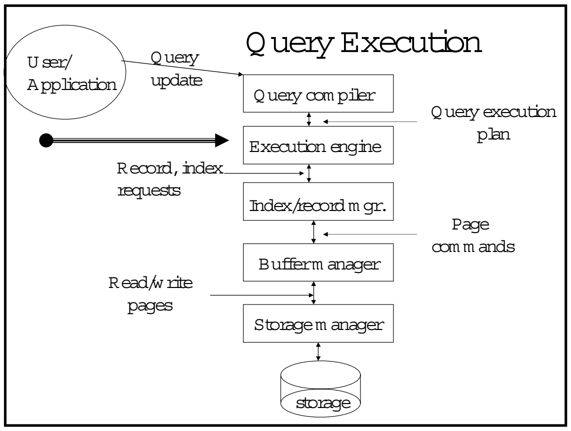
DBMS Internals
Execution and Optimization
May 10th, 2004

Agenda

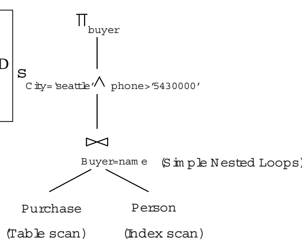
- Questions on phase 2 of the project
- Today: DBMS internals part 2 –
  - Query execution
  - Query optimization
- Next week:
  - Thursday, not Monday.
  - Mostly Phil Bernstein on meta-data management.

Query Execution



Query Execution Plans

```
SELECT S.sname
FROM Purchase P, Person Q
WHERE P.buyer=Q.name AND
    Q.city='seattle' AND
    Q.phone > '5430000'
```



Query Plan:
- logical tree
- implementation choice at every node
- scheduling of operations.

Some operators are from relational algebra, and others (e.g., scan, group) are not.

The Leaves of the Plan: Scans

- Table scan: iterate through the records of the relation.
- Index scan: go to the index, from there get the records in the file (when would this be better?)
- Sorted scan: produce the relation in order. Implementation depends on relation size.

How do we combine Operations?

- The iterator model. Each operation is implemented by 3 functions:
  - Open: sets up the data structures and performs initializations
  - GetNext: returns the the next tuple of the result.
  - Close: ends the operations. Cleans up the data structures.
- Enables pipelining!
- Contrast with data-driven materialize model.
- Sometimes it's the same (e.g., sorted scan).

Implementing Relational Operations

- We will consider how to implement:
  - Selection (σ) Selects a subset of rows from relation.
  - Projection (π) Deletes unwanted columns from relation.
  - Join ($\bowtie$) Allows us to combine two relations.
  - Set-difference Tuples in reln. 1, but not in reln. 2.
  - Union Tuples in reln. 1 and in reln. 2.
  - Aggregation (SUM, MIN, etc.) and GROUP BY

Schema for Examples

Purchase (buyer:string, *seller*: string, *product*: integer),

Person (name:string, *city*:string, *phone*: integer)

- Purchase:
  - Each tuple is 40 bytes long, 100 tuples per page, 1000 pages (i.e., 100,000 tuples, 4MB for the entire relation).
- Person:
  - Each tuple is 50 bytes long, 80 tuples per page, 500 pages (i.e, 40,000 tuples, 2MB for the entire relation).

Simple Selections

```
SELECT *
FROM Person R
WHERE R.phone < '543%'
```

- Of the form $\sigma_{R.attr\ op\ value}(R)$
- With no index, unsorted: Must essentially scan the whole relation; cost is M (#pages in R).
- With an index on selection attribute: Use index to find qualifying data entries, then retrieve corresponding data records. (Hash index useful only for equality selections.)
- Result size estimation:

  (Size of R) * reduction factor.

  More on this later.

Using an Index for Selections

- Cost depends on #qualifying tuples, and clustering.
  - Cost of finding qualifying data entries (typically small) plus cost of retrieving records.
  - In example, assuming uniform distribution of phones, about 54% of tuples qualify (500 pages, 50000 tuples). With a clustered index, cost is little more than 500 I/Os; if unclustered, up to 50000 I/Os!
- Important refinement for unclustered indexes:
  1. Find sort the rid's of the qualifying data entries.
  2. Fetch rids in order. This ensures that each data page is looked at just once (though # of such pages likely to be higher than with clustering).

Two Approaches to General Selections

- First approach: Find the most selective access path, retrieve tuples using it, and apply any remaining terms that don't match the index:
  - Most selective access path: An index or file scan that we estimate will require the fewest page I/Os.
  - Consider city="seattle AND phone<"543%" :
    - A hash index on city can be used; then, phone<"543%" must be checked for each retrieved tuple.
    - Similarly, a b-tree index on phone could be used; city="seattle" must then be checked.

Intersection of Rids

- Second approach
  - Get sets of rids of data records using each matching index.
  - Then intersect these sets of rids.
  - Retrieve the records and apply any remaining terms.

Implementing Projection

```
SELECT DISTINCT
       R.name,
       R.phone
FROM Person R
```

- Two parts:
  (1) remove unwanted attributes,
  (2) remove duplicates from the result.
- Refinements to duplicate removal:
  - If an index on a relation contains all wanted attributes, then we can do an index-only scan.
  - If the index contains a subset of the wanted attributes, you can remove duplicates locally.

Equality Joins With One Join Column

```
SELECT *
FROM Person R, Purchase S
WHERE R.name=S.buyer
```

JOIN

- $R \bowtie S$ is a common operation. The cross product is too large. Hence, performing $R \times S$ and then a selection is too inefficient.
- Assume: M pages in R, p_R tuples per page, N pages in S, p_S tuples per page.
  - In our examples, R is Person and S is Purchase.
- Cost metric: # of I/Os. We will ignore output costs.

Discussion

- How would you implement join?

Simple Nested Loops Join

For each tuple r in R do
  for each tuple s in S do
    if $r_i == s_j$ then add <r, s> to result

- For each tuple in the outer relation R, we scan the entire inner relation S.
  - Cost: M + (p_R * M) * N = 1000 + 100*1000*500 I/Os: 140 hours!
- Page-oriented Nested Loops join: For each page of R, get each page of S, and write out matching pairs of tuples <r, s>, where r is in R-page and S is in S-page.
  - Cost: M + M*N = 1000 + 1000*500 (1.4 hours)

Index Nested Loops Join

foreach tuple r in R do
  foreach tuple s in S where $r_i == s_j$ do
    add <r, s> to result

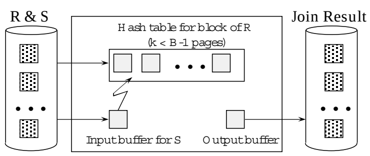
- If there is an index on the join column of one relation (say S), can make it the inner.
  - Cost: M + ((M*p_R) * cost of finding matching S tuples)
- For each R tuple, cost of probing S index is about 1.2 for hash index, 2-4 for B+ tree. Cost of then finding S tuples depends on clustering.
  - Clustered index: 1 I/O (typical), unclustered: up to 1 I/O per matching S tuple.

Examples of Index Nested Loops

- Hash-index on name of Person (as inner):
  - Scan Purchase: 1000 page I/Os, 100*1000 tuples.
  - For each Person tuple: 1.2 I/Os to get data entry in index, plus 1 I/O to get (the exactly one) matching Person tuple. Total: 220,000 I/Os. (36 minutes)
- Hash-index on buyer of Purchase (as inner):
  - Scan Person: 500 page I/Os, 80*500 tuples.
  - For each Person tuple: 1.2 I/Os to find index page with data entries, plus cost of retrieving matching Purchase tuples. Assuming uniform distribution, 2.5 purchases per buyer (100,000 /40,000). Cost of retrieving them is 1 or 2.5 I/Os depending on clustering.

Block Nested Loops Join

- Use one page as an input buffer for scanning the inner S, one page as the output buffer, and use all remaining pages to hold ``block'' of outer R.
  - For each matching tuple r in R-block, s in S-page, add <r, s> to result. Then read next R-block, scan S, etc.



Sort-Merge Join ($R \bowtie_{i=j} S$)

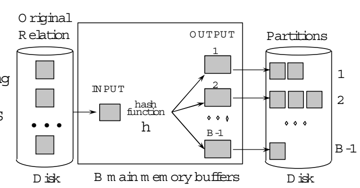
- Sort R and S on the join column, then scan them to do a ``merge'' on the join column.
  - Advance scan of R until current R-tuple >= current S tuple, then advance scan of S until current S-tuple >= current R tuple; do this until current R tuple = current S tuple.
  - At this point, all R tuples with same value and all S tuples with same value match; output <r, s> for all pairs of such tuples.
  - Then resume scanning R and S.

Cost of Sort-Merge Join

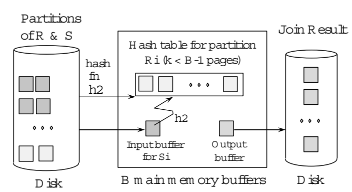
- R is scanned once; each S group is scanned once per matching R tuple.
- Cost: M log M + N log N + (M+N)
- But really, we can do it in 3(M+N) with some trickery.
  - The cost of scanning, M+N, could be M*N (unlikely!)
- With 35, 100 or 300 buffer pages, both Person and Purchase can be sorted in 2 passes; total: 7500. (75 seconds).

Hash-Join

- Partition both relations using hash fn h: R tuples in partition i will only match S tuples in partition i.



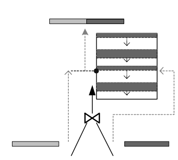
- Read in a partition of R, hash it using **h2 (<> h!)**. Scan matching partition of S, search for matches.



Cost of Hash-Join

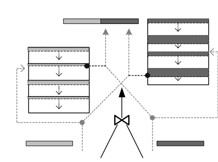
- In partitioning phase, read+write both relations; 2(M+N). In matching phase, read both relations; M+N I/Os.
- In our running example, this is a total of 4500 I/Os. (45 seconds!)
- Sort-Merge Join vs. Hash Join:
  - Given a minimum amount of memory both have a cost of 3(M+N) I/Os. Hash Join superior on this count if relation sizes differ greatly. Also, Hash Join shown to be highly parallelizable.
  - Sort-Merge less sensitive to data skew; result is sorted.

Double Pipelined Join (Tukwila)



Hash Join

- Partially pipelined: no output until inner read
- Asymmetric (inner vs. outer) – optimization requires source behavior knowledge



Double Pipelined Hash Join

- Outputs data immediately
- Symmetric – requires less source knowledge to optimize

Query Optimization

Discussion

- How would you build a query optimizer?

Query Optimization Process (simplified a bit)

- Parse the SQL query into a logical tree:
  - identify distinct blocks (corresponding to nested sub-queries or views).
- Query rewrite phase:
  - apply algebraic transformations to yield a cheaper plan.
  - Merge blocks and move predicates between blocks.
- Optimize each block: join ordering.
- Complete the optimization: select scheduling (pipelining strategy).

Building Blocks

- Algebraic transformations (many and wacky).
- Statistical model: estimating costs and sizes.
- Finding the best join trees:
  - Bottom-up (dynamic programming): System-R
- Newer architectures:
  - Starburst: rewrite and then tree find
  - Volcano: all at once, top-down.

Key Lessons in Optimization

- There are many approaches and many details to consider in query optimization
  - Classic search/optimization problem!
  - Not completely solved yet!
- Main points to take away are:
  - Algebraic rules and their use in transformations of queries.
  - Deciding on join ordering: System-R style (Selinger style) optimization.
  - Estimating cost of plans and sizes of intermediate results.

Operations (revisited)

- Scan ([index], table, predicate):
  - Either index scan or table scan.
  - Try to push down sargable predicates.
- Selection (filter)
- Projection (always need to go to the data?)
- Joins: nested loop (indexed), sort-merge, hash, outer join.
- Grouping and aggregation (usually the last).

Algebraic Laws

- Commutative and Associative Laws
  - $R \cup S = S \cup R$, $R \cup (S \cup T) = (R \cup S) \cup T$
  - $R \cap S = S \cap R$, $R \cap (S \cap T) = (R \cap S) \cap T$
  - $R >< S = S >< R$, $R >< (S >< T) = (R >< S) >< T$
- Distributive Laws
  - $R >< (S \cup T) = (R >< S) \cup (R >< T)$

Algebraic Laws

- Laws involving selection:
  - $s_{C\ AND\ C'}(R) = s_C(s_{C'}(R)) = s_C(R) \cap s_{C'}(R)$
  - $s_{C\ OR\ C'}(R) = s_C(R) \cup s_{C'}(R)$
  - $s_C(R >< S) = s_C(R) >< S$
    - When C involves only attributes of R
  - $s_C(R - S) = s_C(R) - S$
  - $s_C(R \cup S) = s_C(R) \cup s_C(S)$
  - $s_C(R \cap S) = s_C(R) \cap S$

Algebraic Laws

- Example: R(A,B,C,D), S(E,F,G)
  - $s_{F=3}(R \underset{D=E}{><} S) =$?
  - $s_{A=5\ AND\ G=9}(R \underset{D=E}{><} S) =$?

Algebraic Laws

- Laws involving projections
  - $P_M(R >< S) = P_N(P_P(R) >< P_Q(S))$
    - Where N,P,Q are appropriate subsets of attributes of M
  - $P_M(P_N(R)) = P_{M,N}(R)$
- Example R(A,B,C,D), S(E,F,G)
  - $P_{A,B,G}(R \underset{D=E}{><} S) = P_?(P_?(R) \underset{D=E}{><} P_?(S))$

Query Rewrites: Sub-queries

```
SELECT Emp.Name
FROM Emp
WHERE Emp.Age < 30
      AND Emp.Dept# IN
        (SELECT Dept.Dept#
         FROM Dept
         WHERE Dept.Loc = "Seattle"
          AND    Emp.Emp#=Dept.Mgr)
```

The Un-Nested Query

```
SELECT Emp.Name
FROM Emp, Dept
WHERE   Emp.Age < 30
     AND  Emp.Dept#=Dept.Dept#
     AND  Dept.Loc = "Seattle"
     AND  Emp.Emp#=Dept.Mgr
```

Converting Nested Queries

```
Select distinct x.name, x.maker
From product x
Where x.color= "blue"
 AND x.price >= ALL (Select y.price
                     From product y
                     Where x.maker = y.maker
                       AND y.color="blue")
```

Converting Nested Queries

Let's compute the complement first:

```
Select distinct x.name, x.maker
From product x
Where x.color= "blue"
 AND x.price < SOME (Select y.price
                     From product y
                     Where x.maker = y.maker
                       AND y.color="blue")
```

Converting Nested Queries

This one becomes a SFW query:

```
Select distinct x.name, x.maker
From product x, product y
Where x.color= "blue" AND x.maker = y.maker
 AND y.color="blue" AND x.price < y.price
```

This returns exactly the products we DON'T want, so…

Converting Nested Queries

```
(Select x.name, x.maker
From product x
Where x.color = "blue")

EXCEPT

(Select x.name, x.maker
From product x, product y
Where x.color= "blue" AND x.maker = y.maker
 AND y.color="blue" AND x.price < y.price)
```

Semi-Joins, Magic Sets

- You can't always un-nest sub-queries (it's tricky).
- But you can often use a semi-join to reduce the computation cost of the inner query.
- A magic set is a superset of the possible bindings in the result of the sub-query.
- Also called "sideways information passing".
- Great idea; reinvented every few years on a regular basis.

Rewrites: Magic Sets

```
Create View DepAvgSal AS
  (Select E.did, Avg(E.sal) as avgsal
   From Emp E
   Group By E.did)

Select E.eid, E.sal
From Emp E, Dept D, DepAvgSal V
Where E.did=D.did AND D.did=V.did
  And E.age < 30 and D.budget > 100k
  And E.sal > V.avgsal
```

Rewrites: SIPs

Select E.eid, E.sal
From Emp E, Dept D, DepAvgSal V
Where E.did=D.did AND D.did=V.did
  And E.age < 30 and D.budget > 100k
  And E.sal > V.avgsal

- DepAvgsal needs to be evaluated only for departments where V.did IN

  Select E.did
  From Emp E, Dept D
  Where E.did=D.did
    And E.age < 30 and D.budget > 100K

Supporting Views

1. Create View PartialResult as
   (Select E.eid, E.sal, E.did
   From Emp E, Dept D
   Where E.did=D.did
     And E.age < 30 and D.budget > 100K)
2. Create View Filter AS
   Select DISTINCT P.did FROM PartialResult P.
2. Create View LimitedAvgSal as
   (Select F.did Avg(E.Sal) as avgSal
   From Emp E, Filter F
   Where E.did=F.did
   Group By F.did)

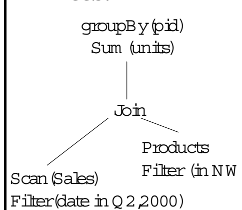
And Finally…

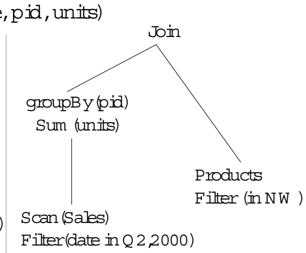
Transformed query:

Select P.eid, P.sal
From PartialResult P, LimitedAvgSal V
Where P.did=V.did
  And P.sal > V.avgsal

Rewrites: Group By and Join

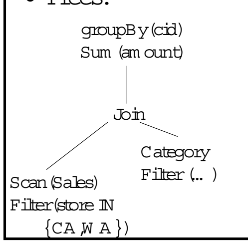
- Schema:
  - Product (pid, unitprice,…)
  - Sales(tid, date, store, pid, units)
- Trees:

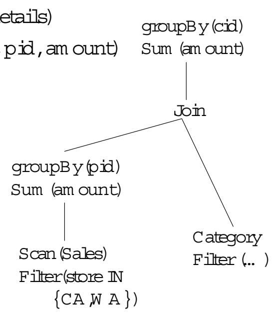




Rewrites: Operation Introduction

- Schema: (pid determines cid)
  - Category (pid, cid, details)
  - Sales(tid, date, store, pid, amount)
- Trees:



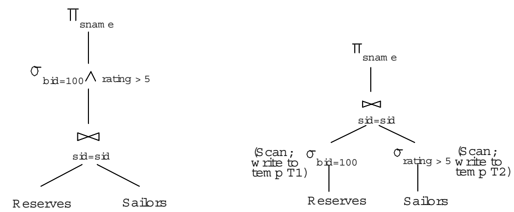


Schema for Some Examples

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

- Reserves:
  - Each tuple is 40 bytes long, 100 tuples per page, 1000 pages (4000 tuples)
- Sailors:
  - Each tuple is 50 bytes long, 80 tuples per page, 500 pages (4000 tuples).

Query Rewriting: Predicate Pushdown



The earlier we process selections, less tuples we need to manipulate higher up in the tree.
Disadvantages?

Query Rewrites: Predicate Pushdown (through grouping)

```
Select bid, Max(age)
From   Reserves R, Sailors S
Where  R.sid=S.sid
GroupBy bid
Having Max(age) > 40
```

```
Select bid, Max(age)
From   Reserves R, Sailors S
Where  R.sid=S.sid and
       S.age > 40
GroupBy bid
```

- For each boat, find the maximal age of sailors who've reserved it.
- Advantage: the size of the join will be smaller.
- Requires transformation rules specific to the grouping/aggregation operators.
- Will it work work if we replace Max by Min?

Query Rewrite: Predicate Movearound

Sailing wiz dates: when did the youngest of each sailor level rent boats?

```
Select sid, date
From   V1, V2
Where  V1.rating = V2.rating and
       V1.age = V2.age
```

```
Create View V1 AS
Select rating, Min(age)
From   Sailors S
Where  S.age < 20
Group By rating
```

```
Create View V2 AS
Select sid, rating, age, date
From   Sailors S, Reserves R
Where  R.sid=S.sid
```

Query Rewrite: Predicate Movearound

Sailing wiz dates: when did the youngest of each sailor level rent boats?

First, move predicates up the tree.

```
Select sid, date
From   V1, V2
Where  V1.rating = V2.rating and
       V1.age = V2.age, age < 20
```

```
Create View V1 AS
Select rating, Min(age)
From   Sailors S
Where  S.age < 20
Group By rating
```

```
Create View V2 AS
Select sid, rating, age, date
From   Sailors S, Reserves R
Where  R.sid=S.sid
```

Query Rewrite: Predicate Movearound

Sailing wiz dates: when did the youngest of each sailor level rent boats?

First, move predicates up the tree.

```
Select sid, date
From   V1, V2
Where  V1.rating = V2.rating and
       V1.age = V2.age, and age < 20
```

Then, move them down.

```
Create View V1 AS
Select rating, Min(age)
From   Sailors S
Where  S.age < 20
Group By rating
```

```
Create View V2 AS
Select sid, rating, age, date
From   Sailors S, Reserves R
Where  R.sid=S.sid, and
       S.age < 20.
```

Query Rewrite Summary

- The optimizer can use any semantically correct rule to transform one query to another.
- Rules try to:
  - move constraints between blocks (because each will be optimized separately)
  - Unnest blocks
- Especially important in decision support applications where queries are very complex.
- In a few minutes of thought, you'll come up with your own rewrite. Some query, somewhere, will benefit from it.
- Theorems?

Cost Estimation

- For each plan considered, must estimate cost:
  - Must estimate cost of each operation in plan tree.
    - Depends on input cardinalities.
  - Must estimate size of result for each operation in tree!
    - Use information about the input relations.
    - For selections and joins, assume independence of predicates.
- We'll discuss the System R cost estimation approach.
  - Very inexact, but works ok in practice.
  - More sophisticated techniques known now.

Statistics and Catalogs

- Need information about the relations and indexes involved. Catalogs typically contain at least:
  - # tuples (NTuples) and # pages (NPages) for each relation.
  - # distinct key values (NKeys) and NPages for each index.
  - Index height, low/high key values (Low/High) for each tree index.
- Catalogs updated periodically.
  - Updating whenever data changes is too expensive; lots of approximation anyway, so slight inconsistency ok.
- More detailed information (e.g., histograms of the values in some field) are sometimes stored.

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.
- Reduction factor (RF) associated with each term reflects the impact of the term in reducing result size. Result cardinality = Max # tuples * product of all RF's.
  - Implicit assumption that terms are independent!
  - Term col=value has RF 1/NKeys(I), given index I on col
  - Term col1=col2 has RF 1/MAX(NKeys(I1), NKeys(I2))
  - Term col>value has RF (High(I)-value)/(High(I)-Low(I))

Histograms

- Key to obtaining good cost and size estimates.
- Come in several flavors:
  - Equi-depth
  - Equi-width
- Which is better?
- Compressed histograms: special treatment of frequent values.

Histograms

- Statistics on data maintained by the RDBMS
- Makes size estimation much more accurate (hence, cost estimations are more accurate)

Histograms

Employee(<u>ssn</u>, name, salary, phone)

- Maintain a histogram on salary:

Salary:	0..20k	20k..40k	40k..60k	60k..80k	80k..100k	> 100k
Tuples	200	800	5000	12000	6500	500

- T(Employee) = 25000, but now we know the distribution

Histograms

Ranks(rankName, salary)

- Estimate the size of Employee $\bowtie_{Salary}$ Ranks

Employee	0..20k	20k..40k	40k..60k	60k..80k	80k..100k	> 100k
	200	800	5000	12000	6500	500

Ranks	0..20k	20k..40k	40k..60k	60k..80k	80k..100k	> 100k
	8	20	40	80	100	2

Histograms

- Assume:
  - V(Employee, Salary) = 200
  - V(Ranks, Salary) = 250
- Then T(Employee $\bowtie_{Salary}$ Ranks) =

  $= S_{i=1,6}\, T_i T_i' / 250$

  $= (200x8 + 800x20 + 5000x40 + 12000x80 + 6500x100 + 500x2)/250$

  $= \ldots$.

Plans for Single-Relation Queries (Prep for Join ordering)

- Task: create a query execution plan for a single Select-project-group-by block.
- Key idea: consider each possible access path to the relevant tuples of the relation. Choose the cheapest one.
- 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).

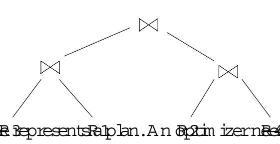
Example

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

- If we have an Index on rating:
  - (1/NKeys(I)) * NTuples(R) = (1/10) * 40000 tuples retrieved.
  - Clustered index: (1/NKeys(I)) * (NPages(I)+NPages(R)) = (1/10) * (50+500) pages are retrieved (= 55).
  - Unclustered index: (1/NKeys(I)) * (NPages(I)+NTuples(R)) = (1/10) * (50+40000) 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.
- Doing a file scan: we retrieve all file pages (500).

Determining Join Ordering

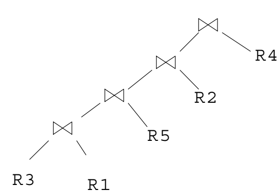
- R1 $\bowtie$ R2 $\bowtie$ …. $\bowtie$ Rn
- Join tree:



- A join tree represents a plan. An optimizer needs to inspect many (all?) join trees

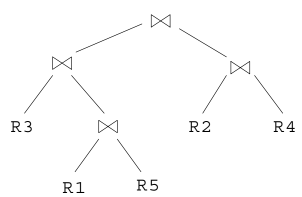
Types of Join Trees

- Left deep:



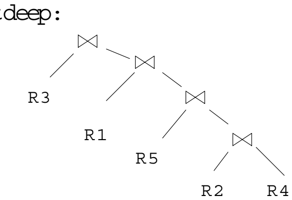
Types of Join Trees

- Bushy:



Types of Join Trees

- Right deep:



Problem

- Given: a query R1 ⋈ R2 ⋈ … ⋈ Rn
- Assume we have a function cost() that gives us the cost of every join tree
- Find the best join tree for the query

Dynamic Programming

- Idea: for each subset of {R1, …, Rn}, compute the best plan for that subset
- In increasing order of set cardinality:
  - Step 1: for {R1}, {R2}, …, {Rn}
  - Step 2: for {R1,R2}, {R1,R3}, …, {Rn-1, Rn}
  - …
  - Step n: for {R1, …, Rn}
- A subset of {R1, …, Rn} is also called a subquery

Dynamic Programming

- For each subquery Q ⊆ {R1, …, Rn} compute the following:
  - Size(Q)
  - A best plan for Q: Plan(Q)
  - The cost of that plan: Cost(Q)

Dynamic Programming

- Step 1: For each {Ri} do:
  - Size({Ri}) = B(Ri)
  - Plan({Ri}) = Ri
  - Cost({Ri}) = (cost of scanning Ri)

Dynamic Programming

- Step i: For each Q ⊆ {R1, …, Rn} of cardinality i do:
  - Compute Size(Q) (later…)
  - For every pair of subqueries Q', Q''
    s.t. Q = Q' U Q''
    compute cost(Plan(Q') ⋈ Plan(Q''))
  - Cost(Q) = the smallest such cost
  - Plan(Q) = the corresponding plan

Dynamic Programming

- Return Plan({R1, …, Rn})

Dynamic Programming

- Summary: computes optimal plans for subqueries:
  - Step 1: {R1}, {R2}, …, {Rn}
  - Step 2: {R1,R2}, {R1,R3}, …, {Rn-1,Rn}
  - …
  - Step n: {R1, …, Rn}
- We used naïve size/cost estimations
- In practice:
  - more realistic size/cost estimations (next)
  - heuristics for Reducing the Search Space
    - Restrict to left linear trees
    - Restrict to trees "without cartesian product"
  - need more than just one plan for each subquery:
    - "interesting orders"