UNIT 15
Query Optimization

Contents

15.1 Introduction to Query Optimization

The Problem

- How to choose an efficient strategy for <u>evaluating</u> a given <u>expression</u> (a query).
  - Expression (a query):

    e.g. select distinct S.SNAME
    from S, SP
    where S.S# =SP.S# and SP.P#= 'p2'
  - Evaluate:
  - <u>Efficient strategy</u>:
    - **First class**

      e.g. (A join B) where condition-on-B
      ≡ (A join (B where condition-on-B)) e.g. SP.P# = 'p2'
    - **Second class**

      e.g. from S, SP ==> S join SP

      §15.5 Implementing the Join Operators

      How to implement join operation efficiently?
  - “Improvement"

    may not be an "optimal" version.

Query Processing in the DBMS

Query in SQL:

SELECT CUSTOMER. NAME
FROM CUSTOMER, INVOICE
WHERE REGION = 'N.Y.' AND
AMOUNT > 10000 AND
CUTOMER.C#=INVOICE.C#

Internal Form :

$P(\sigma (S \bowtie SP)$

Operator :

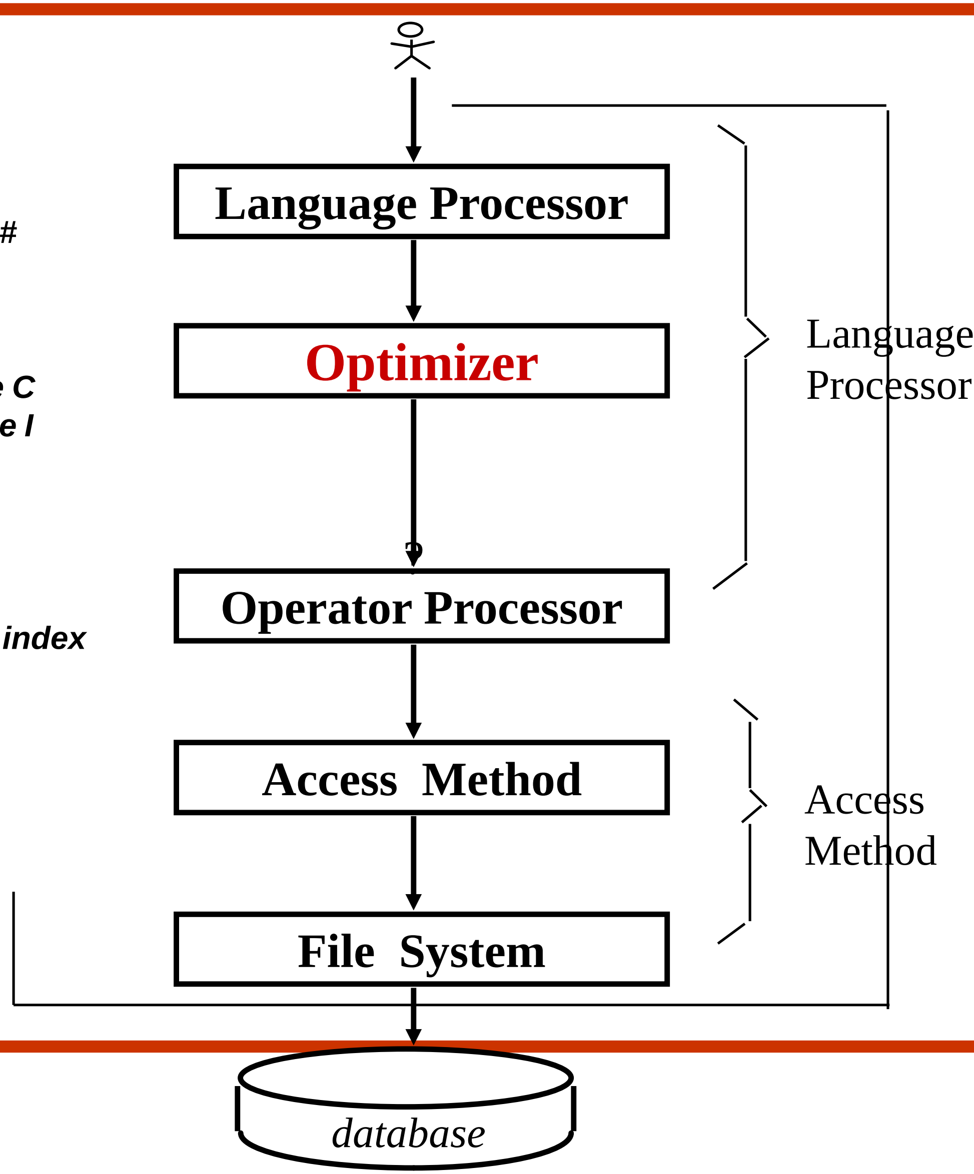
SCAN C using region index, create C
SCAN I using amount index, create I
SORT C?and I?on C#
JOIN C?and I?on C#
EXTRACT name field

Calls to Access Method:

OPEN SCAN on C with region index
GET next tuple
.
.

Calls to file system:

GET10th to 25th bytes from
block #6 of file #5



An Example

Suppose: |S| = 100,

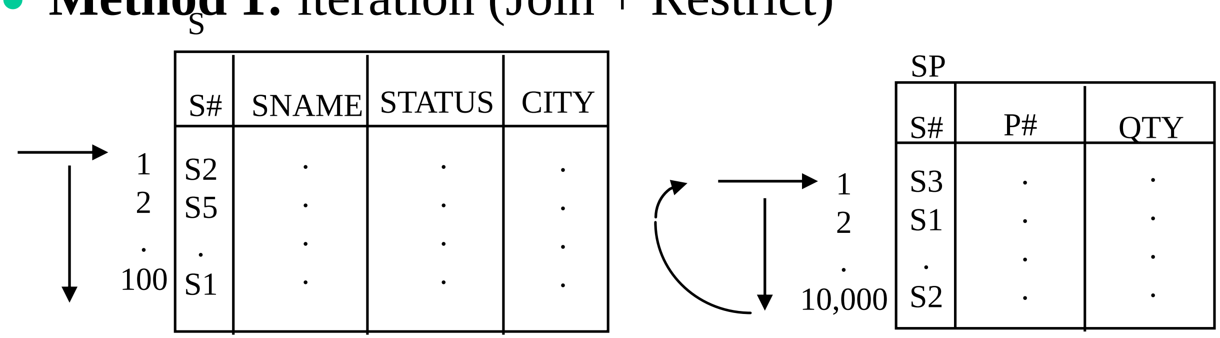
|SP| = 10,000, and there are 50 tuples in SP with p# = 'p2'?

Results are placed in Main Memory.

Query in SQL:

```
SELECT  S.*
FROM    S,SP
WHERE  S.S# = SP.S#  AND SP.P# = 'p2‘
```

- **Method 1:** iteration (Join + Restrict)



Cost = 100 * 10,000 = 1,000,000 tuple I/O's

An Example (cont.)

- **Method 2:** Restriction → iteration Join

SP

	S#	P#	QTY
1	S3	P4	.
2	S1	P2	.
.	.	.	.
10000	S2	P2	.

restrict ⇒ p#= 'p2'

SP'

	S#	P#	QTY
1	S1	P2	.
2	S3	P2	.
.	.	.	.
50	S2	P2	.

S

	S#	SNAME	STATUS	CITY
1	S2	.	.	.
2	S5	.	.	.
.	.	.	.	.
100	S1	.	.	.

SP'

	S#	P#	QTY
1	S1	P2	.
2	S3	P2	.
.	.	.	.
50	S2	P2	.

cost = 10,000 + 100 * 50 = 15,000 I/O

An Example (cont.)

- **Method 3:** Sort-Merge Join + Restrict

  Suppose S, SP are sorted on S#.

S

	S#	SNAME	STATUS	CITY
1	S1	.	.	.
2	S2	.	.	.
.	.	.	.	.
.	.			
.	.			
100	S100	.	.	.

SP

	S#	P#	QTY
1	S1	.	.
2	S1	.	.
.	.	.	.
.	.		
.	.		
10,000	S100	.	.

cost = 100 + 10,000 = 10,100 I/O

15.2 The Optimization Process: An Overview

(1) Query => internal form

(2) Internal form => efficient form

(3) Choose candidate low-level procedures

(4) Generate query plans and choose the cheapest one

Step 1: Cast the query into some internal representation

Query: "get names of suppliers who supply part p2"

SQL: **select distinct** S.SNAME
from S,SP
where S.S# = SP.S# **and** SP.P# = 'p2'

Query tree:

result
|
project (SNAME)
|
restrict (SP.P# = 'p2')
|
join (S.S# = SP.S#)
/ \
S SP

Algebra:

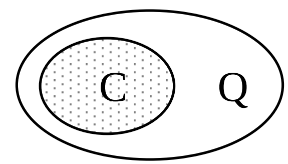
$((S \text{ join}_{S.S\# = SP.S\#} SP) \text{ where } P\#= \text{'P2'}) [SNAME]$ or $\pi_{SNAME}(\sigma_{\text{'P2'}}(S \bowtie SP))$

Step 2: Convert to equivalent and efficient form

- Def: Canonical Form

  Given a set Q of queries, for q1, q2 belong to Q, q1 are equivalent to q2 (q1 ≡ q2) iff they produce the same result, Subset C of Q is said to be a set of canonical forms for Q iff

  $$\forall q \in Q \quad \exists! \quad c \in C \quad \ni \quad q \equiv c$$



- Note: Sufficient to study the small set C
- Transformation Rules

Step2

output of step1 → trans. → equivalent and more efficient form

Algebra → **Efficient Algebra**

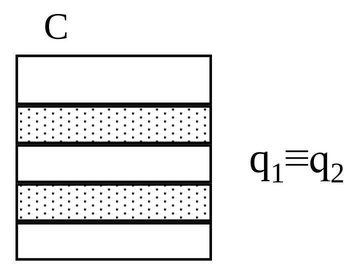
Step 2: Convert to equivalent and efficient form (cont.)

e.g.1 [restriction first]

(A join B) where restriction_B q_1

⇩

A join (B where restriction_B) q_2



e.g.2 [More general case]

(A join B) where restriction_A and restriction_B

⇩

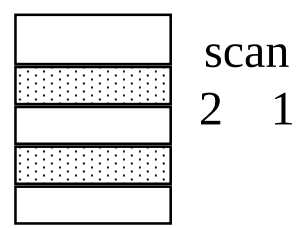
(A where rest_on_A) join (B where rest_on_B)

e.g.3 [Combine restriction]

(A where rest_1) where rest_2

⇩

A where rest_1 and rest_2



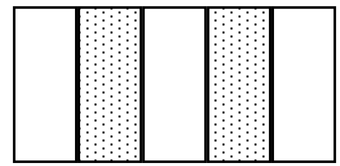
Step 2: Convert to equivalent and efficient form (cont.)

e.g.4 [projection] last attribute

(A [attribute_list_1]) [attri_2]

⇩

A [attri_2]

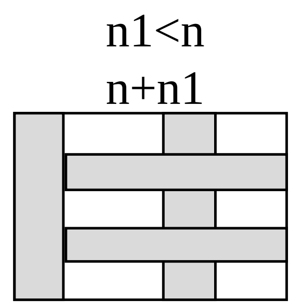


e.g.5 [restriction first]

(A [attri_1]) where rest_1

⇩

(A where rest _1) [attri_1]



.
.
.
.

Step 2: Convert to equivalent and efficient form (cont.)

e.g.6 [Introduce extra restriction]

SP JOIN (P WHERE P.P#= 'P2')
sp.p# = p.p#

⇩ if restriction on join attribute

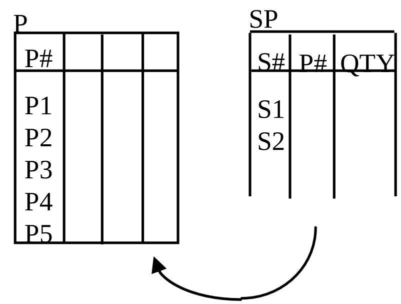
(SP WHERE SP.P# = 'P2') JOIN (P WHERE P.P# = 'P2')

e.g.7 [Semantic transformation]

(SP join P) [S#]
sp.p# = p.p#

⇩ if SP.P# is a foreign key matching
the primary term P.P#

SP[S#]



Note: a very significant improvement.

Ref.[17.27] P.571 J. J. King, VLDB81

Step 3: Choose candidate low-level procedures

- **Low-level procedure**
  - e.g. Join, restriction are low-level operators
  - there will be a set of procedures for implementing each operator,

    e.g. Join (ref p.11-31)

    <1> Nested Loop (a brute force)

    <2> Index lookup (if one relation is indexed on join attribute)

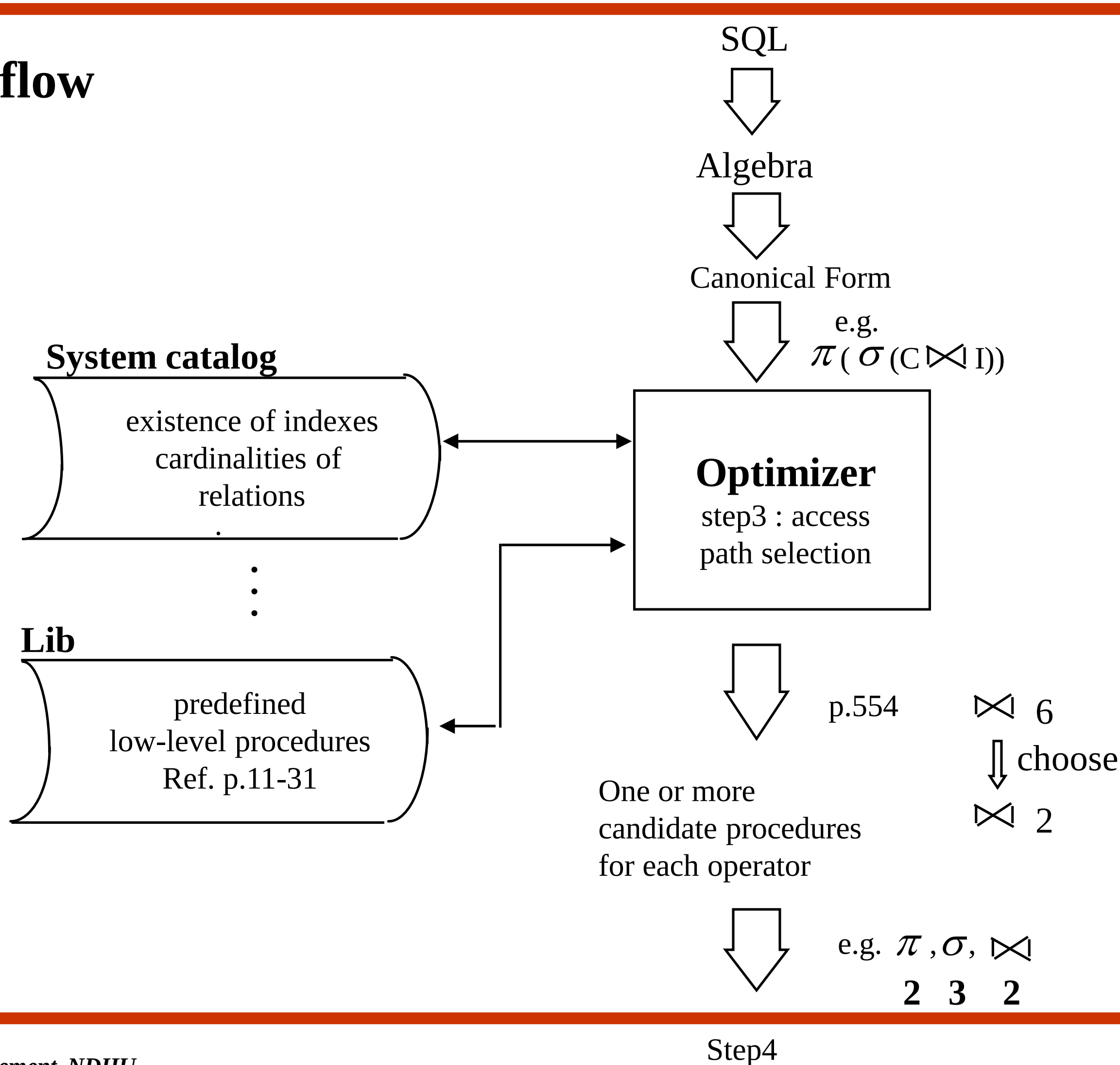
    <3> Hash lookup (if one relation is hashed by join attribute)

    <4> Merge (if both relations are indexed on join attribute)

⋮

Step 3: Choose candidate low-level procedures (cont.)

- **Data flow**



Step 4: Generate query plans and choose the cheapest

- **Query plan**
  - is built by combing together a set of candidate implementation procedures
  - for any given query

⇩

many many reasonable plans

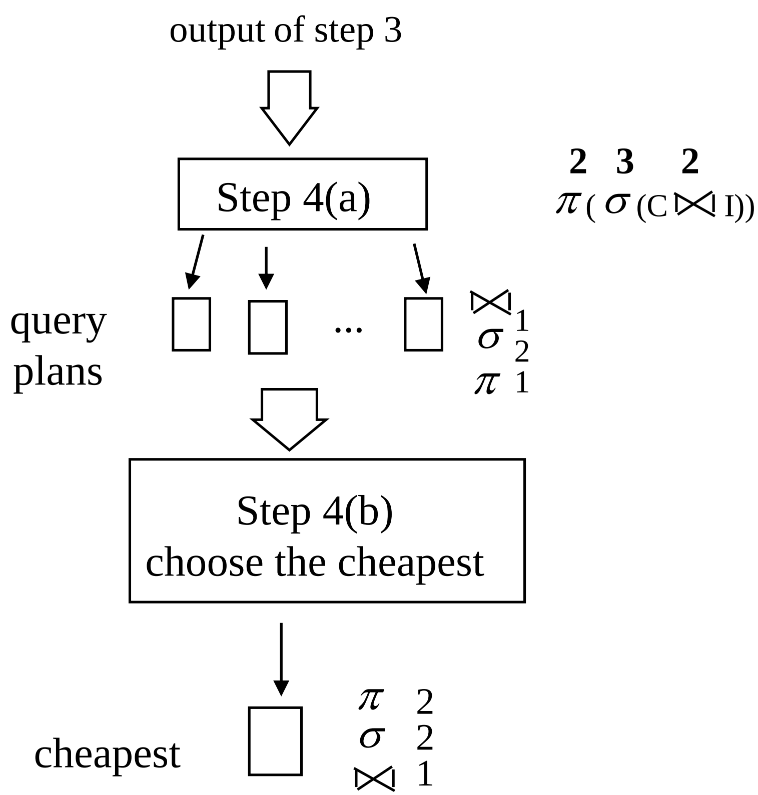
Note: may not be a good idea to generate all possible plans.

⇩

heuristic technique "keep the set within bound"
(reducing the search space)

Step 4: Generate query plans and choose the cheapest (cont.)

- **Data flow**



Step 4: Generate query plans and choose the cheapest (cont.)

- **Choosing the cheapest**
  - require a method for assigning a <u>cost</u> to any given <u>plan</u>.
  - factor of cost formula:

    (1) # of disk I/O

    (2) CPU utilization

    (3) size of intermediate results

    ⋮
  - a difficult problem [Jarke 84, 17.3. p.564 ACM computing surveys] [Yao 79, 17.8 TODS]

15.3 Optimization in System R

Optimization in System R

- Only minor changes to DB2 and SQL/DS.
- Query in System R (SQL) is a set of "select-from-where" block
- System R optimizer

  step1: choosing block order first

  in case of nested => innermost block first

  step2: optimizing individual blocks

  Note: certain possible query plan will never be considered.
- The statistical information for optimizer

  Where: from the system catalog

  What: 1. # of tuples on each relation

  2. # of pages occupied by each relation.

  3. percentage of pages occupied by each relation.

  4. # of distinct data values for each index.

  5. # of pages occupied by each index.

  ⋮

  Note: not updated every time the database is updated. (overhead??)

Optimization in System R (cont.)

Given a query block

case 1. involves just a restriction and/or projection

1. statistical information (in catalog)
2. formulas for size estimates of intermediate results.
3. formulas for cost of low-level operations (next section)

⇓

choose a strategy for constructing the query operation.

case 2. involves one or more join operations

e.g. A join B join C join D

⇓

((A join B) join C) join D

Never: (A join B) join (C join D)

Why? See next page

Optimization in System R (cont.)

((A join B) join C) join D

Never: (A join B) join (C join D)

Note:

1. "reducing the search space"
2. heuristics for choosing the sequence of joins are given in [17.34] P.573
3. (A join B) join C

   not necessary to compute entirely before join C

   i.e. if any tuple has been produced

pass to join **C**

It may never be necessary to finish relation "A $\bowtie$ B ", **why ?**

∵ C has run out ??

Optimization in System R (cont.)

- How to determine the order of join in System R ?
  - consider only sequential execution of multiple join.

    <e.g.> $((A \bowtie B) \bowtie C) \bowtie D$

    $(A \bowtie B) \bowtie (C \bowtie D)$ $\times$

STEP1: Generate all possible sequences

<e.g.>

(1) $((A \bowtie B) \bowtie C) \bowtie D$

(2) $((A \bowtie B) \bowtie D) \bowtie C$

(3) $((A \bowtie C) \bowtie B) \bowtie D$

(4) $((A \bowtie C) \bowtie D) \bowtie B$

(5) $((A \bowtie D) \bowtie B) \bowtie C$

(6) $((A \bowtie D) \bowtie C) \bowtie B$

(7) $((B \bowtie C) \bowtie A) \bowtie D$

(8) $((B \bowtie C) \bowtie D) \bowtie A$

(9) $((B \bowtie D) \bowtie A) \bowtie C$

(10) $((B \bowtie D) \bowtie C) \bowtie A$

(11) $((C \bowtie D) \bowtie A) \bowtie B$

(12) $((C \bowtie D) \bowtie B) \bowtie A$

Total # of sequences = (4!)/ 2 = 12

Optimization in System R (cont.)

STEP 2: Eliminate those sequences that involve Cartesian Product

- if A and B have no attribute names in common, then

$$A \bowtie B = A \times B$$

STEP 3: For the remainder, estimate the cost and choose a cheapest.

15.4 Optimization in INGRES

Query Decomposition

- a general idea for processing queries in INGRES.
- basic idea: break a query involving multiple tuple variables down into a sequence of smaller queries involving one such variable each, using **detachment** and tuple **substitution**.
  - avoid to build Cartesian Product.
  - keep the # of tuple to be scanned to a minimum.

<e.g> "Get names of London suppliers who supply some red part weighing less than 25 pounds in a quantity greater than 200"

Initial query:

```
Q0: RETRIEVE (S.SNAME) WHERE   S.CITY= 'London'
                      AND      S.S# = SP.S#
                      AND      SP.QTY > 200
                      AND      SP.P# = P.P#
                      AND      P.COLOR = Red
                      AND      P.WEIGHT < 2 5
```

⇩ detach P

Query Decomposition (cont.)

D1: RETRIEVE INTO P' (P.P#) WHERE P.COLOR= 'Red'
AND P.WEIGHT < 25

Q1: RETRIVE (S.SNAME) WHERE S.CITY = 'London'
AND S.S# = SP.S#
AND SP.QTY > 200
AND SP.P# = P'.P#

S join SP join P'

⇓ detach SP

D2: RETRIEVE INTO SP' (SP.S#, SP.P#)
WHERE SP.QTY > 200

Q2: RETRIEVE (S.SNAME) WHERE S.CITY = 'London'
AND S.S#=SP'.S#
AND SP'.P#=P'.P#

⇓ detach S

Query Decomposition (cont.)

D3: RETRIEVE INTO S' (S.S#, S.SNAME)
WHERE S.CITY = 'LONDON'
Q3: RETRIEVE (S'.SNAME) WHERE S'.S# =SP'.S# AND SP'.P# = P'.P#

⇓ detach P' and SP'

D4: RETRIEVE INTO SP''(SP'.S#)
WHERE SP'.P# =P'.P#
Q4: RETRIEVE (S'.SNAME) WHERE S'.S# = SP''.S#

⇓ D4: two var. --> tuple substitution
(Suppose D1 evaluate to {P1, P3 }

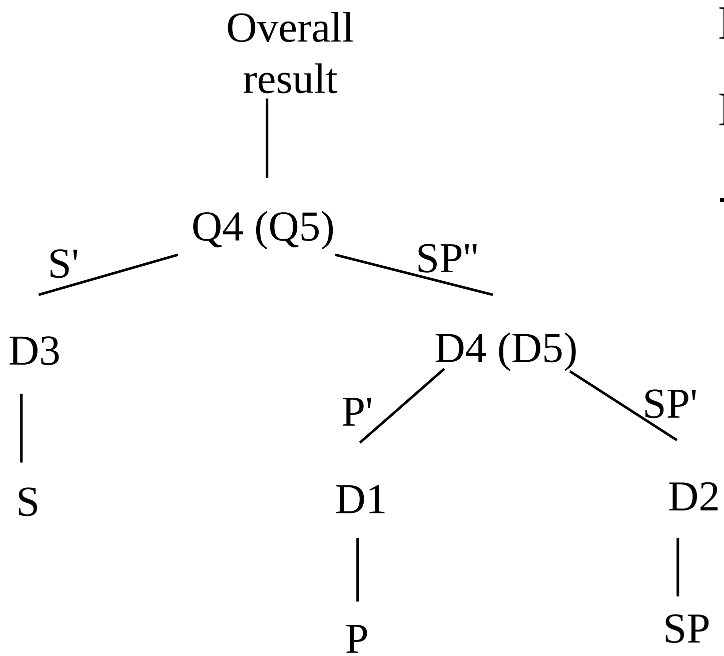
D5: RETRIEVE INTO SP''(SP'.S#)
WHERE SP'.P# = 'P1'
OR SP'.P#= 'P3'

⇓ Q4 : two var. --> tuple substitution
(Suppose D5 evaluate to { S1, S2, S4})

Q5: RETRIEVE (S'.SNAME) WHERE S'.S# = 'S1'
OR S'.S# = 'S2'
OR S'.S# = 'S4'

Query Decomposition (cont.)

- Decomposition tree for query Q_0:



D1, D2, D3: queries involve only one variable => evaluate

D4, Q4: queries involve tow variable => tuple substitution

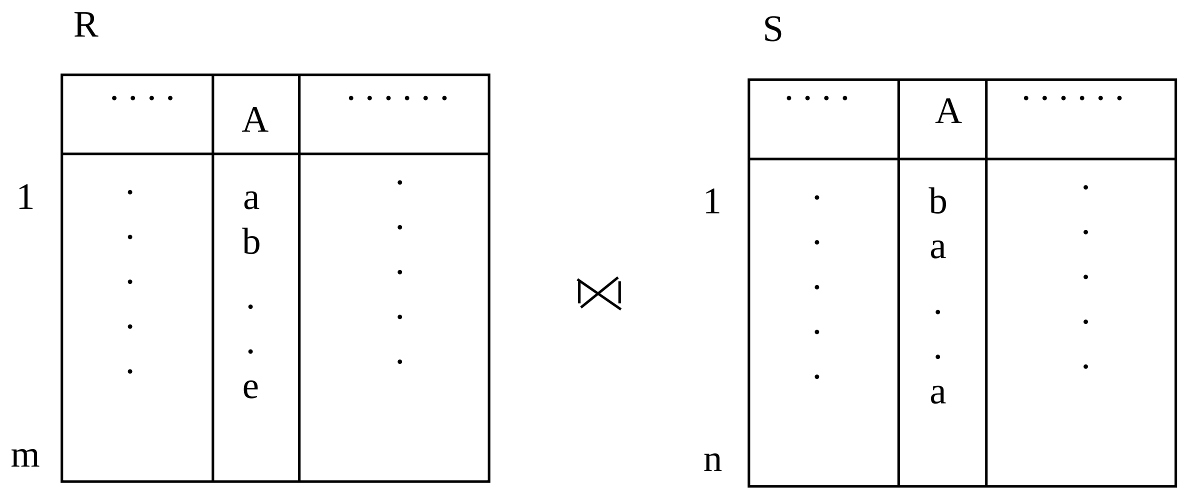
- **Objectives :**
  - **avoid to build Cartesian Product.**
  - **keep the # of tuple to be scanned to a minimum.**

15.5 Implementing the Join Operators

- Method 1: Nested Loop
- Method 2: Index Lookup
- Method 3: Hash Lookup
- Method 4: Merge

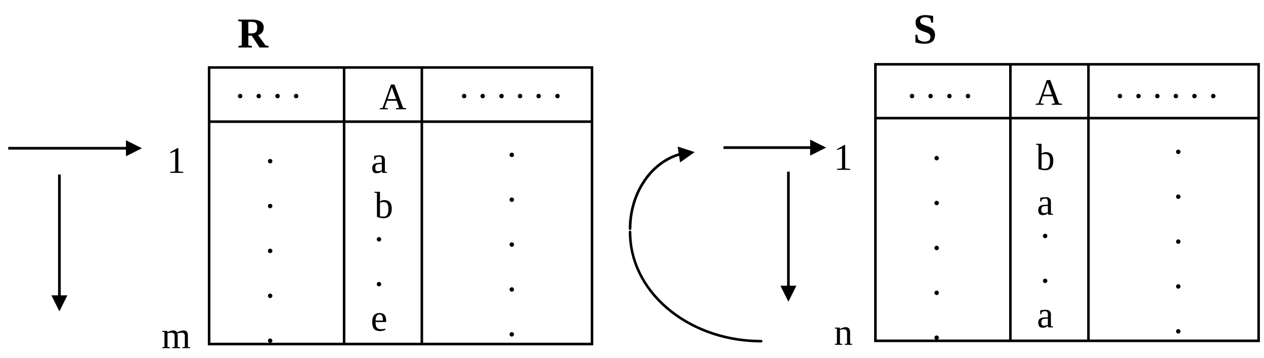
Join Operation

- Suppose R ⋈ S is required, R.A and S.A are join attributes.



Method 1: Nested Loop

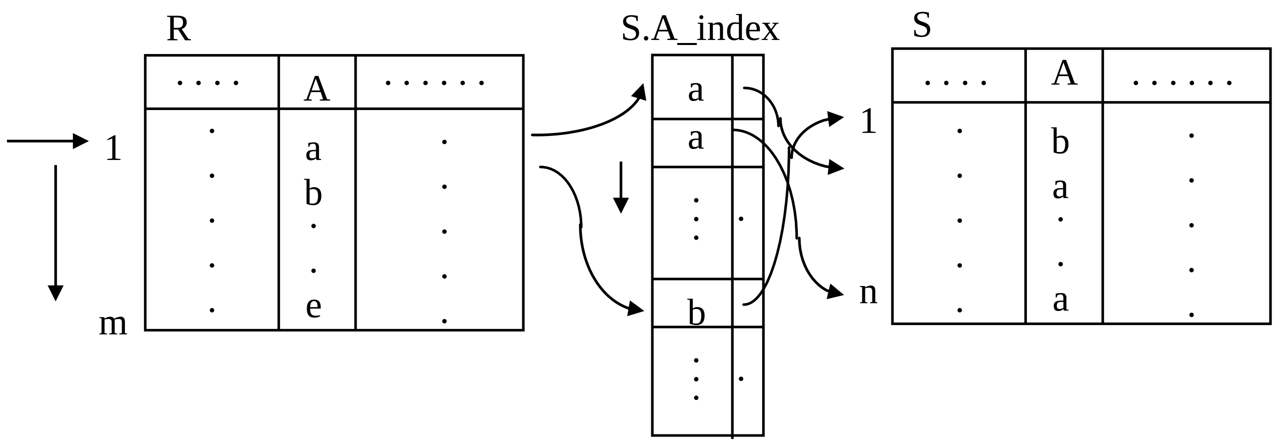
- Suppose R and S are not sorted on A.



- O (mn)
- the worst case
- assume that S is neither indexed nor hashed on A
- will usually be improved by constructing index or hash on S.A dynamically and then proceeding with an index or hash lookup scan.

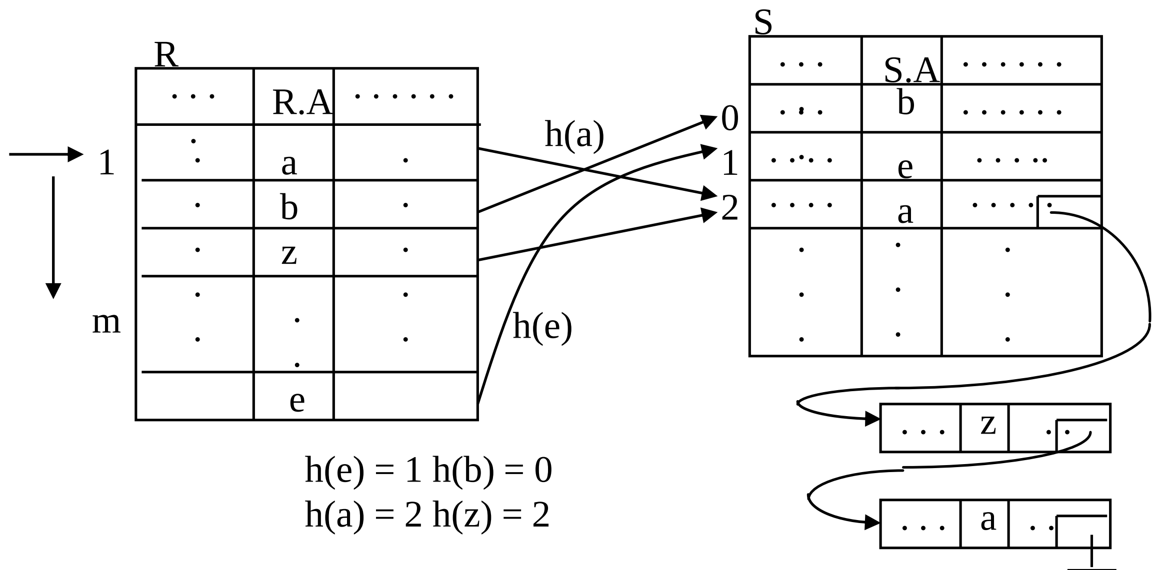
Method 2: Index Lookup

- Suppose S in indexed on A



Method 3: Hash Lookup

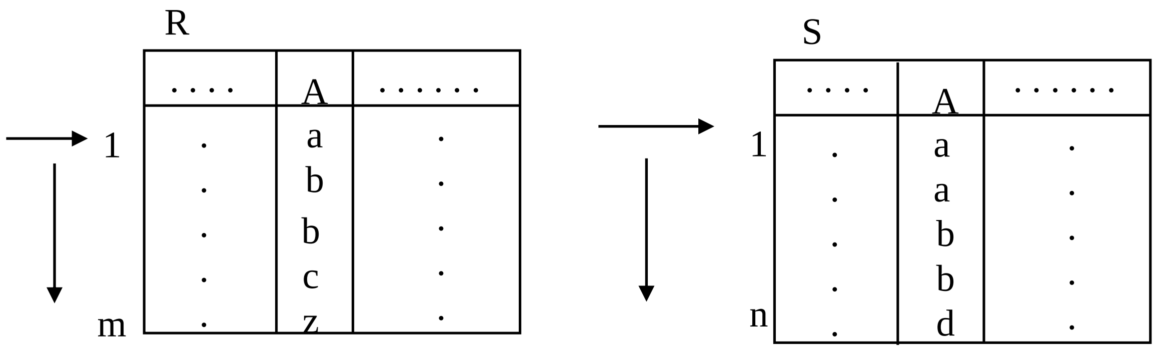
- Suppose S is hashed on A.

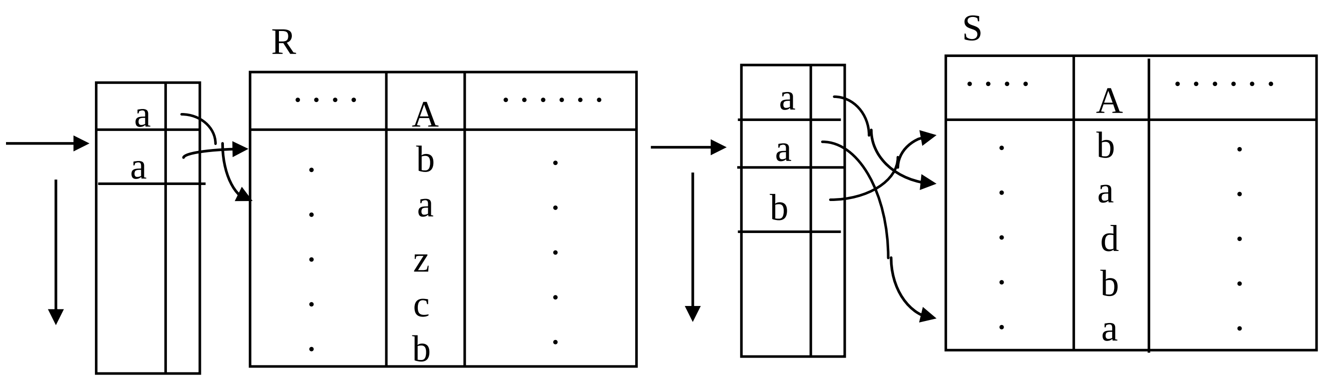


-Calculate hash function is faster than search in index.

Method 4: Merge

- Suppose R and S are both sorted (for indexed) on A.





– Only index is retrieved for any unmatched tuple.

end of unit 15