Relational Query Optimization

Review

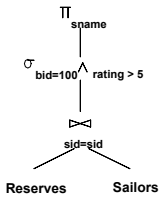
- **Choice of single-table operations**
  - Depends on indexes, memory, stats,…
- **Joins**
  - Blocked nested loops:
    - simple, exploits extra memory
  - Indexed nested loops:
    - best if 1 rel small and one indexed
  - Sort/Merge Join
    - good with small amount of memory, bad with duplicates
  - Hash Join
    - fast (enough memory), bad with skewed data
- **These are "rules of thumb"**
  - On their way to a more principled approach…

Query Optimization Overview

- Query can be converted to relational algebra
- Rel. Algebra converts to tree
- Each operator has implementation choices
- Operators can also be applied in different orders!

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

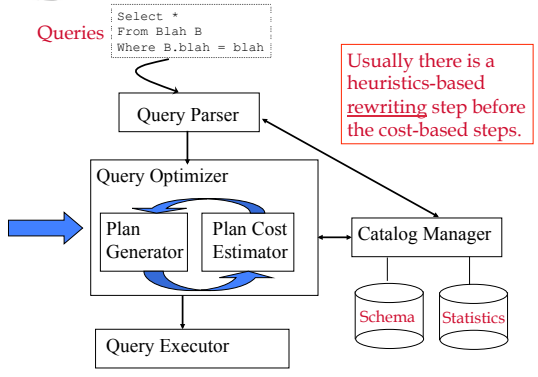
$\pi_{(sname)}\sigma_{(bid=100 \wedge rating > 5)}$
(Reserves ⋈ Sailors)



Query Optimization Overview (cont.)

- ***Plan:*** Tree of R.A. ops (and some others) with choice of algorithm for each op.
  - Recall: Iterator interface (next()!)
- **Three main issues:**
  - For a given query, what plans are considered?
  - How is the cost of a plan estimated?
  - How do we "search" in the "plan space"?
- **Ideally: Want to find best plan.**
- **Reality: Avoid worst plans!**

Cost-based Query Sub-System



Let's go through some examples

- **Just to get a flavor…**

Schema for Examples

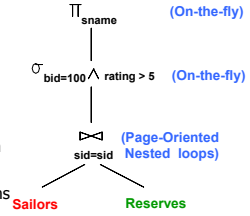
Sailors (*sid*: integer, *sname*: string, *rating*: integer, *age*: real)
Reserves (*sid*: integer, *bid*: integer, *day*: dates, *rname*: string)

- As seen in previous lectures…
- **Reserves:**
  - Each tuple is 40 bytes long, 100 tuples per page, 1000 pages.
  - Assume there are 100 boats
- **Sailors:**
  - Each tuple is 50 bytes long, 80 tuples per page, 500 pages.
  - Assume there are 10 different ratings
- Assume we have 5 pages in our buffer pool!

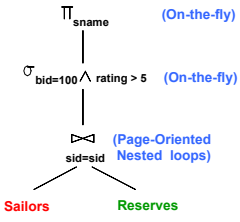
Motivating Example

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

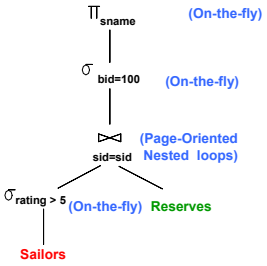
- Cost: 500+500*1000 I/Os
- By no means the worst plan!
- Misses several opportunities:
  - selections could be 'pushed' down
  - no use made of indexes
- *Goal of optimization:* Find faster plans that compute the same answer.



Alternative Plans – Push Selects (No Indexes)

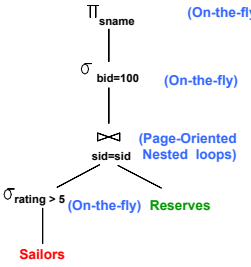


500,500 IOs

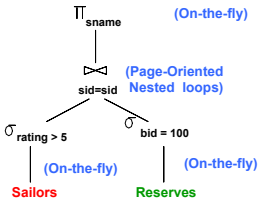


250,500 IOs

Alternative Plans – Push Selects (No Indexes)

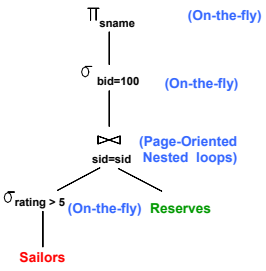


250,500 IOs

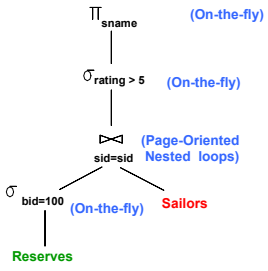


250,500 IOs

Alternative Plans – Push Selects (No Indexes)

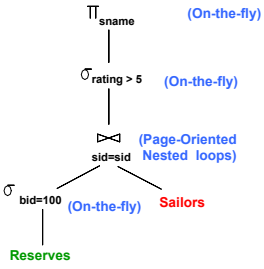


250,500 IOs

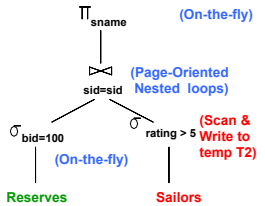


6000 IOs

Alternative Plans – Push Selects (No Indexes)

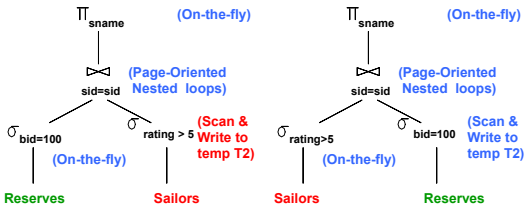


6000 IOs



4250 IOs
1000 + 500+ 250 + (10 * 250)

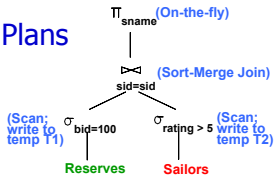
Alternative Plans – Push Selects (No Indexes)



4250 IOs

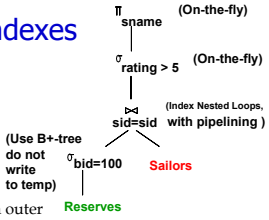
4010 IOs
500 + 1000 +10 +(250 *10)

More Alternative Plans (No Indexes)



- ***Sort Merge Join***
- **With 5 buffers, cost of plan:**
  - Scan Reserves (1000) + write temp T1 (10 pages) = 1010.
  - Scan Sailors (500) + write temp T2 (250 pages) = 750.
  - Sort T1 (2*2*10) + sort T2 (2*4*250) + merge (10+250) = 2300
  - Total: 4060 page I/Os.
- If use Chunk NL join, join = 10+4*250, total cost = 2770.
- Can also `push' projections, but must be careful!
  - T1 has only *sid*, T2 only *sid*, *sname*:
  - T1 fits in 3 pgs, cost of Chunk NL under 250 pgs, total < 2000.

More Alt Plans: Indexes



- **With clustered index on *bid* of Reserves, we access:**
  - 100,000/100 = 1000 tuples on 1000/100 = 10 pages.
- **INL with outer not materialized.**
  - Projecting out unnecessary fields from outer doesn't make an I/O difference.
- Join column *sid* is a key for Sailors.
  - At most one matching tuple, unclustered index on *sid* OK.
- Decision not to push *rating>5* before the join is based on availability of *sid* index on Sailors.
- Cost: Selection of Reserves tuples (10 I/Os); then, for each, must get matching Sailors tuple (1000*1.2); total 1210 I/Os.

Summing up

- There are *lots* of plans
  - Even for a relatively simple query
- People often think they can pick good ones by hand
  - MapReduce is based on that assumption
- Not so clear that's true!
  - Machines are better at enumerating options than people
  - But we will see soon how optimizers make simplifying assumptions

What is needed for optimization?

- Given: A closed set of operators
  - Relational ops (table in, table out)
  - Encapsulation (e.g. based on iterators)

1. **Plan space**
   - Based on relational equivalences, different implementations
2. **Cost Estimation** based on
   - Cost formulas
   - Size estimation, in turn based on
     - Catalog information on base tables
     - Selectivity (Reduction Factor) estimation
3. **A search algorithm**
   - To sift through the plan space and find lowest cost option!

Query Optimization

- We'll focus on "System R" (Selinger) style optimizers

Access Path Selection
in a Relational Database Management System

P. Griffiths Selinger
M. M. Astrahan
D. D. Chamberlin
R. A. Lorie
T. G. Price

IBM Research Division, San Jose, California 95193

ABSTRACT: In a high level query and data manipulation language such as SQL, requests are stated non-procedurally, without reference to access paths. This paper describes how System R chooses access paths retrieval. Nor does a user specify in what order joins are to be performed. The System R optimizer chooses both join order and an access path for each table in the SQL statement. Of the many possible

Highlights of System R Optimizer

Works well for 10-15 joins.

1. **Plan Space: Too large, must be pruned.**
   - Many plans share common, "overpriced" subtrees
     - ignore them all!
   - In some implementations, only consider *left-deep plans*
   - In some implementations, Cartesian products avoided
2. **Cost estimation**
   - Very inexact, but works ok in practice.
   - Stats in system catalogs used to estimate sizes & costs
   - Considers combination of CPU and I/O costs.
   - System R's scheme has been improved since that time.
3. **Search Algorithm: Dynamic Programming**

Query Optimization

1. **Plan Space**
2. **Cost Estimation**
3. **Search Algorithm**

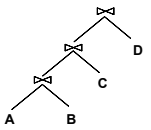
Query Blocks: Units of Optimization

- Break query into *query blocks*
- Optimize one block at a time
- Uncorrelated nested blocks computed once
- Correlated nested blocks like function calls
  - But sometimes can be "decorrelated"
  - Beyond the scope of CS186!

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

Outer block *Nested block*

❖ For each block, the plans considered are:
- All available access methods, for each relation in FROM clause.
- All *left-deep join trees*
  - right branch always a base table
  - consider all join orders and join methods



Schema for Examples

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

- **Reserves:**
  - Each tuple is 40 bytes long, 100 tuples per page, 1000 pages. 100 distinct bids.
- **Sailors:**
  - Each tuple is 50 bytes long, 80 tuples per page, 500 pages. 10 ratings, 40,000 sids.

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"
GROUP BY S.sid

HAVING COUNT (*) >= 2
```

For each sailor with 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"
GROUP BY S.sid

HAVING COUNT (*) >= 2
```

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

Relational Algebra Equivalences

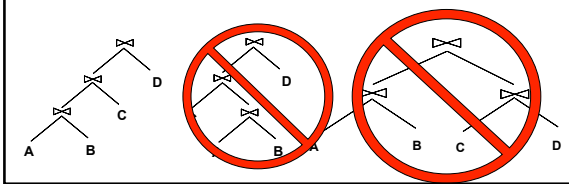
- Allow us to choose different join orders and to "push" selections and projections ahead of joins.
- **Selections:**
  - $\sigma_{c1 \wedge \ldots \wedge cn}(R) \equiv \sigma_{c1}(\ldots(\sigma_{cn}(R))\ldots)$ (*cascade*)
  - $\sigma_{c1}(\sigma_{c2}(R)) \equiv \sigma_{c2}(\sigma_{c1}(R))$ (*commute*)
- **Projections:**
  - $\pi_{a1}(R) \equiv \pi_{a1}(\ldots(\pi_{a1, \ldots, an-1}(R))\ldots)$ (*cascade*)
- **Cartesian Product**
  - $R \times (S \times T) \equiv (R \times S) \times T$ (associative)
  - $R \times S \equiv S \times R$ (commutative)
  - *This means we can do joins in any order.*
    - But…beware of cartesian product!
    - $R \times (S \bowtie T) \equiv (R \times S) \bowtie T$

More Equivalences

- **Eager projection**
  - Can cascade and "push" (some) projections thru selection
  - Can cascade and "push" (some) projections below one side of a join
  - Rule of thumb: can project anything not needed "downstream"
- **Selection on a cross-product is equivalent to a join.**
  - If selection is comparing attributes from each side
- **A selection on attributes of R commutes with R ⋈ S.**
  - i.e., $\sigma(R \bowtie S) \equiv \sigma(R) \bowtie S$
  - but only if the selection doesn't refer to S!

Queries Over Multiple Relations

- **A System R heuristic: *only left-deep join trees* considered.**
  - Restricts the search space
  - Left-deep trees allow us to generate all *fully pipelined* plans.
    - Intermediate results not written to temporary files.
    - Not all left-deep trees are fully pipelined (e.g., SM join).



Query Optimization

1. Plan Space
2. **Cost Estimation**
3. Search Algorithm

Cost Estimation

- **For each plan considered, must estimate total cost:**
  - Must estimate *cost* of each operation in plan tree.
    - Depends on input cardinalities.
    - We've already discussed this for various operators
      - sequential scan, index scan, joins, etc.
  - Must estimate *size of result* for each operation in tree!
    - Because it determines downstream input cardinalities!
    - 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 + *CPU-factor* * #tuples
- Q: Is "cost" the same as estimated "run time"?

Statistics and Catalogs

- **Need info on relations and indexes involved.**
- *Catalogs* **typically contain at least:**

Statistic	Meaning
NTuples	# of tuples in a table (cardinality)
NPages	# of disk pages in a table
Low/High	min/max value in a column
Nkeys	# of distinct values in a column
IHeight	the height of an index
INPages	# of disk pages in an index

- **Catalogs updated periodically.**
  - Too expensive to do continuously
  - Lots of approximation anyway, so a little slop here is ok.
- **Modern systems do more**
  - Esp. keep more detailed statistical information on data values
    - e.g., histograms

Size Estimation and Selectivity

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

- **Max output cardinality = product of input cardinalities**
- ***Selectivity (sel)* associated with each *term***
  - reflects the impact of the *term* in reducing result size.
  - |output| / |input|

  Result cardinality = Max # tuples * $\prod sel_i$
  - *Book calls selectivity "Reduction Factor" (RF)*
- **Avoid confusion:**
  - "highly selective" in common English is opposite of a high selectivity value (|output|/|input| high!)

Result Size Estimation

- *Result cardinality* = **Max # tuples * product of all RF's.**
- Term *col=value* (given Nkeys(I) on *col*)
  RF = *1/NKeys(I)*
- Term *col1=col2* (handy for joins too…)
  RF = *1/MAX(NKeys(I1), NKeys(I2))*

  Why MAX? See bunnies on next slide…
- Term *col>value*
  RF = *(High(I)-value)/(High(I)-Low(I) + 1)*

  Implicit assumptions: values are uniformly distributed and *terms* are independent!
- *Note, if missing the needed stats, assume **1/10**!!!*

P(LeftEar = RightEar)

- 100 bunnies
- 2 distinct LeftEar colors
  $\{C_1, C_2\}$
- 10 distinct RightEar colors
  $C_1..C_{10}$
- independent ears
- $P(L=R) = \Sigma_i P(C_i,C_i)$
  $= P(C_1,C_1)+P(C_2,C_2)+P(C_3,C_3)+\ldots$
  $= (½ * 1/10) + (½ * 1/10) + (0 * 1/10) + \ldots$
  $= 1/10$
  $= 1/MAX(2,10)$

Postgres 9.4: src/include/utils/selfuncs.h

```
/* default selectivity estimate
   for equalities such as "A = b"
   */
   #define DEFAULT_EQ_SEL  0.005

   /* default selectivity
   estimate for inequalities such
   as "A < b" */
   #define DEFAULT_INEQ_SEL
   0.3333333333333333

   /* default selectivity
   estimate for range
   inequalities "A > b AND A < c"
   */
   #define DEFAULT_RANGE_INEQ_SEL
   0.005
```

```
/* default selectivity estimate
   for pattern-match operators
   such as LIKE */
   #define DEFAULT_MATCH_SEL
   0.005

   /* default number of distinct
   values in a table */
   #define DEFAULT_NUM_DISTINCT
   200

   /* default selectivity
   estimate for boolean and null
   test nodes */
   #define DEFAULT_UNK_SEL
   0.005
   #define DEFAULT_NOT_UNK_SEL
   (1.0 - DEFAULT_UNK_SEL)
```

src/backend/optimizer/path/clausesel.c

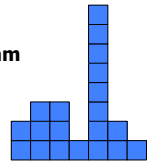
```
/*
 * This is not an operator, so we guess at the selectivity.
 * THIS IS A HACK TO GET V4 OUT THE DOOR.  FUNCS SHOULD BE
 * ABLE TO HAVE SELECTIVITIES THEMSELVES.     -- JMH 7/9/92
 */
s1 = (Selectivity) 0.3333333;
```

Reduction Factors & Histograms

- **For better estimation, use a histogram**

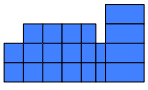
equiwidth

No. of Values	2	3	3	1	8	2	1
Value	0-.99	1-1.99	2-2.99	3-3.99	4-4.99	5-5.99	6-6.99



equidepth

No. of Values	2	3	3	3	3	2	4
Value	0-.99	1-1.99	2-2.99	3-4.05	4.06-4.67	4.68-4.99	5-6.99



Note: 10-bucket equidepth histogram divides the data into *centiles*
- "order statistics" like median, quantiles, etc.

Query Optimization

1. Plan Space
2. Cost Estimation
3. **Search Algorithm**

Enumeration of Alternative Plans

- **There are two main cases:**
  - Single-relation plans (base case)
  - Multiple-relation plans (induction)
- **Single-table queries include selects, projects, and grouping/aggregate ops:**
  - Consider each available access path (file scan / index)
    - Choose the one with the least estimated cost
  - Selection/Projection done on the fly
  - Result pipelined into grouping/aggregation

Cost Estimates for Single-Relation Plans

- **Index I on primary key matches selection:**
  - *Cost is Height(I)+1 for a B+ tree.*
- **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*

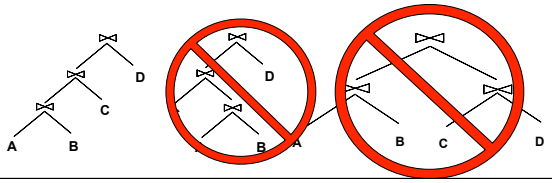
Example

```
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
  - Clustered index: (1/NKeys(I)) * (NPages(I)+NPages(R)) = (1/10) * (50+500) = 55 pages are retrieved. (This is the ***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 unclustered index, 50+40000.
- **Doing a file scan:**
  - We retrieve all file pages (500).

Queries Over Multiple Relations

- **A System R heuristic: *only left-deep join trees* considered.**
  - Restricts the search space
  - Left-deep trees allow us to generate all *fully pipelined* plans.
    - Intermediate results not written to temporary files.
    - Not all left-deep trees are fully pipelined (e.g., SM join).



Enumeration of Left-Deep Plans

- **Left-deep plans differ in**
  - the order of relations
  - the access method for each relation
  - the join method for each join.
- **Enumerated using N passes (if N relations joined):**
  - Pass 1: Find best 1-relation plan for each relation
  - Pass i: Find best way to join result of an (i -1)-relation plan (as outer) to the i'th relation. (i between 2 and N.)
- **For each subset of relations, retain only:**
  - Cheapest plan overall, plus
  - Cheapest plan for each *interesting order* of the tuples.

The Dynamic Programming Table

Subset of tables in FROM clause	Interesting-order columns	Best plan	Cost
{R, S}	<none>	hashjoin(R,S)	1000
{R, S}	<R.a, S.b>	sortmerge(R,S)	1500

A Note on "Interesting Orders"

- **An intermediate result has an "interesting order" if it is sorted by any of:**
  - ORDER BY attributes
  - GROUP BY attributes
  - Join attributes of *yet-to-be-added* (downstream) joins

Enumeration of Plans (Contd.)

- **Match an *i* -1 way plan with another table *only if***
  a) there is a join condition between them, *or*
  b) all predicates in WHERE have been "used up".
     - i.e., avoid Cartesian products if possible.
- **ORDER BY, GROUP BY, aggregates etc. handled as a final step**
  - via `interestingly ordered' plan if chosen (free!)
  - or via an additional sort/hash operator
- **Despite pruning, this is exponential in #tables.**
- **Recall: in practice, COST considered is**
  #IOs + factor * #tuples

Example

Sailors:
Hash, B+ on *sid*
Reserves:
Clustered B+ tree on *bid*
B+ on *sid*
Boats
B+ on *color*

```
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
```

- **Pass1: Best plan(s) for each relation**
  - Sailors, Reserves: File Scan
  - Also B+ tree on Reserves.bid as interesting order
  - Also B+ tree on Sailors.sid as interesting order
  - Boats: B+ tree on color

Subset of tables in FROM clause	Interesting-order columns	Best plan	Cost
{Sailors}	--	filescan	
{Reserves}	--	Filescan	
{Boats}	--	B-tree on color	
{Reserves}	(bid)	B-tree on bid	
{Sailors}	(sid)	B-tree on sid	

Pass 2

- **For each plan 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)
  - Reserves Btree on bid (outer) with Boats (inner)
  - Reserves Btree on bid (outer) with Sailors (inner)
  - File Scan Sailors (outer) with Boats (inner)
  - File Scan Sailors (outer) with Reserves (inner)
  - Boats Btree on color with Sailors (inner)
  - Boats Btree on color with Reserves (inner)
- **Retain cheapest plan for each (pair of relations, order)**

Subset of tables in FROM clause	Interesting-order columns	Best plan	Cost
{Sailors}	--	filescan	
{Reserves}	--	Filescan	
{Boats}	--	B-tree on color	
{Reserves}	(bid)	B-tree on bid	
{Sailors}	(sid)	B-tree on sid	
{Boats, Reserves}	(B.bid) (R.bid)	SortMerge(B-tree on Boats.color, filescan Reserves)	
Etc...			

Pass 3 and beyond

- **Using Pass 2 plans as outer relations, generate plans for the next join**
  - E.g. Boats B+-tree on color with Reserves (bid) (sortmerge) inner Sailors (B-tree sid) sort-merge
- **Then, add cost for groupby/aggregate:**
  - This is the cost to sort the result by sid, *unless it has already been sorted by a previous operator.*
- **Then, choose the cheapest plan**

Physical DB Design

- Query optimizer does what it can to use indices, clustering etc.
- DataBase Administrator (DBA) is expected to set up physical design well
- Good DBAs understand query optimizers very well

One Key Decision: Indexes

- **Which tables**
- **Which field(s) should be the search key?**
- **Multiple indexes?**
- **Clustering?**

Index Selection

- **A greedy approach:**
  - Consider most important queries in turn.
  - Consider best plan using the current indexes
  - See if better plan is possible with an additional index.
  - If so, create it.
- **But consider impact on updates!**
  - Indexes can make queries go faster, updates slower.
  - Require disk space, too.

Issues to Consider in Index Selection

- **Attributes mentioned in a WHERE clause are candidates for index search keys.**
  - Range conditions are sensitive to clustering
  - Exact match conditions don't require clustering
    - Or do they???? :-)
- **Choose indexes that benefit many queries**
- **NOTE: only one index can be clustered per relation!**
  - So choose it wisely!

Example 1

```
SELECT  E.ename, D.mgr
FROM  Emp E, Dept D
WHERE  E.dno=D.dno AND D.dname= 'Toy'
```

- **B+ tree index on *D.dname* supports 'Toy' selection.**
  - Given this, index on D.dno is not needed.
- **B+ tree index on *E.dno* allows us to get matching (inner) Emp tuples for each selected (outer) Dept tuple.**
- **What if WHERE included: `` ... AND E.age=25'' ?**
  - Could retrieve Emp tuples using index on *E.age*, then join with Dept tuples satisfying *dname* selection.
    - Comparable performance to strategy that used *E.dno* index.
  - So, if *E.age* index is already created, this query provides much less motivation for adding an *E.dno* index.

Example 2

```
SELECT  E.ename, D.mgr
FROM  Emp E, Dept D
WHERE  E.sal BETWEEN 10000 AND 20000
 AND E.hobby= 'Stamps'  AND E.dno=D.dno
```

- **All selections are on Emp so it should be the outer relation in any Index NL join.**
  - Suggests that we build a B+ tree index on *D.dno.*
- **What index should we build on Emp?**
  - B+ tree on *E.sal* could be used, OR an index on *E.hobby* could be used.
  - Only one of these is needed, and which is better depends upon the selectivity of the conditions.
    - As a rule of thumb, equality selections more selective than range selections.
- ***Helps to understand optimizers to get this right!***

Examples of Clustering

```
SELECT  E.dno
FROM  Emp E
WHERE  E.age>40
```

- **B+ tree index on E.age can be used to get qualifying tuples.**
  - How selective is the condition?
  - Is the index clustered?
- **Consider the GROUP BY query.**
  - If many tuples have *E.age* > 10, using *E.age* index and sorting the retrieved tuples may be costly.
  - Clustered *E.dno* index may be better!
- **Equality queries and duplicates:**
  - Clustering on *E.hobby* helps!

```
SELECT  E.dno,  COUNT (*)
FROM  Emp E
WHERE  E.age>10
GROUP BY E.dno
```

```
SELECT  E.dno
FROM  Emp E
WHERE  E.hobby=Stamps
```

Index-Only Plans

- **Answer query without going to heap file!**

<*E.dno*>	SELECT D.mgr FROM Dept D, Emp E WHERE D.dno=E.dno
<*E.dno,E.eid*>	SELECT D.mgr, E.eid FROM Dept D, Emp E WHERE D.dno=E.dno
<*E.dno*>	SELECT E.dno, COUNT(*) FROM Emp E GROUP BY E.dno
<*E.dno,E.sal*> *B-tree trick!*	SELECT E.dno, MIN(E.sal) FROM Emp E GROUP BY E.dno
<*E. age,E.sal*> or <*E.sal, E.age*>	SELECT AVG(E.sal) FROM Emp E WHERE E.age=25 AND E.sal BETWEEN 3000 AND 5000

Horizontal Decompositions

- **Typical decomposition: Relation is replaced by collection of relations that are *projections*. Most important case.**
  - We will talk about this at length as part of Conceptual DB Design
- **Sometimes, might want to replace relation by a collection of relations that are *selections.***
  - Each new relation has same schema as original, but subset of rows.
  - Collectively, new relations contain all rows of the original.
  - Typically, the new relations are disjoint.

Horizontal Decompositions (Contd.)

- Contracts (Cid, Sid, Jid, Did, Pid, Qty, Val)
- **Suppose that contracts with value > 10000 are subject to different rules.**
  - So queries on Contracts will often say *WHERE val>10000.*
- **One approach: clustered B+ tree index on the *val* field.**
- **Second approach: replace contracts by two new relations, LargeContracts and SmallContracts, with the same attributes (CSJDPQV).**
  - Performs like index on such queries, but no index overhead.
  - Can build clustered indexes on other attributes, in addition!

Masking Conceptual Schema Changes

```
CREATE VIEW Contracts(cid, sid, jid, did, pid, qty, val)
        AS SELECT *
        FROM LargeContracts
        UNION
        SELECT *
        FROM SmallContracts
```

- **Horizonal Decomposition from above**
- **Masked by a view.**
  - NOTE: queries with condition *val>10000* must be asked wrt LargeContracts for efficiency: so some users may have to be aware of change.
    - I.e. the users who were having performance problems
    - Arguably that's OK -- they wanted a solution!

Index Tuning "Wizards"

- **Both IBM's DB2 and MS SQL Server have automated index advisors**
  - Some info in Section 20.6 of the book
- **Basic idea:**
  - They take a workload of queries
    - Possibly based on logging what's been going on
  - They use the optimizer cost metrics to estimate the cost of the workload over different choices of sets of indexes
  - Enormous # of different choices of sets of indexes:
    - Heuristics to help this go faster

Tuning Queries and Views

- **If a query runs slower than expected, check if an index needs to be re-clustered, or if statistics are too old.**
- **Sometimes, the DBMS may not be executing the plan you had in mind. Common areas of weakness:**
  - Selections involving null values (bad selectivity estimates)
  - Selections involving arithmetic or string expressions (ditto)
  - Selections involving OR conditions (ditto)
  - Complex subqueries (more on this later)
  - Failed size estimation (a common problem in large queries)
  - Lack of evaluation features like index-only strategies or certain join methods.
- **Check the plan that is being used! Then adjust the choice of indexes or rewrite the query/view.**
  - E.g. check via POSTGRES "Explain" command
  - Some systems rewrite for you under the covers (e.g. DB2)
    - Can be confusing and/or helpful!

Forcing the optimizer

- **Many DBMSs allow you to override or "hint" the optimizer if you believe it's messing up**
  - PostgreSQL allows you to "disable" individual physical operators per query (indexscans, specific join algorithms, etc.)
  - MS SQL Server allows hints on specific parts of your query (each Join, Union, etc.). Also allows you to specify a plan in a special XML syntax.
  - Etc.

More Guidelines for Query Tuning

- **Minimize the use of DISTINCT: don't need it if duplicates are acceptable, or if answer contains a key.**
- **Minimize the use of GROUP BY and HAVING:**

```
SELECT MIN (E.age)
FROM Employee E
GROUP BY E.dno
HAVING E.dno=102
```

```
SELECT MIN (E.age)
FROM Employee E
WHERE E.dno=102
```

- **Consider DBMS use of index when writing math:**
  - E.age = 2*D.age might only match index on E.age!
- **A good optimizer should do all these things for you**

Guidelines for Query Tuning (Contd.)

- **Avoid using intermediate relations:**

```
SELECT * INTO Temp
FROM Emp E, Dept D
WHERE E.dno=D.dno
      AND D.mgrname= 'Joe
```

and

```
SELECT T.dno, AVG(T.sal)
FROM Temp T
GROUP BY T.dno
```

vs.

```
SELECT E.dno, AVG(E.sal)
FROM Emp E, Dept D
WHERE E.dno=D.dno
      AND D.mgrname= 'Joe'
GROUP BY E.dno
```

- Does not materialize the intermediate reln Temp.
- If there is a dense B+ tree index on Emp <*dno, sal*>, an index-only plan can be used to avoid retrieving Emp tuples in the left query!

Points to Remember

- **Want to understand DB design (tables, indexes)?**
  - *Must* understand query optimization
- **Three parts to optimizing a query:**
  - Plan space
    - E.g., left-deep plans only
    - avoid Cartesian products.
    - Prune plans with *interesting orders* separate from unordered plans
  - Cost Estimation
    - Output cardinality and cost for each plan node.
    - *Key issues*: Statistics, indexes, operator implementations.
  - Search Strategy
    - we learned "bottom-up" dynamic programming

Points to Remember

- **Single-relation queries:**
  - All access paths considered, cheapest is chosen.
  - *Issues*:
    - Selections that *match* index
    - whether index key has all needed fields
    - whether index provides tuples in an interesting order.

More Points to Remember

- **Multiple-relation queries:**
  - All single-relation plans are first enumerated.
    - Selections/projections considered as early as possible.
  - Use best 1-way plans to form 2-way plans. Prune losers.
  - Use best (i-1)-way plans and best 1-way plans to form i-way plans
  - At each level, for each subset of relations, retain:
    - best plan for each interesting order (including no order)

Summary

- Optimization is the reason for the lasting power of the relational system
- But it is primitive in some SQL databases, and in the Big Data stack
- New areas: many!
  - Smarter statistics (fancy histograms, "sketches")
  - Auto-tuning statistics
  - Adaptive runtime re-optimization (e.g. *Eddies*)
  - Multi-query optimization
  - Parallel scheduling issues