Query Optimization

Using Heuristics in Query Optimization

- Process for heuristics optimization
  - The parser of a high-level query generates an initial internal representation;
  - Apply heuristics rules to optimize the internal representation.
  - A query execution plan is generated to execute groups of operations based on the access paths available on the files involved in the query.
- The main heuristic is to apply first the operations that reduce the size of intermediate results.
  - E.g., Apply SELECT and PROJECT operations before applying the JOIN or other binary operations.

Example

- Example:
  - For every project located in 'Stafford', retrieve the project number, the controlling department number and the department manager's last name, address and birthdate.
- Relation algebra:

$$\pi_{\text{Pnumber, Dnum, Lname, Address, Bdate}}\,(((\sigma_{\text{Plocation='Stafford'}}(\text{PROJECT}))$$
$$\bowtie_{\text{Dnum=Dnumber}}(\text{DEPARTMENT})) \bowtie_{\text{Mgr\_ssn=Ssn}}(\text{EMPLOYEE}))$$

- SQL query:

```
SELECT   P.Pnumber, P.Dnum, E.Lname, E.Address, E.Bdate
FROM     PROJECT P, DEPARTMENT D, EMPLOYEE E
WHERE    P.Dnum=D.Dnumber AND D.Mgr_ssn=E.Ssn AND
         P.Plocation= 'Stafford';
```

Example (cont'd)

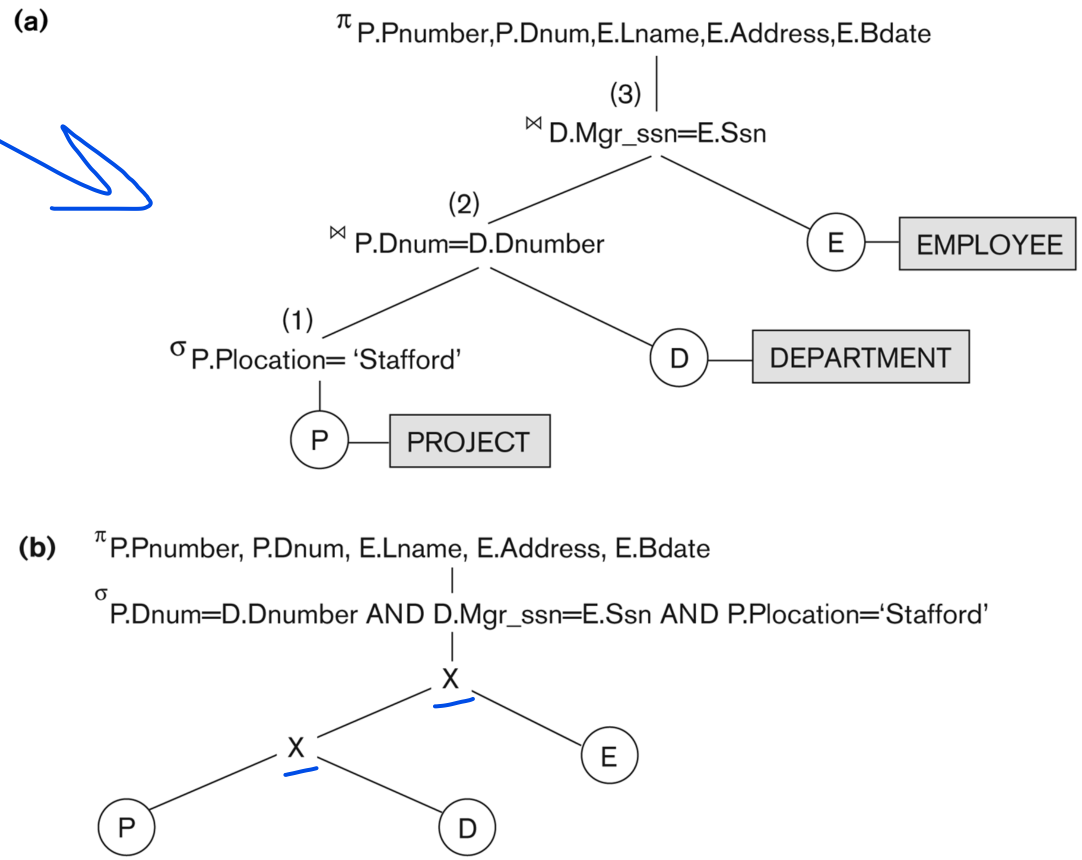


Figure 15.4
Two query trees for the query Q2. (a) Query tree corresponding to the relational algebra expression for Q2. (b) Initial (canonical) query tree for SQL query Q2. (c) Query graph for Q2.

Selection and projections

- Cascading of selections
  - $\sigma_{c_1 \wedge c_2 \wedge \ldots \wedge c_n}(R) \equiv \sigma_{c_1}(\sigma_{c_2}(\ldots(\sigma_{c_n}(R))))$
- Commutativity
  - $\sigma_{c_1}(\sigma_{c_2}(R)) \equiv \sigma_{c_2}(\sigma_{c_1}(R))$
- Cascading of projections
  - $\pi_{a_1}(R) \equiv \pi_{a_1}(\pi_{a_2}(\ldots(\pi_{a_n}(R))\ldots)$
  - iff $a_i \subseteq a_{i+1}$, $i = 1, 2, \ldots n-1$

Cartesian products and joins

- Commutativity
  - $R \times S \equiv S \times R$
  - $R \bowtie S \equiv S \bowtie R$
- Assosiativity
  - $R \times (S \times T) \equiv (R \times S) \times T$
  - $R \bowtie (S \bowtie T) \equiv (R \bowtie S) \bowtie T$
- Their combination
  - $R \bowtie (S \bowtie T) \equiv R \bowtie (T \bowtie S) \equiv (R \bowtie T) \bowtie S$
    $\equiv (T \bowtie R) \bowtie S$

Other operations

- Selection-projection commutativity
  - $\pi_a (\sigma_c (R)) \equiv \sigma_c (\pi_a (R))$
  - iff *every attribute in c* is *included* in the *set of attributes a*
- Combination (join definition)
  - $\sigma_c (R \times S) \equiv R \bowtie_c S$
- Selection-Cartesian/join commutativity
  - $\sigma_c (R \times S) \equiv \sigma_c(R) \times S$
  - iff the *attributes in c* appear *only in R* and *not in S*
- Selection distribution/replacement
  - $\sigma_c(R \bowtie S) \equiv \sigma_{c_1 \wedge c_2} (R \bowtie S) \equiv \sigma_{c_1} (\sigma_{c_2} (R \bowtie S)) \equiv \sigma_{c_1}(R) \bowtie \sigma_{c_2}(S)$
  - iff c_1 *is relevant only to R* and c_2 *is relevant only to S*

Other operations (cont.)

- Projection-Cartesian product commutativity
  - $\pi_a (R \times S) \equiv \pi_{a_1}(R) \times \pi_{a_2}(S)$
  - iff a_1 *is the subset of attributes in a appearing in R* and a_2 *is the subset of attributes in a appearing in S*
- Projection-join commutativity
  - $\pi_a (R \bowtie_c S) \equiv \pi_{a_1}(R) \bowtie_c \pi_{a_2}(S)$
  - iff *same as before* and *every attribute in c appears in a*
- Attribute elimination
  - $\pi_a(R \bowtie_c S) \equiv \pi_a(\pi_{a_1}(R) \bowtie_c \pi_{a_2}(S))$
  - iff a_1 *subset of attributes in R appearing in either a or c* and a_2 *is the subset of attributes in S appearing in either a or c*

Query Optimization using Equivalence Rules

1. Break up any select operations with conjunctive conditions into a cascade of select operations.
2. Move each select operation as far down the query tree as is permitted by the attributes involved in the select condition.
3. Rearrange the leaf nodes of the tree so that the leaf node relations with the most restrictive select operations are executed first in the query tree representation.
4. Combine a Cartesian product operation with a subsequent select operation in the tree into a join operation.
5. Break down and move lists of projection attributes down the tree as far as possible by creating new project operations as needed.
6. Identify subtrees that represent groups of operations that can be executed by a single algorithm.

Query Execution Plans

- An execution plan for a relational algebra query consists of a combination of the relational algebra query tree and information about the access methods to be used for each relation as well as the methods to be used in computing the relational operators stored in the tree.
- **Materialized evaluation**: the result of an operation is stored as a temporary relation.
- **Pipelined evaluation**: as the result of an operator is produced, it is forwarded to the next operator in sequence.

Using Selectivity and Cost Estimates in Query Optimization

- **Cost-based query optimization**:
  - Estimate and compare the costs of executing a query using different execution strategies and choose the strategy with the lowest cost estimate.
  - More suitable for compiled queries (Stored Procedure).
  - (Compare to heuristic query optimization)
- Issues
  - Cost function
  - Number of execution strategies to be considered

Cost estimation

- A *plan* is a tree of operators
- *Two parts* to *estimating* the *cost* of a plan
  - For *each node*, estimate the *cost* of *performing* the corresponding *operation*
  - For *each node*, *estimate* the *size* of the *result* and any *properties* it might have (*e.g.*, sorted)
- *Combine* the *estimates* and *produce* an *estimate* for the *entire plan*
- We have seen *various storage methods* and *algorithms*
  - And *know the cost* of *using each* one, *depending* on the *input cardinality*
- The *problem* is *estimating* the *output cardinality* of the *operations*
  - Namely, *selections* and *joins*

Selectivity factor

- The *maximum number of tuples* in the *result* of any *query* is the *product* of the *cardinalities* of the *participating relations*
- Every *predicate* in the *where-clause eliminates some* of these *potential results*
- *Selectivity factor* of a *single predicate* is the *ratio* of the *expected result size* to the *maximum result size*
- *Total result size* is *estimated* as the *maximum size times* the *product* of the *selectivity factors*

Various selectivity factors

- *column = value* $\rightarrow \dfrac{1}{\#keys(column)}$
  - Assumes *uniform distribution* in the values
  - Is itself an *approximation*
- $column_1 = column_2 \rightarrow \dfrac{1}{\max(\#keys(column_1), \#keys(column_2))}$
  - *Each value* in $column_1$ has a *matching value* in $column_2$; *given* a *value* in $column_1$, the *predicate* is just a *selection*
  - Again, an *approximation*

Various selectivity factors (cont.)

- *column > value* $\rightarrow \frac{(high(column) - value)}{(high(column) - low\ (column))}$
- $value_1 < column < value_2 \rightarrow \frac{(value2 - value1)}{(high(column) - low\ (column))}$
- *column in list* → *number of items in list times s.f. of column = value*
- *column in sub-query* → *ratio* of *subquery's estimated size* to the *number of keys* in column
- *not (predicate)* → *1 - (s.f. of predicate)*
- $P_1 \vee P_2 \rightarrow f_{P_1} + f_{P_2} - f_{P_1} \cdot f_{P_2}$

Key assumptions made

- The *values across columns* are *uncorrelated*
- The *values* in a *single column* follow a *uniform distribution*
- *Both* of these assumptions *rarely hold*
- The *first assumption* is *hard to lift*
  - Only recently have researchers started tackling the problem
- The *uniform distribution* assumption can be *lifted* with *better statistical methods*
  - In our case, *histograms*

Histograms

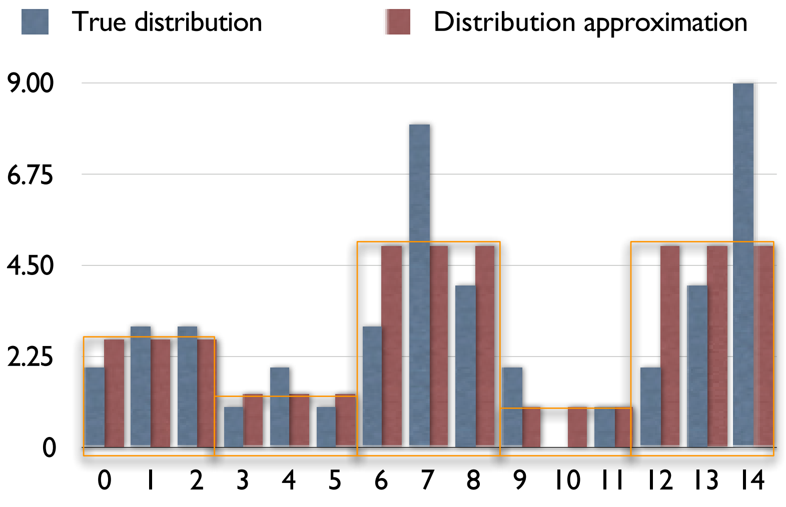
- *Elegant data structures* to *capture value distributions*
  - *Not affected* by the *uniform distribution* assumption (though this is *not entirely true*)
- They offer *trade-offs* between *size* and *accuracy*
  - The *more memory* that is dedicated to a histogram, the *more accurate* it is
  - But also, the *more expensive* to manipulate
- *Two* basic classes: *equi-width* and *equi-depth*

Desirable histogram properties

- *Small*
  - Typically, a DBMS will allocate a *single page* for a histogram!
- *Accurate*
  - Typically, less than *5% error*
- *Fast access*
  - *Single lookup* access and *simple algorithms*

Equi-width histogram construction

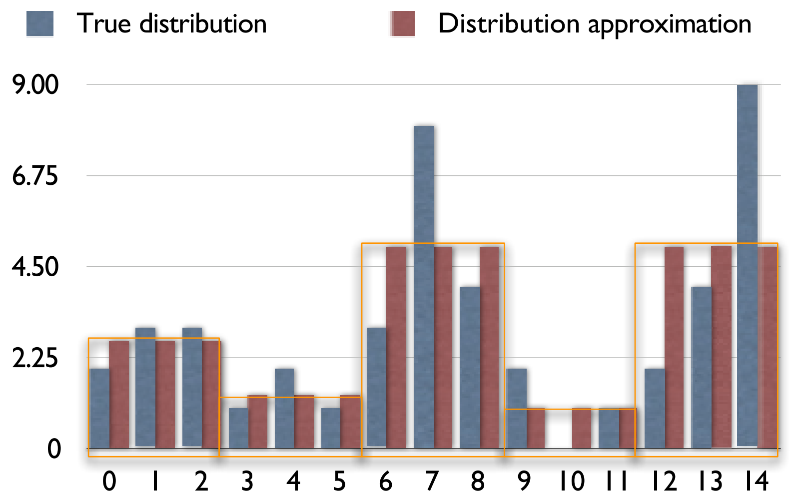
- The *total range* is *divided* into *sub-ranges* of *equal width*
- Each *sub-range* becomes a *bucket*
- The *total number of tuples* in *each bucket* is *stored*



min	max	count
0	2	8
3	5	4
6	8	15
9	11	3
12	14	15

Equi-width histogram estimation

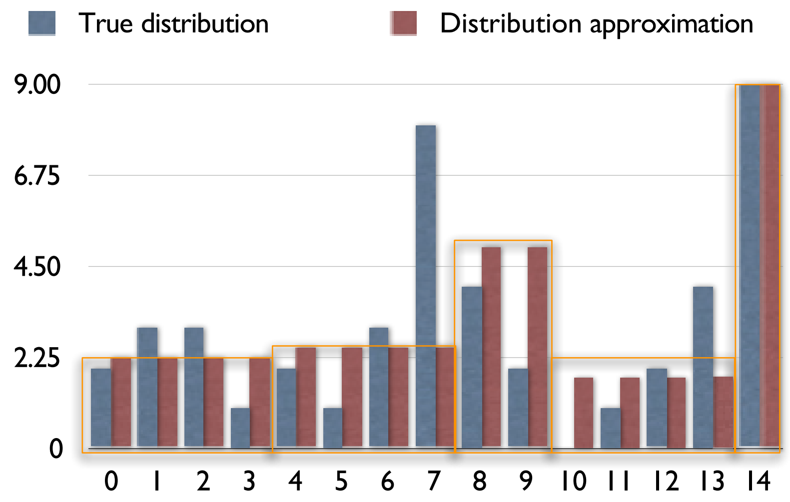
- To *estimate* the output cardinality of a range query
  - The *starting bucket* is *identified*
  - The *histogram* is then *scanned forward* until the *ending bucket* is *identified*
  - The *numbers of tuples* in the *buckets* of the range are *summed*
  - *Within each bucket* the *uniform distribution assumption* is made
- $6 \leq v \leq 10$: $\frac{3}{3} \cdot 15 + \frac{2}{3} \cdot 3 = 17$



min	max	count
0	2	8
3	5	4
6	8	15
9	11	3
12	14	15

Equi-depth histogram construction and estimation

- The *total range* is *divided* into *sub-ranges* so that the *number of tuples* in *each range* is (approximately) *equal*
- Each *sub-range* becomes a *bucket*
- The *same schema* as in *equi-width* histograms is used
- In fact, the *same algorithm* is used for *estimation* (!)
- $6 \leq v \leq 10$:
  $\frac{2}{4} \cdot 10 + \frac{2}{2} \cdot 10 + \frac{1}{4} \cdot 7 \approx 17$



min	max	count
0	3	8
4	7	10
8	9	10
10	13	7
14	14	9

Using Selectivity and Cost Estimates in Query Optimization

- Catalog Information Used in Cost Functions
  - Information about the size of a file
    - number of records (tuples) (r),
    - record size (R),
    - number of blocks (b)
    - blocking factor (bfr)
  - Information about indexes and indexing attributes of a file
    - Number of levels (x) of each multilevel index
    - Number of first-level index blocks (bI1)
    - Number of distinct values (d) of an attribute
    - Selectivity (sl) of an attribute
    - Selection cardinality (s) of an attribute. (s = sl * r)

Using Selectivity and Cost Estimates in Query Optimization (Cont'd)

- Examples of Cost Functions for SELECT (in terms of block transfers)
- S1. Linear search (brute force) approach
  - $C_{S1a} = b$;
  - For an equality condition on a key, $C_{S1a} = (b/2)$ if the record is found; otherwise $C_{S1a} = b$.
- S2. Binary search:
  - $C_{S2} = \log_2 b + \lceil (s/bfr) \rceil - 1$
  - For an equality condition on a unique (key) attribute, $C_{S2} = \log_2 b$
  - Where s is the selection cardinality
- S3. Using a primary index (S3a) or hash key (S3b) to retrieve a single record
  - $C_{S3a} = x + 1$; $C_{S3b} = 1$ for static or linear hashing;
  - $C_{S3b} = 2$ for extendible hashing;
  - Where x is number of index levels

Using Selectivity and Cost Estimates in Query Optimization (Cont’d)

- Examples of Cost Functions for SELECT (contd.)
- S4. Using an ordering index to retrieve multiple records:
  - For the comparison condition on a key field with an ordering index,
  - $C_{S4} = x + (b/2)$
- S5. Using a clustering index to retrieve multiple records:
  - $C_{S5} = x + \lceil (s/bfr) \rceil$
- S6. Using a secondary (B+-tree) index:
  - For an equality comparison, $C_{S6a} = x + s$;
  - For an comparison condition such as >, <, >=, or <=,
  - $C_{S6a} = x + (b_{I1}/2) + (r/2)$

Using Selectivity and Cost Estimates in Query Optimization (Cont'd)

- Examples of Cost Functions for SELECT (contd.)
- S7. Conjunctive selection:
  - Use either S1 or one of the methods S2 to S6 to solve.
  - For the latter case, use one condition to retrieve the records and then check in the memory buffer whether each retrieved record satisfies the remaining conditions in the conjunction.
- S8. Conjunctive selection using a composite index:
  - Same as S3a, S5 or S6a, depending on the type of index.

Example of Cost Functions for SELECT

- EMPLOYEE file has $r=10{,}000$ records stored in $b=2000$ blocks with $bfr=5$ records/block and the following access path:
  - A clustering index on Salary with levels $x_{Salary}=3$ and average selection cardinality $s_{Salary}=20$.
  - A secondary index on the key attribute Ssn with $x_{ssn}=4$ ($s_{ssn}=1$)
- For the statement $\text{Sigma}_{ssn=123456789}(\text{Employee})$
- Using S_1: Cost = $b/2=1000$
- Using S_{6a}: Cost = $x_{ssn}+1 = 4+1 = 5$

Cost-based query optimisation

- The *paradigm* employed is *cost-based query optimisation*
  - Simply put: *enumerate* alternative *plans*, *estimate* the *cost* of *each plan*, *pick* the *plan* with the *minimum cost*
- For *cost-based optimisation*, we need a *cost model*
  - Since *what "hurts"* performance *is I/O*, the *cost model* should *use I/O* as its *basis*
  - Hence, the *cardinality-based cost model*
    - *Cardinality* is the *number* of *tuples* in a *relation*

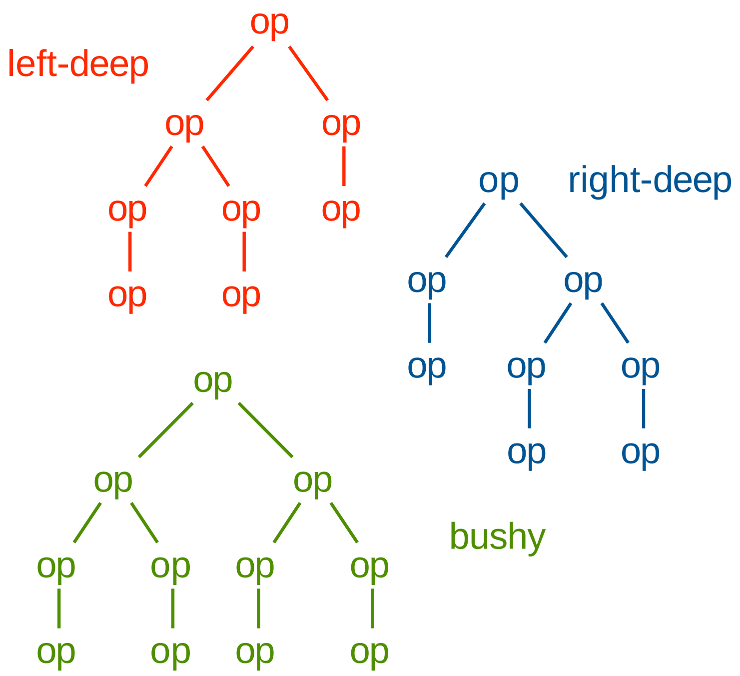
Plan enumeration

- *Plan enumeration* consists of *two parts* (again, *not necessarily independent* from one another)
  - *Access method selection* (*i.e.*, what is the *best way* to *access a relation* that appears in the query?)
  - *Join enumeration* (*i.e.*, what is the *best algorithm* to *join* two relations, and *when should we apply it*?)
- *Access methods*, *join algorithms* and their various *combinations* define a *search space*
  - The *search space* can be *huge*
  - *Plan enumeration* is the *exploration* of this *search space*

Search space exploration

- As was stated, the *search space* is *huge*
  - *Exhaustive exploration* is *out of the question*
  - Because it *could be the case* that *exploring* the search space might *take longer than* actually *evaluating* the query
  - The *way* in which we *explore* the *search space* describes a *query optimisation method*
    - *Dynamic programming*, *rule-based* optimisation, *randomised* exploration, . . .

Types of plan



- There are *two types of plan*, *according to their shape*
  - *Deep* (*left or right*)
  - *Bushy*
- *Different shapes* for *different objectives*

Just an idea . . .

- A query over *five relations*, only *one access method*, only *one join algorithm*, only *left-deep plans*
  - Remember, *cost*($R \bowtie S$) != *cost*($S \bowtie R$)
  - So, the number of *possible plans* is $5! = 120$
  - If we add *one extra access method*, the number of *possible plans* becomes $2^5 \cdot 5! = 3840$
  - If we add one *extra join algorithm*, the number of *possible plans* becomes $2^4 \cdot 2^5 \cdot 5! = 61440$

Cardinality-based cost model

- A *cardinality-based cost model* means we need *good ways of* doing the following
  - *Using cardinalities* to *estimate costs* (*e.g.*, accurate cost functions)
  - *Estimating output cardinalities after* we apply *certain operations* (*e.g.*, after a selection the cardinality will change; it will not change after a projection)
    - *Because* these *output cardinalities will be used as inputs* to the *cost functions* of *other operations*

Rule-based optimisation

- *Basically* an issue of *if-then rules*
  - *If* (*condition list*) *then apply some transformation* to the plan constructed so far
    - *Estimate* the *cost* of the *new plan*, *keep* it *only if* it is *cheaper than* the *original*
  - The *order* in which the *rules are applied* is *significant*
  - As a *consequence*, rules are applied *by precedence*
    - For instance, *pushing down selections* is given *high precedence*
    - Combining two relations with a *Cartesian product* is given *low precedence*

Randomised exploration

- *Mostly useful* in *big queries* (more than 15 joins or so)
- The *problem* is one of *exploring a bigger portion* of the search space
  - So, *every once in a while* the *optimiser “jumps”* to some *other part* of the search space *with some probability*
- As a *consequence*, it gets to *explore parts* of the search space it would *not have explored otherwise*

Dynamic programming

- In the beginning, there was *System R*, which had an *optimiser*
- *System R's optimiser* was using *dynamic programming*
  - An *efficient way* of *exploring* the search space
- *Heuristics*: use the *equivalence rules* to *push down selections* and *projections*, *delay Cartesian products*
  - *Minimise input cardinality* to, and *memory* requirements of the *joins*
- *Constraints*: *left-deep plans*, *nested loops* and *sort-merge join* only
  - *Left-deep plans* took better *advantage of pipelining*
  - *Hash-join* had *not* been *developed* back then

Dynamic programming steps

- *Identify* the *cheapest* way to *access every* single *relation* in the query, *applying local predicates*
  - For *every relation*, *keep* the *cheapest access method overall* and the *cheapest access method* for an *interesting order*
- For *every access method*, and for *every join predicate*, find the *cheapest way* to *join* in a *second relation*
  - For *every join result keep* the *cheapest plan overall* and the *cheapest plan* in an *interesting order*
- *Join* in the *rest* of the *relations* using the *same principle*

Selinger Optimizer

R ← set of relations to join (e.g., ABCD)

For ∂ in {1…|*R*|}:

  for *S* in {all length ∂ subsets of *R*}:

    optjoin(S) = *a* join (*S*-*a*),

      where *a* is the single relation that minimizes:

        cost(**optjoin**(*S*-*a*)) +

        min. cost to join (*S*-*a*) to *a* +

        min. access cost for *a*

optjoin(*S*-*a*) is cached from previous iteration

Example

Cache

Subplan	Best choic e	Cost
A	index	100
B	seq scan	50
…		

optjoin(ABCD) – assume all joins are NL

∂=1

A = best way to access A

(e.g., sequential scan, or predicate pushdown into index...)

B = best way to access B

C = best way to access C

D = best way to access D

Total cost computations: *choose*(N,1), where N is number of relations

Example

Cache

Subplan	Best choice	Cost
A	index	100
B	seq scan	50
{A,B}	BA	156
{B,C}	BC	98
…		

optjoin(ABCD)

∂=2

{A,B} = AB or BA

(using previously computed best way to access A and B)

{B,C} = BC or CB

{C,D} = CD or DC

{A,C} = AC or CA

{A,D} = AD or DA

{B,D} = BD or DB

Total cost computations: *choose*(N,2) x 2

Example

Cache

Subplan	Best choice	Cost
A	index	100
B	seq scan	50
{A,B}	BA	156
{B,C}	BC	98
{A,B,C}	BCA	125
…		
{B,C,D}	BCD	115

optjoin(ABCD)

Already computed – lookup in cache

∂=3

{A,B,C} = remove A, compare A({B,C}) to ({B,C})A

remove B, compare B({A,C}) to ({A,C})B

remove C, compare C({A,B}) to ({A,B})C

{A,B,D} = remove A, compare A({B,D}) to ({B,D})A

....

{A,C,D} = …

{B,C,D} = …

- Total cost computations: *choose*(N,3) x 3 x 2

Example

optjoin(ABCD)

Already computed –
lookup in cache

∂=4

{A,B,C,D} = remove A, compare A({B,C,D}) to ({B,C,D})A

remove B, compare B({A,C,D}) to ({A,C,D})B

remove C, compare C({A,B,D}) to ({A,B,D})C

remove D, compare D({A,B,C}) to ({A,B,C})D

Final answer is plan with minimum cost of these four

Total cost computations: *choose*(N,4) x 4 x 2

Cache

Subplan	Best choice	Cost
A	index	100
B	seq scan	50
{A,B}	BA	156
{B,C}	BC	98
{A,B,C}	BCA	125
{B,C,D}	BCD	115
{A,B,C,D}	ABCD	215

Complexity

choose(n,1) + *choose*(n,2) + … + *choose*(n,n) total subsets considered

All subsets of a size n set = power set of n = 2^n

Equiv. to computing all binary strings of size n

000,001,010,100,011,101,110,111

Each bit represents whether an item is in or out of set

Complexity (continued)

For each subset,

k ways to remove 1 join

$k < n$

m ways to join 1 relation with remainder

Total cost: $O(nm2^n)$ plan evaluations

$n = 20$, $m = 2$

4.1×10^7

Interesting Orders

- Some queries need data in sorted order
  - Some plans produce sorted data (e.g., using an index scan or merge join
- May be non-optimal way to join data, but overall optimal plan
  - Avoids final sort
- In cache, maintain best overall plan, plus best plan for each interesting order
- At end, compare cost of

  best plan + sort into order

  to

  best in order plan
- Increases complexity by factor of k+1, where k is number of interesting orders

Example

SELECT A.f3, B.f2 FROM A,B where A.f3 = B.f4
ORDER BY A.f3

Subplan	Best choice	Cost	Best in A.f3 order	Cost
A	index	100	index	100
B	seq scan	50	seqscan	50
{A,B}	BA hash	156	AB merge	180

compare:

cost(sort(output)) + 156

to

180

Overview of Query Optimization in Oracle

- Oracle DBMS V8
  - **Rule-based query optimization**: the optimizer chooses execution plans based on heuristically ranked operations.
    - (Currently it is being phased out)
  - **Cost-based query optimization**: the optimizer examines alternative access paths and operator algorithms and chooses the execution plan with lowest estimate cost.
    - The query cost is calculated based on the estimated usage of resources such as I/O, CPU and memory needed.
  - Application developers could specify hints to the ORACLE query optimizer.
  - The idea is that an application developer might know more information about the data.

Semantic Query Optimization

- **Semantic Query Optimization**:
  - Uses constraints specified on the database schema in order to modify one query into another query that is more efficient to execute.
- Consider the following SQL query,

```
SELECT    E.LNAME, M.LNAME
FROM      EMPLOYEE E M
WHERE     E.SUPERSSN=M.SSN AND E.SALARY>M.SALARY
```

- Explanation:
  - Suppose that we had a constraint on the database schema that stated that no employee can earn more than his or her direct supervisor. If the semantic query optimizer checks for the existence of this constraint, it need not execute the query at all because it knows that the result of the query will be empty. Techniques known as theorem proving can be used for this purpose.