \frac{high(Column) - Value}{high(Column) - low(Column)}$$

⇒ Condition is Value1 < Column < Value2

$$S_c = \frac{Value2 - Value1}{high(Column) - low(Column)}$$

• these two also follow from uniform dist. assumption (area under curve)



⇒ Condition is Column in list

$$S_c = \frac{len(list)}{\#Keys(Column)}$$

} For value in list check Column = Value ($P(C=v_1) + P(C=v_2) + \ldots$)

⇒ Condition is Column in Subquery

$$S_c = \frac{|q|}{\#Keys(Column)}$$

- let $|q|$ be estimated size of subquery
- treat like list

⇒ $S_{\bar{c}} = 1 - S_c$ (c is the Predicate, $\bar{c}$ is its negation)

⇒ $S_{c_1 or c_2} = S_{c_1} + S_{c_2} - S_{c_1} \cdot S_{c_2}$

- (assumes indep of c_1, c_2)
- addition rule of Prob.

• We have so far assumed

① Values in a Single Column follow a uniform distribution
- rarely holds
- Can be relaxed with better Statistical methods; namely, histograms.

② Values across Columns are uncorrelated.
(e.g. for indepence of c_1, c_2)
- rarely holds and had to relax

* Histograms

→ Data structure to capture the actual Probability distribution of Column values

→ Trade off between size and accuracy
- larger size → more accuracy but more hard to manipulate

→ Two classes: equi-width & equi-depth

• Desirable Properties: Small, accurate, fast access
  - small: e.g. single block
  - accurate: error ≤ 5%
  - fast access: single look up access

• Equi-Width Histogram

→ total range of Column value is divided into subranges of equal width. (buckets)

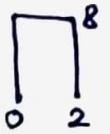
⟶ histogram associates each bucket with the no. of tuples in its range

e.g., if Column value ranges from 0 to 14 then a Possible equi-width histogram is

min	max	Count
0	2	8
3	5	4
6	8	15
9	11	3
12	14	15
		Σ=45

- width = 3
- we will assume that all values in same bucket follow a uniform dist.

i.e.,



$P(0 \le V \le 2) = \frac{8}{45}$

$P(V=0) = P(V=1) = P(V=2)$

$= \frac{8}{45} \cdot \frac{1}{3}$

$$S_{6 \le V \le 10} = P(6 \le V \le 10)$$

$$= \underbrace{\frac{3}{3} \cdot \frac{15}{45}}_{6 \le V < 8} + \underbrace{\frac{2}{3} \cdot \frac{3}{45}}_{9 \le V \le 10} = \frac{17}{45}$$

- if we were to assume a uniform dist.

$$S_{6 \le V \le 10} = \frac{10-6}{14-0} = \frac{12}{42}$$

it gets much worse if there's an outlier at 30.

* Note that from this point onwards, S_c will denote the output size and $\hat{S}_c$ will denote selectivity (s_L in slides)

- Equi-depth Histogram

→ The total range of the Column value is divided into subranges (buckets) such that the no. of tuples in each is approx. equal.

min	max	Count
0	3	8
4	7	10
8	9	10
10	13	7
14	14	9
		T=44

→ to construct, set a threshold and look for smallest subrange satisfying it. (x ≥ 7)

Some estimation algo.

$$S_{6 \le V \le 10} = \frac{2}{4} \cdot \frac{10}{44} + \frac{2}{2} \cdot \frac{10}{44} + \frac{1}{4} \cdot \frac{7}{44} = \frac{17}{44}$$

4 ≤ V ≤ 7 | 8 ≤ V ≤ 9 | 10 ≤ V ≤ 10

→ Can be proven to be more accurate than equi-width. (esp. continuous data* with no outliers)

- Catalog Information used in Cost Functions

→ (to estimate cost of various exec. steps)

Info about File Size
→ no. of records (r)
→ record size (R)
→ no. of blocks (b)
→ blocking factor (bfr)

Info about indexes
→ No. of levels (x)
→ No. of 1st level index blocks (bI1)
→ No. of distinct values of attribute (d)
→ Selectivity of attribute given condition ($\hat{S}$)
→ Selection cardinality of attr. ($\hat{S} * r = S$)

C_{Si}, Si = method

- Examples of Cost Functions for Select (in terms of #IOs of blocks)

S1. Linear Search by brute force

→ Search all File blocks (non-Key or not found)

$$C_{S1} = b$$

→ if it's a Key (no duplicates)

$$C_{S1} = b/2$$

S2. Binary Search

→ Non-Key

$\checkmark$ to reach 1st block

$$C_{S2} = \log_2 b + \left\lceil \frac{S}{bfr} \right\rceil - 1$$

to read the remaining blocks

- if block holds 10 records and output cardinality S is 50 records then 4 more blocks to read

→ Key

$$C_{S2} = \log_2 b$$

(even if unique, S = 1 and formula above holds)

S3a. Use a Primary index to retrieve a single record

$$C_{S3a} = X + 1$$

- X: to reach 1st index level (leaves)
- 1: Final I/O to get needed data block

S3b. Use a hash key to retrieve a single record

$$C_{S3b} = 1 \quad \text{(linear hashing)}$$

- Read the right bucket

$$C_{S3b} = 2 \quad \text{(extendible hashing)}$$

- directory is also a block

S4. Using an Ordering index to retrieve multiple records

Clustered or Prim.

$$C_{S4} = X + \frac{b}{2}$$

- X: to reach leaves
- $\frac{b}{2}$: assuming an inequality condition, roughly, on avg. half records qualify.

S5. Using a Clustered index to retrieve mult. records

$$C_{S5} = X + \lceil S/bfr \rceil$$

- like about but S not necessarily $\frac{1}{2}r$ ($S_c = \frac{1}{2}$ not forced).

S6a. Using a secondary index for equality search

$$C_{S6a} = X + S$$

- S: block access for each qualifying record in index.

- replace S with S+1 if block of pointers (unique index)

S6b. Secondary Index for multiple records

<, ≤, >, ≥,

$$C_{S6b} = X + \frac{b_{I1}}{2} + \frac{r}{2}$$

- X: to reach leaves
- $\frac{b_{I1}}{2}$: 1/2 of block's quality (leaves) (reach blocks of pointers)
- $\frac{r}{2}$: and they correspond to 1/2 the records
  - i.e., r/2 block pointer on avg.

S7. Conjunctive Selection

→ Use S1 (linear search)

or

→ Use one of the conditions to retrieve the records (S2 to S6)
- (which condition depends on sort/hash options) then check the rest of the conditions in memory.

S8. Conjunctive Selection using Composite index

→ Special case of S3a, S5, S6a

Example:

$r = 10000 \quad b = 2000 \quad bfr = 5$

Access Paths:
- Clustering index on salary with 3 levels and $S_{Sal} = 20$
- Secondary index on key attrib. SSN with 4 levels and $S_{SSN} = 1$

Query: $\sigma_{SSN=1234}(Employee)$

→ S_1: $C_{S1} = b/2 = 1000$

→ S_{6a}: $C_{S6a} = X + 1 = 4 + 1$

* The cost model for cost-based optimization in our case is #I/O
  → depends on cardinalities of inputs hence its a Cardinality-based cost model

Trivial

. Cost-based Query Optim. = Plan enumeration + Selection (min cost)

The combinations of these two define the search space.

- access method selection
- join algorithm selection (enum.)

• Search space is huge
• Plan enumeration = search space exploration

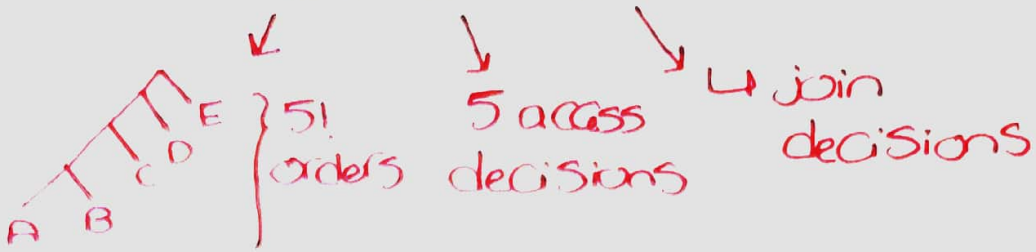
→ exhaustive exploration is hence a bad idea
  • often takes more than just evaluating the query

• Query optimization method = way to explore search space

- Dynamic Programming
- Randomised exploration
- Rule-based opt.

→ Why not just brute force?
  • Consider
    5 relations, 2 access methods for each and 3 possible join algorithms with left-deep plan

- In this case there are $5! \times 2^5 \times 3^4$ Possible Plans



* Dynamic Programming

1. Identify Cheapest way to access each relation given local Predicates (Conditions)
   - if one of the ways isn't cheapest overall but cheapest w.r.t an interesting order (helpful in future) then keep it as well. ①

2. For every relation and for every join condition, find the cheapest way to join in a second
   - ①
3. Keep applying Principle 2 until Costs for the whole Plan is found

Example:

- Want to optimally join A,B,C,D (3 available join algos)

1st iteration:

→ Find best way to access A, B, C, D

Cache	best	Cost
A	Index	100
B	SeqScan	50
C	...	
D		

- 2nd iteration
  - → 4 relations ⇒ $^4C_2$ Possible Pairs (joins)

{A, B} ← A⋈B or B⋈A
{A, C}
{A, D}
{B, C}
{B, D}
{C, D}

} given the best access methods estimate best way to join each (order of relations and join method)
e.g.
{A,B} → A⋈B with sort merge
... record all with the cost in cache.

- if 3 join methods, we need $3 \times 2 \times {}^4C_2$ Computations and $^nC_2$ results

(4 is no. of rel.)

- 3rd iteration
  - → $^4C_3$ Possible trios and each ordered in $2 \times 3 =$ 6 Possible ways (with 3 join Possible for each)

{} • exec A⋈B or C

e.g. {A,B,C}
exec. B → B{A,C}
exec C → {A,B}C
exclude A → A{B,C}
{A,C}B
{B,C}A
C{A,B}

- The SubProblem in { } has already been Solved!

e.g. (A⋈B)⋈C
sort merge, block nested

* Find best way to Compute {A,B,C}
  → 6 × 3 Possibilities
  (6 = # join methods)

- In total, $^4C_3 \times 6 \times 3$ Computations and $^4C_3$ results

- 4th iteration
  - → $^4C_4 \times 4 \times 2 \times 3$ Computations and $^4C_4$ Outputs
    - only {A,B,C,D}
    - exclude A, B, C, D
    - joins
    - before or after {}

- Clearly, final result gives the best plan.

→ Called Selinger Optimization

R ← Set of relations to join

For l in 1, 2, ..., |R|

For each subset S of length l of R

① OPTjoin(S) = a.join(S-a)

where a is the relation that minimizes Cost(OPTjoin(S-a)) + min cost to join (S-a) and a + min cost to access a

① e.g. For the subset {A, B, C} we choose to join one of the three (a) with optimal result of the other two (S-a). while taking into account the cost of the new join and cost to access a.

- Complexity

→ $\sum_{i=1}^{n} {}^{n}C_i$ Subsets are considered in total $= 2^n - 1$

- the 'Powerset' mathematically includes all subsets even {} and has 2^n elements

- if we take it to be 2^n, this equivalent to the no. of ways to write a binary of length n.

- For each subset length (K) we have K possibilities
  (removing each of the relations and using previous subplans)
  → K < n
  → Suppose we have m ways to evaluate the join (6 in prev. example)
  ↙ ↘
  Comm. 3 methods

$$\text{Total Cost} = O(nm2^n)$$

→ Interesting Orders
- non-optimal joins producing sorted data can help in future (e.g. if final result will be sorted)
- In cache maintain best overall plan and best interesting order plan

⇒ Complexity becomes $O(2nm2^n)$

- double evaluations due to interesting order

* Suppose there were I interesting orders to consider

$$O((I+1)nm2^n)$$

- each time I+1 costs are estimated.

* Rule-based Optimization

- An issue of if-then rules
  if (condition_list) // Not where clause
  apply transformation on plan so far.
  → keep the new plan only if cheaper than original

- Order of rules matter (Precedence)

higher Precedence →
Push σ down | Use Cartesian

* Randomised Exploration

→ Useful in big queries with huge search spaces

→ to explore a bigger portion of it; let optimizer jump to a far state with some probability (periodically).

. Otherwise can get stuck in a smaller region of search space.

. Oracle DBMS V8

Uses

→ Rule-based optimization (through heuristics)

→ Cost-based optimization (optimize IO, CPU & memory)

→ allow developer hints to optimizer (they might know more)

Semantic Query Optimization

- uses constraints from DB schema (e.g. unique attr.)
- to modify query so its more efficient to execute (generally).

e.g. query with EMP.salary > Manag.salary

. There is an insertion constraint that employee can't have higher salary

→ hence semantic query optimizer knows this has $S = 0$