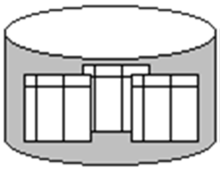
QUERY PROCESSING: How to Optimize Relational Queries?



Introduction

- We've covered the implementation of single relational operations
  - Choices depend on indexes, memory, statistics,…
  - Joins
    - Blocked nested loops:
      - simple, exploits extra memory
    - Indexed nested loops:
      - best if one relation small and one indexed
    - Sort/Merge Join:
      - good with small amount of memory, bad with duplicates
    - Hash Join:
      - fast (enough memory), bad with skewed data

Introduction

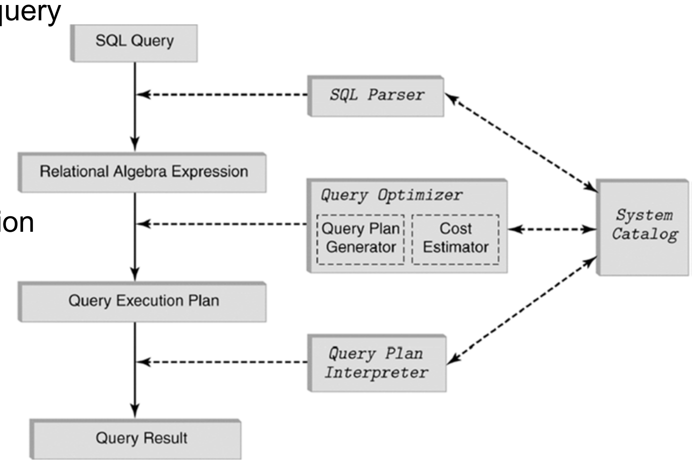
- Query optimization is an important task in a relational DBMS
- Must understand optimization in order to understand the performance impact of
  - a given database design (relations, indexes)
  - on a workload (set of queries)
- Two parts to optimize a query:
  - Consider a set of alternative execution plans
    - Must prune search space; typically, left-deep plans only
      - This reduces optimization complexity and generates plans amenable to pipelined evaluation
  - Must estimate cost of each execution plan that is considered
    - Must estimate size of result and cost for each plan node (operator)
    - Key issues: Statistics, indexes, operator implementations

Basic Terminology

- Query Processing: the activities involved in retrieved data from the database
  - How to take a query in high level language (typically SQL) into a correct and efficient execution strategy, and then execute this strategy
- Query Plan: queries are compiled into logical query plans (often like relation algebra) and then converted into physical query plan (by selecting an implementation for each operator)
- Query Optimisation: the activity of choosing an efficient execution strategy for processing a query
  - Many transformations of the same high-level query
  - Choose one that minimises some system resource

Query Processing Steps/Architecture

❶ Input: User-defined query
❷ Parsing
❸ Query validation
❹ View resolution
❺ Query optimization
❻ Execution plan creation
❼ Code creation
❽ Execution
❾ Output: Query result



- Main difference to language compilers: translation is data dependent!

Query Evaluation

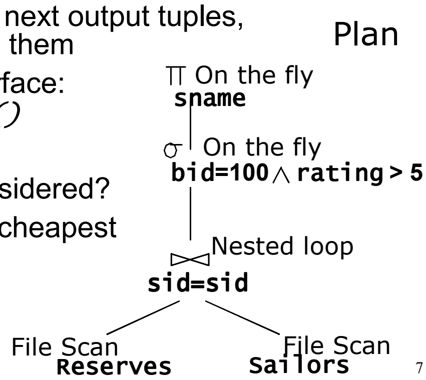
```
SELECT S.sname
FROM   Reserves R, Sailors S
WHERE  R.sid=S.sid AND
     R.bid=100 AND S.rating>5
```

- Problem: An SQL query is declarative (`select from where` filter) i.e., does not specify a query execution plan
  - ◆A relational algebra expression is procedural
    - there is an associated query execution plan
- Solution: Convert SQL query to an equivalent relational algebra expression and evaluate it using the associated query execution plan (i.e., choice of an implementation algorithm)
  - ◆But which equivalent expression is best?
    - Operators can be applied in different order!

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

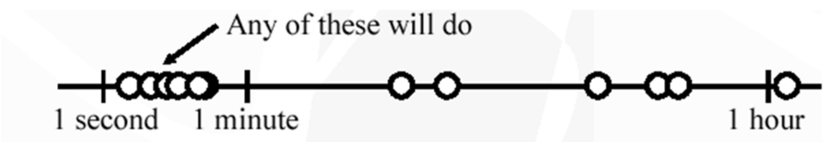
Query Execution Plan (QEP)

- Convert relational algebra expression to an operation tree where each node is annotated to indicate:
  - ◆which access method to use for each relation
  - ◆which implementation method to use for each operation
- Each operation tree is typically implemented using pipeline:
  - when an operator is `pulled' for the next output tuples, it `pulls' on its inputs and computes them
  - nodes support uniform *iterator* interface: *open(), get_next(), close()*
- Two main issues:
  - ◆For a given query, what plans are considered?
    - Algorithm to search plan space for cheapest (estimated) plan
  - ◆How is the cost of a plan estimated?
- Ideally: Want to find best plan
- Reality: Avoid worst plans!



Query Optimizer

- The evaluation of relational algebra expressions involves:
  - ◆estimating the cost of a relational algebra expression
  - ◆transforming one relational algebra expression to an equivalent one
  - ◆choosing access methods for evaluating the subexpressions
- Too expensive to consider all algebraically equivalent plans: might take longer to find an optimal plan than to compute query brute-force !!!
  - ◆Consider only a subset of plans using heuristic algorithms
- Query optimizers do not "optimize"
  - ◆just try to find "reasonably good" evaluation strategies



Schema and Base for Examples

```
Sailors (sid:integer, sname:string, rating:integer, age:real)
Reserves (sid:integer, bid:integer, day:dates, rname:string)
```

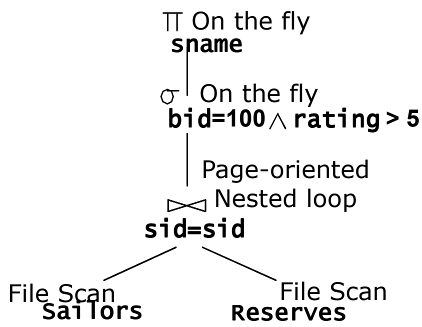
- `Sailors`:
  - Each tuple is 50 bytes long, 80 tuples per page, 500 pages
  - Assume there are 10 different ratings
- `Reserves`:
  - Each tuple is 40 bytes long, 100 tuples per page, 1000 pages
  - Assume there are 100 boats

Motivating Example

```
SELECT S.sname
FROM   Reserves R, Sailors S
WHERE  R.sid=S.sid AND
       R.bid=100 AND S.rating>5
```

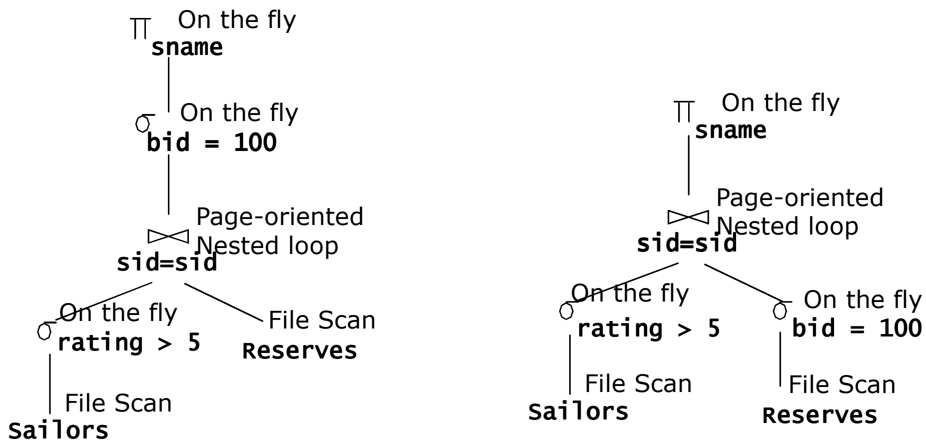
- Cost: 500+500*1000 = 500500 I/Os
- By no means the worst plan!
- Misses several opportunities:
  - selections could have been `pushed' earlier
  - no use is made of any available indexes, etc.
- Goal of optimization:
  - To find more efficient plans that compute the same answer

Plan



500500 IOs

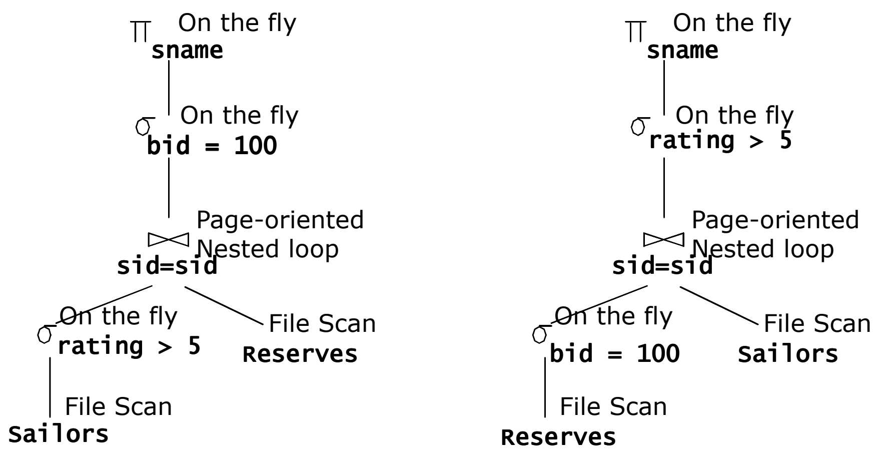
Alternative Plans: Push Selects (No Indexes)



250500 IOs　　　　250500 IOs

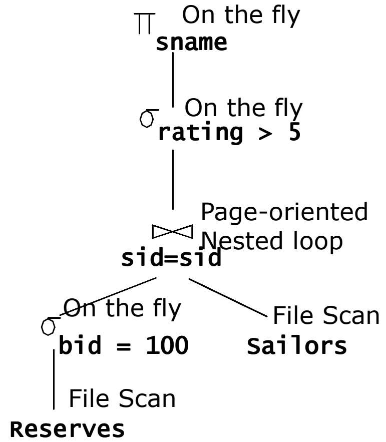
- With "Reserves" as outer 501000 IOs

Alternative Plans: Push Selects (No Indexes)

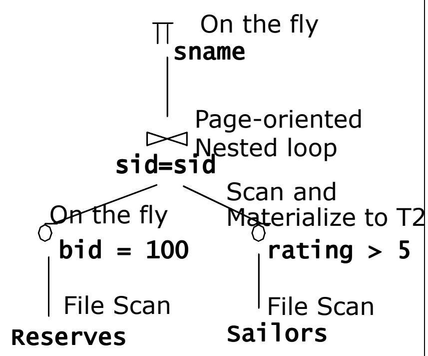


250500 IOs　　　　6000 IOs

Alternative Plans: Push Selects (No Indexes)

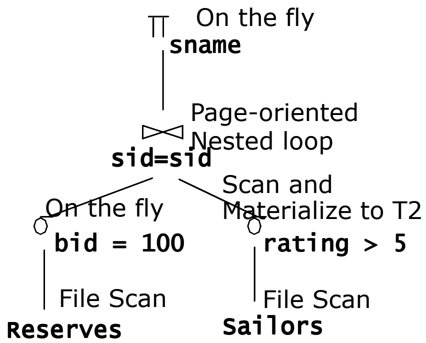


6000 IOs

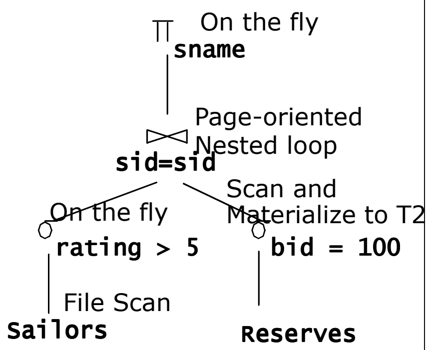


4250 IOs

Alternative Plans: Push Selects (No Indexes)



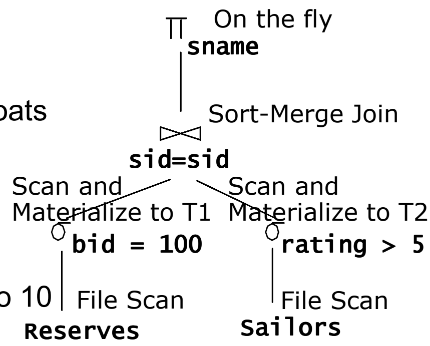
4250 IOs



4010 IOs

More Alternative Plans: Join Algos (No Indexes)

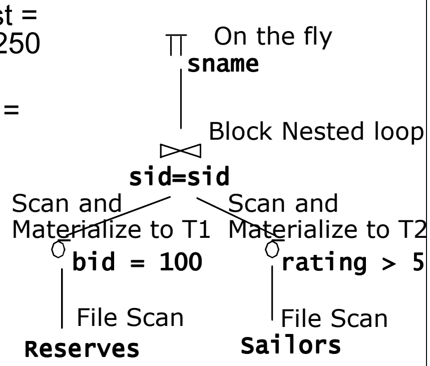
- Cost of $\sigma_{\texttt{bid}=100}$ (R) = Cost to scan R + Cost to write T1 = 1000 + ***size****(T1)* IOs
  - Estimate ***size****(T1)*: Assume uniform distribution of reservations over 100 boats
    - ***size****(T1)* = 1000/100 = 10 IOs
- Cost of $\sigma_{\texttt{rating}>5}$ (S) =Cost to scan S + Cost to write T2 = 500 + ***size****(T2)* IOs
  - Estimate ***size****(T2)*: Assume uniform distribution of ratings over range of 1 to 10
    - ***size****(T2)* = 500/2 = 250 IOs



- With 5 buffers, cost of sort-merge join of *T1* & *T2*
  - Cost Sort *T1* (2 pass)= 2*2*10 = 40 IOs
  - Cost Sort *T2* (4 pass)= 2*4*250 = 2000 IOs
  - Cost Merge *T1* and *T2*= 10+250 = 260 IOs
- Total cost = cost of selection + cost of join = 1760 + 2300 = 4060 IOs

More Alternative Plans: Join Algos (No Indexes)

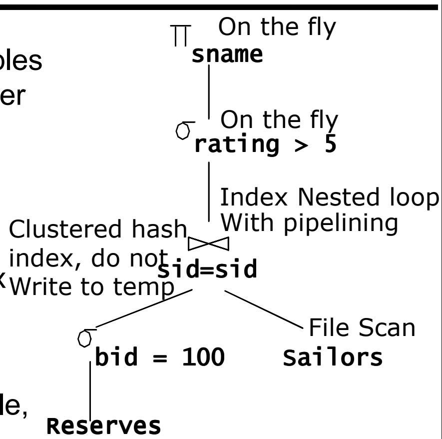
- Cost of block nested loops join: *T1* as outer rel
  - Every 3 page block of *T1*, scan *T2:* Cost = Scan *T1* + Scan *T2* (4 times) = 10 + 4*250 = 1010 IOs
- Total cost = cost of selection + cost of join = 1760 + 1010 = 2770 IOs
- Push projection ahead of join
  - Only ***sid*** of *T1* and ***sid,sname*** of *T2* needed
  - Remove unwanted attributes as *T1* and *T2* are scanned during selection



- Reduce size of *T1* and *T2* substantially
  - *T1* fit into 3 buffer page
    - Perform block nested loops join with a single scan of *T2*
  - Cost of join ~ 250 IOs
- Total cost of plan ~ 2000 (1750 + 250) IOs

More Alternative Plans: Join Algos (Indexes)

- Cost of $\sigma_{bid=100}$ (R): use clustered hash index on *R.bid* to retrieve matching R tuples
  - Uniform distribution of reservations over 100 boats the estimated # of selected tuples = 100000/100 = 1000
  - Clustered index, 1000 tuples in same bucket stored sequentially, Cost of selection = 10 IOs
- For each selected R tuple use hash index on *S.sid* to retrieve matching S tuples
  - Join attribute *sid* is a key for Sailors; At most one S tuple match
  - Average 1.2 page IO to retrieve a tuple, Cost of join = 1200 IOs
- For each tuple in join result
  - Perform selection `rating>5` on-the-fly
  - Perform projection on *sname* on-the-fly
- Total cost = 1210 IOs



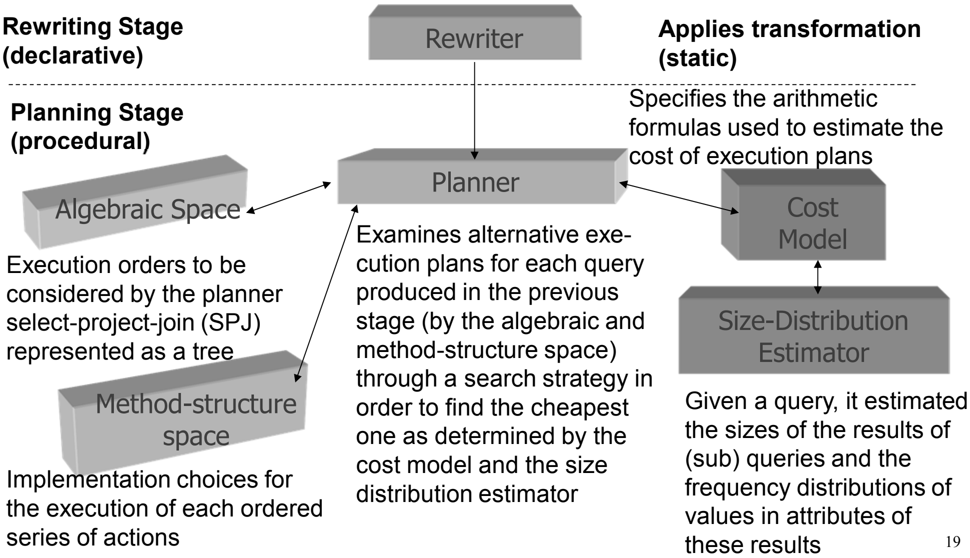
- Projection not push ahead
  - Selected R tuples not materialized
  - Join pipelined
- Selection `rating>5` not push ahead
  - Index on *sid* available

What is Needed for Query Optimization?

- A closed set of operators
  - Relational operators (table `in`, table `out`)
  - Encapsulation based on iterators
- Plan space based on
  - Relational algebra equivalences
- Cost estimation based on
  - Cost formulas
  - Size estimation, based on
    - Catalog information on base tables
    - Selectivity (*Reduction Factor*) estimation
- A search algorithm
  - Enumeration of plans
    - Single/Multiple-Relation queries
  - To sift through the plan space based on cost!

- Optimize a relational algebra expression:
  - Enumerate alternative execution plans
  - Estimate cost of each enumerated plan
  - Choose plan with least cost

Overview of a Typical Query Optimizer



Translating SQL to Relational Algebra

```
SELECT  S.sid, MIN (R.day)
FROM  Sailors S, Reserves R, Boats B
WHERE  S.sid = R.sid AND R.bid = B.bid AND
       B.color = “red” AND
       S.rating = ( SELECT MAX (S2.rating)
                    FROM Sailors S2 )
GROUP BY S.sid
HAVING COUNT (*) >= 2
```

- For each sailor with the highest rating (over all sailors), and at least two reservations for red boats, find the sailor id and the earliest date on which the sailor has a reservation for a red boat

Translating SQL to Relational Algebra

```
SELECT  S.sid, MIN (R.day)
FROM  Sailors S, Reserves R, Boats B
WHERE  S.sid = R.sid AND R.bid = B.bid AND
       B.color = "red" AND
       S.rating = ( SELECT MAX (S2.rating)
                    FROM Sailors S2 )
GROUP BY S.sid
HAVING COUNT (*) >= 2
```

Inner Block

$\pi_{\texttt{S.sid}, \mathrm{MIN}(\texttt{R.day})}$ (HAVING $_{\mathrm{COUNT}(*)>2}$ (GROUP BY $_{\texttt{S.sid}}$ ($\sigma_{\texttt{B.color = "red"} \wedge \texttt{S.rating = val}}$ (Sailors $\bowtie$ Reserves $\bowtie$ Boats))))

Units of Optimization: Query Blocks

- An SQL query is parsed into a collection of *query blocks*, and these are optimized one block at a time
  - ◆Nested blocks are usually treated as calls to a subroutine, made once per outer tuple
  - ◆Exactly one SELECT and one FROM clause
  - ◆At most one WHERE, one GROUP BY and one HAVING clause

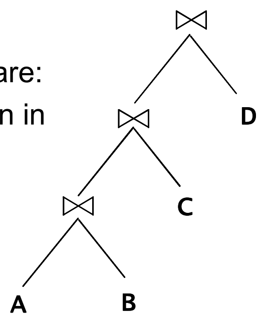
```
SELECT  S.sname
FROM  Sailors S
WHERE  S.age IN
     (SELECT  MAX (S2.age)
        FROM  Sailors S2
        GROUP BY  S2.rating)
```

nested block

```
SELECT  S.sname
FROM  Sailors S
WHERE  S.age IN
 Reference to nested block
```

Units of Optimization: Query Blocks

- *Query blocks* expressed in relational algebra as
  - ◆Cross-product of all relations in the FROM clause
  - ◆Selections in the WHERE clause
  - ◆Projections in the SELECT clause
- For each block, the execution plans considered are:
  - ◆All available access methods, for each relation in FROM clause
  - ◆All left-deep join trees (multi-relation) i.e.,
    - right branch always a base table,
    - consider all join orders and join methods
- Intricacies of SQL complicate query optimization
  - ◆E.g. nested subqueries



The Issue of Nested Queries (Uncorrelated)

- Find names of sailors who reserve boat # 103

```
SELECT S.sname
FROM Sailors S
WHERE S.sid IN (SELECT R.sid
                FROM Reserves R
                WHERE R.bid=103)
```

  - ◆Nested subquery evaluated once
  - ◆Result is a collection of `sids` C
  - ◆For each S tuple, check if `sid` is in C
    - Nested loops join of S and C

The Issue of Nested Queries (Uncorrelated)

- Find names of sailors with highest rating

```
SELECT S.sname
FROM Sailors S
WHERE S.rating = (SELECT MAX (S2.rating)
                  FROM Sailors S2)
```

  - ◆Nested subquery evaluated once
  - ◆Result is a single value
  - ◆Incorporated into top-level query

The Issue of Nested Queries (Correlated)

- Conceptually, a nested subquery is a function
  - ◆Variables from outer level query are the parameters

```
SELECT S.sname
FROM Sailors S
WHERE EXISTS (SELECT *
              FROM Reserves R
              WHERE R.bid=103 AND S.sid=R.sid)
```

- Correlated evaluation: the subquery is separated evaluated for each tuple in the outer level query
  - ◆Tuple variable S from top-level query appears in nested query
  - ◆Evaluate subquery for each S tuple
- Correlated evaluation may be quite inefficient since
  - ◆a large number of calls may be made to the nested query
  - ◆there may be unnecessary random I/O as a result

Query Rewriting: Nested Queries (Correlated)

- Nested block is optimized independently, with the outer tuple considered as providing a selection condition
- Outer block is optimized with the cost of `calling' nested block computation taken into account
  - Implicit ordering of these blocks means that some good strategies are not considered
- SQL optimizers attempt to rewrite nested subqueries into joins where possible, enabling use of efficient join techniques
  - Nested query has equivalent query without nesting
  - Correlated query has equivalent query without correlation
  - The non-nested decorrelated version of the query is typically optimized better

Nested block to optimize:

```
SELECT  *
 FROM  Reserves R
 WHERE R.bid=103 AND
       R.sid= outer
value
```

Equivalent non-nested query:

```
SELECT S.sname
FROM Sailors S, Reserves R
WHERE  S.sid=R.sid AND
       R.bid=103
```


Query Rewriting

- Syntactic and semantic query analysis detects & rejects incorrect queries:
  - Type errors
  - Semantically incorrect (disconnected query graph)
- Normalize query predicates expressions into
  - conjunctive or disjunctive normal form
- Simplify statements:
  - Eliminate ANY / ALL operators

```
... val > ANY (x, y) => val > x OR value > y
... x> ALL(SELECT y FROM R WHERE z=10)
=>... NOT ( x <= ANY (SELECT...)
=>...NOT EXISTS (SELECT y FROM R
                  WHERE z= 10 AND x <= y)
```

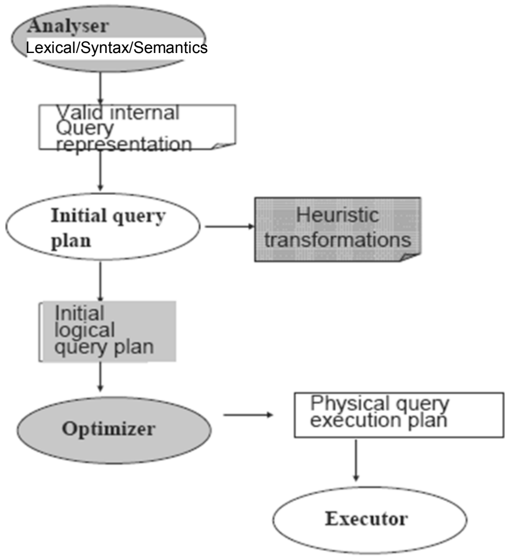
  - Eliminate more baroque constructs (BETWEEN)
  - Evaluates expressions as far as possible

```
... x > 0.5* z/100 * 4 => x > z/50
```

- Rewrite calculus query into relational algebra expressions

Logical vs. Physical of Optimization

- Logical level optimization
  - ◆data dictionary (system catalog) independent (algebraic) transformation of the query according to the algebraic laws of relational algebra
- Physical level optimization
  - ◆physical level using indexes ("internal")
    - schema based
  - ◆cost based selection of optimal plan using database statistics
    - value based



Logical Optimization

- Transforms query according to algebraic laws of relational algebra
  - ◆Focus is application of "algebraic transformation rules“
- Two relational algebra expressions over the same set of input relations are equivalent if they produce the same result on all instances of the input relations
  - ◆To transform a relational expression into another equivalent expression we need transformation rules that preserve equivalence
- Each transformation rule
  - ◆Is provably correct (i.e, does preserve equivalence)
  - ◆Has a heuristic associated with it

Relational Algebra Equivalences: Selections & Projections

- Selection Cascading $\sigma_{c1 \wedge \ldots \wedge cn}(R) \equiv \sigma_{c1}\left(\ldots\ \sigma_{cn}(R)\right)$
  - ◆Combine several selections into one
  - ◆Replace a selection involving several conjuncts with several smaller selection operations
- Commutative Selections $\sigma_{c1}\left(\sigma_{c2}(R)\right) \equiv \sigma_{c2}\left(\sigma_{c1}(R)\right)$
  - ◆Test conditions c1 and c2 in either order
- Projection Cascading $\pi_{a1}(R) \equiv \pi_{a1}\left(\ldots\left(\pi_{an}(R)\right)\right)$
  - ◆Successively eliminating columns from R is simply eliminating all but the columns retained by the final projection
    - If each a_i is a set of attributes of R, $a_i \subset a_{i+1}$
- Commute projection and selection:

$$\pi_{Att}\left(\sigma_{\text{Cond}}(R)\right) \equiv \sigma_{\text{Cond}}\left(\pi_{Att}(R)\right)$$

  - ◆A projection commutes with a selection that only uses attributes retained by the projection
    - if *attr* $\supseteq$ all attributes in *Cond*

Relational Algebra Equivalences: Joins & Cartesian Product

- Commutative

$$(R \bowtie S) \equiv (S \bowtie R)$$

  - ◆used to reduce cost of nested loop evaluation strategies (smaller relation should be in outer loop)
- Associative

$$R \bowtie (S \bowtie T) \equiv (R \bowtie S) \bowtie T$$

  - ◆used to reduce the size of intermediate relations in computation of multi-relational join – first compute the join that yields smaller intermediate result
- This implies order-independence of joins

$$R \bowtie (S \bowtie T) \equiv (T \bowtie R) \bowtie S$$

- N-way join has *T(N)* × *N*! different evaluation plans (more latter)
  - ◆*T(N)* is the number of parenthesized expressions
  - ◆*N*! is the number of permutations

Pushing Selections and Projections

- $\sigma_{Cond}(R \times S) \equiv R \bowtie_{Cond} S$
  - A selection between attributes of the two arguments of a cross-product converts cross-product to a join
    - *Cond* relates attributes of both R and S
  - Reduces size of intermediate relation since rows can be discarded sooner
- $\sigma_{Cond}(R \times S) \equiv \sigma_{Cond}(R) \times S$
  - A selection on just attributes of R commutes with cross-product
    - *Cond* involves only the attributes of R
  - Reduces size of intermediate relation since rows of R are discarded sooner
- $\pi_{attr}(R \times S) \equiv \pi_{attr}(\pi_{attr'}(R) \times S)$
  - A projection following a join can be `pushed' by retaining only attributes of R (and S) that are needed for the join or are kept by the projection
    - if $attributes(R) \supseteq attr' \supseteq attr$
  - Reduces the size of an operand of product

Equivalence Example

$$\sigma_{C1 \wedge C2 \wedge C3}(R \times S) \equiv$$
$$\sigma_{C1}(\sigma_{C2}(\sigma_{C3}(R \times S))) \equiv$$
$$\sigma_{C1}(\sigma_{C2}(R) \times \sigma_{C3}(S)) \equiv$$
$$\sigma_{C2}(R) \bowtie_{C1} \sigma_{C3}(S)$$

- assuming *C2* involves only attributes of R,
- *C3* involves only attributes of S,
- and *C1* relates attributes of R and S

Choice of Execution Plans

- Must consider the interaction of evaluation techniques when choosing execution plans:
  - ◆choosing the cheapest algorithm for each operation independently may not yield best overall algorithm:
    - merge-join may be costlier than hash-join, but may provide a sorted output which reduces the cost for an outer level aggregation
    - nested-loop join may provide opportunity for pipelining
- Systems may use *heuristics* to reduce the number of choices that must be made in a cost-based fashion
  - ◆Heuristic optimization transforms the query-tree by using a set of rules that typically (but not in all cases) improve execution performance
  - ◆Heuristics do not always reduce the cost, but work in many cases
- Practical query optimizers incorporate elements of the following two broad approaches:
  - ❶Uses heuristics to choose a plan
  - ❷Search all the plans and choose the best plan in a cost-based fashion

Physical Optimization

- Rule-based (“internal”)
  - ◆Chooses access method to result data according to heuristic rules e.g.
    - to push selections and projections down the query tree (System R)
    - to repeatedly pick “best” relation to join next (Oracle)
      - Starting from each of *n* starting points and pick best among these
  - ◆Precedence among access operators (record level interface, Oracle):
    - ❶`Single row access by rowid ....`
    - ❸`Single row by primary key ....`
    - ❽`Single column index ....`
    - ❿`Full table scan`

  Optimizer always chooses lowest rank operator independent from data
- Cost-based optimization
  - ◆Utilizes statistical properties of the DB state
  - ◆Among the top secrets of DB vendors
  - ◆DB2 (IBM) seems to have the most sophisticated optimizer (!)
  - ◆Basic principle: make the typical (simple) case fast

Typical Rules in Heuristic Optimization

❶ Split conjunctive selections to enable sift of partial selection predicates

❷ Shift selections towards the leaves of the execution tree

❸ Rearrange leaves such that most restrictive selections are far left in the tree

❹ Replace Cartesian product operations that are followed by a selection condition by join operations

❺ Split projections (and create new ones) and move towards leaves (but beware of mutating joins to cross products)

❻ Identify those subtrees whose operations can be pipelined, and execute them using pipelining

Example: Limitation of Heuristics

- Consider two logical plans:

❶ $\sigma_{age \geq 18}$ ((Sailors) $\bowtie_{\text{Sailors.sid = Reserves.sid}}$ Reserves)

❷ ($\sigma_{age \geq 18}$(Sailors)) $\bowtie_{\text{Sailors.sid = Reserves.sid}}$ Reserves)

- Plan ❷ is created by the heuristic rule pushing selections as close to relation as possible (do selections early)
- If all sailors have age>18, and no sailor appears in reserve, Plan ❶ is better than Plan ❷

CSD Univ. of Crete Fall 2012

Heuristic vs. Cost based Optimization

- Heuristic query optimization
  - ◆Sequence of single query plans
  - ◆Each plan is (presumably) more efficient than the previous
  - ◆Search is linear
- Cost-based query optimization
  - ◆Many query plans generated
  - ◆The cost of each is estimated, with the most efficient chosen
  - ◆Search is multi-dimensional (various performance metrics!)
- Both logical and physical plan optimization can be based on cost estimation
  - ◆For logical plan optimization, estimated sizes of intermediate results can be used to choose better logical plans
  - ◆For physical plan optimization, estimated disk I/O can be used to generate (or enumerate) and to choose better physical plans
  - ◆For scans using secondary indices, some optimizers take into account the probability that the page containing the tuple is in the buffer

39

CSD Univ. of Crete Fall 2012

Cost-based Query Sub-System

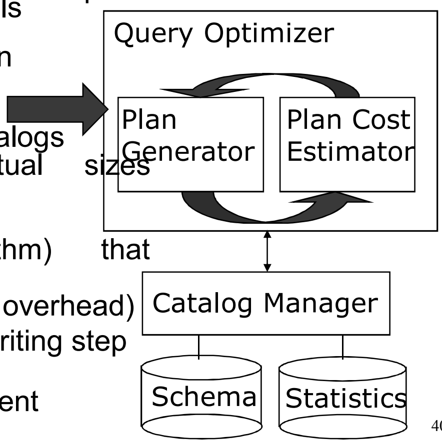
❶ Performance Metrics
- ◆Data volume to transfer in terms of page I/Os
- ◆CPU cost (path length of query: i.e., number of instructions)
- ◆Resource allocation, e.g. buffer pools required, cache misses, etc.
- ◆Communication, power consumption (depends upon domain)

❷ Cost Estimation
- ◆Statistics maintained by system catalogs
- ◆Estimates are approximations to actual sizes and costs

❸ Plan Enumeration algorithm
- ◆search strategy (optimization algorithm) that searches through the plan space
- ◆has to be efficient (low optimization overhead)

- Usually there is a heuristics-based rewriting step before the cost-based QO
  - ◆Identify alternative plans for equivalent relational algebra expressions

Goal: Given an initial Logical Plan generate a physical plan that is "Optimal" with respect to one or more performance metrics



40

Cost Estimation of Execution Plans

- Must estimate *cost* of each operation in execution plan tree
  - We've already discussed how to estimate the cost of operations (sequential scan, index scan, joins, etc.)
    - Number of pages of input relations
    - Indexes available
    - Pipelining or temporary relations created (materialize)
- Must estimate *size of result* and *sort order* for each operation in tree!
  - Needed for the input of the operation corresponding to the parent node
    - Use information about the input relations
    - For selections and joins, assume *independence* of predicates
- In System R, cost is boiled down to a single number consisting of

  `#I/O +` *`factor`* `* #CPU instructions`

System Catalogs

- Maintain statistics about relations
  - Cardinality: Number of tuples `NTuples(R)` for each relation R
  - Size: Number of pages `Npages(R)` for each relation R
- Maintain statistics about indexes
  - Index cardinality and size
    - Number of distinct key values `Nkeys(I)` for each index I
    - Number of pages `INPages(I)` for each index I
  - Index height
    - Number of non leaf levels `Iheight(I)` for tree index I
  - Index range
    - Minimum current key value `ILow(I)` and
    - maximum current key value `IHigh(I)` for each index I
- Statistics updated periodically
  - Expensive to update whenever data change
  - Approximations anyway
- May store more detailed statistical information
  - Histograms of the values in some attribute (more latter)

Size Estimation and Reduction Factors

```
SELECT  attribute list
FROM  relation list
WHERE  term1 AND ... AND termk
```

- Consider a query block:
  - Maximum # tuples in result is the product of the cardinalities of relations in the FROM clause
  - Every term in the WHERE clause eliminates some of the potential result tuples
- Reduction factor (RF) associated with each *term* reflects the impact of the *term* in reducing result size
  - Assume conditions tested by each term in the WHERE clause are statistically independent and values are uniformly distributed
  - Model effect of WHERE clause on result size by associating a RF with each term
  - Size of result = Max # tuples* product of RF for all the terms
  - RF usually called “selectivity”

Result Size Estimation for Selections

- Term `column = value`

  `RF = 1 / NKeys(I)`
  - `I` is an index on `column`
- Term `column1 = column2`

  `RF = 1 / max( NKeys(I1), NKeys(I2) )`
  - `I1` and `I2` are indexes on `column1` and `column2` respectively
- Term `column > value`

  `RF = ( High(I) - value) / ( High(I) - Low(I) )`
  - `I` is an index on `column`
- Note, if missing indexes, assume 1/10 (default) !!!

- Reduction factor due to complex condition
  - `RF(Cond1 AND Cond2)` = `RF(Cond1)` × `RF(Cond2)`
  - `RF(Cond1 OR Cond2)` = `min(1, RF(Cond1)` + `RF(Cond2))`

Result Size Estimation for Joins

- Given a join of R and S, what is the range of possible result sizes (in # of tuples)?
  - ◆Assume relations R and S, with `NTuples(R)` and `NTuples(S)`
- *Natural join*:
  - ❶R ∩ S = ∅, then cost of R ⋈ S = `NTuples(R) * NTuples(S)`
  - ❷R ∩ S = *key* for R (R ∩ S = key for S is symmetric)
    - a tuple of S will join with at most one tuple from R
      ⇒ cost of R ⋈ S ≤ `NTuples(S)`
  - ❸Also, if R ∩ S = foreign key of S referencing R
      ⇒ cost of R ⋈ S ≤ `NTuples(S)`
  - ❹R ∩ S = {A} neither key for R nor S
    - estimate each tuple r of R generates `NTuples(S)/ NKeys(A,S)` result tuples
      ⇒ cost of R ⋈ S = `NTuples(R) * NTuples(S)/ NKeys(A,S)`
    - but can also consider it starting with S
      ⇒ cost of R ⋈ S = `NTuples(S) * NTuples(R)/ NKeys(A,R)`
    - If these two estimates differ, take the lower one!

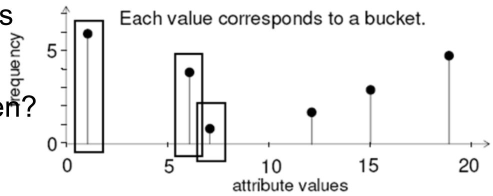
Result Size Estimation for Set Operations

- Unions/intersections of selections on the same relation: rewrite and use size estimate for selections
  - ◆$\sigma_{\theta 1}$ (R) ∩ $\sigma_{\theta 2}$ (R) can be rewritten as $\sigma_{\theta 1}$ $\sigma_{\theta 2}$ (R)
- Operations on different relations:
  - ◆Estimated size of `R ∪ S` = `NTuples(R)` + `NTuples(S)`
  - ◆Estimated size of `R ∩ S` = `min(NTuples(R)`, `NTuples(S))`
  - ◆Estimated size of `R - S` = `NTuples(R)`
- Inaccurate, upper bounds on the sizes

Better Estimation Using Histograms

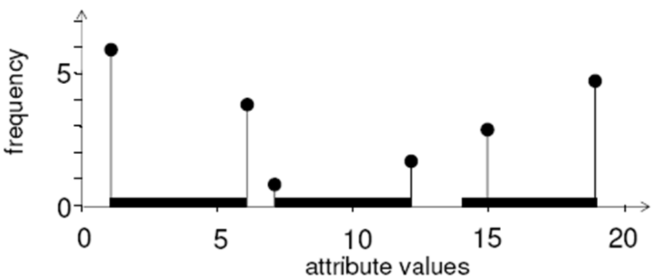
- Motivation
  - `NTuples(R)`, `High(R,A)`, `Low(R,A)`
    - Too little information
  - Actual distribution of R.A: (v_1, f_1), (v_2, f_2), …, (v_n, f_n)
    - f_i is the frequency of v_i i.e., the number of times v_i appears in `R.A`
    - Too much information (perfect histogram); Anything in between?
- Idea
  - Partition the domain of `R.A` into intervals = buckets (compression)
  - Store a small summary of the distribution within each bucket
  - Number of buckets are the “knob” that controls the resolution

attribute value	1	6	7	12	15	19
frequency	6	4	1	2	3	5



More compact representation, as compared to original list of values

Equi-width Histogram



attribute values	[1, 6]	[7, 12]	[14, 19]
frequency	7 + 4 = 12	1 + 2 = 3	3 + 5 = 8

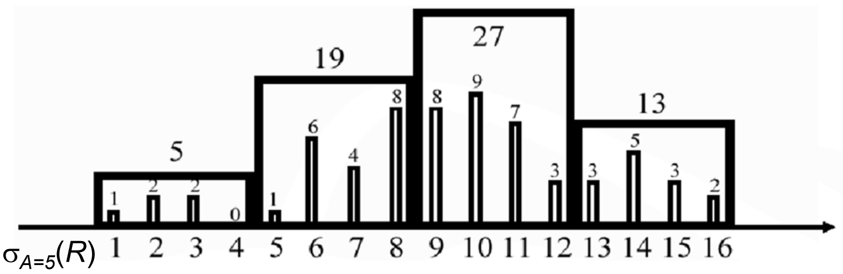
- Divide the domain into B buckets of equal width
  - Store the bucket boundaries and the sum of frequencies of the values with each bucket

24

Construction and Maintenance

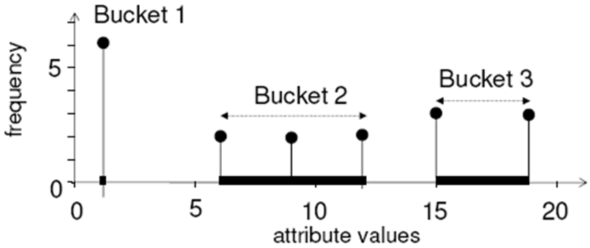
- Construction:
  - ◆Scan in one pass R to construct an accurate equi-width histogram
    - Keep a running count for each bucket
  - ◆If scanning is not acceptable, use sampling
    - Construct a histogram on R_{sample}, and scale the frequencies by `NTuples(R) / NTuples(R`$_{sample}$`)`
- Maintenance:
  - ◆Incremental maintenance: for each update on R, increment/ decrement the corresponding bucket frequencies
  - ◆Periodical re-computation: because distribution changes slowly

Using an Equi-width Histogram



- Q: $\sigma_{A=5}(R)$
  - ◆5 is in bucket [5,8] (with 19 tuples)
  - ◆Assume uniform distribution within the bucket
  - ◆Thus |Q| ≈ 19/4 ≈ 5
  - ◆Note that the actual value of A is 1
- Q: $\sigma_{A>=7\ \&\ A <= 16}(R)$
  - ◆[7,16] covers [9,12] (27 tuples) and [13,16] (13 tuples)
  - ◆[7,16] partially covers [5,8] (19 tuples)
  - ◆Thus |Q| ≈ 19/2 + 27 + 13 ≈ 50
  - ◆Note that the actual value range of A is 52

Equi-height Histogram



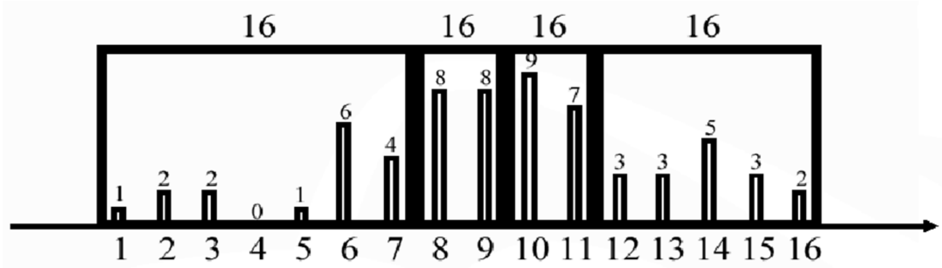
attribute values	1	[6, 12]	[15, 19]
frequencies	6	3 x 2 = 6	2 x 3 = 6

- Divide the domain into B buckets with roughly the same number of tuples in each bucket
  - ◆Store the sum of frequencies and the bucket boundaries
- Intuition: high frequencies are more important than low frequencies

Construction and Maintenance

- Construction:
  - ◆Sort all R.A values, and then take equally spaced slights
    - Example: 1 2 2 3 4 7 8 9 10 10 10 10 11 11 12 12 14 16 …
  - ◆Sampling also works
- Maintenance:
  - ◆Incremental maintenance
    - Merge adjacent buckets with small counts
    - Split any bucket with a large count
      - Select the median value to split
      - Need a sample of the values within this bucket to work well
  - ◆Periodic re-computation also works

Using an Equi-height Histogram



- Q: $\sigma_{A=5}(R)$
  - 5 is in bucket [1,7] (with 16 tuples)
  - Assume uniform distribution within the bucket
  - Thus $|Q| \approx 16/7 \approx 2$ (actual value = 1)
- Q: $\sigma_{A>=7 \;\&\; A <= 16}(R)$
  - [7,16] covers [8,9], [10,11],[12,16] (all with tuples)
  - [7,16] partially covers [1,7] (16 tuples)
  - Thus $|Q| \approx 16/7 + 16 + 16 + 16 \approx 50$ (actual value range = 52)

Histogram Tricks

- Store the number of distinct values in each bucket
  - To get rid of the effects of the values with 0 frequency
    - These values tend to cause underestimation
- Compressed histogram
  - Store (v_i, f_i) pairs explicitly if f_i is high (perfect histogram)
  - For other values, use an equi-width or equi-height histogram
- More Histograms
  - V-optimal histogram
    - Avoid putting very different frequencies into the same bucket
    - Partition in a way to minimize $\Sigma_i var_i$, where var_i is the frequency variance within bucket i
  - MaxDiff Histogram
    - Define area to be the product of the frequency of a value and its "spread" (the difference between this value and the next value with non-zero frequency)
    - Insert bucket boundaries where two adjacent areas differ by large amounts

Value-Independence Assumption

- Assumption so far
  $$p(A_1=v_1,\ A_2=v_2) = p(A_1=v_1)*p(A_2=v_2)$$
- Given this assumption, we can predict frequencies of combinations of attribute values based on frequencies of individual values
  - i.e., one would only need histograms for the individual attributes
- Popular approach, but quality of predictions is low
  - Multi-dimensional histograms become necessary
  - Problems mentioned before (which histogram variant, which granularity) now more urgent and more difficult at same time

Enumeration of Alternative Plans: Search Space

- Given the search space of equivalent operator trees for a given query
  - Query optimization process tries to identify the least expensive plan (according to a cost model) in a search space
  - Constrained to the time it takes to evaluate the search space versus a potential execution time
- There are two main cases:
  - Single-relation plans
  - Multiple-relation plans
- For queries over a single relation, queries consist of a combination of selects, projects, and aggregate operations:
  - Each available access method (file scan / index) is considered, and the one with the least estimated cost is chosen
  - The different operations are essentially carried out together (e.g., if an index is used for a selection, projection is done for each retrieved tuple, and the resulting tuples are pipelined into the aggregate computation)

Schema and Base for Examples

```
Sailors (sid:integer, sname:string, rating:integer, age:real)
Reserves (sid:integer, bid:integer, day:dates, rname:string)
```

- `Sailors`:
  - Each tuple is 50 bytes long, 80 tuples per page, 500 pages
  - Assume there are 10 ratings, 40000 sids
- `Reserves`:
  - Each tuple is 40 bytes long, 100 tuples per page, 1000 pages
  - Assume there are 100 distinct bids

Single-Relation Queries

- No joins
  - Only one selection or projection or aggregate operation
- Combination of selection, projection and aggregate operations
  - Plans without indexes
  - Plans with index
- Plans without indexes
  - Scan relation and apply selection and projection operations to each retrieved tuple
  - Cost:
    - File scan
    - Write out tuples after select and project
    - Sort tuples for GROUP BY
    - No additional IO for HAVING
    - Aggregation done on the fly

Single-Relation Queries: Example

$\pi_{(\texttt{rating, sname})}$
$\sigma_{(\texttt{rating} > 5 \wedge \texttt{age}=20)}$
(Sailors)

```
SELECT S.rating, COUNT(*)
FROM Sailors S
WHERE S.rating > 5 AND S.age = 20
GROUP BY S.rating
HAVING COUNT DISTINCT (S.sname) > 2
```

- File scan = Npages(Sailors) = 500 IOs
- Write out tuples after select and project
  - Result tuple size ratio = size of <S.rating,S.sname>/size of Sailor tuple = 0.8
  - RF for rating selection = 0.5
  - RF for age selection = 0.1 (default)
  - Cost = Npages(Sailors) * 0.8 * 0.5 * 0.1 = 20 IOs
- Sort tuples for GROUP BY S.rating
  - Assume enough buffer pages to sort the Temp relation in two passes
  - Cost = 3 * Npages(Temp) = 60 IOs
- Total Cost = 500+20+60 = 580 IOs

Single-Relation Queries

- Plans with indexes
  - Single-index access method
    - Several indexes match selection conditions
    - Each matching index offers an alternative access method
    - Choose access method that retrieves fewest pages
    - Apply project, non primary selection terms
    - Compute grouping and aggregation
  - Multiple-index access method
    - Several indexes match selection conditions
    - Use each index to retrieve a set of rids
    - Intersect these sets of rids
    - Sort result by page id and retrieve tuples
    - Apply project, non primary selection terms
    - Compute grouping and aggregation

Single-Relation Queries

- Sorted index access method
  - List of grouping attributes is a prefix of a tree index
  - Use tree index to retrieve tuples in the order required by the GROUP BY clause
  - Apply selection conditions on each retrieve tuple
  - Remove unwanted fields
  - Compute aggregate operations for each group
  - Strategy works well for clustered index

Single-Relation Queries

- Index-only access method
  - All attributes in query are included in search key of a dense index on the relation in FROM clause
    - Data entries in index contain all the attributes of a tuple needed for the query
    - One index entry per tuple
  - Use an index-only scan to compute answers
    - No need to retrieve actual tuple from relation
    - Apply selection conditions to data entries in index
    - Remove unwanted attributes
    - Sort result for grouping
    - Compute aggregate functions within each group
  - Applicable even if index does not match selection conditions
    - If index match selection, then examine a subset of index entries
    - Otherwise, scan all index entries
  - Independent of whether index is clustered

Single-Relation Queries: Example

- Indexes
  - B+-tree index on `S.rating`
  - Hash index on `S.age`
  - B+-tree index on `<rating,sname,age>`

```
SELECT S.rating, COUNT(*)
FROM Sailors S
WHERE S.rating > 5 AND S.age = 20
GROUP BY S.rating
HAVING COUNT DISTINCT (S.sname) > 2
```

- Single-index access method
  - Use hash index on *age* to retrieve `Sailors` tuples such that *S.age=20*
  - Apply condition *S.rating>5* to retrieved tuples
  - Project out unwanted attributes
  - Sort Temp relation on rating to identify groups
  - Apply HAVING condition to eliminate some groups

Single-Relation Queries: Example

- Multiple-index access method
  - Use B+-tree index on `rating` to get rids of tuples with *rating>5*
  - Use index on age to get rids of tuples with *age=20*
  - Get the intersection of the two sets of rids
  - Sort rids by page number
  - Retrieve corresponding `Sailors` tuples …
  - Compute aggregate functions in the HAVING and SELECT clauses on-the-fly
- Sorted index access method
  - Use B+-tree index on grouping attribute `rating` to retrieve tuples with *rating>5*
  - Retrieved tuples ordered by `rating`
  - Compute aggregate functions in the HAVING and SELECT clauses on-the-fly
- Index-only access method
  - Use B+ tree index on `<rating,sname,age>` to retrieve data entries with *rating>5*
  - Retrieved entries *sorted* by `rating`
  - Choose entries with *age=20*
  - Compute aggregate functions in the HAVING and SELECT clauses on-the-fly
  - No `Sailors` tuples are accessed

Cost Estimates for Single-Relation Plans

- Index *I* on primary key matches selection:
  - Cost is *Height(I) + 1* for a B+-tree, about *1.2* for hash index
- Clustered index *I* matching one or more selects:
  - *(NPages(I) + NPages(R)) * product of RF's of matching selects*
- Non-clustered index *I* matching one or more selects:
  - *(NPages(I) + NTuples(R)) * product of RF's of matching selects*
- Sequential scan of file:
  - *NPages(R)*

❋ Recall: Must also charge for duplicate elimination if required by `distinct` clause

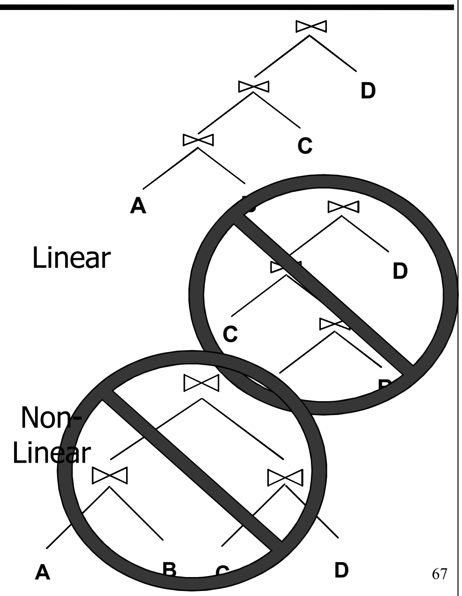
Cost Estimates for Single-Relation Plans: Example

```
SELECT S.sid
FROM   Sailors S
WHERE  S.rating=8
```

- If we have an index on *rating* :
  - Cardinality = (1/NKeys(I)) * NTuples(R) = (1/10) * 40000 tuples
  - Cost clustered index: (1/NKeys(I)) * (NPages(I)+NPages(R)) = (1/10) * (50+500) = 55 pages are retrieved
  - Cost unclustered index: (1/NKeys(I)) * (NPages(I)+NTuples(R)) = (1/10) * (50+40000) = 4005 pages are retrieved
- If we have an index on *sid* :
  - Would have to retrieve *all* tuples/pages!!!
  - With a clustered index, the cost is 50+500
  - With an unclustered index, the cost is 50+40000
- Doing a file scan is better:
  - We retrieve all file pages, the cost is 500

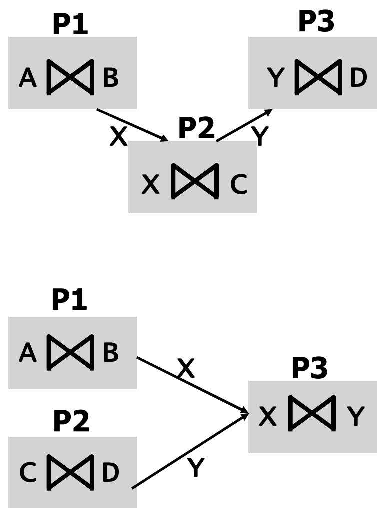
Multiple-Relation Queries

- Query blocks with two or more relations in FROM clause
  - Require joins
  - Cost of plan affected by the order in which relations are joined
- Fundamental heuristic in System R: *only left-deep join trees* are considered
  - As the number of joins increases, the number of alternative plans grows rapidly
    - *we need to prune the search space of alternative plans*



Associativity of Joins and Left-deep Plans

- *Left-deep join*: once a row of X has been output by P1, it need not be output again, but C may have to be processed several times in P2 for successive portions of X
  - Advantage: none of the intermediate relations (X, Y) have to be completely materialized and saved on disk
    - Important if one such relation is very large, but the final result is small
- Non-linear join: Each row of X must be processed against all of Y
  - Hence all of Y (can be very large) must be stored in P3, or P2 has to recomputed it several times



Left-deep Plans

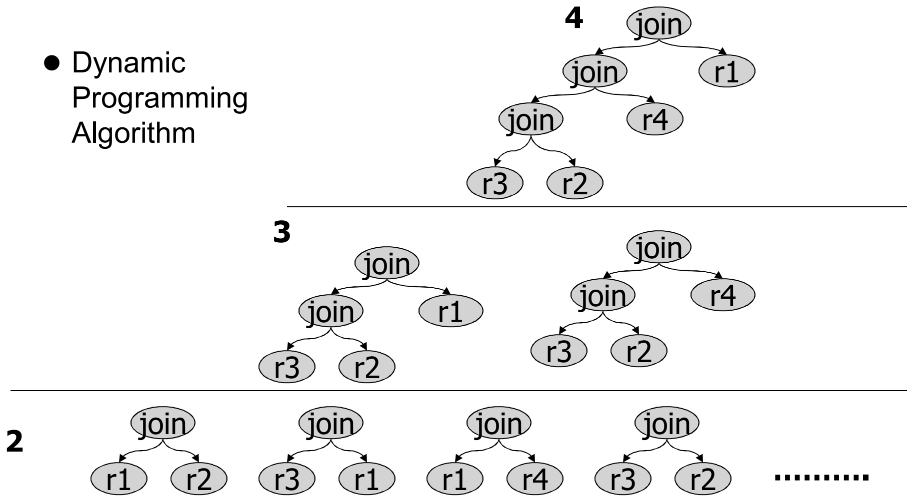
- Left-deep trees allow us to generate all *fully pipelined* plans
  - Intermediate join results not written to temporary files
    - Always materialize inner (base) relations
    - Examine entire inner relation for each tuple in outer relation
  - Not all left-deep trees are fully pipelined (e.g., Sort-Merge join)
- Enumeration of left-deep plans
  - Left-deep plans differ only in
    - Order of relations
    - access method for each relation
    - Join implementation method for each join
  - Multiple-pass algorithm
    - Using dynamic programming, the least-cost join order for any subset of $\{R_1, R_2, \ldots R_n\}$ is computed only once and stored for future use
    - N passes for join of N relations
  - In spite of pruning plan space, this approach is still costly (N! permutations)

Enumeration of Left-Deep Plans

- Pass 1
  - Enumerate all single-relation plans: can involve σ, π
    - All possible access methods (file scan, single index, multiple indexes, sorted index, index-only)
  - Keep best 1-relation plan for each relation
- Pass 2
  - Enumerate plans with one join
    - Join result of each 1-relation plan (outer) obtain in Pass 1 to every other relation (inner)
    - Tuples generated by outer plan pipelined into join
  - Keep best 2-relation plans
- Pass N
  - Enumerate N-relation plans (N-1 joins)
    - Contain all relations in query
    - Join result of each (N-1)-relation plan generated in Pass N-1 to every other relation

Dynamic Programming Plan Enumeration Algorithm

- Dynamic Programming Algorithm



Choosing Best or nearly Best Plans

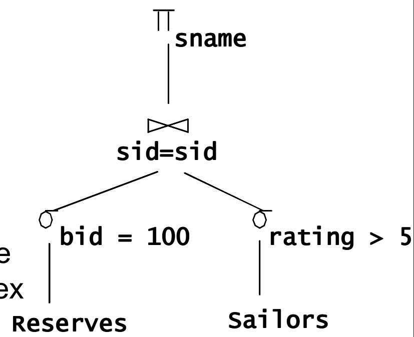
- An N-1 way plan is not combined with an additional relation unless there is a join condition between them, unless all predicates in WHERE have been used up
  - i.e., avoid Cartesian products if possible
- For each subset of relations, retain only:
  - Cheapest overall plan (best) for query
  - Cheapest plan for producing answers in some interesting order (nearly best) of the tuples i.e., if it is sorted by any of:
    - ORDER BY attributes
    - GROUP BY attributes
    - Join attributes of yet-to-be-planned joins
- ORDER BY, GROUP BY, aggregates etc. handled as a final step, using either an interestingly ordered plan or an additional sort/hash operator

36

Dealing with "Interesting Orders"

- An interesting sort order is a particular sort order of tuples that could be useful for a later operation: e.g., ($R_1 \bowtie R_2 \bowtie R_3) \bowtie R_4 \bowtie R_5$
  - Generating the result of $R_1 \bowtie R_2 \bowtie R_3$ sorted on the attributes common with $R_4 \bowtie R_5$ may be useful, but generating it sorted on the attributes common to only R1 and R2 is not useful
  - Using merge-join to compute $R_1 \bowtie R_2 \bowtie R_3$ may be costlier, but may provide an output sorted in an interesting order
- When picking the optimal plan
  - Comparing their costs is not enough
    - Plans are not totally ordered by cost anymore
  - Comparing interesting orders is also needed
    - Plans are now partially ordered
    - Plan X is better than plan Y if Cost of X is lower than Y and Interesting orders produced by X subsume those produced by Y
- Need to keep a set of optimal plans for joining every combination of *k* rels
  - Typically one for each interesting order
    - Find a "nearby" plan by making one change to one operator
    - Choose the best nearby plan according to estimated cost

Enumeration of Left-Deep Plans: Example



- Indexes
  - B+-tree index on `S.rating`
  - Hash index on `S.sid`
  - B+-tree index on `R.bid`
- Pass 1
  - "Sailors" relation 3 access methods: B+-tree, hash index, file scan
    - Selection *rating>5* matches B+-tree
      - Lower cost compared to hash index and file scan
      - However, if this selection is expected to retrieve a lot of tuples, and index is unclustered, file scan may be cheaper
  - "Reserves" relation 2 access methods: B+-tree, file scan
    - Selection *bid=100* matches B+-tree
      - Lower cost compared to file scan

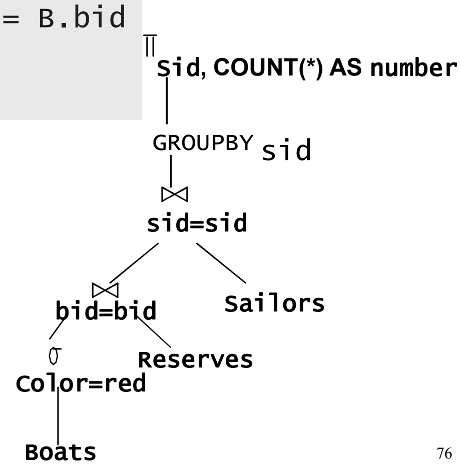
Enumeration of Left-Deep Plans: Example

- Pass 2
  - ◆Take relation computed by plan for R in Pass 1 and join it (as the outer) with S
    - Alternative access methods
      - Retrieve S tuples with *rating>5* and *sid=value* where *value* is some value from an outer R tuple
      - Selection *sid=value* matches hash index on `sid`
      - Selection *rating>5* matches B+ tree index on `rating`
      - Hash index cheaper since equality selection has lower RF
    - Alternative join methods e.g,. sort-merge join
  - ◆Take relation computed by plan for S in Pass 1 and join it (as the outer) with R
    - Alternative access methods
      - Retrieve R tuples with *bid=100* and *sid=value* where *value* is some value from an outer S tuple
      - Use B+ tree index on `bid`
    - Alternative join methods e.g,. block-nested loop join
  - ◆ Retain cheapest overall plan

Join Order Selection: First Example

```
Select S.sid, COUNT(*) AS number
FROM  Sailors S, Reserves R, Boats B
WHERE S.sid = R.sid AND R.bid = B.bid
      AND B.color = "red"
GROUP BY S.sid
```

- Sailors:
  - ◆B+-tree on *sid*
  - ◆Hash index on *sid*
- Reserves:
  - ◆B+-tree on *sid*
  - ◆Clustered B+-tree on *bid*
- Boats:
  - ◆B+-tree on *color*
  - ◆Hash index on *color*



Pass1: First Example

- Find best plan for each relation
  - ◆Reserves, Sailors
    - No selection match index
    - File scan
  - ◆Boats
    - Hash index on `color` match selection: Cheaper
    - Also retain B+-tree on `color`
      - Returns tuples in sorted order by `color`

Pass2: First Example

- For each of the plans in pass 1, generate plans joining another relation as the inner, using all join methods (and matching inner access methods)
  - ◆File Scan Reserves (outer) with Boats (inner)
  - ◆File Scan Reserves (outer) with Sailors (inner)
  - ◆File Scan Sailors (outer) with Boats (inner)
  - ◆File Scan Sailors (outer) with Reserves (inner)
  - ◆Boats accessed via hash on `color` with Sailors (inner)
  - ◆Boats accessed via B+-tree on `color` with Sailors (inner)
  - ◆Boats accessed via hash on `color` with Reserves (inner) (sort-merge)
  - ◆Boats accessed via B+-tree on `color` with Reserves (inner) (BNL)

- Retain cheapest plan for each pair of relations

Pass2: First Example

- Join of Boats accessed via hash index on `color` and Reserves (inner)
  - ◆Plan A: Index nested loops accessing Reserves via B+-tree index on `bid`
  - ◆Plan B: Access Reserves via B+-tree index on `bid` and use sort-merge join
    - Generate tuples in sorted order by `bid`
    - Retained (interesting order) although Plan A is cheaper
- Good heuristic: Avoid cross-products
  - ◆ Will not consider following joins

Outer	Inner
Scan of Sailors	Boats
Boats accessed via B+ tree on `color`	Sailors
Boats accessed via hash index on `color`	Sailors

Pass3: First Example

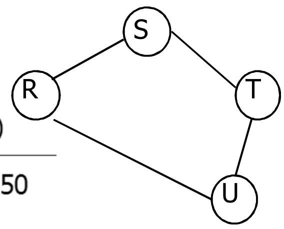
- Take each plan retained in Pass 2 as outer, and join remaining relation as inner
- Example:
  - ◆Access Boats via hash index on `color`
  - ◆Access Reserves via B+ tree on `bid`
  - ◆Join using sort-merge join
  - ◆Take result as outer and join with Sailors (accessed via B+-tree on `sid`) using sort-merge join
  - ◆Result of first join sorted by `bid`
  - ◆Second join requires input to be sorted by `sid`
  - ◆Result of second join sorted by `sid`

Beyond Pass3 : First Example

- Next, consider GROUP BY clause
  - ◆Require sorting on `sid`
  - ◆For each plan retained in Pass 3, if result is not sorted on `sid`, add cost of sorting
  - ◆Example plan in Pass 3 produce tuples in `sid` order
    - May be the cheapest even if there is a cheaper plan joining all three relations but does not produce tuples in `sid` order
- Aggregation on the fly
- Finally, choose the cheapest plan

Join Order Selection: Second Example

- Consider the Join: R $\bowtie$ S $\bowtie$ T $\bowtie$ U
- Statistics: each relation has 1000 tuples



R(a,b)	S(b,c)	T(c,d)	U(d,a)
V(R,a)=100			V(U,a)=50
V(R,b)=200	V(S,b)=100		
	V(S,c)=500	V(T,c)=20	
		V(T,d)=50	V(U,d)=1000

- Cost estimation: take the size (in tuple) of intermediate results as the cost
  - ◆ the simplest estimation of cost
- Plans: sequence of relations in the left-deep join tree
  - ◆Example: R,S,T,U means R $\bowtie$ S $\bowtie$ T $\bowtie$ U

41

Join Order Selection: Second Example

- Cost taken to be size of intermediate results
  - Actually should take I/O cost or some other cost metric
- Best sub-plans involving one relation

	{R}	{S}	{T}	{U}
size	1000	1000	1000	1000
cost	0	0	0	0
best plan	R	S	T	U

- Best sub-plans involving 2 relations

	{R,S}	{R,T}	{R,U}	{S,T}	{S,U}	{T,U}
size	5000	1M	10000	2000	1M	1000
cost	0	0	0	0	0	0
best plan	R ⋈ S	R ⋈ T	R ⋈ U	S ⋈ T	S ⋈ U	T ⋈ U

Join Order Selection: Second Example

- Best sub-plans involving 3 relations

	{R,S,T}	{R,S,U}	{R,T,U}	{S,T,U}
size	10000	50000	10000	2000
cost	2000	5000	1000	1000
best plan	(S⋈ T)⋈ R	(R⋈ S)⋈ U	(T⋈ U)⋈ R	(T⋈ U)⋈ S

- Best plans involving 4 relations
- Only left-deep trees considered

Plan	Cost
((S⋈T)⋈ R)⋈ U	12k
((R⋈S)⋈ U)⋈ T	55k
((T⋈U)⋈ R)⋈ S	11k
((T⋈U)⋈ S)⋈ R	3k

Join Order Selection: Second Example

- Based on the previous cost estimation
- The best single join is T $\bowtie$ S, with cost 1000
- There are two possible longer plans
  - (T $\bowtie$ U) $\bowtie$ R or (T $\bowtie$ U) $\bowtie$ S
- Choose the second plan of which the cost is 2000
- The final best order is then ((T $\bowtie$ U) $\bowtie$ S) $\bowtie$ R with cost 3000

Derivation of the Number of Possible Join Orderings

- Let J(n) denote the number of different join orderings for a join of n argument relations
  - Obviously, J(n) = T(n) * n! . . . with T(n) the number of different binary tree shapes and n! the number of leaf permutations
- We can now derive T(n) inductively:

  $T(1) = 1,$

  $T(n) = {}_1\Sigma^{n-1}\ T(i) * T(n - i)$

. . .namely, $T(n) = \Sigma_{allpossibilities} T(leftsubtree) * T(rightsubtree)$

- It turns out that T(n) = C(n − 1), for C(n) the n-th Catalan number,

$$C(n) = \frac{1}{n+1}\binom{2n}{n} = \frac{(2n)!}{(n+1)!\cdot n!}$$

- Substituting T(n) = C(n − 1), we obtain

  $T(n) * n! = (2(n - 1))!/(n - 1)!$

43

Dynamic Programming Algorithm: Complexity

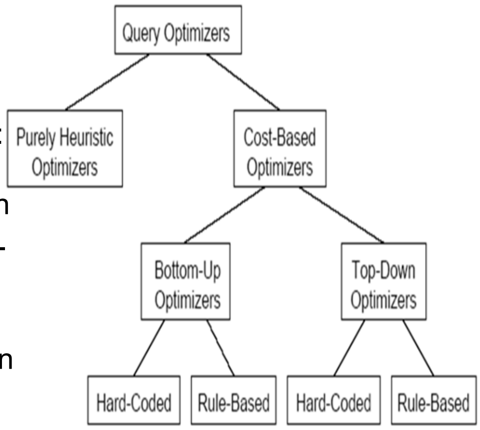
- Consider finding the best join-order for $R_1 \bowtie R_2 \bowtie \ldots R_n$.
  - There are $(2(n-1))!/(n-1)!$ different join orders for above expression
    - With $n = 7$, the number is 665280, with $n = 10$, the number is greater than 176 billion!
- Cost-based optimization is expensive, even with dynamic programming (bushy trees)
  - time complexity is $O(3^n)$
    - With $n = 10$, this number is 59000 instead of 176 billion!
  - space complexity is $O(2^n)$
- If only left-deep trees are considered
  - time complexity of finding best join order is $O(n\ 2^n)$ while
  - space complexity remains at $O(2^n)$
- Cost-based optimization is expensive, but worthwhile for queries on large datasets (typical queries have small n, generally < 10)

Points to Remember

- Consider single-relation queries:
  - All access methods considered, cheapest is chosen
  - Selections that *match* index, whether index key has all needed fields and/or provides tuples in a desired order
- Compute multiple-relation queries in a left-deep manner:
  - All single-relation plans are first enumerated
    - Selections/projections considered as early as possible
  - Next, for each 1-relation plan, all ways of joining another relation (as inner) are considered
  - Next, for each 2-relation plan that is `retained', all ways of joining another relation (as inner) are considered, etc.
  - At each level, for each subset of relations, only best and nearly best plans are `retained'
    - a plan is considered nearly best if its output has some interesting order, e.g., is sorted

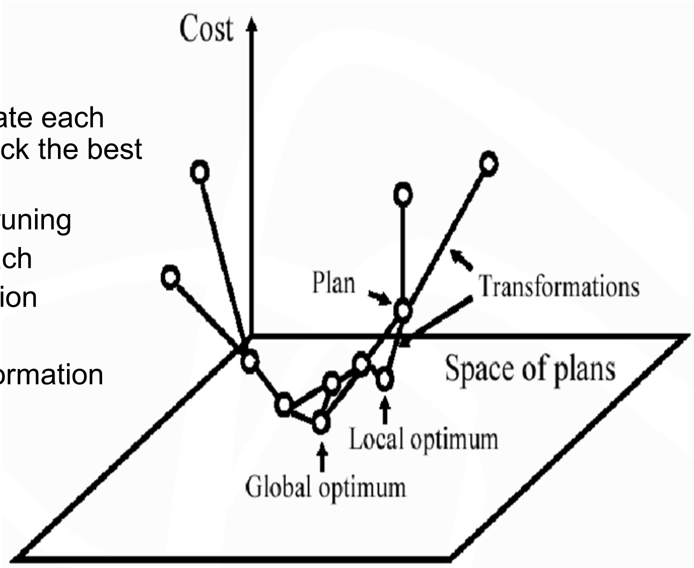
Structure of Query Optimizers

- Some query optimizers integrate heuristic selection and the generation of alternative access plans
  - ◆System R and Starburst use a hierarchical procedure based on the nested-block concept of SQL: heuristic rewriting followed by cost-based join-order optimization
- Even with the use of heuristics, cost-based query optimization imposes a substantial overhead
  - ◆This expense is usually more than offset by savings at query-execution time, particularly by reducing the number of slow disk accesses



Join Enumeration (Plan Search) Algorithms

- Exhaustive: enumerate each possible plan, and pick the best
- Bottom-up dynamic programming with pruning
  - ◆System-R approach
- Top-down memoization
- Greedy Techniques
- Randomized/Transformation Techniques



Branch-and-Bound Pruning

- Use heuristics to find a reasonably good physical plan, compute its cost C and set it as initial bound
  - build other physical plans piece by piece, each unfinished plan is called a sub-plan
- When expanding a sub-plan, if the cost of the expanded sub-plan is higher than the bound, abandon the sub-plan
  - If a new complete plan has a cost C' that is lower than the bound, set C' as the new bound and continue
  - May stop when the bound is low enough
- Observation: Pruning performance very dependent upon order that plans are derived
  - Important to find a good plan early!

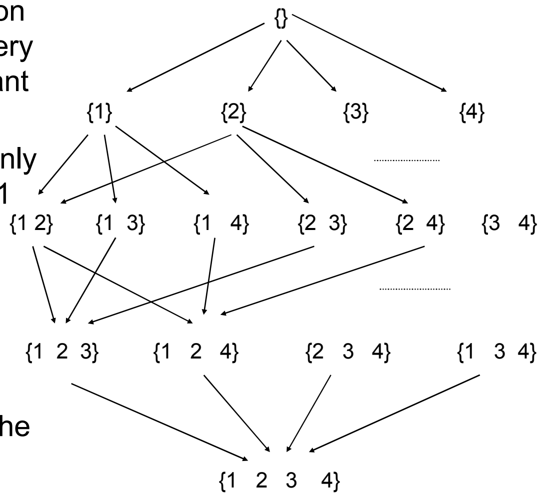
Bottom-up Plan Generation

- Bottom-up generation of optimal plans:
  - Compute the optimal plan for joining k relations
  - Suboptimal plans are pruned
  - From these plans, derive the optimal plans for joining k+1 relations
- The dynamic programming algorithm proceeds by considering increasingly larger subsets of the set of all relations
- Assumption 1: Once we have joined k relations, the method of joining this result further with another relation is independent of the previous join methods (true?)
- Assumption 2: Any subplan of an optimal plan must also be optimal, otherwise we could replace the subplan to get a better overall plan (true?)

CSD Univ. of Crete — Fall 2012

Bottom-up Plan Generation

- Elegant iterative process
- Search space is pruned based on the principal of optimality for every sub-expression with no redundant work
- If restricted to left-deep plans, only requires keeping sizes 1 and k-1 to calculate size k
- First plan not completed until nearly done
- Do we really need the optimal plan for every sub-expression?
  - ◆We only need to guarantee the overall plan is optimal



93

CSD Univ. of Crete — Fall 2012

Improvement: “Interesting Order”

- Assumption 2 does not always hold!!
  - ◆Example: R(A,B) $\bowtie$ S(A,C) $\bowtie$ T(A,D)
  - ◆Best plan for R $\bowtie$ S: hash join (beats sort-merge join)
  - ◆Best overall plan: sort-merge join R and S, then sort-merge join with T
    - Subplan of the optimal plan is not optimal
- Why?
  - ◆The result of the sort-merge join of R and S is sorted on A
  - ◆This is an interesting order that can be exploited by later processing (e.g., join, duplicate elimination, GROUP BY, ORDER BY, etc.)!
- Not sufficient to find the best join order for each subset of the set of k given relations (Hill Climbing)
  - ◆Must find the best join order for each subset, for each interesting sort order of the join result for that subset
  - ◆Simple extension of earlier dynamic programming algorithm

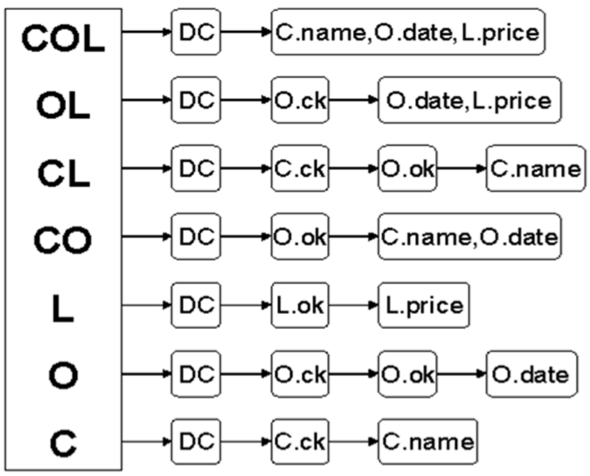
94

Bottom-up Dynamic Programming

```
Procedure OptLDPlan(Q):
  for each R_i ∈ Q
    initialize Table[R_i,*] to dummy plan with cost ∞
    for each plan p that accesses R_i
      for each interesting order o that p satisfies
        if (Cost(p) < Cost(Table[R_i, o])) then Table[R_i, o] := p
  for k = 2 to |Q|
    for each S ⊆ Q of size |S| = k
      initialize Table[S,*] to dummy plan with cost ∞
      for each R_i ∈ S
        let S_i = S \ {R_i}
        generate all plans for S_i ⋈ R_i from Table[S_i,*] and Table[R_i,*]
        for each such plan p
          for each interesting order o that p satisfies
            if (Cost(p) < Cost(Table[S, o])) then Table[S, o] := p
    if (k >= 3) then delete from Table[] entries for size k-1
  return Table[Q,*]
```


Bottom-up Dynamic Programming

```
SELECT *
FROM C, O, L
WHERE C.ck = O.ck
AND O.ok = L.ok
ORDER BY C.name,
  O.date, L.price
```



Top-down Memoization

- Any bottom-up DP algorithm can be re-written as top-down memoization
- Main idea:
  - Don't compute plans for sub-expressions until needed
  - Save optimal plans in lookup table to avoid redundant work
- Observations:
  - Natural functional programming
  - First complete plan created early
  - Potentially yields the same plans as DP, but generated in different order
  - Overhead due to "random" table lookups
  - "Random" access to table forces entire table to be kept until algorithm completes (unlike DP for left-deep plans)
  - If interesting orders not identified ahead of time, avoiding redundant work requires all considered plans to be remembered
  - Generating complete plans early on enables Branch&Bound pruning!
    - Initialize upper bound to ∞
    - Compute plan cost as assembled top-down
    - Abandon any (sub-)plan as soon as cost exceeds current bound
    - Use completed plans to refine upper bound

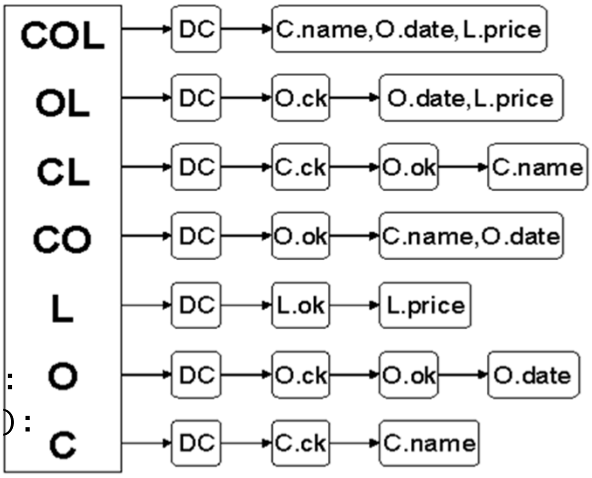
Top-down Memoization

```
Function OptLDPlan(Query Q, Order o):
   if NotEmpty(Table[Q,o])
      return Table[Q,o]
   optplan := dummy plan with cost ∞
   for each R_i ∈ Q
      let S_i := Q \ {R_i}
      for each join method Op that will satisfy o
         plan p := Op(OptLDPlan(S_i, o_1), OptLDPlan(R_i, o_2))
            where o_1 and o_2 are the orders required by Op
         if (Cost(p) < Cost(optplan))
            optplan := p
   Table[Q,o] := optplan
   return optplan
```

Top-down Memoization

```
SELECT *
FROM C, O, L
WHERE C.ck = O.ck
AND O.ok = L.ok
ORDER BY C.name,
   O.date, L.price
```

```
HJ((CO,DC),(L,DC)):
HJ((C,DC),(O,DC)):
------
MSJ((C,C.ck),(O,C.ck)):
MSJ((CO,O.ok),(L,L.ok)):
NLJ((O,O.ok), (C,DC)):
```



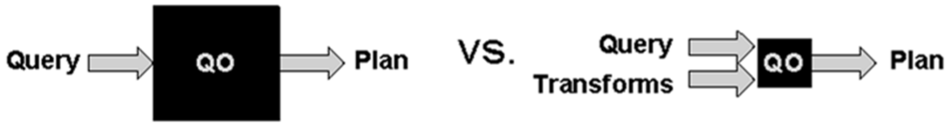
Join Enumeration Algorithms Comparison

Bottom-up DP:
- Finds optimal solution in $O(3^n)$ time and space (bushy)
- First complete plan generated late
- Interesting orders generated eagerly, so caution needed in defining what is an interesting order

Top-down Memoization:
- ◆Finds optimal solution in $O(3^n)$ time and space (bushy)
- ◆Time constant larger than DP due to table lookup
- ◆Space constant larger b/c everything must be remembered
- Generates complete plans early
- Only optimizes requested interesting orders, so definition can be loose without necessarily impacting performance
- Allows branch-and-bound pruning

Motivation for Transformation-based QO

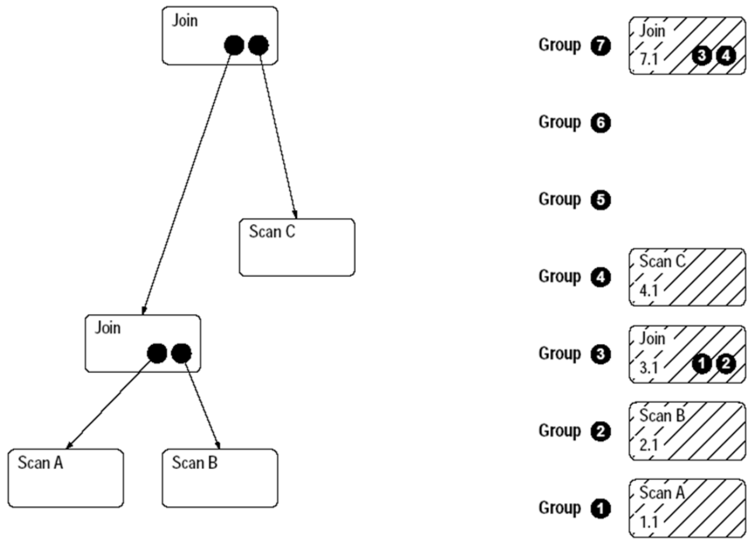
- Provide single framework for QO
  - Remove rigid separation between logical rewriting and cost-based join enumeration
  - Allow any decision to be either cost-based or heuristic, independent of stage at which it occurs
- Use encapsulation to remove pruning semantics from the search algorithm
  - Simplifies comprehension
  - Improves extensibility
  - Provides flexibility in modifying search space



Structure of a Transformation-based Optimizer

- Query representation: algebraic tree
  - Logical & physical operators
  - Operator order always defined (unlike Query Graph Model (QGM))
- Multiple trees stored in MEMO structure composed of two entities:
  - Multi-expressions
    - Operator having groups as inputs
    - Logical or physical
  - Groups
    - Set of logically equivalent multi-expressions
    - Mixture of logical and physical multi-expressions
    - “Winner’s circle” identifying optimal multi-expression for each requested physical property
- Optimization = Tasks + Memo
  (Programs = Algorithms + Data Structures)

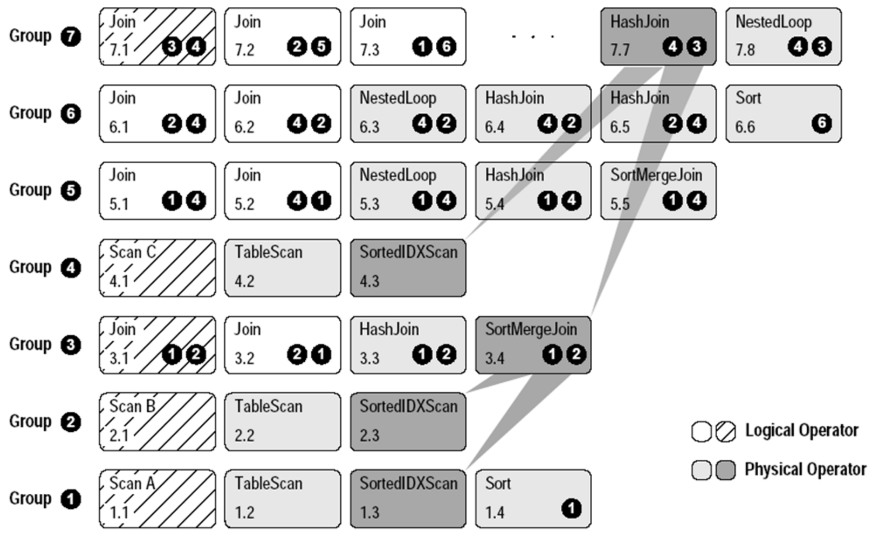
Initial Plan MEMO Structure



Types of Transformations

- Logical Substitution (LogOp → LogOp)
  - Old multi-expression is replaced by new
  - For guaranteed-win scenarios, or when you want to make heuristic decisions to minimize search space
    - E.g. predicate pushdown, subquery-to-join rewriting
- Logical Transformation (LogOp → LogOp)
  - Both old and new multi-expression are kept and optimized further
  - For cost-based decision between different logical plans
    - E.g. join ordering, group_by pushdown/pullup
- Physical Transformation (LogOp → PhysOp)
  - Generate physical multi-expression that implements logical
    - E.g. access method selection, join algorithm selection

MEMO Structure Representing Alternative Plans



How Optimization Proceeds?

- Every transform specifies a pattern
- To optimize multi-expression e:
  - First optimize each input group, possibly specifying required physical properties
  - Apply every rule whose pattern matches e
  - Insert resulting multi-expression into MEMO rooted in same group
- To optimize group g given physical property p
  - Optimize each multi-expression in g
  - Store identity of optimal multi-expression for p in winner's circle

Comparison to Join Enumeration

- Query tree is traversed top-down; memoization of subexpressions is used; Similarities to top-down join enumeration with memoization:
  - ❶Complete plans generated early
  - ❷Branch-and-bound pruning can be used
  - ❸Entire memo structure must be remembered
- Differences:
  - ❶Cost-based optimization for arbitrary logical transforms
  - ❷Search space determined by current set of rules
    - Easily modify search space by adding/removing rules
    - Search algorithm (application of transform rules) independent from semantics of transforms
      - Improves extensibility
      - Redundantly derives same expression by different paths (leads to O(4n) work for join enumeration unless rules disable each other)
  - ❸Optimizes global query, not local to SPJG blocks
    - Memory consumption proportional to size of global query, not just size of single block (interacts with Similarity #3)

History of Bottom-up Join Enumeration

- SYSTEM-R (basis of early DB2 optimizers): Selinger et al., “Access method selection in a Relational Database Management System”, SIGMOD 1979
  - ◆Proposed dynamic programming approach
  - ◆Introduced concept of “interesting orders”
- STARBURST (basis of DB2 UDB): Haas et al., “Extensible Query Processing in Starburst”, SIGMOD 1989; Ono and Lohman, “Measuring the Complexity of Join Enumeration in Query Optimization”, VLDB 1990
  - ◆Focus on making QO extensible for new query operators and optimization rules
  - ◆Propose two-phase optimization: logical rewrite followed by grammar-based join enumeration (evaluated bottom-up)
  - ◆Parameterize the join enumeration algorithm to tailor search space
    - Customization limited by predetermined switches
    - E.g. Bushy vs. left-deep, Cartesian products or not, etc.

History of Top-down Join Enumeration

- Naïve Translation of System-R algorithm Chaudhuri et al., "Optimizing queries with materialized views", SIGMOD 1995
  - Re-cast System-R DP as functional algorithm with memoization
  - Purely conceptual reasons, no attempt to exploit top-down
- Sybase ASA Bowman and Paulley, "Join Enumeration in a Memory-Constrained Environment", ICDE 2001
  - Back-tracking search without memoization (to minimize memory usage)
  - Aggressive branch-and-bound pruning
  - "Optimizer governor" heuristically guides search order
    - Attempts to locate good plans early to provide good upper bounds
    - Decides when to terminate search
  - Not guaranteed to find optimal plan (would take O(n!) time)
- COLUMBIA (actually a transformational optimizer) Shapiro et al.,"Exploiting Upper and Lower Bounds in Top-Down Query Optimization", IDEAS 2001
  - Focuses on value of branch-and-bound pruning during join enumeration (humorously christened "group pruning")

History of Transformation-based QO

- EXODUS Graefe and DeWitt, "The EXODUS Optimizer Generator", SIGMOD 1987
  - Iustrated plan generation via transforms
  - Efficiency problems due to poor memorization
- VOLCANO Graefe and McKenna, "The Volcano Optimizer Generator: Extensibility and Efficient Search", ICDE 1993
  - MEMO structure to avoid repeated work
  - Flexible cost models and physical properties (generalization of interesting orders)
  - Two phases: full logical expansion, then physical generation
  - Conservative branch-and-bound pruning (no lower-bounding of inputs)
- Microsoft SQL Server Pellenkoft et al., "The Complexity of Transformation-based Join Enumeration", VLDB 1997
  - Proves independent transformations lead to redundant work due to multiple derivation paths; must analyze inter-rule dependencies to avoid

History of Transformation-based QO

- CASCADES (basis of SQL Server and Tandem NonStop SQL) Graefe, "The Cascades Framework for Query Optimization", IEEE Data Eng. Bull. 18(3), 1995
  - Interleave logical and physical optimization
  - Generates first physical plan earlier
  - Makes branch-and-bound pruning more useful
  - Minor modifications of VOLCANO to improve efficiency and extensibility
- COLUMBIA (see previous slide) Shapiro et al., "Exploiting Upper and Lower Bounds in Top-Down Query Optimization", IDEAS 2001
  - Added lower-bounding of inputs to branch-and-bound pruning of CASCADES
  - Prove certain set of transforms enumerate all bushy trees and guarantee first plan generated for each group will be left-deep

Summary

- Query optimization is a very complex task
  - Combinatorial explosion
  - The task is to find one good query execution plan, not the best one
- No optimizer optimizes all queries adequately
  - Heuristic optimization is more efficient to generate, but may not yield the optimal query execution plan
  - Cost-based optimization relies on statistics gathered on the relations
- Until query optimization is perfected, query tuning is a fact of life
  - It has a localized effect and is thus relatively attractive
  - It is a time-consuming and specialized task
  - It makes the queries harder to understand
  - This is not likely to change any time soon
- Optimization is the reason for the lasting power of the relational DBMS
  - But it is primitive in some ways
  - New areas: random statistical approaches (e.g., *simulated annealing)*, adaptive runtime re-optimization (e.g., *eddies*)

References

- Based on slides from:
  - R. Ramakrishnan and J. Gehrke
  - H. Garcia Molina
  - J. Hellerstein
  - S. Chaudhuri
  - A. Silberschatz, H. Korth and S. Sudarshan
  - P. Lewis, A. Bernstein and M. Kifer
  - D. DeHaan
  - L. Mong Li

Highlights of System R Optimizer

❶ Find all plans for accessing each base table

❷ For each table
  - Save cheapest unordered plan
  - Save cheapest plan for each interesting order
  - Discard all others

❸ Try all ways of joining pairs of 1-table plans; save cheapest unordered + interesting ordered plans

❹ Try all ways of joining 2-table with 1-table

❺ Combine k-table with 1-table till you have full plan tree

❻ At the top, to satisfy GROUP BY and ORDER BY
  - Use interesting ordered plan
  - Add a sort node to unordered plan

System R Example: Search Trees

EMP

NAME	DNO	JOB	SAL
SMITH	50	12	8500
JONES	50	5	15000
DOE	51	5	9500

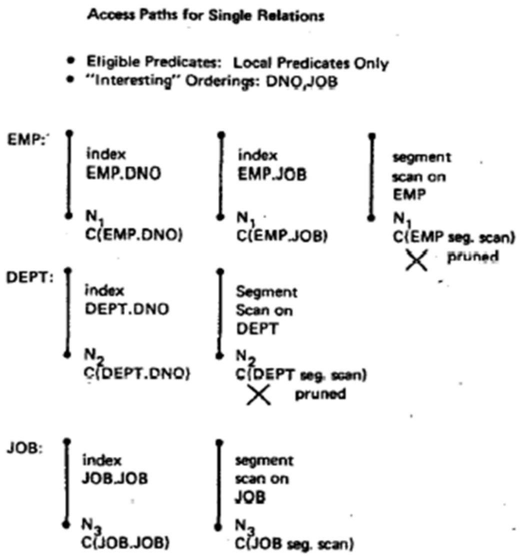
DEPT

DNO	DNAME	LOC
50	MFG	DENVER
51	BILLING	BOULDER
52	SHIPPING	DENVER

JOB

JOB	TITLE
5	CLERK
6	TYPIST
9	SALES
12	MECHANIC

```
SELECT  NAME, TITLE, SAL, DNAME
FROM    EMP, DEPT, JOB
WHERE   TITLE='CLERK'
AND     LOC='DENVER'
AND     EMP.DNO=DEPT.DNO
AND     EMP.JOB=JOB.JOB
```



Source: Selinger et al, "Access Method Selection in a Relational Database Management System"

System R Example: Search Trees

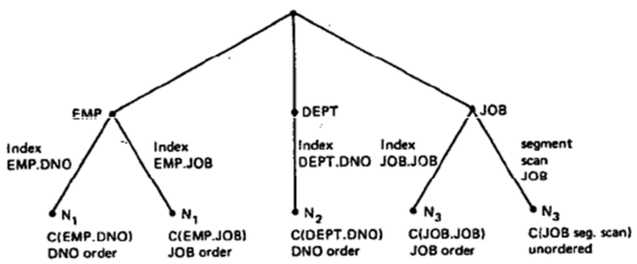


Figure 3. Search tree for single relations

Source: Selinger et al, "Access Method Selection in a Relational Database Management System"

System R Example: Search Trees

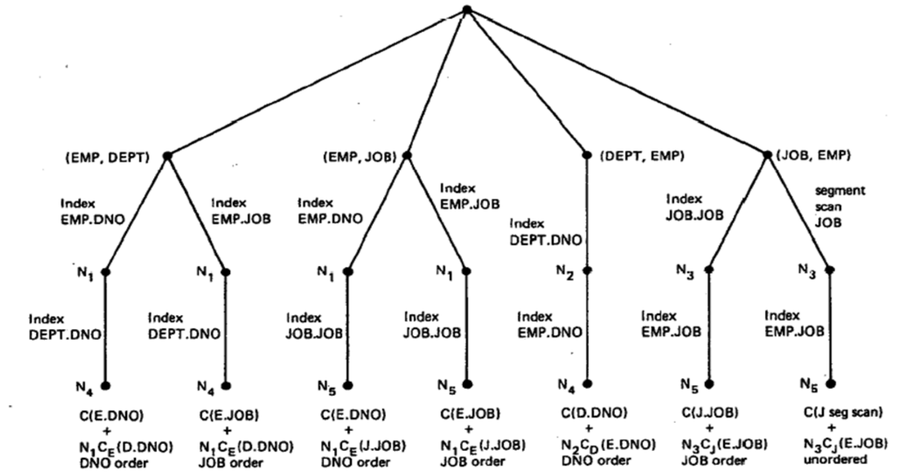


Figure 4. Extended search tree for second relation (nested loop join)

Source: Selinger et al, "Access Method Selection in a Relational Database Management System"

System R Example: Search Trees

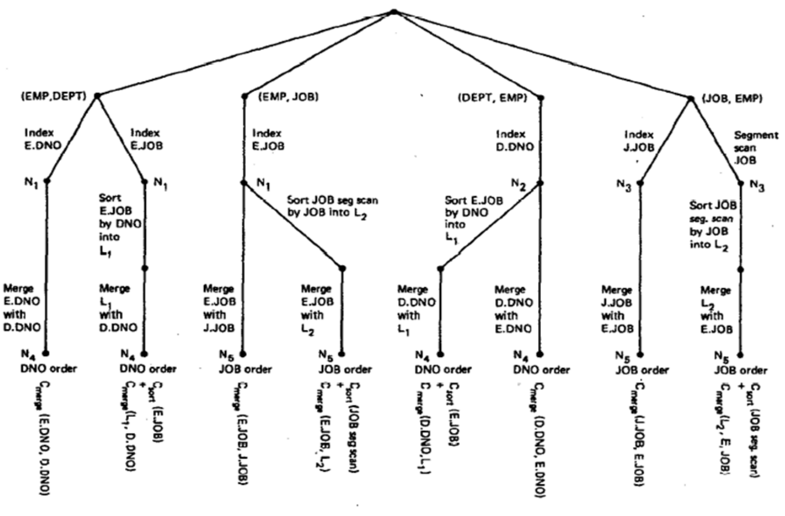


Figure 5. Extended search tree for second relation (merge join)

Source: Selinger et al, "Access Method Selection in a Relational Database Management System"

CSD Univ. of Crete Fall 2012

System R Example: Search Trees

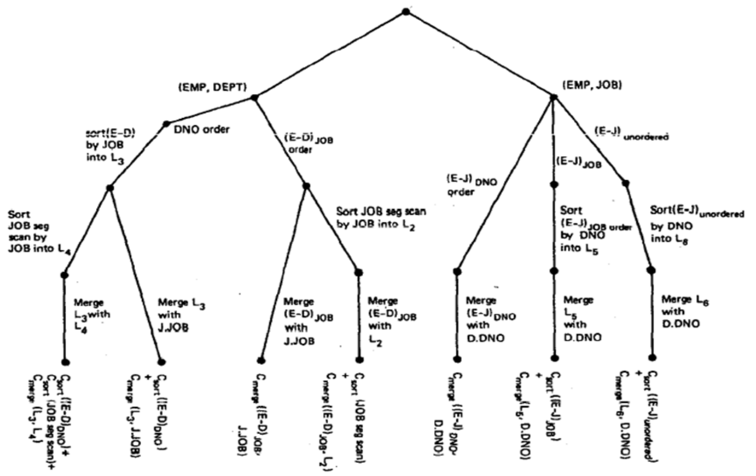


Figure 6. Extended search tree for third relation

119

Source: Selinger et al, "Access Method Selection in a Relational Database Management System"

CSD Univ. of Crete Fall 2012

Highlights of System R Optimizer

- Impact:
  - Most widely used currently; works well for < 10 joins
- Cost estimation:
  - Very inexact, but works ok in practice
  - Statistics, maintained in system catalogs, used to estimate cost of operations and result sizes
  - Considers combination of CPU and I/O costs
  - More sophisticated techniques known now
- Plan Space: Too large, must be pruned
  - Only the space of *left-deep plans* is considered
  - Cartesian products avoided

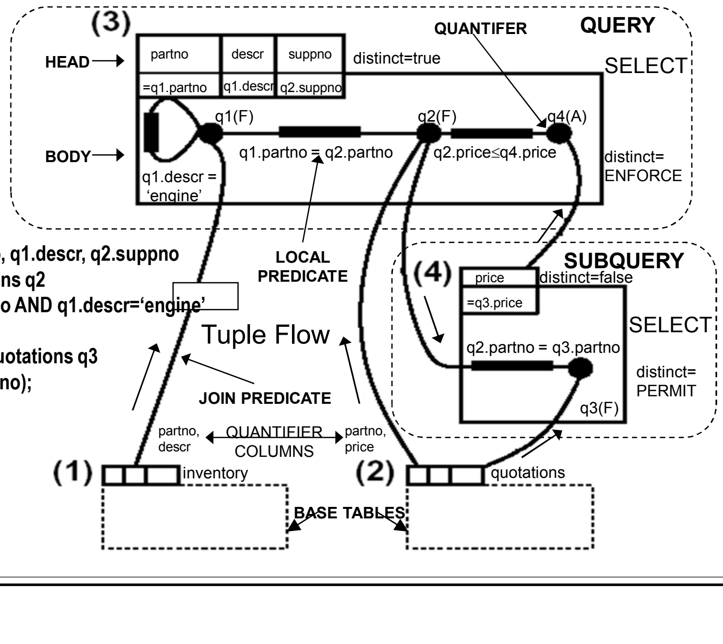
120

Highlights of Starburst Optimizer

- SQL is not a pure declarative query language as it has imperative features
- Complex queries can contain subqueries and views
  - These naturally divide a query into nested blocks and
  - can create evaluation path expressions
- Traditional DBMS only perform plan optimization on a single query block at a time and perform *no cross-block optimization*
  - The result: Query optimizers are often forced to choose a sub-optimal plan
  - The problem: Query generators can produce very complex queries and databases are getting bigger
    - The penalty for poor planning is getting larger

Starburst: The Query Graph Model (QGM)

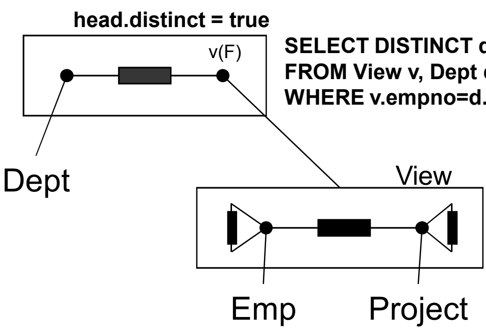
SELECT DISTINCT q1.partno, q1.descr, q2.suppno
FROM inventory q1, quotations q2
WHERE q1.partno = q2.partno AND q1.descr='engine'
AND q2.price ≤ ALL
(SELECT q3.price FROM quotations q3
WHERE q2.partno=q3.partno);



Starburst: Rule Based Query Rewrite (QRW)

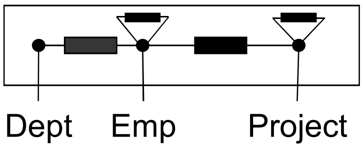
- The goals of query rewriting
  - Make queries as declarative as possible
    - Transform “procedural” queries
    - Perform unnesting/flattening on nested blocks
  - Retain the semantics of the query (same answer)
- How?
  - Perform natural heuristics *E.g. “predicate pushdown”*
  - Production rules encapsulate a set of Query Rewrite heuristics
- A Single Rewrite Philosophy
  - “Whenever possible, a query should be converted to a single select operator”
- The Result
  - The Standard optimizer is given the maximum latitude possible

Example of Rule 1: SELECT Merge



SELECT DISTINCT d.deptno, v.lastname
FROM View v, Dept d
WHERE v.empno=d.mgmo

SELECT empno, lastname
FROM Emp, Project
WHERE salary<20000 AND
workno=projno AND
pbudget>500000



SELECT DISTINCT d.deptno,e.lastname
FROM Emp e, Dept d, Project p
WHERE e.empno=d.mgmo AND
e.salary<20000 AND
e.workno=p.projno AND
p.pbudget>500000

Highlights of Starburst Optimizer

- The problem: Complex SQL queries can contain nested blocks that can't be optimized using the standard plan optimizer
- The solution: By rewriting the query to a semantically equivalent query with fewer boxes the (near) optimal plan can be found
- The Query Graph Model (QGM) provides an abstract view of queries that is suitable for most rule transformations
  - Mechanisms are provided for dealing with duplicates
- Rewrite Rule Engine: Condition->action rules where LHS and RHS are arbitrary C functions on
  - QGM representation
  - Rules and Classes for search control
  - Conflict Resolution Schemes
- Bottom up enumeration of plans
- Grammar-like set of production rules to generate execution plans
  - LOLEPOP: terminals (physical operators)
  - STAR: production rules (alternative implementations of query graph blocks)
  - GLUE: additional rules for achieving a given property (order)

Highlights of Volcano Optimizer

- Query as an algebraic tree
- Transformation rules
  - Logical rules,
  - Implementation rules
- Optimization goal
  - Logical expression,
  - Physical properties,
  - Estimated cost
- Top down algorithm
  - Logical expressions optimized on demand
    - Enumerate possible moves
    - Implement operator
    - Enforce property
    - Apply transformation rules
  - Select move based on promise
  - Branch and bound