Introduction to Query Optimization

Reading:

- Selinger, et al., "Access Path Selection in a Relational Database Management System"
- Chauduri, "An Overview of Query Optimization in Relational Systems"

SQL is a declarative language:

- how SQL queries are mapped to efficient programs is AI or Software Engineering problem of "Automatic Programming"
- this lecture reviews the core algorithm for mapping an SQL query to an efficient program
- note: non-standard way of describing algorithms... but these are the basic abstractions to understand, implement

How to optimize this query?

```
select  A.q, D.p
from    A, B, C, D
where   A.x = B.x  and  A.y = B.y
        B.x = C.x  and  A.q > 40  and D.p ≤ 7
```

Query Evaluation Procedure

1. Parse query into a *query graph*

2. Map query graph into one or more *logical access plans*

3. Map each logical access plan into a *physical access plan*

4. Generate the space of all equivalent logical access plans and their physical access plans

5. Estimate the cost of each physical access plan

6. Efficiently search the space of physical access plans and return the cheapest

Goal: because heuristics are used (both to estimate costs and to avoid examining all plans), want to avoid clearly inefficient plans and to identify a "good" plan.

- comment on optimality, buffering...

1: Mapping Query Text to a Query Graph

Assume conjunctive, non-nested, non-aggregate queries

Given an SQL statement S, let:

- Q(S,R) - *local predicate* for relation R is the conjunction of clauses that reference R but are not join clauses
- P(S,R) - *local projection* for relation R is list of attributes of R mentioned in the select or projection clauses of S

Example:

```
select  A.q, D.p
from    A, B, C, D
where   A.x = B.x and A.y = B.y and B.x = C.x
        A.q > 40 and D.p ≤ 7
order by D.p
```

Q(S,A) =	P(S,A) =
Q(S,B) =	P(S,B) =
Q(S,C) =	P(S,C) =
Q(S,D) =	P(S,D) =

Query Graphs

A *query graph* is an undirected graph where:

- nodes are relations labeled with relation name or executable expression

  implicitly labeled with their local predicates and projection lists

- edges show relations to be joined, labeled with join conditions

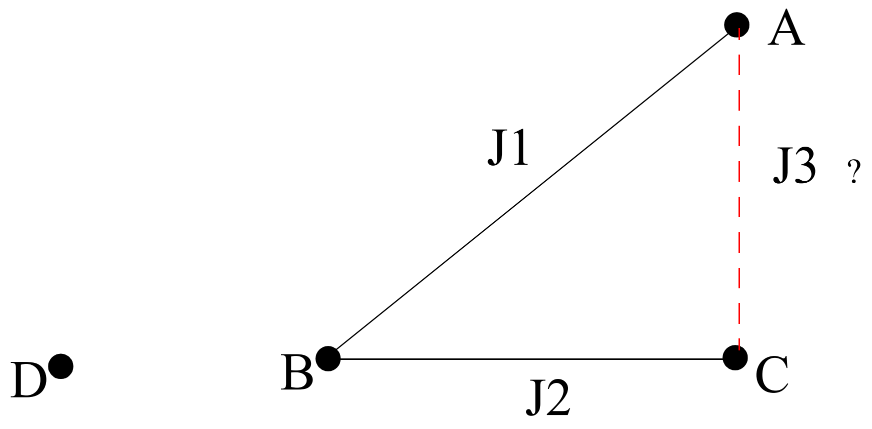
Example:

```
select  A.q, D.p
from    A, B, C, D
where   A.x = B.x and A.y = B.y and B.x = C.x
        A.q > 40 and d.p ≤ 7
order by D.p
```

Query Graph:

Query Graphs (Cont)

Note: join clauses, in addition to those listed in S, can be inferred…



J1 is	A.x = B.x and A.y = B.y
J2 is	B.x = C.x
J3 is	?

May not be done often … why?

2: Map Query Graph to Access Plan

Reduce query graph to an expression called a *logical access plan* using graph rewrite rules

- the plan is *logical* because no algorithms are specified; only *operations* (like join, retrieval...) are used

There can be many possible logical access plans for a query; each plan is generated by applying a unique sequence of graph rewrite rules

To show how this is done, we need to distinguish:

- *stored files* — files on disk; base relations
- *stream files* — result of computations on stored files

“how to think”

Logical Access Plans

A *logical access plan* is a composition of the following set of operations on stored and stream files:

Operation	Semantics
ret(F, O)	retrieve tuples from stored file F in O order. Implicit parameters are P(S,F) and Q(S,F). O must be * or member of P(S,F)
join(F1, F2, J, O)	produce the join of stored or stream files F1 and F2 over join predicate J in O order. O must be * or a member of P(S,F1) or P(S,F2). The projection list of resulting stream is P(S,F1) union P(S,F2)
prod(F1,F2,O)	produce the cross product of stored or stream files F1 and F2 in O order. O must be * or a member of P(S,F1) or P(S,F2). the projection list of resulting stream is P(S,F1) union P(S,F2).
end(F, O)	F is a stream file. end() sorts F in order O designated by S and trims F of all fields not listed in the select clause of S

- all non-nested, non-aggregate, conjunctive SQL queries can be processed by composing these operations…
- *relational algebra is never really used ... why?*

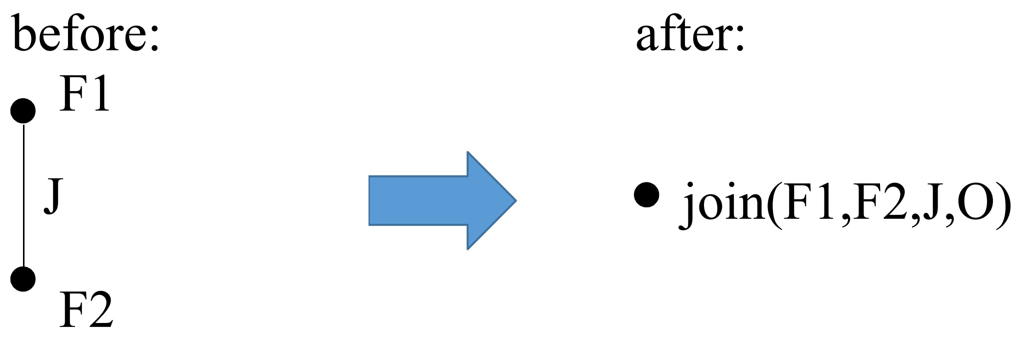
Rewrite Rule #1: Local Processing

before:		after:
• F	$\Rightarrow$	• ret(F,O)

Purpose of Rule #1:

- to convert stored files to stream files
- to eliminate unnecessary fields and unqualified records (i.e., relational projection and selection)

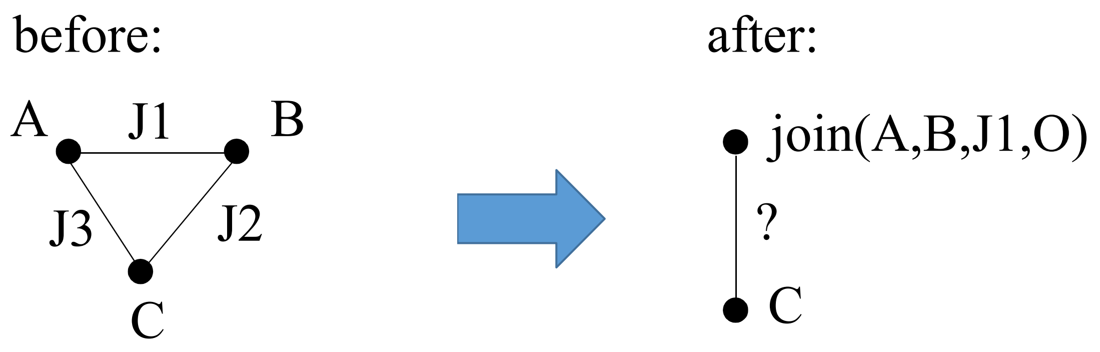
Rewrite Rule #2: Joins (Edge Removal)



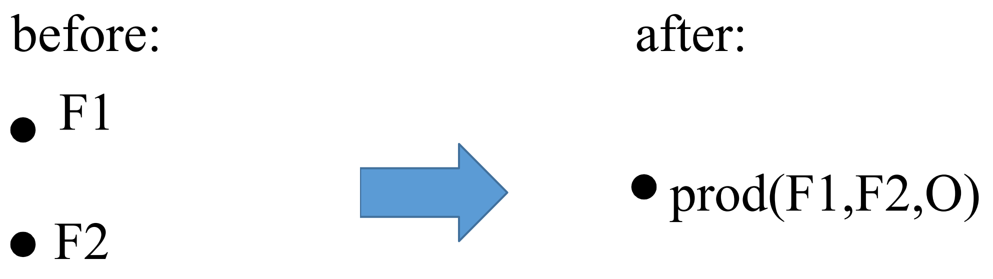
Purpose of Rule #2:

- to join two (stream) files into a single stream
- to simplify the query graph by eliminating a node, edges

Note: side effect where joined nodes connected to a common node:



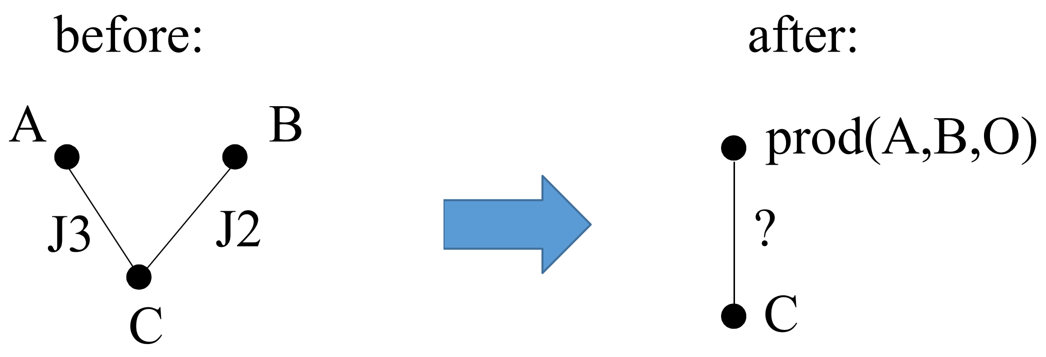
Rewrite Rule #3: Cross Products



Purpose of Rule #3:

- to join two (stream) files that have TRUE for a join predicate
- to simplify the query graph by eliminating a node

Note: side effect again:



- cross product is a not-often-used special case of join

Rewrite Rule #4: End

before: after: (no node left)

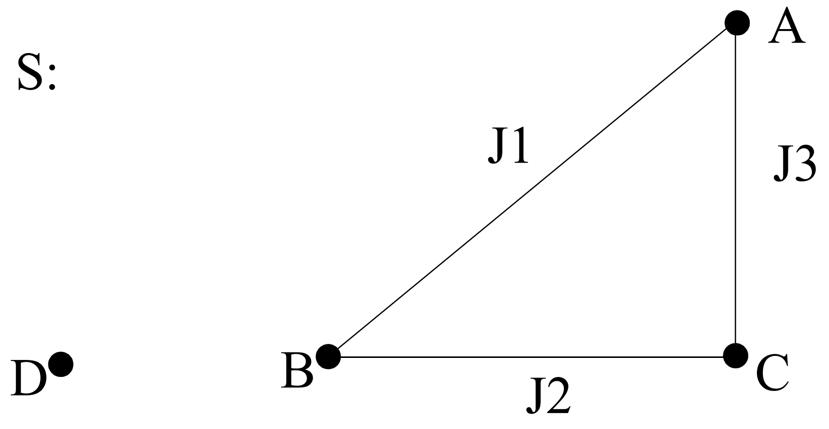
● F $\Rightarrow$ end(F,S)

Purpose of Rule #4:

- to sort stream file F into order designated by SQL statement S and trim tuples of F of all fields not listed in the select clause of S

- last step in converting graph (of a single node) into a logical access plan

Example



Step 1: apply local processing to A, B, C, D

Step 2: join A with C

Step 3: join A-C with B

Step 4: cross product A-B-C with D

Step 5: end

3: Convert Logical Access Plans into Physical Access Plans

Operations are abstract computations that can be implemented by one or more *algorithms*

- ret(…) and join(…) are operations
- merge-join, nested loops are algorithms that implement join operations

A *physical access plan* is derived from a logical access plan by replacing each operation with an implementing algorithm

- logical plans are compositions of *operations*
- physical plans are compositions of *algorithms*

List of algorithms that implement an operation expressed as a *catalog* of rewrite rules

$$\begin{array}{lcll} \text{operation} & \rightarrow & A_1 & \text{; algorithm}_1 \text{ preconditions} \\ & & A_2 & \text{; algorithm}_2 \text{ preconditions} \\ & & \ldots & \\ & & A_n & \text{; algorithm}_n \text{ preconditions} \end{array}$$

- note: all algorithms have *at least* same parameters as operations

Retrieval Examples

$$
\begin{array}{lll}
\text{ret(F,O)} & \rightarrow & \text{segment\_scan\_and\_sort(F,O)} \\
 & & \text{use\_ordering\_index(F,O)} \\
 & & \text{use\_nonordering\_index\_and\_sort(F,O)} \\
 & & \text{use\_multiple\_indices\_and\_sort(F,O)} \\
 & & \text{bitmap indices} \\
 & & \text{others??}
\end{array}
$$

What are these algorithms?

- *segment_scan_and_sort* examines each tuple in a relation, qualified tuples are saved and then sorted
- *use_ordering_index* index on the sort key is traversed (thus retrieving tuples in sorted order). Qualified tuples are immediately output
- *use_nonordering_index_and_sort* uses a single index to retrieve tuples, qualified tuples are saved and then sorted
- *use_multiple_indices_and_sort* uses multiple indices to retrieve tuples; qualified tuples are saved and sorted
- *bitmap indices* see lecture on data warehouses

Join Examples

join(F1,F2,J,O) $\rightarrow$ nested_loop(F1,F2,J,O)

merge_join(F1,F2,J,O)

hash_join(F1,F2,J,O)

others?

- note: algorithms have same parameters as operations

Nested loops $O(n^2)$

```
foreach i in F1 do
   foreach j in F2 where i joins_with j do
      output(i,j);
```

Merge-join O(n log n) — note: cross-product possibilities

```
S1 = sort F1 in join key order;
S2 = sort F2 in join key order;
join_output = merge S1 and S2 where tuples
                 have identical join keys;
```

Nested Loops with Query Modification

Suppose **`B.X`** is indexed in the query:

```
select * from A,B
where B.Y=y and B.X=A.X and A.Q=q
```

A standard way to join relations **`A`** and **`B`** via nested loops (which incidentally isn't that bad in terms of performance)

- modify the local query for **`B`** from **`Y=y`** to parameterized query **`p(x) =`** "**`Y=y and X=x`**"

Optimizer may select to process the **`B`** retrieval using index for **`X`** — but since this query is parameterized, it must be invoked many times, once for each tuple of relation **`A`**:

```
foreach a in A do
   foreach b in B where p(a.X) do
      output(a,b);
```

This is a common form of *query modification*

Hash Join Algorithms

Papers:

- Kitsuregawa, Tanaka, and Moto-Oka, “Application of Hash to Database Machine and Its Architecture”, *New Generation Computing*, 1983.
- L. Shapiro, “Join Processing in Database Systems with Large Memories”, *ACM TODS*, 1986.

Motivation:

- exploit large main memories
- linear algorithms!
- Shapiro shows hash joins better than sort merge

Large class of hash join algorithms

- classical hash join
- Grace hash join (original)
- ...

Classical Hash Join

Block nested loops

```
foreach block of tuples i in F1 do
    foreach block of tuples j in F2 do
        output tuples of i that join with j;
```

- constant times faster than nested loops

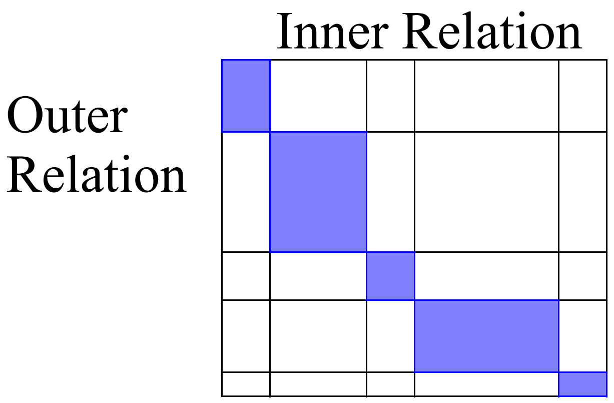
Classical hash join is a variation of block nested loops

- buffer entire outer relation and store in hash table, hashing on join key
- scan inner file and probe table for joining tuples
- performance: linear!
  (each tuple of inner relation (**`F2`**) examined only once)

Question: what if there isn't enough memory?

Grace Hash Join (One of Many Variants)

Basic idea: partition both relations via hashing and join corresponding partitions



assume
k buffers
or buckets

7. hash outer relation O into *k* buckets, where O_j is the partition in bucket # j
8. flush buckets to disk
9. do same with inner file I, where I_j is the partition in bucket #j

   (note: important that same hash function be used for partitioning both relations)

10. for j=1…*k* do join(O_j,I_j)
    (where join could be sort merge or classical hash join)
11. merge results of the *k* joins

Mapping Logical to Physical Plans

A *physical access plan* is derived from a logical plan by replacing each operation of the logical plan with an implementing algorithm

- "for now" notation: drop ordering, join predicate parameters
- example: join(join(ret(A), ret(B)), ret(C))

Questions: given a clique (fully connected graph) of *n* nodes and there are *j* join algorithms and *r* retrieval algorithms…

- how many logical access plans are there? # of join orders
- how many physical access plans does a single logical access plan have?
- what does this tell us about the size of the physical access plan search space?

generally, *n*, *j*, *r* are small... what if they are big?

4: Generate Space of Equivalent Plans

We know that the output of all centralized query optimization algorithms are expressions (logical access plans) that are compositions of 4 basic operations

- actual algorithms apply Rewrite Rules 1-4 in a premeditated way to reduce query graphs to expressions

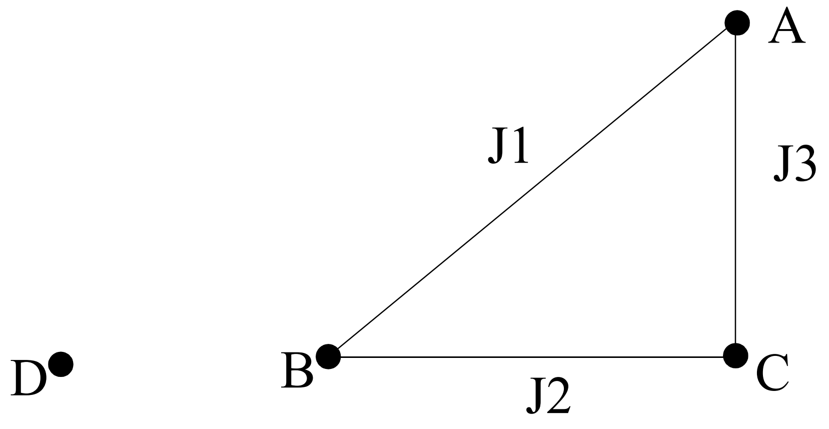
Example: System R algorithm

- enumerates the search space of (almost) all equivalent expressions and chooses the cheapest
- can be expressed nondeterministically…

System R Algorithm: SysR(G,S)

1. **(select sink)** Nondeterministically select node F in G and a retrieval order O. Process F by a local processing rewrite. The resulting node is the sink F_0. Let $i = 0$. *(Choose retrieval algorithm).*
2. **(main loop)** While G contains more than one node, repeat steps 2.1, 2.2.
    - 2.1.**(edge removal)** If there are 1 or more edges connected to F_i, nondeterministically select an edge and a join order O_i. Locally process the connected node and eliminate the edge by a join rewrite. The resulting node is sink F_{i+1}. Increment i. *(Choose join algorithm)*
    - 2.2.**(cross product)** If no edge is connected to F_i and graph G contains more than one node, nondeterministically select a node in G (different from F_i) and an order O_i. Locally process the connected node and combine both nodes with a cross product rewrite. The resulting node is sink F_{i+1}. Increment i.
3. **(final step)** Graph G has been reduced to a single node F_i. Apply the end rewrite rule to F_i. The resulting expression is an access plan for S.

Example



5: Estimate Cost of a Plan

What is the cost of a physical access plan P?

- costs are weighted averages of I/O and CPU used
- simple mathematical equations that estimate "costs" given parametric input (called descriptors)

A *descriptor* (object) is a set of variables that models some database entity

Example: descriptor variables for stored files:

Variable	Definition
n	number of tuples in file
r	number of tuples per block/page
h	height of a file structure
b	number of blocks in file structure (note: $b \geq n/r$)
…	

- values come from statistics DBMS collects

Descriptors for stream files, queries?

Descriptors

Example: descriptor variables for stream files:

Variable	Definition
n	number of tuples in stream
order	attribute in which stream is ordered
cost	cost of generating stream

Example: variables for query descriptors:

Variable	Definition
q	selectivity of query

- note: *selectivity* is the fraction of tuples that satisfy the predicate

  so if *n* is the number of tuples, and *q* is the selectivity of a predicate, *n*q* is the expected number of tuples that satisfy the predicate.

Example: variables for join predicate descriptors:

Variable	Definition
q	selectivity of join query

Characteristic Functions

Let A be a file and ΔA be its descriptor

Let ΔA.v be the value of parameter v for ΔA

operation	ret(A,Q…)	B+ tree
cost function	$ret(ΔA,ΔQ)	ΔA.b ; if scan ΔA.h ; if key search

stream descriptor	Δret(ΔA,ΔQ)	n = ΔA.n ∗ ΔQ.q
		cost = $ret(ΔA,ΔQ)

More examples:

ret(A,Q...)	unordered file	hash file
$ret(ΔA,ΔQ)	ΔA.b	ΔA.b ; if scan 1 ; if key search
Δret(ΔA, ΔQ)		

More Examples

operation	join(A,B,J…)	nested loops
cost function	\$join(ΔA, ΔB,ΔJ)	ΔA.cost + ΔA.n *ΔB.cost

stream descriptor	Δjoin(ΔA,ΔB,ΔJ)	n = ΔA.n * ΔB.n* ΔJ.q
		cost = \$join(ΔA, ΔB,ΔJ)

More examples:

join(A,B, J,…)	classical hash join	merge-join	block-nested loops
\$join(ΔA, ΔB,ΔJ)	ΔA.cost + ΔB.cost (merge is free)	ΔA.b + ΔB.b + \$sort(ΔA) + \$sort(ΔB) + \$merge(free)	
Δjoin(ΔA,ΔQ,ΔJ)			

Relationship of Algorithms to Cost Functions

Suppose we are to choose an implementation for operation ret(F,Q,O). How do we do it?

ret(F,Q,O) $\rightarrow$ segment_scan_and_sort(F,Q,O)

use_ordering_index(F,Q,O)

use_nonordering_index_and_sort(F,Q,O)

Answer: How cost and descriptor functions are defined:

\$ret(ΔF, ΔQ) =

Δret(ΔF, ΔQ): cost = \$ret(ΔF, ΔQ)

n = ΔF.n * ΔQ.q

More Examples

join(F1,F2,J,O) $\rightarrow$ nested_loop(F1, F2, J, O)

merge_join(F1, F2, J, O)

hash_join(F1, F2, J, O)

\$join(ΔF1, ΔF2, ΔJ) =

Δjoin(ΔF1, ΔF2, ΔJ): cost = \$join(ΔF1, ΔF2, ΔJ)

n = ΔF1.n * ΔF2.n * ΔJ.q

Estimating Costs of Complex Expressions

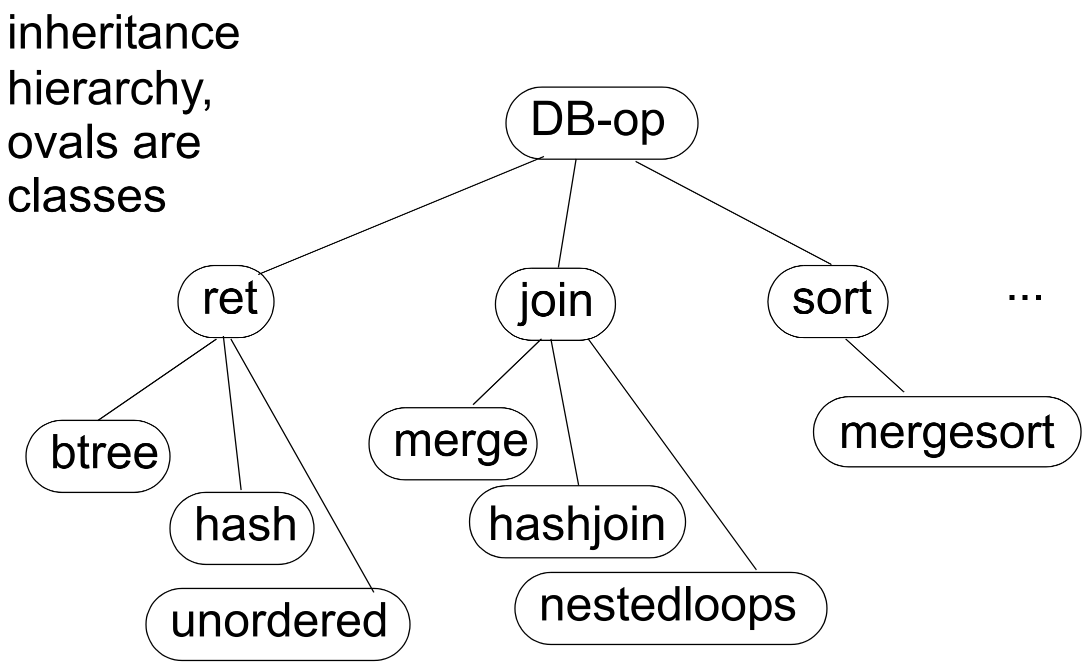
Evaluate cost and descriptor functions recursively

- example:

  P = join(join(ret(A,...) ret(B, ...), ...), ret(C,...))

- what is ΔP?

Natural Object-Oriented Design



- DB-op is an abstract class or interface
- interface for most database operations include the following set of methods:

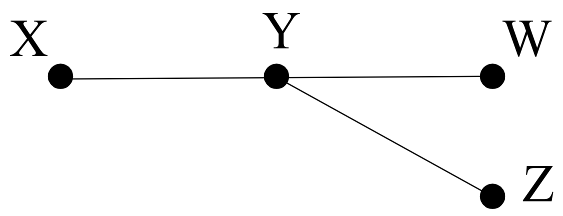
`getFirstRecord( )` — get first record of stream
`getNextRecord( )` — get next record of stream
`atEnd( )` — at end of stream?
`cost( )` — estimated cost of executing operation
`size( )` — number of records returned

6: Search Space of Equivalent Plans Efficiently, Returning the Cheapest Plan

Dynamic programming underlies pruning

- each iteration, identify set of access plans that generate the same result
- eliminate all but the cheapest plan to reduce the search space
- "for now" notation: ignore stream ordering parameters, join predicate parameters …

Consider a SQL select statement that has the following query graph:



- reduction into a logical/physical access plan in four iterations, one per rewrite
- each iteration generates plans and prunes

System R Query Graph Reduction

Iteration #1: identify all possible sinks

- identify all expressions that reference a single relation

<u>sink X</u>   <u>sink Y</u>   <u>sink Z</u>   <u>sink W</u>

- note: X means stream of tuples produced by ret(X, ...)
  note: no pruning (a lie)

Reduction (Continued)

Iteration #2: joining of two relations

- starting with the sinks of last iteration, remove a single edge

<u>sink X</u>   <u>sink Y</u>   <u>sink Z</u>   <u>sink W</u>

- note: X-Y means stream of X,Y tuples produced as a result of join(X,Y, ...)

Now prune: choose cheapest among equivalence set

- note: *xy* denotes cheapest way to produce the stream that joins relations X and Y

Reduction (Cont)

Iteration #3: joining of three relations

- starting with the sinks of last iteration, remove a single edge

<u>sink xy</u>　　　<u>sink yz</u>　　　<u>sink yw</u>

- note: xy-Z means join stream xy with Z (which denotes ret(Z,...))

Now prune: choose cheapest among equivalence set

- note: *xyz* denotes minimum cost to generate the stream that joins X, Y, Z

Reduction (Cont)

Iteration #4: joining of four relations

- starting with the sinks of last iteration, remove a single edge

<u>sink xyz</u> <u>sink xyw</u> <u>sink yzw</u>

Now prune:

Asymptotic Complexity of the System R Algorithm

Assume worst case scenario:

- *n* relations to be joined
- query graph is fully connected (i.e., clique)

System R only generates *left-deep* trees (i.e., inner input of a join operator is the retrieval of a stored file)

- easy to see that the search space has size O(*n!*)
- but dynamic programming reduces complexity

Remember:

- at each iteration, the optimizer constructs optimal plans for each possible join of *j* relations (*j*=1..*n*).
- after each iteration, only the best plan for joining a given subset of relations is maintained
- thus, the algorithm complexity is determined by the total number of operator trees that need to be examined at each stage

Complexity Continued

$$\sum_{j=0}^{n-1} \binom{n}{j} \times (n-j) = n \times 2^{n-1}$$

Complexity has exponential worst-case complexity in the number of relations to be joined. Better than *n!*

What's Next and Other Big Picture Issues

Rewrite engine/approach works well for optimizing join orders; doesn't scale to handle other operations

- other logical operations (e.g., group-by)

```
SELECT E.id, AVE(E.sal) FROM Emp E
WHERE  E.age<30
GROUP BY E.id
```

- nested queries, correlated queries

```
SELECT E.Name FROM Emp E
WHERE  E.Dept IN (
   SELECT D.Name FROM   Dept
   WHERE  D.Location = 'Austin' AND
          E.id = D.mangerId )
```

- *outerjoins* — joins that preserves tuples of relations
  left outerjoin, right outerjoin, symmetric outerjoin

```
SELECT * FROM employees FULL[1] OUTER JOIN
orders ON employees.eno = orders.eno;
```

All are optimized uniformly by rewriting expressions — towards rule-based query optimization

1. FULL — meaning symmetric — can be replaced with LEFT or RIGHT

Semi-joins (Bernstein 1979)

Many improvements in query optimization use *semi-joins*

Want to compute join(A,B,…), where A and B are at different sites (more on distributed query optimization later)

- observation: shipping all tuples of B is expensive. Ship only those tuples of B that join with A.

- idea: perform half of a join or “semi-join”

semijoin(B,A,J) algorithm:

- ship join column of A to B’s site

  note: implicit retrieval of A if A is stored

  note: implicit storage and projection of A, shipment to B’s site in any case

- perform join of B with A’s join column (using any join algorithm)

  note: only tuples of B that join with A tuples remain

Bloom Filters (Cont)

Similar to semi-joins, but more efficient

bloom(B,A,J) algorithm:

1. **(bloom)** define a bit map *M* of *m* bits
2. **(bloom)** clear *M*
3. **(bloom)** for each join value *v* of A, hash *v* and set the corresponding bit(s) in *M*

   note: implicit retrieval of A tuples in step 3
   note: send M to site of B

4. **(bloom filter)** eliminate B tuples whose join values do not have their corresponding hash bit(s) set in *M*

   note: Bloom filters are probabilistic and hence may not filter *all* unwanted join values

   note: value of *m* is optimizable

Bloom Filters (Cont)

Note advantages:

- semi-joins may require considerable storage for column values and considerable expense for shipping column values

- bloom filters require very small storage space (even for a fairly large bit map), and little expense for shipping bit map

Optimization of Other Logical Operations

Lot of work optimizing aggregation and group-by expressions

- instance of understanding the algebraic equivalences among expressions using these operators

Example: left-outer-join operation   loj(B, C)

loj(join(A, B), C) = join(A, loj(B, C))

Example: Group-by and Join

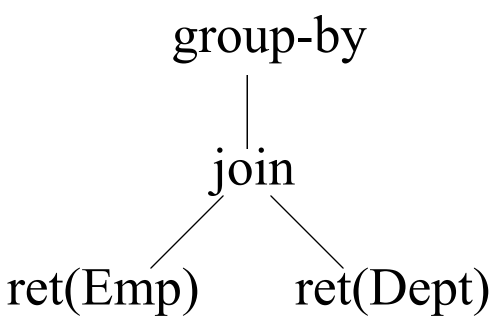
groupBy(join(A, B)) = join(groupBy(A), B)

Above is possible only if you look at the *(above hidden) parameters* of GroupBy. Example:

```
SELECT D.d#, Ave( E.sal )
FROM Emp E, Dept D
WHERE E.d# = D.d# AND E.age < 30 AND
      D.location = 'Austin'
GROUP_By D.d#
```

Look Closer

Original operator tree:



Example relations:

Emp

e#	sal	d#
A	12	1
B	8	1
C	20	2
D	30	3

Dept

d#	name	location
1	d1	austin
2	d2	austin

Emp * Dept

e#	sal	d#	name	location
A	12	1	d1	austin
B	8	1	d1	austin
C	20	2	d2	austin

assume above Emps have age < 30 and Depts are in austin

qualify Emp * Dept, group_by yields:

d#	ave(sal)
1	10
2	20

Note: we could have computed same result differently (and maybe a *cheaper* way)

- no guarantee that it is cheaper, but let cost equations decide this
- key idea is to offer the optimizer *choices*

Continued

Compute aggregation on **`Emp`** directly:

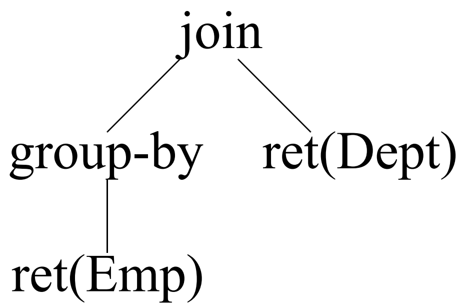
```
create view as Temp (
   select d#, ave(sal) as asal
   from Emp
   where age < 30
   group_by d#
)
```

Temp

d#	ave(sal)
1	10
2	20
3	30

Notice: join with **`Dept`** (because it is a foreign key join — each employee works in a single department — this constraint is important!) just filters **`Temp`** rows. So equivalent query is:

```
select T.d# T.avesal
from Temp T, Dept D
where D.d# = T.d# and D.location = 'Austin'
```



Example of join really being a "semi-join"

Nested and Correlated SubQueries

Hard to write, easy to generate, but can they be processed efficiently? Try this **`g≠h≠p`**:

```
SELECT Ck
FROM   Ri
WHERE  Cg > 40 AND Ch > (
   SELECT AGG(Cm)
   FROM   Rj
   WHERE  Cn = Ri.Cp )
```

- one way to process is via nested loops: loop over locally-qualified **`Ri`** tuples, perform inner selection and aggregation as a method call for each **`Ri`** tuple.

Kim showed a more efficient way by rewriting nested queries into separately evaluable queries:

- look at above query — for each value of **`Ri.Cp`**, compute the aggregate
- note: redundant aggregations are computed when the same **`Ri.Cp`** value appears several times
- better: compute the aggregate for each unique **`Rj.Cn`** value in a **`Temp`** relation, and then join the **`Temp`** relation with **`Ri`**.

Nested and Correlated SubQueries (Continued)

Create all aggregations at once

```
CREATE VIEW KimTemp AS (
   SELECT Cn, AGG(Cm) AS Agg
   FROM   Rj
   GROUP BY Cn )
```

Compute the result of the original query:

```
SELECT Ri.Ck
FROM   Ri, KimTemp T
WHERE  Ri.Cp = T.Cn AND Ri.Ch > T.Agg
       AND Ri.Cg > 40
```

Remember:

- variety of different query rewrite rules (such as above)
- again, gives optimizer more choices to look at — only after the cost functions are evaluated is it clear if the rewrite was good or not

Magic Sets

Kim's rewrites transform query expressions into other query expressions and he provides cost functions that evaluate the benefit of doing so

- but there are still other possibilities
- Magic Sets use horn clauses of Prolog but the essential idea is this: there are other non-Kim rewrite rules that involve semi-joins

Look again at the query considered earlier

```
SELECT Ck
FROM   Ri
WHERE  Cg > 40 AND Ch > (
   SELECT AGG(Rj.Cm)
   FROM   Rj
   WHERE  Cn = Ri.Cp )
```

- Kim creates a relation (`KimTemp`) that computes the aggregate for *all* `Rj.Cn` values
- but *not all* `Rj.Cn` values are needed, so potentially unnecessary work is being done
- instead, use a semi-join to eliminate unnecessary `Rj.Cn` values and aggregate computations

Magic Sets (Cont)

First, compute the set of “good” `Rj.Cp` values:

```
CREATE VIEW GOOD_ONES AS (
   SELECT Cp
   FROM   Ri
   WHERE  Cg > 40 )
```

Next, compute the aggregate relation for only the good values

```
CREATE VIEW MagicTemp AS (
   SELECT Rj.Cn, AGG(Rj.Cm) AS Agg
   FROM   Rj, GOOD_ONES
   WHERE  Rj.Cn = GOOD_ONES.Cp   // semi-join
   GROUP BY Rj.Cn )
```

Lastly, compute result of original query:

```
SELECT Ri.Ck
FROM   Ri, MagicTemp T
WHERE  Ri.Cp = T.Cn AND Ri.Ch > T.Agg
       AND Ri.Cg > 40 - - - - - - - - - - needed?
```

Note: lots of Magic Set techniques like this...

Query Optimizer Directions

Most likely direction for large DBMSs are those based on algebraic rewrite rules

- 2 kinds of rules:

  transformation rules — equivalences among logical expressions

  implementation rules — logical operation to physical operation mappings

- idea is to convert SQL query into an expression, and progressively rewrite the expression into an equivalent, more efficiently executable counterpart

- 2 systems that do this:

  Starburst (IBM) — generate set of logical plans first, and then convert into set of efficient physical plans

  Volcano/Cascade (Microsoft) — intermix generation of logical and physical plans (in top down manner)

- Remains to be seen if ad hoc techniques will prevail…

Other Variants

We presented the “standard” model of query optimization, but there are “enhancements” that others have examined

Basic idea: we’re composing operators (e.g., X·Y))

- each operation has a separate implementation
- but there are unified algorithms of “combined” operators XY = X·Y that are much more efficient

Example: DeWitt et al. looked at the following operator combination:

- UNIQUE·SORT and SORT·UNIQUE
- algorithm is straightforward: using a merge-sort, whenever duplicates are discovered, they are removed
- Lo! better performance

General model proposed by Freytag that applies general program rewriting techniques

- expansion into primitives, reshuffling of code, simplifying using rewrite rules

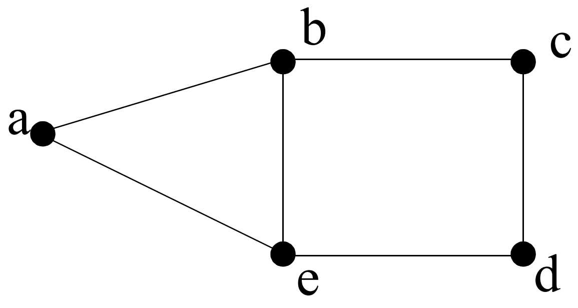
Distributed Query Optimization

Reading:

- Mackert and Lohman, “R* Optimizer Validation and Performance Evaluation for Distributed Queries”

Distributed database (DDBMS) connects different DBMSs over a network to provide a “centralized” view of their databases

- *homogeneous* if same kind of DBMSs connected
- *heterogeneous* otherwise



a-e are sites

edges are direct connections

- at each site is a centralized DBMS
- DDBMSs assumes no site or communication failures
- DDBMS is simply client of available network protocols
- “sees” site information: relation A is at site a, B is at b

Distributed Query Optimization (Cont)

How does distribution of data affect query optimization?

Ans #1: *Distributed Ingres* solution

- *home* site is site at which query originated
- all retrievals are performed at the site of their relations
- all other operations are performed at the home site

Background: look at the expression:

$$P = \text{end}(\ \text{join}(\ \text{ret}(A), \text{join}(\ \text{ret}(B), \text{ret}(C)\ )\ )\ )$$

- "for now" notation: join and retrieval sort orders ignored, join and selection predicates ignored
- let O^s denote that operation O is executed at site s
- how many new access plans do we have with the Distributed Ingres strategy?

  ans:

  $$P = \text{end}^{\text{home}}(\ \text{join}^{?}\ (\ \text{ret}^{?}(A), \text{join}^{?}\ (\ \text{ret}^{?}\ (B), \text{ret}^{?}(C)\ )\ )\ )$$

Distributed Query Optimization (Cont)

Ans #2: *R** (distributed version of System R) solution:

- retrievals only at site of their files (*F* is at site *f*)
- joins at site of either the inner file, outer file, or a “wild-card” site (denoted *)
- end operation at home site

Centralized Operation	Becomes	Distributed Operation
ret(F)	→	$\text{ret}^{f}(F)$
join(X,Y)	→	$\text{join}^{x}(X^{x},Y^{y})$
		$\text{join}^{y}(X^{x},Y^{y})$
		$\text{join}^{*}(X^{x},Y^{y})$
end(X)	→	$\text{end}^{\text{home}}(X^{x})$

Example: what are w = { } and u = { } below?

$$P = \text{end}^{\text{home}}(\ \text{join}^{w}(\ \text{ret}^{a}(A),\ \text{join}^{u}(\ \text{ret}^{b}(B),\ \text{ret}^{c}(C)\ )\ )\ )$$

Distributed Query Optimization (Cont)

Note implicit use of xfer( ) (transfer/shipment) operation:

- $xfer^b(A^a)$ transfers stream A at site a to site b
- thus, $foo^b(A^a)$ is really a shorthand for $foo^b(xfer^b(A^a))$

Note algebraic identity or simplification:

$$foo^b(xfer^b(A^a)) \;=\; foo^b(A^a)$$

Note: implicit use of xfer( ) in OO parlance is a type conversion

Hint: we want to understand query processing algorithms as algebraic rewrites…

Distributed Query Optimization (Cont)

Analysis: given an centralized access plan P with *n* join operations and *n+1* retrieval operations, how may distributed *logical access plans* can be generated from P

- using the Distributed Ingres algorithm, given centralized Ingres algorithm produces *CI(n)* plants

- using the R* algorithm without wild cards?

- using the R* algorithm with wild cards?

Comments?

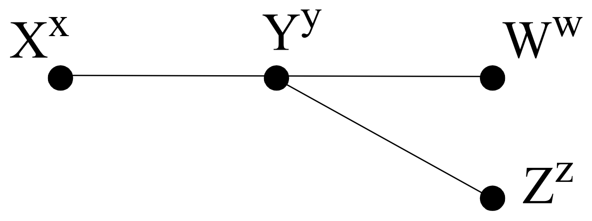
Distributed Query Optimization (Cont)

Distributed Ingres query optimization algorithm is:

- basically no different than the centralized Ingres algorithm
- negligible additional information kept, no new plans must be considered

R* query optimization algorithm is:

- basically no different than that of System R
- must keep track of additional plans with site information
- recall previous plan



R* Algorithm

Pass 1 produces sinks: X^x, Y^y, Z^z, W^w

Pass 2 produces sinks: XY^x, XY^y, YZ^y, YZ^z, YW^y, YW^w

Pass 3 produces sinks: XYZ^x, XYZ^y, XYZ^z, XYW^x, XYW^y, XYW^w, YZW^y, YZW^z, YZW^w

Pass 4 produces sinks: $XYZW^x$, $XYZW^y$, $XYZW^z$, $XYZW^w$

Pass 5 yields $XYZW^{home}$

Cost Functions

Straightforward extension of centralized cost model:

total_cost = CPU
+ I/O
+ Number_of_messages *// new*
+ Bytes_transferred *// new*

Note: weighted average of the above, either in time or in $$

- R* uses milliseconds

Note: depending on the network speed, different weights will be used:

- *slow network will cause optimization algorithm to minimize the volume of data transmitted (and cause the selection of very different join algorithms, join/semi-join site selections)*
- *some people advocate simply minimizing transmission times (and delays)!*

Summary

General problems of Relational Query Optimization:

- simple optimizations handled by bottom-up dynamic programming approaches ala System R

  still common today…

- ad hoc extensions (e.g., to rewrite nested queries) have sufficed for now when new operations are introduced

  group-by, aggregation, magic set & Kim rewrites, etc.

- ultimately, seems that rule-based optimizers might be the standard for the future

- cost models (collecting accurate statistics, estimating the number of records produces, estimating the cost of operators) remains a difficult problem — because it is!

Other topics to cover:

- user-defined functions, parallel execution of relational queries