Query Processing Notes
CSE232

Query Processing

- The query processor turns user queries and data modification commands into a query plan - a sequence of operations (or algorithm) on the database
  - from high level queries to low level commands
- Decisions taken by the query processor
  - Which of the algebraically equivalent forms of a query will lead to the most efficient algorithm?
  - For each algebraic operator what algorithm should we use to run the operator?
  - How should the operators pass data from one to the other? (eg, main memory buffers, disk buffers)

<u>Example</u>

Select B,D
From R,S
Where R.A = "c" $\wedge$ S.E = 2 $\wedge$ R.C=S.C

R	A	B	C
	a	1	10
	b	1	20
	c	2	10
	d	2	35
	e	3	45

S	C	D	E
	10	x	2
	20	y	2
	30	z	2
	40	x	1
	50	y	3

Answer

B	D
2	x

• How do we execute query eventually?

One idea

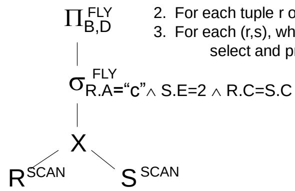
- Scan relations
- Do Cartesian product
- Select tuples
- Do projection

RxS	R.A	R.B	R.C	S.C	S.D	S.E
	a	1	10	10	x	2
	a	1	10	20	y	2
	.					
	.					
Bingo! Got one...	C	2	10	10	x	2
	.					
	.					

Relational Algebra - can be enhanced to describe plans...

Ex: Plan I

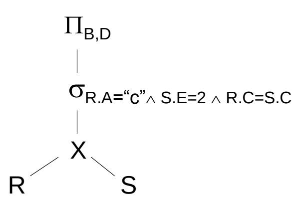
1. Scan R
2. For each tuple r of R scan S
3. For each (r,s), where s in S select and project on the fly



OR: $\Pi^{FLY}_{B,D} [\sigma^{FLY}_{R.A=\text{"c"} \wedge S.E=2 \wedge R.C = S.C} (R^{SCAN} X S^{SCAN})]$

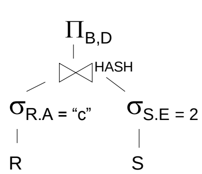
"FLY" and "SCAN" are the defaults

Ex: Plan I



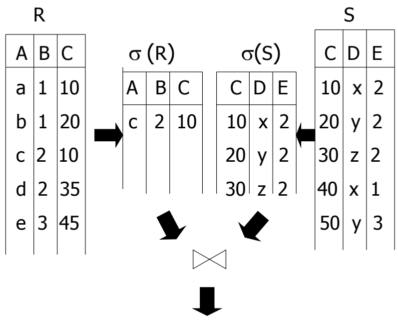
Another idea:

Plan II



$\bowtie$ natural join

Scan R and S, perform on the fly selections, do hash join, project



Plan III

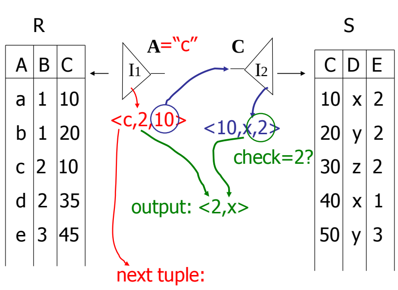
Use R.A and S.C Indexes

(1) Use R.A index to select R tuples with R.A = "c"

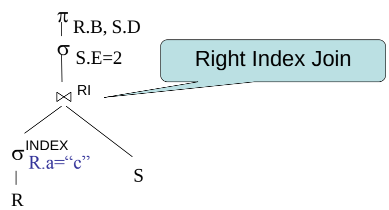
(2) For each R.C value found, use S.C index to find matching join tuples

(3) Eliminate join tuples S.E ≠ 2

(4) Project B,D attributes



Algebraic Form of Plan

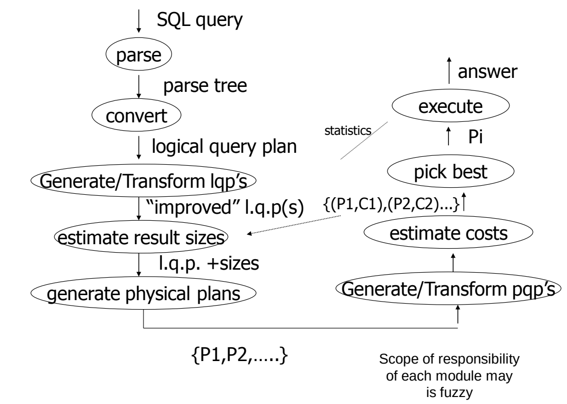


From Query To Optimal Plan

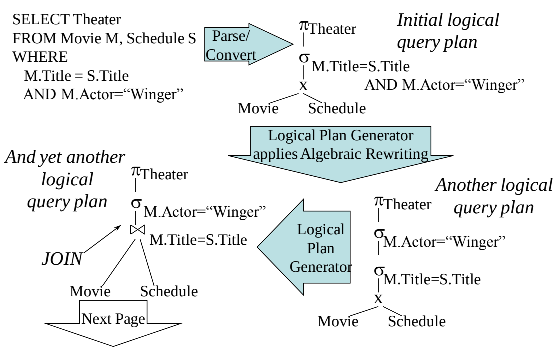
- Complex process
- Algebra-based logical and physical plans
- Transformations
- Evaluation of multiple alternatives

Issues in Query Processing and Optimization

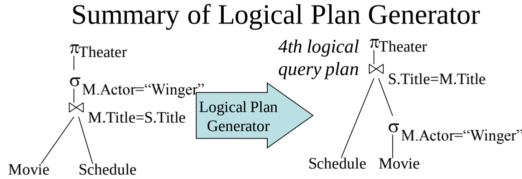
- Generate Plans
  - employ efficient execution primitives for computing relational algebra operations
  - systematically transform expressions to achieve more efficient combinations of operators
- Estimate Cost of Generated Plans
  - Statistics
- "Smart" Search of the Space of Possible Plans
  - always do the "good" transformations (relational algebra optimization)
  - prune the space (e.g., System R)
- Often the above steps are mixed



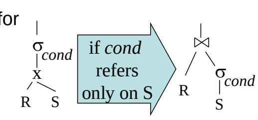
Example: The Journey of a Query



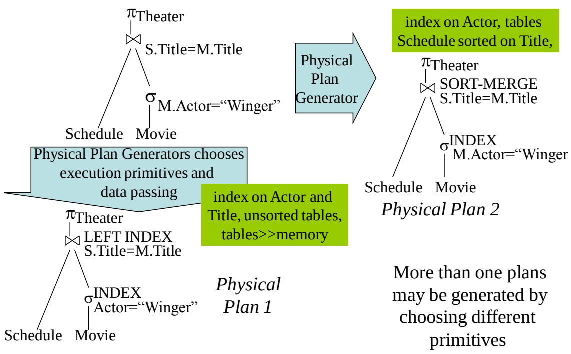
The Journey of a Query cont'd: Summary of Logical Plan Generator



- 4 logical query plans created
- algebraic rewritings were used for producing the candidate logical query plans plans
- the last one is the winner (at least, cannot be a big loser)
- in general, multiple logical plans may "win" eventually



The Journey of a Query Continues at the Physical Plan Generator

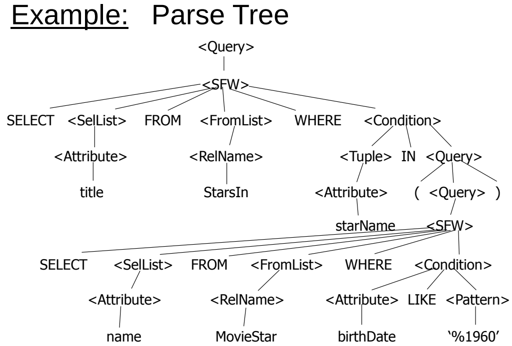


Example: Nested SQL query

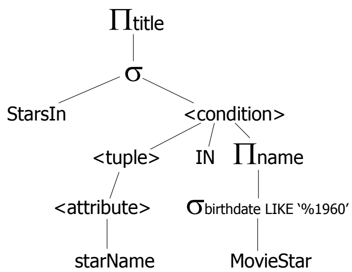
```
SELECT title
FROM StarsIn
WHERE starName IN (
        SELECT name
        FROM MovieStar
        WHERE birthdate LIKE '%1960'
);
```

(Find the movies with stars born in 1960)

Example: Parse Tree

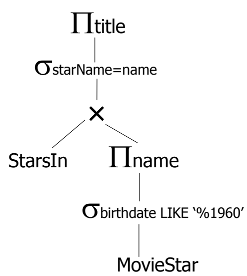


Example: Generating Relational Algebra



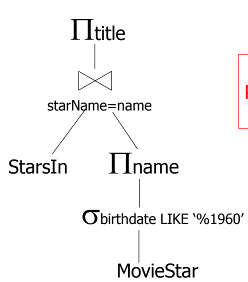
An expression using a two-argument σ, midway between a parse tree and relational algebra

Example: Logical Query Plan (Relational Algebra)



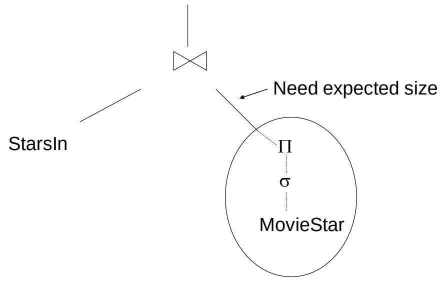
May consider "IN" elimination as a rewriting in the logical plan generator or may consider it a task of the converter

Example: Improved Logical Query Plan

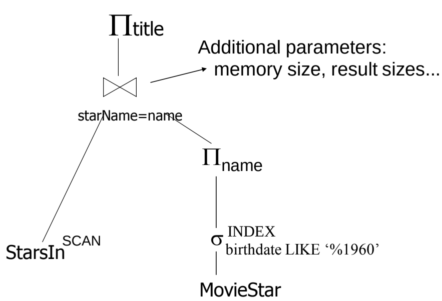


Question: Push project to StarsIn?

Example: Result sizes are important for selecting physical plans



Example: One Physical Plan



Topics

- Bag Algebra, List Algebra and other extensions
  - name & value conversions, functions, aggregation

Algebraic Operators: A Bag version

- *Union of R and S*: a tuple t is in the result as many times as the sum of the number of times it is in R plus the times it is in S
- *Intersection of R and S*: a tuple t is in the result the minimum of the number of times it is in R and S
- *Difference of R and S*: a tuple t is in the result the number of times it is in R minus the number of times it is in S
- $\delta(R)$ converts the bag *R* into a set
  - SQL's *R UNION S* is really $\delta(R \cup S)$
- **Example:** Let *R={A,B,B}* and *S={C,A,B,C}*. Describe the union, intersection and difference...

Extended Projection

- We extend the relational project π_A as follows:
  - The attribute list may include $x \rightarrow y$ in the list *A* to indicate that the attribute *x* is renamed to *y*
  - Arithmetic, string operators and scalar functions on attributes are allowed. For example,
    - $a+b \rightarrow x$ means that the sum of *a* and *b* is renamed into *x*.
    - $c||d \rightarrow y$ concatenates the result of *c* and *d* into a new attribute named *y*
- The result is computed by considering each tuple in turn and constructing a new tuple by picking the attributes names in *A* and applying renamings and arithmetic and string operators
- **Example:**

An Alternative Approach to Arithmetic and Other 1-1 Computations

- Special purpose operators that for every input tuple they produce one output tuple
  - $MULT_{A,B \rightarrow C}R$: for each tuple of *R*, multiply attribute *A* with attribute *B* and put the result in a new attribute named *C*.
  - $PLUS_{A,B \rightarrow C}R$
  - $CONCAT_{A,B \rightarrow C}R$
- **Exercise:** Write the above operators using extended projection. Assume the schema of *R* is *R(A,B,D,E).*

Products and Joins

- *Product of R and S* ($R \times S$):
  - If an attribute named *a* is found in both schemas then rename one column into *R.a* and the other into *S.a*
  - If a tuple *r* is found *n* times in *R* and a tuple *s* is found *m* times in *S* then the product contains *nm* instances of the tuple *rs*
- Joins
  - *Natural Join* $R \bowtie S = \pi_A \sigma_C(R \times S)$ where
    - *C* is a condition that equates all common attributes
    - *A* is the concatenated list of attributes of *R* and *S* with no duplicates
    - you may view tha above as a rewriting rule
  - *Theta Join*
    - arbitrary condition involving multiple attributes

Grouping and Aggregation

- $\gamma_{GroupByList;\ aggrFn1 \rightarrow attr1,\ldots,aggrFnN \rightarrow attrN}$
- Conceptually, grouping leads to nested tables and is immediately followed by functions that aggregate the nested table
- Example: $\gamma_{Dept;\ AVG(Salary) \rightarrow AvgSal,\ldots,\ SUM(Salary) \rightarrow SalaryExp}$

Find the average salary for each department
SELECT Dept, AVG(Salary) AS AvgSal,
 SUM(Salary) AS SalaryExp
FROM Employee
GROUP-BY Dept

Employee

Name	Dept	Salary
Joe	Toys	45
Nick	PCs	50
Jim	Toys	35
Jack	PCs	40

Dept	Nested Table Name Salary
Toys	Joe 45 Jim 35
PCs	Nick 50 Jack 40

Dept	AvgSal	SalaryExp
Toys	40	80
PCs	45	90

Grouping and Aggregation: An Alternate approach

- Operators that combine the *GROUP-BY* clause with the aggregation operator (*AVG,SUM,MIN,MAX*,…)
- $SUM_{GroupbyList;GroupedAttribute \rightarrow ResultAttribute}\ R$ corresponds to SELECT *GroupbyList*,
  SUM(*GroupedAttribute*) AS *ResultAttribute*
  FROM *R*
  GROUP BY *GroupbyList*
- Similar for *AVG,MIN,MAX,COUNT…*
- Note that $\delta(R)$ could be seen as a special case of grouping and aggregation
- **Example**

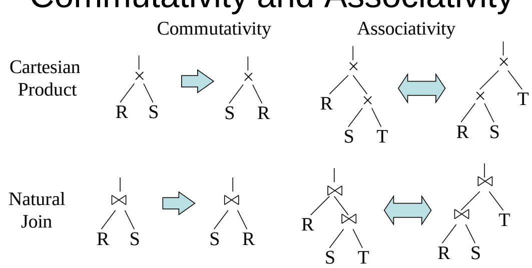
Sorting and Lists

- SQL and algebra results are ordered
- Could be non-deterministic or dictated by SQL ORDER BY, algebra τ
- $\tau_{OrderByList}$
- A result of an algebraic expression o(exp) is ordered if
  - If o is a τ
  - If o retains ordering of exp and exp is ordered
    - Unfortunately this depends on implementation of o
  - If o creates ordering
  - Consider that leaf of tree may be SCAN(R)

<u>Relational algebra optimization</u>

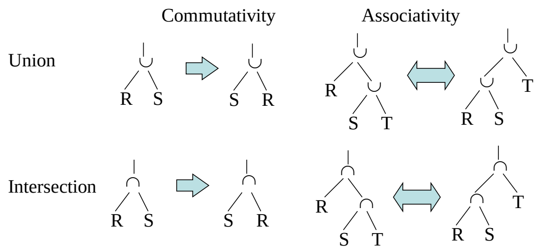
- Transformation rules (preserve equivalence)
- What are good transformations?

Algebraic Rewritings: Commutativity and Associativity



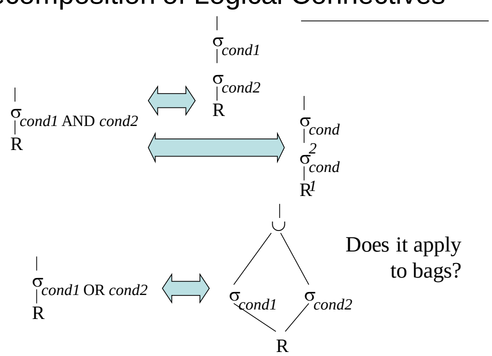
Question 1: Do the above hold for both sets and bags?
Question 2: Do commutativity and associativity hold for arbitrary Theta Joins?

Algebraic Rewritings: Commutativity and Associativity (2)

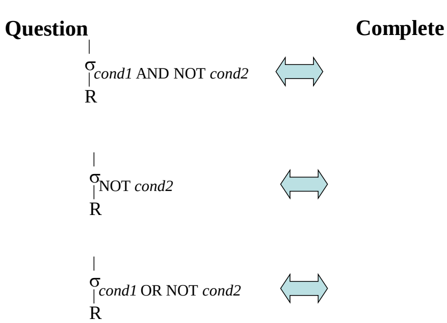


Question 1: Do the above hold for both sets and bags?
Question 2: Is difference commutative and associative?

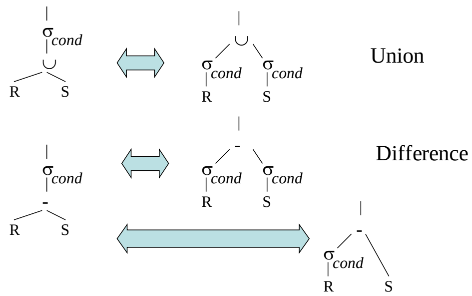
Algebraic Rewritings for Selection: Decomposition of Logical Connectives



Algebraic Rewritings for Selection: Decomposition of Negation

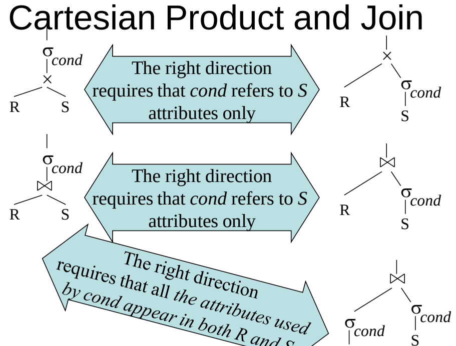


Pushing the Selection Thru Binary Operators: Union and Difference



Exercise: Do the rule for intersection

Pushing Selection thru Cartesian Product and Join



Exercise: Do the rule for theta join

Rules: π,σ combined

Let x = subset of R attributes

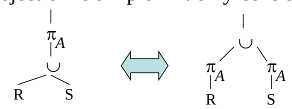
z = attributes in predicate P

(subset of R attributes)

$$\pi_x[\sigma_{p}(R)] = \pi_x\{\sigma_p[\overset{\pi_{xz}}{\cancel{\pi_x}}(R)]\}$$

Pushing Simple Projections Thru Binary Operators

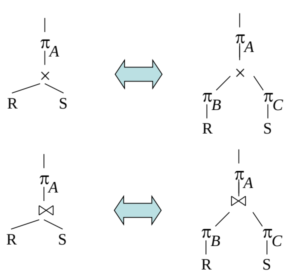
A projection is simple if it only consists of an attribute list



Union

Question 1: Does the above hold for both bags and sets?
Question 2: Can projection be pushed below intersection and difference?
Answer for both bags and sets.

Pushing Simple Projections Thru Binary Operators: Join and Cartesian Product

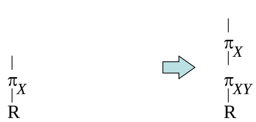


Where *B* is the list of *R* attributes that appear in *A.* Similar for *C.*

Question: What is *B* and *C ?*

Exercise: Write the rewriting rule that pushes projection below theta join.

Projection Decomposition



Some Rewriting Rules Related to Aggregation: SUM

- $\sigma_{cond}\ SUM_{GroupbyList;GroupedAttribute \rightarrow ResultAttribute} R$
  $\Leftrightarrow SUM_{GroupbyList;GroupedAttribute \rightarrow ResultAttribute}\ \sigma_{cond}\ R$,
  if *cond* involves only the *GroupbyList*
- $SUM_{GL;GA \rightarrow RA}(R \cup S)$
  $\Leftrightarrow PLUS_{RA1,\ RA2:RA}$
  $((SUM_{GL;GA \rightarrow RA11}\ R) \bowtie (SUM_{GL;GA \rightarrow RA2}\ S)))$
- $SUM_{GL2;RA1 \rightarrow RA2}\ SUM_{GL1;GA \rightarrow RA1} R \Leftrightarrow$
  $SUM_{GL2:GA \rightarrow RA2} R$
  - **Question:** does the above hold for both bags and sets?

Derived Rules: $\sigma + \bowtie$ combined

More Rules can be Derived:

$\sigma_{p \wedge q}$ (R $\bowtie$ S) =

$\sigma_{p \wedge q \wedge m}$ (R $\bowtie$ S) =

$\sigma_{p \vee q}$ (R $\bowtie$ S) =

p only at **R**, **q** only at **S**, **m** at both **R** and **S**

--> Derivation for first one:

$\sigma_{p \wedge q}$ (R $\bowtie$ S) =

σ_{p} [σ_{q} (R $\bowtie$ S)] =

σ_{p} [R $\bowtie$ σ_{q} (S)] =

[σ_{p} (R)] $\bowtie$ [σ_{q} (S)]

Which are always "good" transformations?

- □ $\sigma_{p1 \wedge p2}(R) \rightarrow \sigma_{p1}[\sigma_{p2}(R)]$
- □ $\sigma_p(R \bowtie S) \rightarrow [\sigma_p(R)] \bowtie S$
- □ $R \bowtie S \rightarrow S \bowtie R$
- □ $\pi_x[\sigma_p(R)] \rightarrow \pi_x\{\sigma_p[\pi_{xz}(R)]\}$

<u>In textbook:</u> more transformations

- Eliminate common sub-expressions
- Other operations: duplicate elimination

<u>Bottom line:</u>

- No transformation is <u>always</u> good at the l.q.p level
- Usually good:
  - early selections
  - elimination of cartesian products
  - elimination of redundant subexpressions
- Many transformations lead to "promising" plans
  - Commuting/rearranging joins
  - In practice too "combinatorially explosive" to be handled as rewriting of l.q.p.

Algorithms for Relational Algebra Operators

- Three primary techniques
  - Sorting
  - Hashing
  - Indexing
- Three degrees of difficulty
  - data small enough to fit in memory
  - too large to fit in main memory but small enough to be handled by a "two-pass" algorithm
  - so large that "two-pass" methods have to be generalized to "multi-pass" methods (quite unlikely nowadays)

The dominant cost of operators running on disk:

- Count # of disk blocks that must be read (or written) to execute query plan

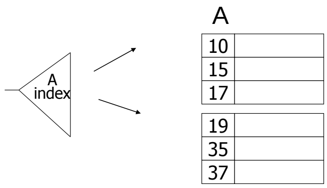
To estimate costs, we use additional parameters:

B(R) = # of blocks containing R tuples
f(R) = max # of tuples of R per block
M = # memory blocks available
Sorting information

HT(i) = # levels in index I
Caching information (eg, first levels of index always cached)
LB(i) = # of leaf blocks in index i

Clustering index

Index that allows tuples to be read in an order that corresponds to a sort order



Clustering can radically change cost

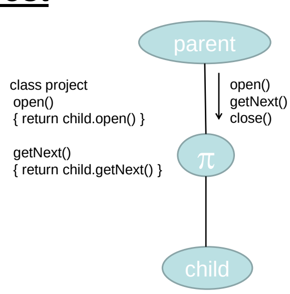
- Clustered relation

  R1 R2 R3 R4 | R5 R5 R7 R8 | …..

- Clustering index

Pipelining can radically change cost

- Interleaving of operations across multiple operators
- Smaller memory footprint, fewer object allocations
- Operators support:
  - open()
  - getNext()
  - close()
- Simple for unary
- Pipelined operation for binary discussed along with physical operators



Example R1 $\bowtie$ R2 over common attribute C

First we will see main memory-based implementations

- Iteration join (conceptually – without taking into account disk block issues)

```
for each r ∈ R1 do
   for each s ∈ R2 do
      if r.C = s.C then output r,s pair
```

- Merge join (conceptually)

```
(1) if R1 and R2 not sorted, sort them
(2) i ← 1; j ← 1;
    While (i ≤ T(R1)) ∧ (j ≤ T(R2)) do
       if R1{ i }.C = R2{ j }.C then outputTuples
       else if R1{ i }.C > R2{ j }.C then j ← j+1
       else if R1{ i }.C < R2{ j }.C then i ← i+1
```

Procedure Output-Tuples

```
While (R1{ i }.C = R2{ j }.C) ∧ (i ≤ T(R1)) do
    [jj ← j;
    while (R1{ i }.C = R2{ jj }.C) ∧ (jj ≤ T(R2)) do
            [output pair R1{ i }, R2{ jj };
             jj ← jj+1  ]
    i ← i+1  ]
```

Example

i	R1{i}.C	R2{j}.C	j
1	10	5	1
2	20	20	2
3	20	20	3
4	30	30	4
5	40	30	5
		50	6
		52	7

- Join with index (Conceptually)

Assume R2.C index

```
For each r ∈ R1 do
  [ X ← index (R2, C, r.C)
    for each s ∈ X do
        output r,s pair]
```

Note: X ← index(rel, attr, value)
then X = set of rel tuples with attr = value

- Hash join (conceptual)
  - Hash function h, range 0 → k
  - Buckets for R1: G0, G1, ... Gk
  - Buckets for R2: H0, H1, ... Hk

<u>Algorithm</u>

(1) Hash R1 tuples into G buckets

(2) Hash R2 tuples into H buckets

(3) For i = 0 to k do

match tuples in Gi, Hi buckets

<u>Simple example</u> hash: even/odd

R1	R2
2	5
4	4
3	12
5	3
8	13
9	8
	11
	14

Buckets

	R1	R2
Even	2 4 8	4 12 8 14
Odd:	3 5 9	5 3 13 11

<u>Factors that affect performance</u>

(1) Tuples of relation stored physically together?

(2) Relations sorted by join attribute?

(3) Indexes exist?

Disk-oriented Computation Model

- There are *M* main memory buffers.
  - Each buffer has the size of a disk block
- The input relation is read one block at a time.
- The cost is the number of blocks read.
- If *B* consecutive blocks are read the cost is *B/d*.
- The output buffers are not part of the *M* buffers mentioned above.
  - *Pipelining* allows the output buffers of an operator to be the input of the next one.
  - We do not count the cost of writing the output.

Notation

- $B(R)$ = number of blocks that R occupies
- $T(R)$ = number of tuples of R
- $V(R,[a_1, a_2, \ldots, a_n])$ = number of distinct tuples in the projection of R on $a_1, a_2, \ldots, a_n$

One-Pass Main Memory Algorithms for Unary Operators

- Assumption: Enough memory to keep the relation
- Projection and selection:
  - Scan the input relation *R* and apply operator one tuple at a time
  - Incremental cost of "on the fly" operators is 0
- Duplicate elimination and aggregation
  - create one entry for each group and compute the aggregated value of the group
  - it becomes hard to assume that CPU cost is negligible
    - main memory data structures are needed

One-Pass Nested Loop Join

- Assume *B(R)* is less than *M*
- Tuples of *R* should be stored in an efficient lookup structure
- **Exercise:** Find the cost of the algorithm below

```
for each block Br of R do
  store tuples of Br in main memory
for each each block Bs of S do
  for each tuple s of Bs
    join tuples of s with matching tuples of R
```

Generalization of Nested-Loops

```
for each chunk of M-1 blocks Br of R do
  store tuples of Br in main memory
  for each each block Bs of S do
    for each tuple s of Bs
      join tuples of s with matching tuples of R
```

Exercise: Compute cost

Simple Sort-Merge Join

- Assume natural join on *C*
- Sort *R* on *C* using the two-phase multiway merge sort
  - if not already sorted
- Sort *S* on *C*
- Merge (opposite side)
  - assume two pointers `Pr,Ps` to tuples on disk, initially pointing at the start
  - sets `R', S'` in memory
- Remarks:
  - Very low average memory requirement during merging (but no guarantee on how much is needed)
  - **Cost:**

```
while Pr!=EOF and Ps!=EOF
  if *Pr[C] == *Ps[C]
    do_cart_prod(Pr,Ps)
  else if *Pr[C] > *Ps[C]
    Ps++
  else if *Ps[C] > *Pr[C]
    Pr++

function do_cart_prod(Pr,Ps)
  val=*Pr[C]
  while *Pr[C]==val
    store tuple *Pr in set R'
  while *Ps[C]==val
    store tuple *Ps in set S';
  output cartesian product
      of R' and S'
```

Efficient Sort-Merge Join

- Idea: Save two disk I/O's per block by combining the second pass of sorting with the ``merge".
- Step 1: Create sorted sublists of size *M* for *R* and *S*
- Step 2: Bring the first block of each sublist to a buffer
  - assume no more than *M* sublists in all
- Step 3:Repeatedly find the least *C* value *c* among the first tuples of each sublist. Identify all tuples with join value *c* and join them.
  - When a buffer has no more tuple that has not already been considered load another block into this buffer.

Efficient Sort-Merge Join Example

R

C	*RA*
1	r_1
2	r_2
3	r_3
…	
20	r_{20}

S

C	*SA*
1	s_1
…	
5	s_5
16	s_{16}
…	
20	s_{20}

Assume that after first phase of multiway sort we get 4 sublists, 2 for *R* and 2 for *S*.
Also assume that each block contains two tuples.

R

3 7	8 10	11 13	14 16	17 18
1 2	4 5	6 9	12 15	19 20

S

1 3	5 17	
2 4	16 18	19 20

Two-Pass Hash-Based Algorithms

- General Idea: Hash the tuples of the input arguments in such a way that all tuples that must be considered together will have hashed to the same hash value.
  - If there are *M* buffers pick *M-1* as the number of hash buckets
- Example: Duplicate Elimination
  - Phase 1: Hash each tuple of each input block into one of the *M-1* bucket/buffers. When a buffer fills save to disk.
  - Phase 2: For each bucket:
    - load the bucket in main memory,
    - treat the bucket as a small relation and eliminate duplicates
    - save the bucket back to disk.
  - **Catch:** Each bucket has to be less than *M.*
  - **Cost:**

Hash-Join Algorithms

- Assuming natural join, use a hash function that
  - is the same for both input arguments *R* and *S*
  - uses only the join attributes
- Phase 1: Hash each tuple of *R* into one of the *M-1* buckets R_i and similar each tuple of *S* into one of S_i
- Phase 2: For *i=1…M-1*
  - load R_i and S_i in memory
  - join them and save result to disk
- **Question:** What is the maximum size of buckets?
- **Question:** Does hashing maintain sorting?

Index-Based Join: The Simplest Version

Assume that we do natural join of *R(A,B)* and *S(B,C)* and there's an index on *S*

```
for each Br in R do
  for each tuple r of Br with B value b
     use index of S to find
          tuples {s1 ,s2 ,...,sn} of S with B=b
     output {rs1 ,rs2 ,...,rsn}
```

Cost: Assuming *R* is clustered and non-sorted and the index on *S* is clustered on *B* then
B(R)+T(R)B(S)/V(S,B) + some more for reading index
Question: What is the cost if *R* is sorted?

Opportunities in Joins Using Sorted Indexes

- Do a conventional Sort-Join avoiding the sorting of one or both of the input operands

• Estimating cost of query plan

(1) Estimating size of results
(2) Estimating # of IOs

Estimating result size

- Keep statistics for relation R
  - T(R) : # tuples in R
  - S(R) : # of bytes in each R tuple
  - B(R): # of blocks to hold all R tuples
  - V(R, A) : # distinct values in R for attribute A

Example

R

A	B	C	D
cat	1	10	a
cat	1	20	b
dog	1	30	a
dog	1	40	c
bat	1	50	d

A: 20 byte string
B: 4 byte integer
C: 8 byte date
D: 5 byte string

T(R) = 5 S(R) = 37
V(R,A) = 3 V(R,C) = 5
V(R,B) = 1 V(R,D) = 4

<u>Size estimates</u> for $W = R1 \times R2$

$T(W) =$

$T(R1) \times T(R2)$

$S(W) =$

$S(R1) + S(R2)$

<u>Size estimate</u> for $W = \sigma_{Z=val}(R)$

$S(W) = S(R)$

$T(W) = ?$

<u>Example</u>

R

A	B	C	D
cat	1	10	a
cat	1	20	b
dog	1	30	a
dog	1	40	c
bat	1	50	d

$V(R,A)=3$
$V(R,B)=1$
$V(R,C)=5$
$V(R,D)=4$

$W = \sigma_{z=val}(R) \quad T(W) = \dfrac{T(R)}{V(R,Z)}$

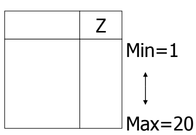
What about $W = \sigma_{z \geq val}(R)$?

$T(W) = ?$

- Solution # 1:
  $T(W) = T(R)/2$
- Solution # 2:
  $T(W) = T(R)/3$

- Solution # 3: Estimate values in range

<u>Example</u> R



$V(R,Z)=10$

$W= \sigma_{z \geq 15}(R)$

$f = \frac{20-15+1}{20-1+1} = \frac{6}{20}$ (fraction of range)

$T(W) = f \times T(R)$

Equivalently:

$f \times V(R,Z)$ = fraction of distinct values

$T(W) = [f \times V(Z,R)] \times \frac{T(R)}{V(Z,R)} = f \times T(R)$

Size estimate for $W = R1 \bowtie R2$

Let x = attributes of R1
y = attributes of R2

$\boxed{\text{Case 1}}$ $X \cap Y = \emptyset$

Same as R1 x R2

$\boxed{\text{Case 2}}$ $W = R1 \bowtie R2$ $X \cap Y = A$

R1

A	B	C

R2

A	D

Assumption:

$\Pi_A R1 \subseteq \Pi_A R2 \Rightarrow$ Every A value in R1 is in R2
(typically A of R1 is foreign key
of the primary key of A of R2)

$\Pi_A R2 \subseteq \Pi_A R1 \Rightarrow$ Every A value in R2 is in R1

"containment of value sets" (justified by primary key – foreign key relationship)

Computing T(W) when A of R1 is the foreign key $\Pi_A R1 \subseteq \Pi_A R2$

R1

A	B	C

R2

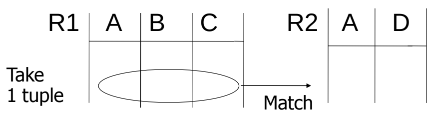
A	D

1 tuple of R1 matches with exactly 1 tuple of R2

so T(W) = T(R1)

Another way to approach when

$\Pi_A R1 \subseteq \Pi_A R2$



1 tuple matches with $\frac{T(R2)}{V(R2,A)}$ tuples...

so $T(W) = \frac{T(R2)}{V(R2, A)} \times T(R1)$

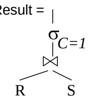
- $V(R1,A) \leq V(R2,A)$ $T(W) = \frac{T(R2)\ T(R1)}{V(R2,A)}$
- $V(R2,A) \leq V(R1,A)$ $T(W) = \frac{T(R2)\ T(R1)}{V(R1,A)}$

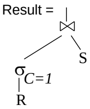
[A is common attribute]

In general $W = R1 \bowtie R2$

$$T(W) = \frac{T(R2)\ T(R1)}{\max\{ V(R1,A), V(R2,A) \}}$$

Combining estimates on subexpressions: Value preservation

Result =


Result =


R(A, C)
$T(R) = 10^3$
$V(A, R) = 10^3$
$V(C, R) = 10^2$

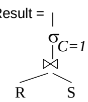
S(A, B)
$T(S) = 10^2$
$V(A, S) = 50$

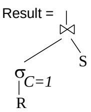
$T(R \bowtie S) =$
$T(R) \times T(S) / \max(V(A,R), V(A, S)) = 10^2$

$V(C, R \bowtie S) = 10^2$ ← (Big) assumption: Value preservation of C

$T(Result) = T(R \bowtie S) / V(C, R \bowtie S) = 1$

Value preservation may have to be pushed to a weird assumption (but there's logic behind it!)

Result =


Result =


Ideally, the size estimation should not depend on which of the two equivalent formulas for Result one uses. However, to achieve this we may need to push the value preservation assumption to artificial intermediate estimates…

R(A, C)
$T(R) = 10^3$
$V(A, R) = 10^3$
$V(C, R) = 10^2$

S(A, B)
$T(S) = 10^2$
$V(A, S) = 50$

$T(R \bowtie S) = 10^2$

$V(C, R \bowtie S) = 10^2$

$T(Result) = 1$

$T(\sigma_{c=1}R) = T(R) / V(C, R) = 10$
$V(A, \sigma_{c=1}R) = \mathbf{10^3}$
$T(Result) =$
$T(\sigma_{c=1}R) \times T(S) / \max(V(A, \sigma_{c=1}R), V(A, S)) = 1$

We had to extend value preservation to the weird assumption that attribute A has more values than the number of tuples in R. In this way the number of S tuples matching an R tuple stays steady

Value preservation of join attribute

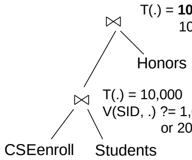
Foreign-to-primary

CSEenroll(EID, SID, …)
T(CSEenroll) = 10,000
V(SID, CSEenroll) = 1,000

Students(SID, …)
T(Students) = 20,000
V(SID, Students) = 20,000

Honors (HID, SID, …)
T(Honors) = 5,000
V(SID, Honors) = 500

$T(CSEenroll(EID, SID, \ldots) \bowtie Students(SID, \ldots) \bowtie Honors (HID, SID, \ldots)) = ?$



T(.) = **10,000 x 5,000 / max(500, <u>20,000</u>) = 2,500 CORRECT**
10,000 x 5,000 / max(500, <u>1,000</u>) = 50,000 WRONG

T(.) = 10,000
V(SID, .) ?= 1,000 (preservation of SIDs in CSEenroll)
or 20,000 (preservation of SIDs in Students) ?

If in doubt, think in terms of probabilities and matching records

- A SID of Student appears in CSEEnroll with probability 1000/20000
  - i.e., 5% of students are enrolled in CSE
- A SID of Student appears in Honors with probability 500/20000
  - i.e., 2.5% of students are honors students

=> An SID of Student appears in the join result with probability 5% x 2.5%

- On the average, each SID of CSEEnroll appears in 10,000/1,000 tuples
  - i.e., each CSE-enrolled student has 10 enrollments
- On the average, each SID of Honors appears in 5,000/500 tuples
  - i.e., each honors' student has 10 honors

⇒Each Student SID that is in both Honors and CSEEnroll is in 10x10 result tuples

⇒ T(result) = 20,000 x 5% x 2.5% x 10 x 10 = 2,500 tuples

Foreign-to-primary

CSEenroll(EID, SID, …)
T(Students) = 10,000
V(SID, Students) = 1,000

Students(SID, …)
T(Students) = 20,000
V(SID, Students) = 20,000

Honors (HID, SID, …)
T(Students) = 5,000
V(SID, Students) = 500

T(CSEenroll(EID, SID, …) ⋈ Students(SID, …) ⋈ Honors (HID, SID, …)) = ?

Plan Enumeration

- A smart exhaustive algorithm
  - According to textbook's Section 16.6
  - no ppt notes
- The INGRES heuristic for plan enumeration

Arranging the Join Order: the Wong-Youssefi algorithm (INGRES)

Sample TPC-H Schema

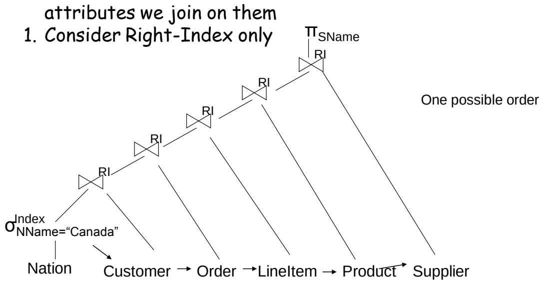
```
Nation(NationKey, NName)
Customer(CustKey, CName, NationKey)
Order(OrderKey, CustKey, Status)
Lineitem(OrderKey, PartKey, Quantity)
Product(SuppKey, PartKey, PName)
Supplier(SuppKey, SName)
```

```
SELECT SName
FROM Nation, Customer, Order, LineItem, Product, Supplier
WHERE Nation.NationKey = Cuctomer.NationKey
      AND Customer.CustKey = Order.CustKey
      AND Order.OrderKey=LineItem.OrderKey
      AND LineItem.PartKey= Product.Partkey
      AND Product.Suppkey = Supplier.SuppKey
      AND NName = "Canada"
```

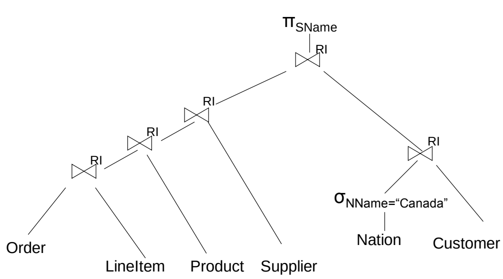
Find the names of suppliers that sell a product that appears in a line item of an order made by a customer who is in Canada

Challenges with Large Natural Join Expressions

For simplicity, assume that in the query
1. All joins are natural
2. whenever two tables of the FROM clause have common attributes we join on them
1. Consider Right-Index only



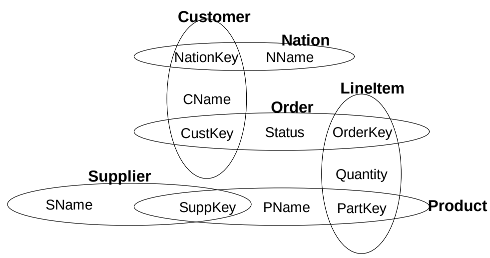
Multiple Possible Orders



Wong-Yussefi algorithm assumptions and objectives

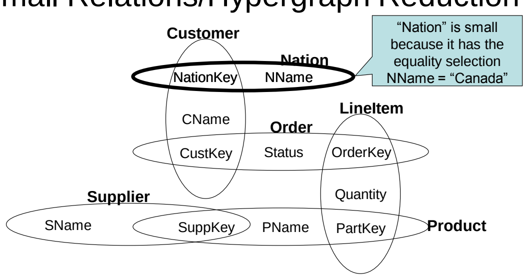
- Assumption 1 (weak): Indexes on all join attributes (keys and foreign keys)
- Assumption 2 (strong): At least one selection creates a *small* relation
  - A join with a small relation results in a small relation
- Objective: Create sequence of index-based joins such that all intermediate results are small

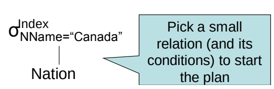
Hypergraphs

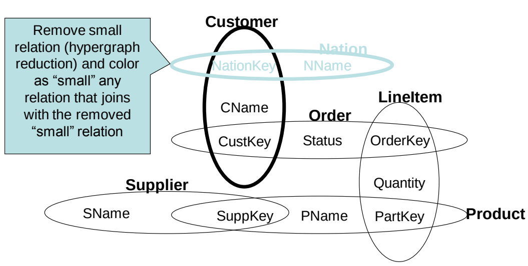


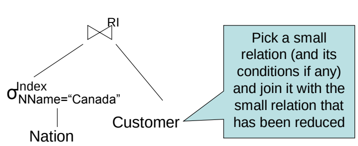
- relation hyperedges
  - two hyperedges for same relation are possible
- each node is an attribute
- can extend for non-natural equality joins by merging nodes

Small Relations/Hypergraph Reduction

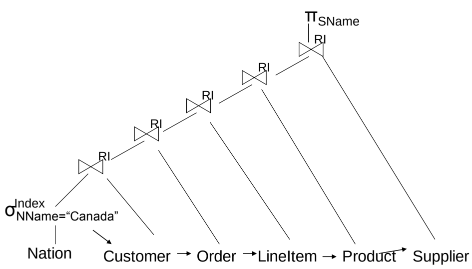








After a bunch of steps…

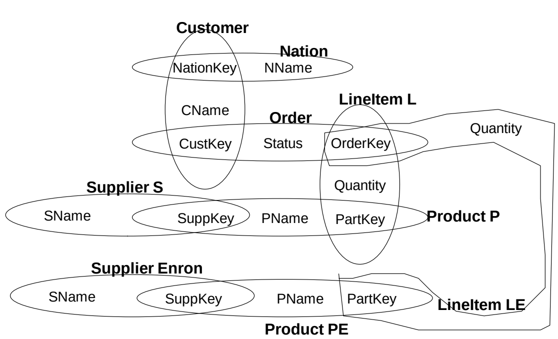


Multiple Instances of Each Relation

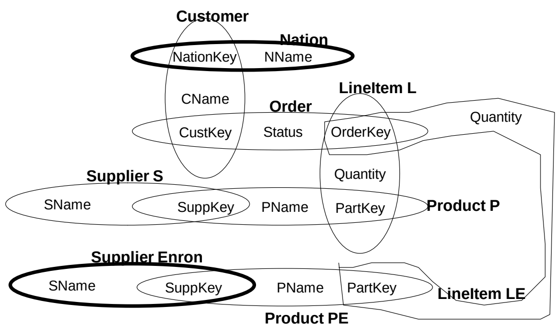
```
SELECT S.SName
FROM Nation, Customer, Order, LineItem L, Product P, Supplier S,
         LineItem LE, Product PE, Supplier Enron
WHERE Nation.NationKey = Cuctomer.NationKey
      AND Customer.CustKey = Order.CustKey
      AND Order.OrderKey=L.OrderKey
      AND L.PartKey= P.Partkey
      AND P.Suppkey = S.SuppKey
      AND Order.OrderKey=LE.OrderKey
      AND LE.PartKey= PE.Partkey
      AND PE.Suppkey = Enron.SuppKey
      AND Enron.Sname = "Enron"
      AND NName = "Cayman"
```

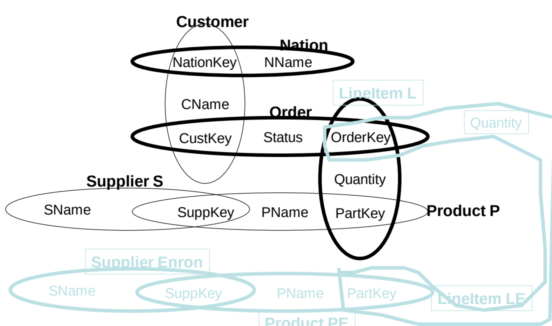
Find the names of suppliers whose products appear in an order made by a customer who is in Cayman Islands and an Enron product appears in the same order

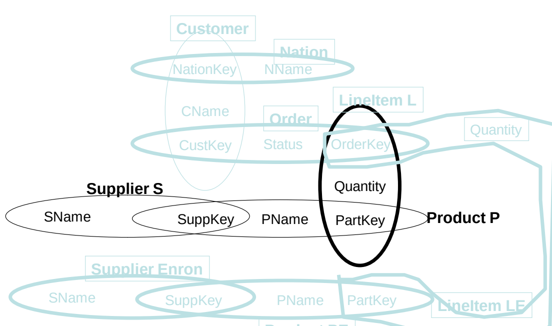
Multiple Instances of Each Relation

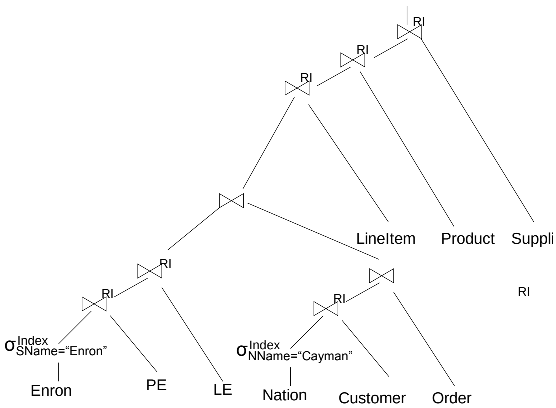


Multiple choices are possible









The basic dynamic programming approach to enumerating plans

for each sub-expression
    $op(e_1\ e_2\ \ldots\ e_n)$ of a logical plan

- (recursively) compute the best plan and cost for each subexpression e_i
- for each physical operator op^p implementing op
  - evaluate the cost of computing op using op^p and the best plan for each subexpression e_i
  - (for faster search) memo the best op^p

Local suboptimality of basic approach and the Selinger improvement

- Basic dynamic programming may lead to (globally) suboptimal solutions
- Reason: A suboptimal plan for e_1 may lead to the optimal plan for $op(e_1\ e_2\ \ldots\ e_n)$
  - Eg, consider $e_1 \bowtie_A e_2$ and
  - assume that the optimal computation of e_1 produces unsorted result
  - Optimal $\bowtie$ is via sort-merge join on A
  - It could have paid off to consider the suboptimal computation of e_1 that produces result sorted on A
- Selinger improvement: memo also any plan (that computes a subexpression) and produces an order that may be of use to ancestor operators

Using dynamic programming to optimize a join expression

- Goal: Decide the join order and join methods
- Initiate with n-ary join $\bowtie_c (e_1\ e_2\ \ldots\ e_n)$, where *c* involves only join conditions
- Bottom up: consider 2-way non-trivial joins, then 3-way non-trivial joins etc
  - "non trivial" -> no cartesian product