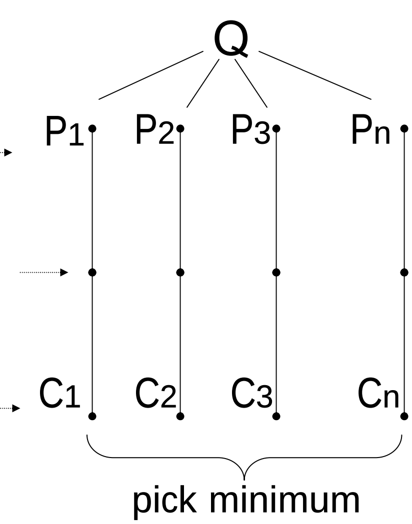
Query Optimization

It is safer to accept any chance that offers itself, and extemporize a procedure to fit it, than to get a good plan matured, and wait for a chance of using it.

Thomas Hardy (1874)
in *Far from the Madding Crowd*

Query optimization

- Generate query plans
- Estimate size of intermediate results
- Estimate cost of plan ($,time,...)



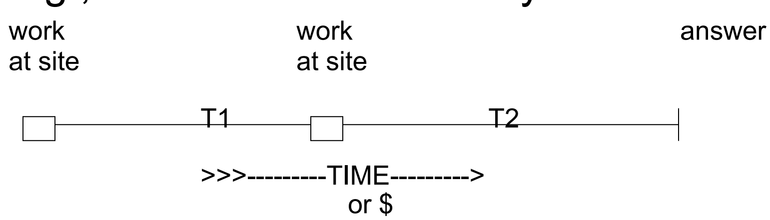
Query Optimization

- Query: R1 $\bowtie$ R2 $\bowtie$ R3
- R1@Site1, R2@Site2, R3@Site3
- Result@Site1
- Possible plans
  - Plan 1: Send R2 and R3 to Site1. Perform query at Site 1
  - Plan 2: Send R3 to Site2; Evaluate I = R2 $\bowtie$ R3; send I to Site1; Evaluate result = I $\bowtie$ R1
  - Many other plans …(including types of joins, number of sites, semijoins, etc)

Cost estimation

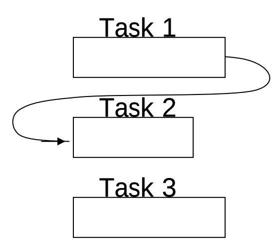
- ☛ As in centralized system: estimate result sizes
- ☛ But: # IOs may not be best metric

e.g., Transmission time may dominate



Another reason why plain IOs not enough: Parallelism

- Cost metrics
  - IOs, Bytes transmitted, $, ...
  - Can add together
- Response time metric
  - cannot add
  - need scheduling and dependency info
  - skew important

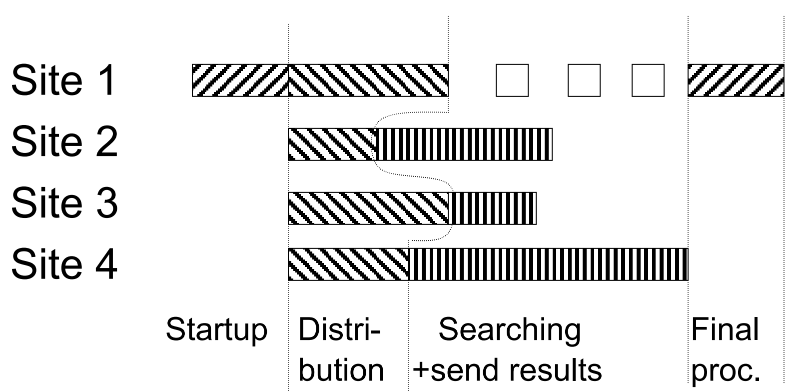


⇔ Take into account:

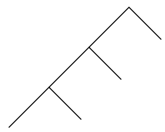
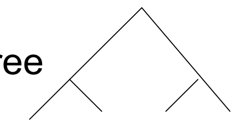
(in parallel/distributed system)

- Start up costs (for parallel operation)
- Data distribution costs/time
- Contention
  - memory, disk, network,...
- Assembling result

Example: Response time



Query Optimizer

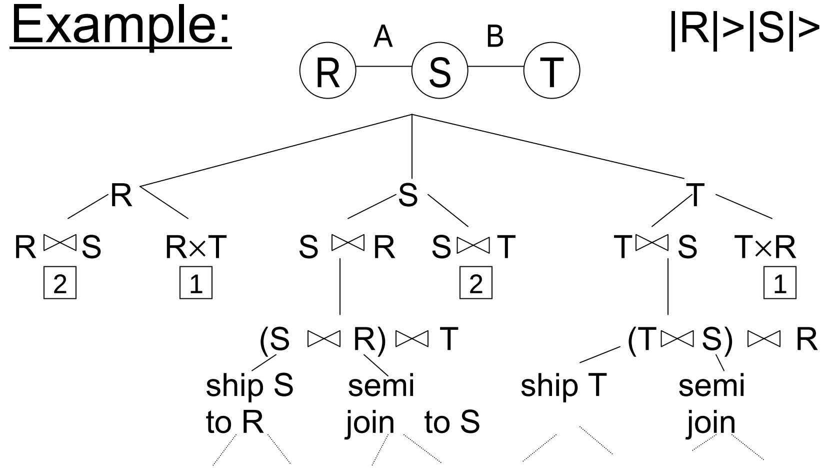
- Cost model
- Plan space
  - Deep tree  vs bushy tree 
- Enumeration/Search strategy
  - Exhaustive (with pruning)
  - Hill climbing (greedy)
  - Query separation
  - SDD-1 (semi-join based)

(1) <u>Exhaustive</u>

- consider “all” query plans
  with a set of techniques
- prune some plans (heuristics)

Example:

|R|>|S|>|T|



Heuristics:

$\boxed{1}$ Prune because cross-product not necessary

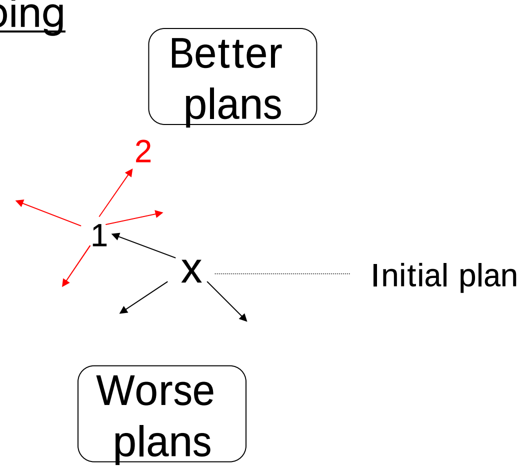
$\boxed{2}$ Prune because larger relation first

☒ In generating plans, keep <u>goal</u> in mind:

e.g.: Goal is parallelism in system with fast net, consider partitioning relation(s) first

e.g.: Goal is reduction of net traffic, consider semi-joins

(2) Hill climbing



Hill Climbing Algorithm

Step 1: Do initial processing

Step 2: Select initial feasible solution (P_0)

- 2.1 Determine the candidate result sites - sites where a relation referenced in the query exist
- 2.2 Compute the cost of transferring all the other referenced relations to each candidate site
- 2.3 P_0 = candidate site with minimum cost

Step 3: Determine candidate splits of P_0 into

$P_1 = \{P_{1a}, P_{1b}\}$

- 3.1 P_{1a} consists of sending one of the relations to the other relation's site
- 3.2 P_{1b} consists of sending the join of the relations to the final result site

Hill Climbing Algorithm

Step 4: Replace P_0 with P_1 that gives

$$\mathit{cost}(P_{1a}) + \mathit{cost}(\text{local join}) + \mathit{cost}(P_{1b}) < \mathit{cost}(P_0)$$

Step 5: Recursively apply steps 3–4 on P_1 until no such plans can be found

Step 6: Check for redundant transmissions in the final plan and eliminate them.

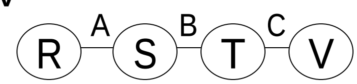
Example $R \bowtie S \bowtie \sigma(U) \bowtie V$

Rel	Site	Size
R	1	10
S	2	20
U	3	90
V	4	40

tuple size = 1

Goal: minimize data transmission

Step 1: R ⋈ S ⋈ T ⋈ V



tuple size = 1

Rel	Site	Size
R	1	10
S	2	20
T	3	30
V	4	40

T = σ(U)
Selectivity is 1/3

Step 2: Initial plan – send relations to one site

What site do we send all relations to?

To site 1: cost=20+30+40=90

To site 2: cost=10+30+40=80

To site 3: cost=10+20+40=70

To site 4: cost=10+20+30=60

P_0: R (1 $\rightarrow$ 4)

S (2 $\rightarrow$ 4)

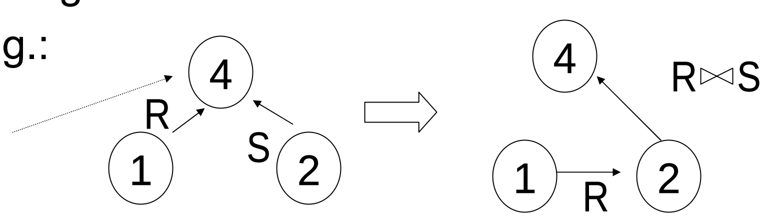
T (3 $\rightarrow$ 4)

Compute R $\bowtie$ S $\bowtie$ T $\bowtie$ V at site 4

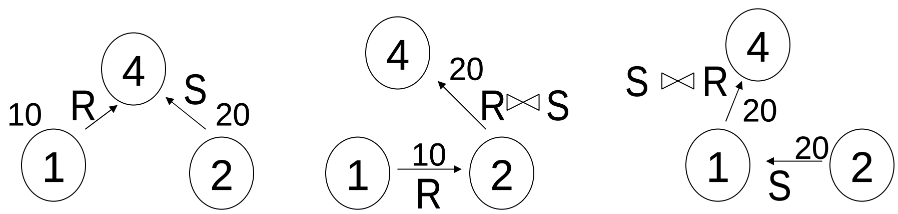
Steps 3 & 4

- Consider sending each relation to neighbor:

e.g.:



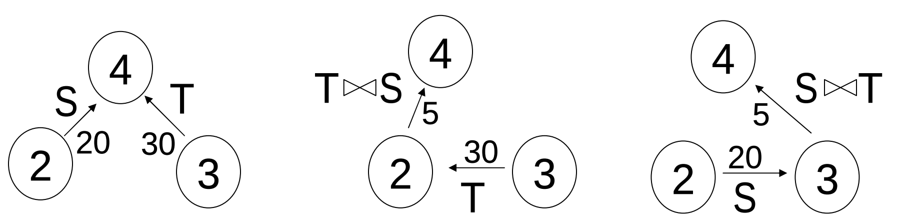
Assume: Size $R \bowtie S = 20$
$S \bowtie T = 5$
$T \bowtie V = 1$



cost = 30

cost = 30
No savings

cost = 40
Worse!



cost = 50

cost = 35
A Win!

cost = 25
A Bigger Win

P1: P1a: S (2 → 3)

α = S ⋈ T

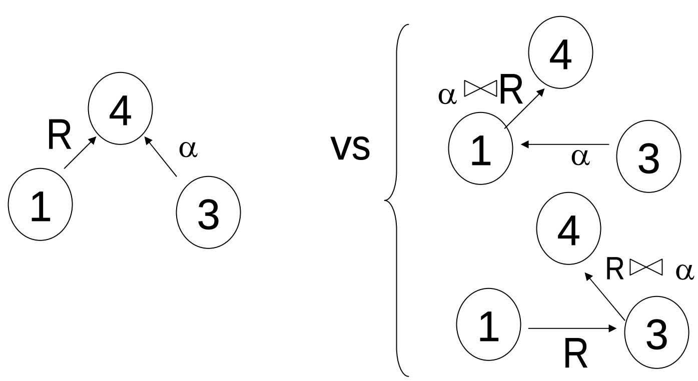
P1b: R (1 → 4)

α (3 → 4)

compute answer at site 4

Step 5: Repeat Steps 3 & 4

- Treat α = S ⋈ T as relation



Hill climbing may miss best plan!

<u>Example:</u> best plan could be:

P_B: T $(3 \rightarrow 4)$ ⊠30

$\beta = T \bowtie V$

β $(4 \rightarrow 2)$ ⊠1

$\beta' = \beta \bowtie S$

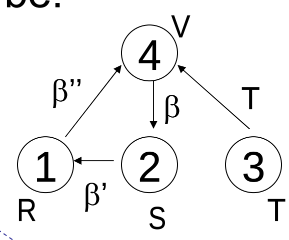
β' $(2 \rightarrow 1)$ ⊠1

$\beta'' = \beta' \bowtie R$

[optional] β'' $(1 \rightarrow 4)$ ⊠1

Compute answer

33 = total

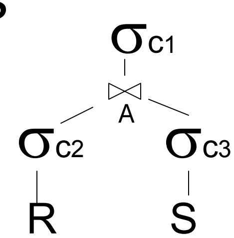


Costs could be low because is very selective

(3) <u>Query separation</u>

- separate query into 2 or more steps
- optimize each step independently

Example: simple queries



1. Compute $R' = \Pi_A[\sigma_{C2} R]$

   $S' = \Pi_A[\sigma_{C3} S]$
2. Compute $J = R' \bowtie S'$
3. Compute

$$Ans = \sigma_{C1}\{[J \bowtie \sigma_{C2} R] \bowtie [J \bowtie \sigma_{C3} S]\}$$

In other words:

(a) Compute A values in answer
(steps 1,2)

(b) Get tuples from sites with matching A values and compute answer
(step 3)

Simple query

- Relations have a single attribute
- Output has a single attribute
  e.g., $J \leftarrow R' \bowtie S'$

Idea

- Decompose query into
  - Local processing
  - Simple query (or queries)
  - Final processing
- Optimize simple query
- Philosophy
  - Hard part is distributed join
  - Do this part with only keys; get rest of data later
  - Simpler to optimize simple queries

SDD-1 Algorithm

Step 1: In the execution strategy (call it *ES*), include all the local processing

Step 2: Reflect the effects of local processing on the database profile

Step 3: Construct a set of beneficial semijoin operations (*BS*) as follows :

$BS = \emptyset$

For each semijoin SJ_i

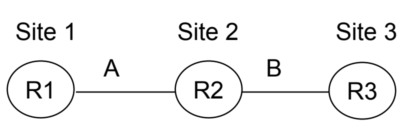
$BS \leftarrow BS \cup SJ_i$ if $cost(SJ_i) < benefit(SJ_i)$

SDD-1 Algorithm – Example

Consider the following query

SELECT R3.C
FROM R1, R2, R3
WHERE R1.A = R2.A
AND R2.B = R3.B

which has the following query graph and statistics:



relation	card	tuple size	relation size
R1	30	50	1500
R2	100	30	3000
R3	50	40	2000

attribute	$SF_{\ltimes}$	$size(\Pi_{attribute})$
R1.A	0.3	36
R2.A	0.8	320
R2.B	1.0	400
R3.B	0.4	80

SF: selectivity factor
Result = SF*R

SDD-1 Algorithm – Example

- Beneficial semijoins: Assume: $T_{MSG} = 0$
  - SJ_1 = R2 ⋉ R1, whose benefit is
    - 2100 = (1 – 0.3)*3000 and cost is 36
  - SJ_2 = R2 ⋉ R3, whose benefit is
    - 1800 = (1 – 0.4)*3000 and cost is 80
- Nonbeneficial semijoins:
  - SJ_3 = R1 ⋉ R2, whose benefit is
    - 300 = (1 – 0.8)*1500 and cost is 320
  - SJ_4 = R3 ⋉ R2, whose benefit is 0 and cost is 400

Cost = transfer semijoin attribute
= $T_{MSG} + T_{TR}$*(size)

Benefit = cost of transferring irrelevant tuples (avoided by the semijoin)
= (1-SF)*size*T_{TR}

SDD-1 Algorithm

Iterative Process

Step 4: Remove the most beneficial SJ_i from *BS* and append it to *ES*

Step 5: Modify the database profile accordingly

Step 6: Modify *BS* appropriately

- compute new benefit/cost values
- check if any new semijoin need to be included in *BS*

Step 7: If *BS* $\neq$ Ø, go back to Step 4.

SDD-1 Algorithm – Example

- Iteration 1:
  - Remove SJ_1 from *BS* and add it to *ES*.
  - Update statistics

    R2' = *size*(R2) = 900 (= 3000×0.3)

    size (R2'.A) = 320*0.3 = 96

    $SF_{\ltimes}$ (R2'.A) = ~0.8 × 0.3 = ~0.24
- Iteration 2:
  - Two beneficial semijoins:

    SJ_2 = R2' $\rtimes$ R3, whose benefit is 540 = (1–0.4) × 900 and cost is 80

    SJ_3 = R1 $\ltimes$ R2', whose benefit is **300=(1–0.8) × 1500** and cost is 96

    NOT 1140=(1-0.24)*1500!!
  - Add SJ_2 to *ES*
  - Update statistics

    size(R2') = 360 (= 900*0.4)

    Note: selectivity of R2' may also change, but not important in this example.

relation	card	tuple size	relation size
R1	30	50	1500
R2'	**30**	**30**	**900**
R3	50	40	2000

attribute	$SF_{\ltimes}$	$size(\Pi_{attribute})$
R1.A	0.3	36
R2'.A	**0.24**	**96**
R2'.B	**1.0**	**120**
R3.B	0.4	80

SDD-1 Algorithm – Example

- Iteration 3:
  - No new beneficial semijoins.
  - Remove remaining beneficial semijoin SJ_3 from *BS* and add it to *ES*.
  - Update statistics

    size(R1) = 1200 (= 1500 × 0.8)

    $SF_{\ltimes}$ (R1.A) = ~0.3 × 0.8 = 0.24

SDD-1 Algorithm

Assembly Site Selection

Step 8: Find the site where the largest amount of data resides and select it as the assembly site

Example:

Amount of data stored at sites:

Site 1: 1200

Site 2: 360

Site 3: 2000

Therefore, Site 3 will be chosen as the assembly site.

SDD-1 Algorithm

Postprocessing

Step 9: For each R_i at the assembly site, find the semijoins of the type $R_i \ltimes R_j$ where the total cost of *ES* without this semijoin is smaller than the cost with it and remove the semijoin from *ES*.

Step 10: Permute the order of semijoins if doing so would improve the total cost of *ES*.

– Example: Final strategy:
  Send (R2 $\ltimes$ R1) $\ltimes$ R3 to Site 3
  Send (R1 $\ltimes$ R2) to Site 3

Summary: Query Optimization

- Cost/result estimation
- Three key components in optimizer
  - Cost model, search space, enumeration algorithm
- In practice, avoid bad plans (rather than find optimal)
- Strategies
  - Exhaustive
  - Hill climbing
  - Separation
  - SDD-1 (semi-join based)