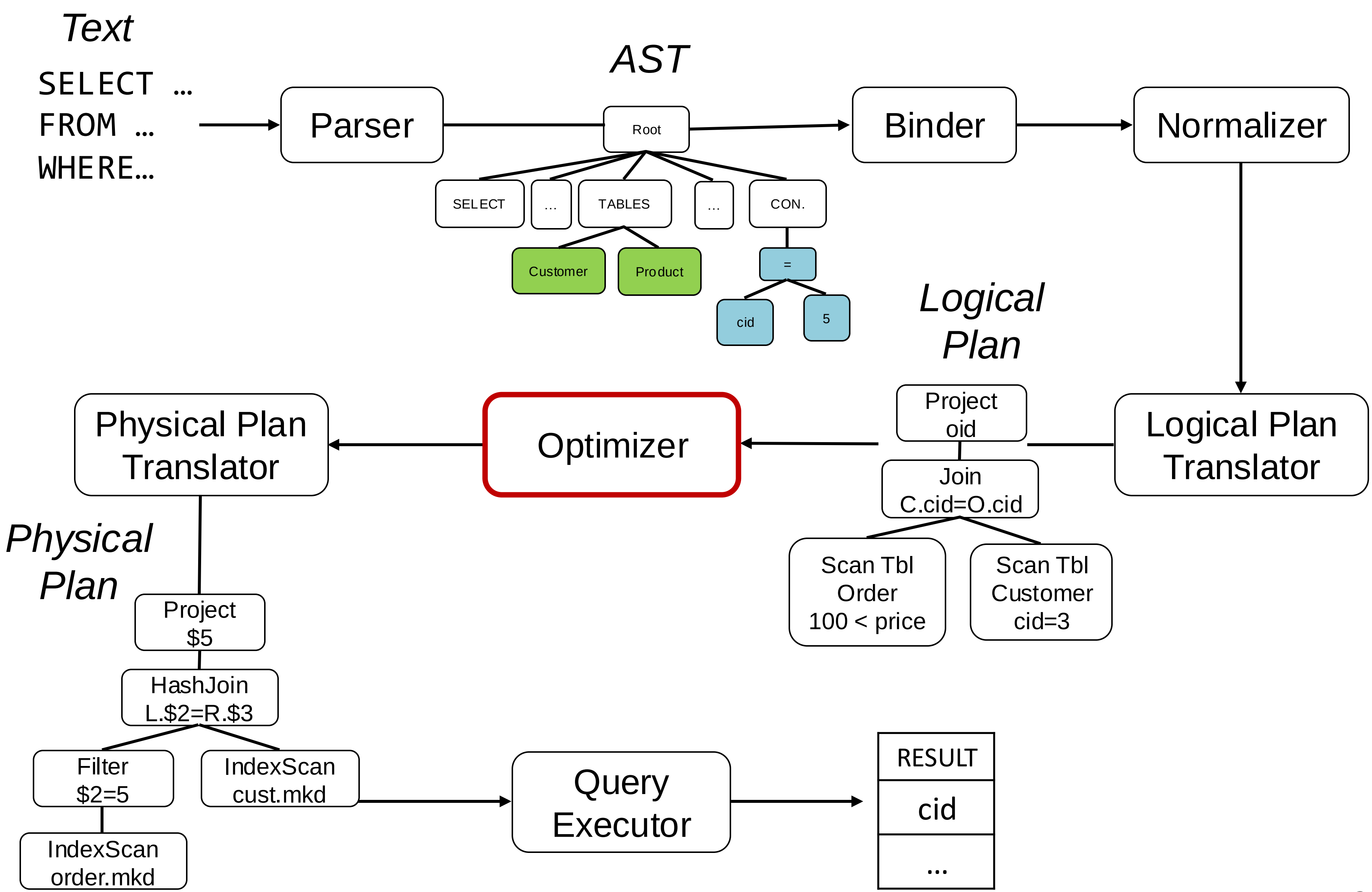
CS 348 Lectures 19-20

Query Optimization

Semih Salihoğlu

March 17-19 2025

Recall: Overview of Compilation Steps



Outline For Today

1. Goal of Query Optimization and Overview of Techniques
2. Cost-based Optimization Principles
3. Cost-based DP Logical Join Plan Optimizer
4. Cardinality Estimation Techniques
5. Rule-based Optimizations/Transformations
6. Final Remarks on Query Optimization & Query Processing

Outline For Today

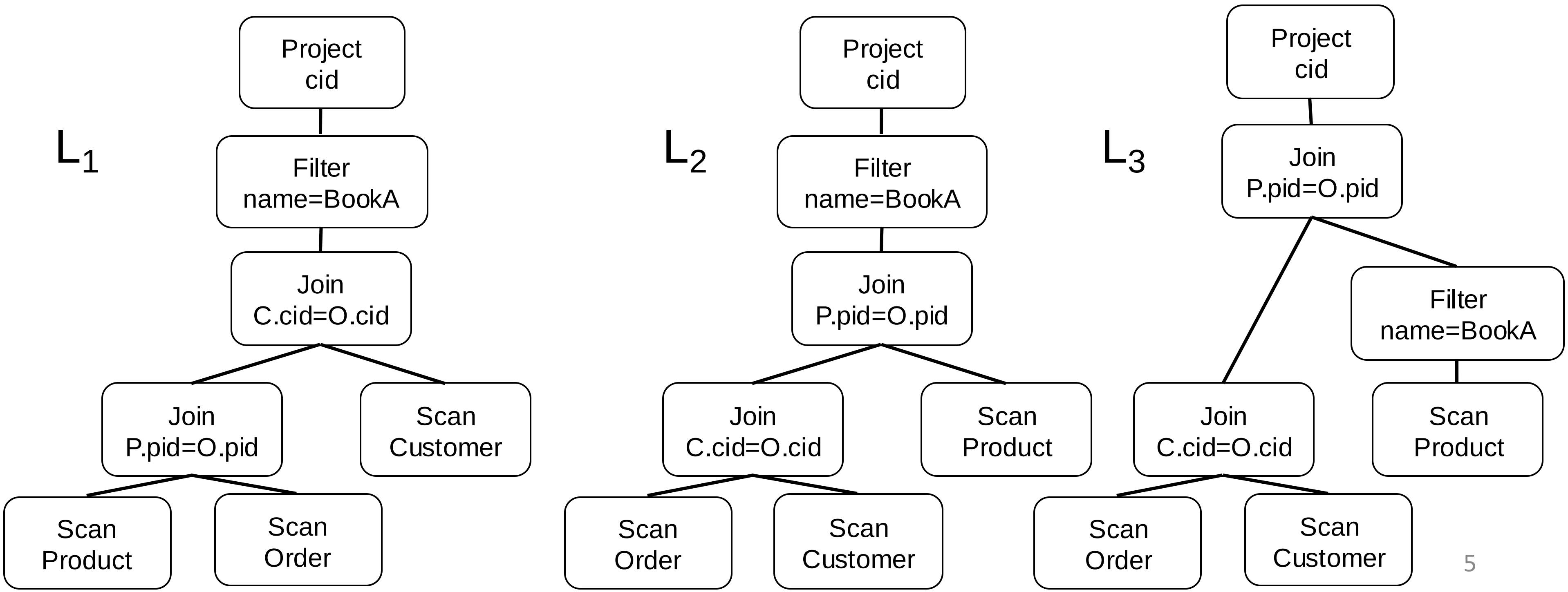
1. Goal of Query Optimization and Overview of Techniques
2. Cost-based Optimization Principles
3. Cost-based DP Logical Join Plan Optimizer
4. Cardinality Estimation Techniques
5. Rule-based Optimizations/Transformations
6. Final Remarks on Query Optimization & Query Processing

Goal of Query Optimization (1)

- Recall ultimately a *physical plan* executes to answer a query
- Given a query Q, many equivalent physical plans exist:
  1. Many equivalent logical plans exist

```
SELECT cid
FROM Customer C, Order O, Product P
WHERE C.cid = O.cid AND O.pid = P.pid
      AND P.name = BookA
```

Logical Plans:

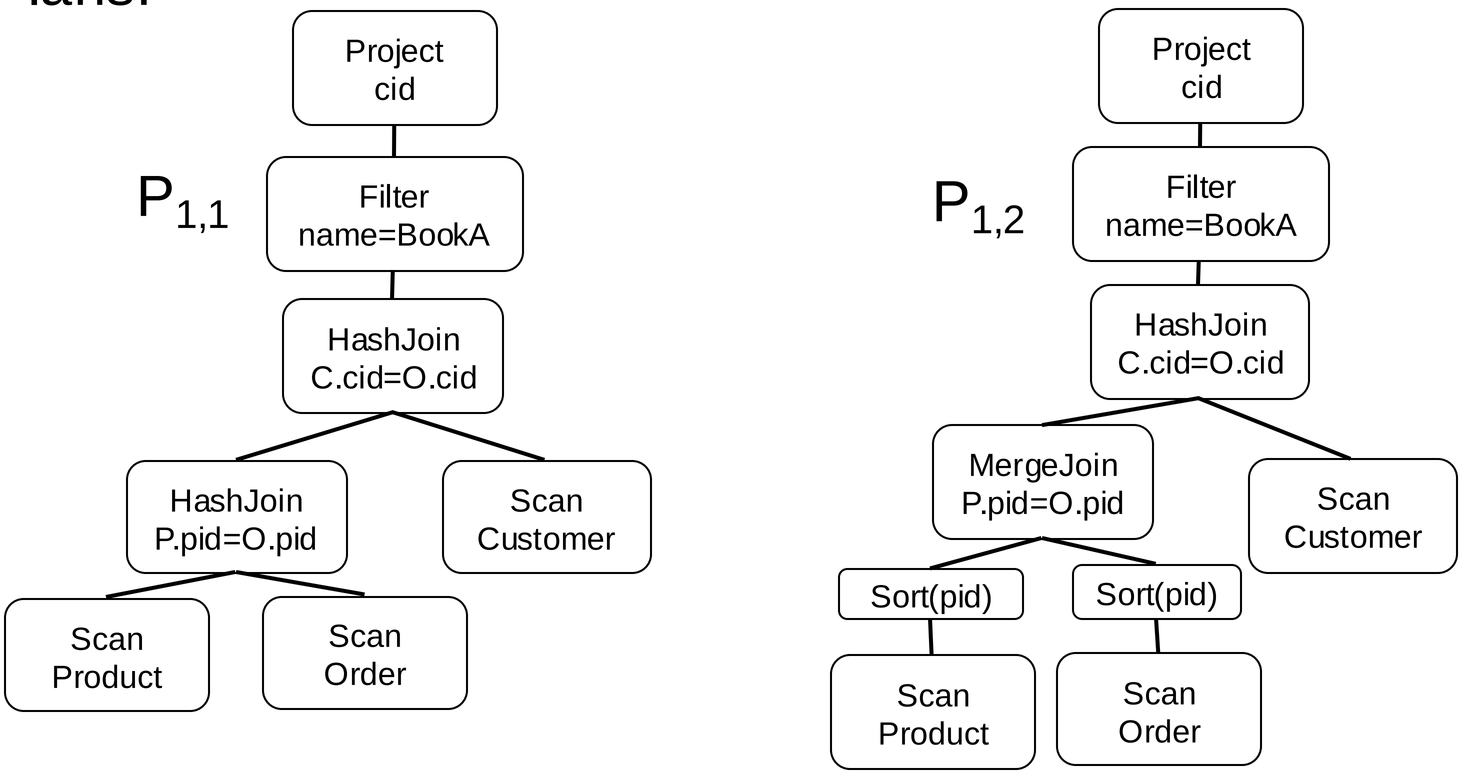


Goal of Query Optimization (1)

- Recall ultimately a *physical plan* executes to answer a query
- Given a query Q, many equivalent physical plans exist:
  1. Many equivalent logical plans exist
  2. Each logical plan can have many equivalent physical plans.

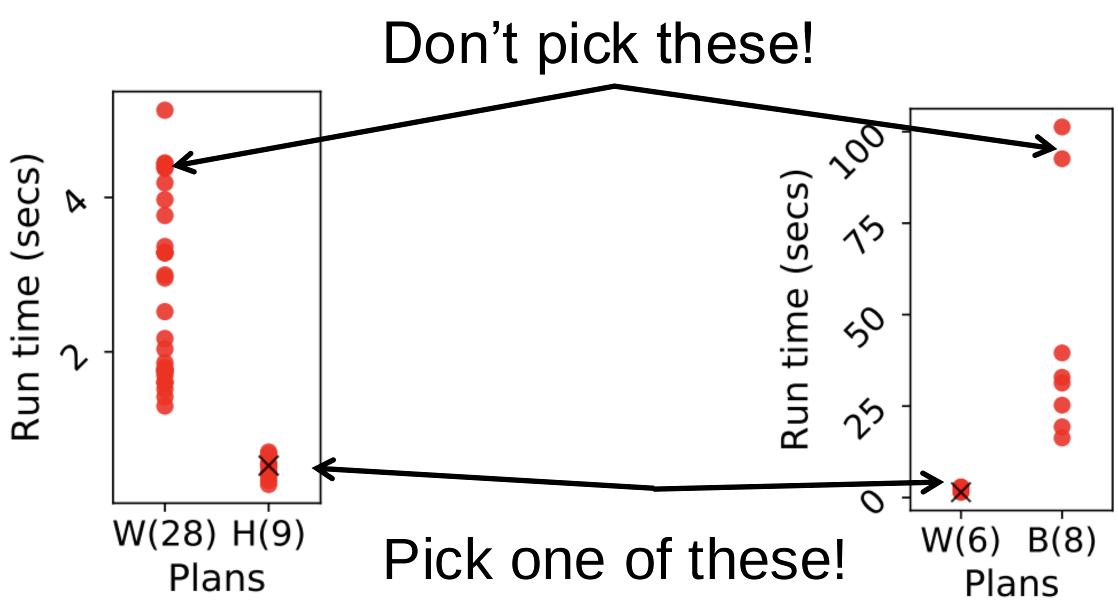
```
SELECT cid
FROM Customer C, Order O, Product P
WHERE C.cid = O.cid AND O.pid = P.pid
      AND P.name = BookA
```

Physical Plans:



Goal of Query Optimization (2)

- Ultimately: Given Q, pick the "best" physical plan for Q:
  - Best: often means fastest, could mean "cheapest"
- DBMS developers are more humble:
  - Pick a reasonably good plan. Do not pick a very bad plan!
  - Example plan spectrum of join-heavy queries



Mhedhbi, Salihoglu, Optimizing Subgraph Queries by Combining Binary and Worst-Case Optimal Joins, VLDB 2019

Overview of Query Opt. Techniques

1. Enumerate a logical plan space (often enumerates all join orders)

(extended) relational algebraic expressions [$L_1, L_2, \ldots, L_k$

2. For one or more of L_i, (optionally) enumerate a physical plan space:

$P_{i,1}, P_{i,2}, \ldots, P_{i,t}$

3. Pick the best $P_{i,1}$

Options for steps 1 & 2:

i. Rule-based

ii. Cost-based

iii. Hybrid rule/cost-based

A common approach:

- ➢ Step 1 is cost-based or hybrid
- ➢ Step 2 is rule-based

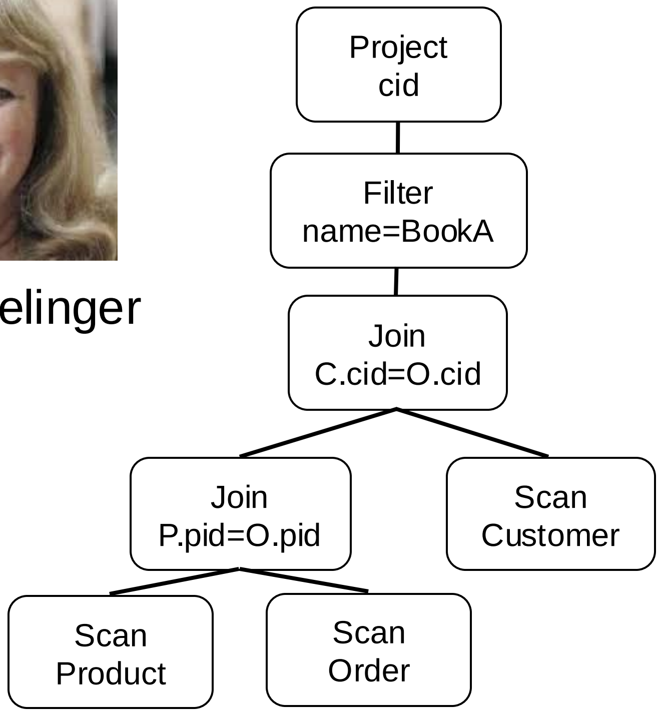
Outline For Today

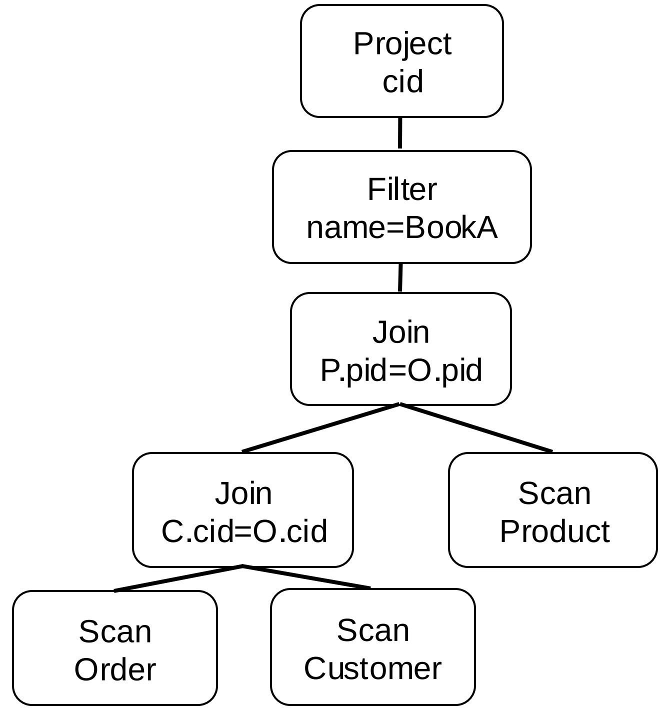
1. Goal of Query Optimization and Overview of Techniques
2. Cost-based Optimization Principles
3. Cost-based DP Logical Join Plan Optimizer
4. Cardinality Estimation Techniques
5. Rule-based Optimizations/Transformations
6. Final Remarks on Query Optimization & Query Processing

Cost-based Optimization Principles

➢ System R ('70s): First prototype relational DBMS (from IBM)

Patricia Selinger





➢ Give each enumerated log/phy plan, e.g., L_i, a $Cost(L_i) = c_i$

➢ Cost is the estimate of the system for how good/bad L_i is.

➢ Pick min cost plan

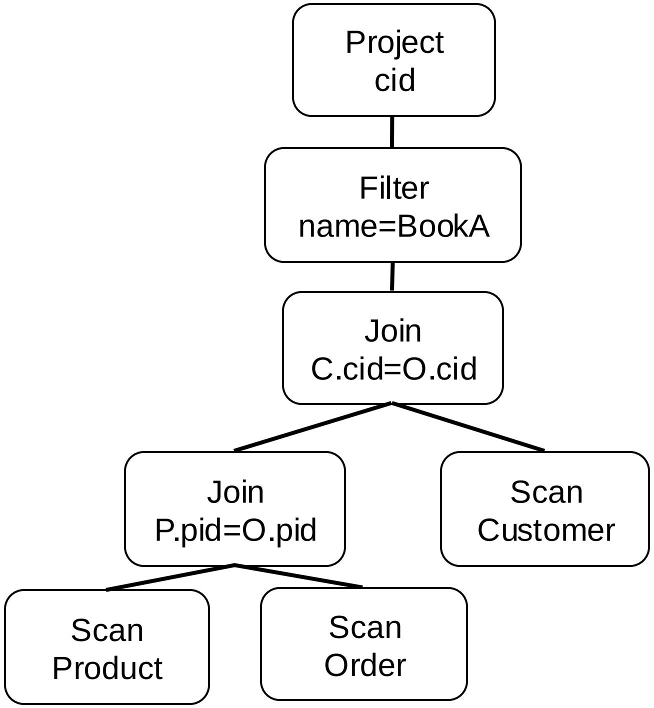
Access Path Selection in a Relational Database Management, Selinger et al. 1979, SIGMOD

Cost-based Optimization Principles (1)

- Naturally: cost definition is broken into costs of operators.

  i.e: $Cost(L_i) = \sum_j cost(o_j \in L_i)$

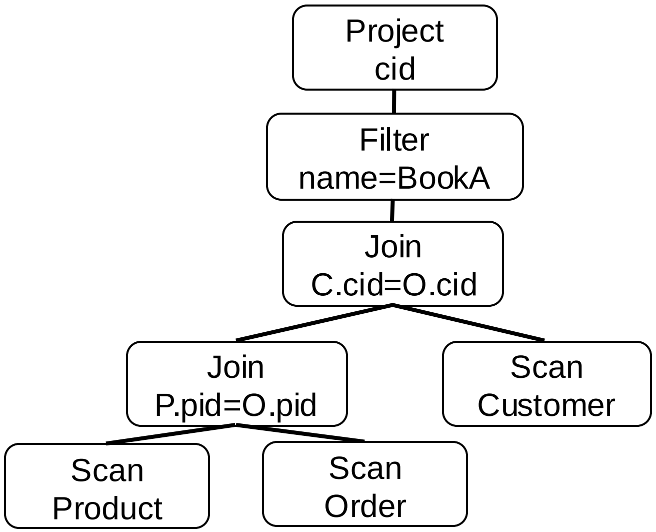
- Example cost metrics or components:
  - # I/Os a plan will make
  - # tuples that will pass through operators
  - # runtime of algorithm o_j is running
    - e.g., nested loop join of R, S: |R|*|S|
  - Combination of above



- For any reasonable metric:
  - Need to estimate cardinality, i.e., size, of tuples o_j will process
  - Cardinality estimation is a notoriously difficult problem

2 Components of Cost-based Optimization

1. What is the cost metric?
   - Can be complicated, e.g., different ops could have different costs
   - But inevitably depends on cardinality, i.e., number, of tuples processed by each operator
2. How do we estimate cardinality of tables processed by each op?
   - Need a “cardinality estimation technique to estimate cardinality

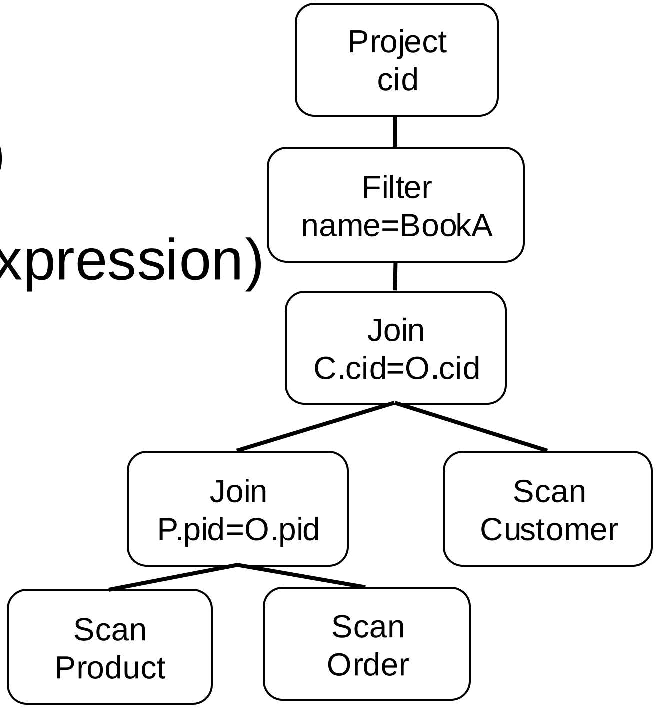


Cardinality Estimation

- Given a database
  1. D: $R_1(A_{1,1},\ldots,A_{1,m1}), \ldots, R_n(A_{n,1},\ldots,A_{n,mn})$
  2. A (sub-) query Q (a relational algebra expression)

What is the |Q|?

- E.g:
  - $\sigma_{name=BookA}(Product)$?
  - $Product \bowtie Order$?
  - $\sigma_{name=BookA}(Product \bowtie Order \bowtie Customer)$?

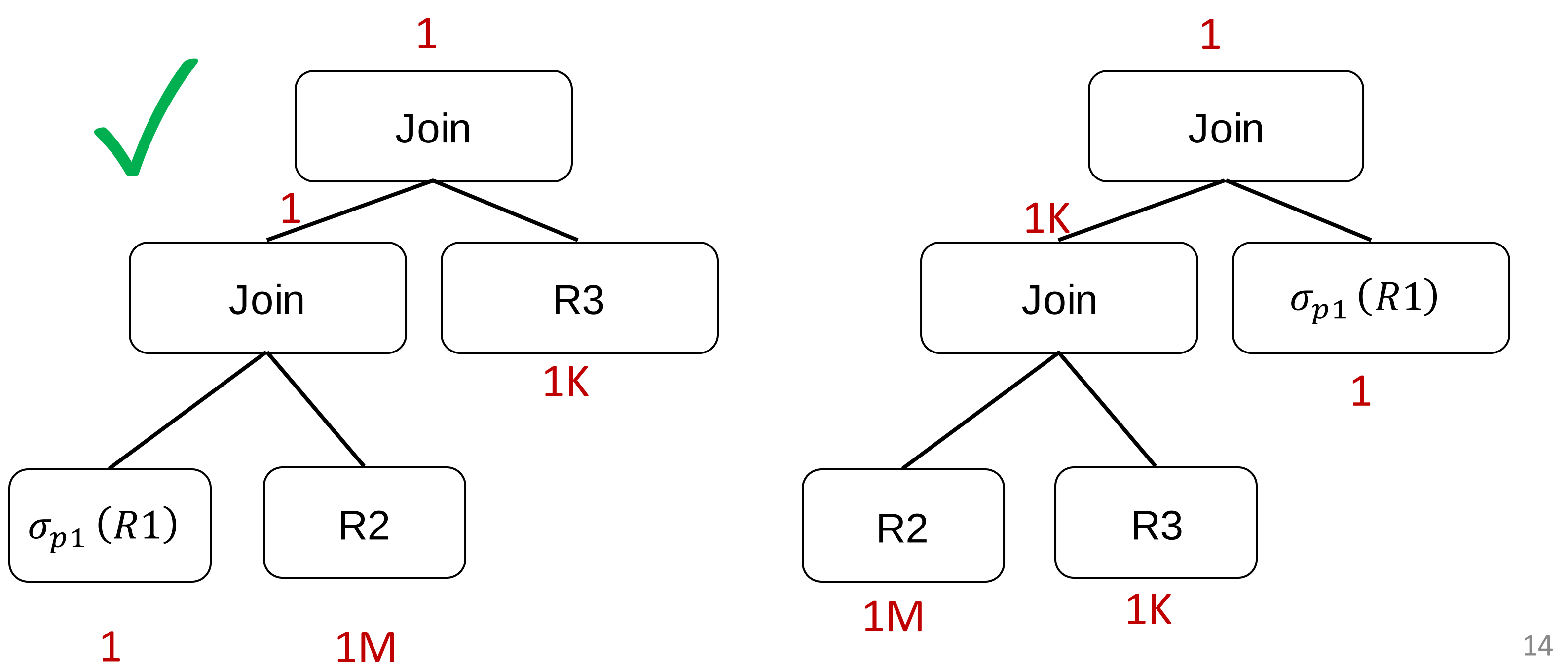


Example Poor Optimizer Choice

- Suppose cost(o_j): # input tuples processed.
- $\sigma_{p1}(R1) \bowtie R2 \bowtie \mathrm{R3}$
- Suppose $\sigma_{p1}(R1)$ = 1M but DBMS underestimates as 1
- Suppose |R2| = 1M and |R3| = 1K
- Suppose output of join has the size of the minimum input relation

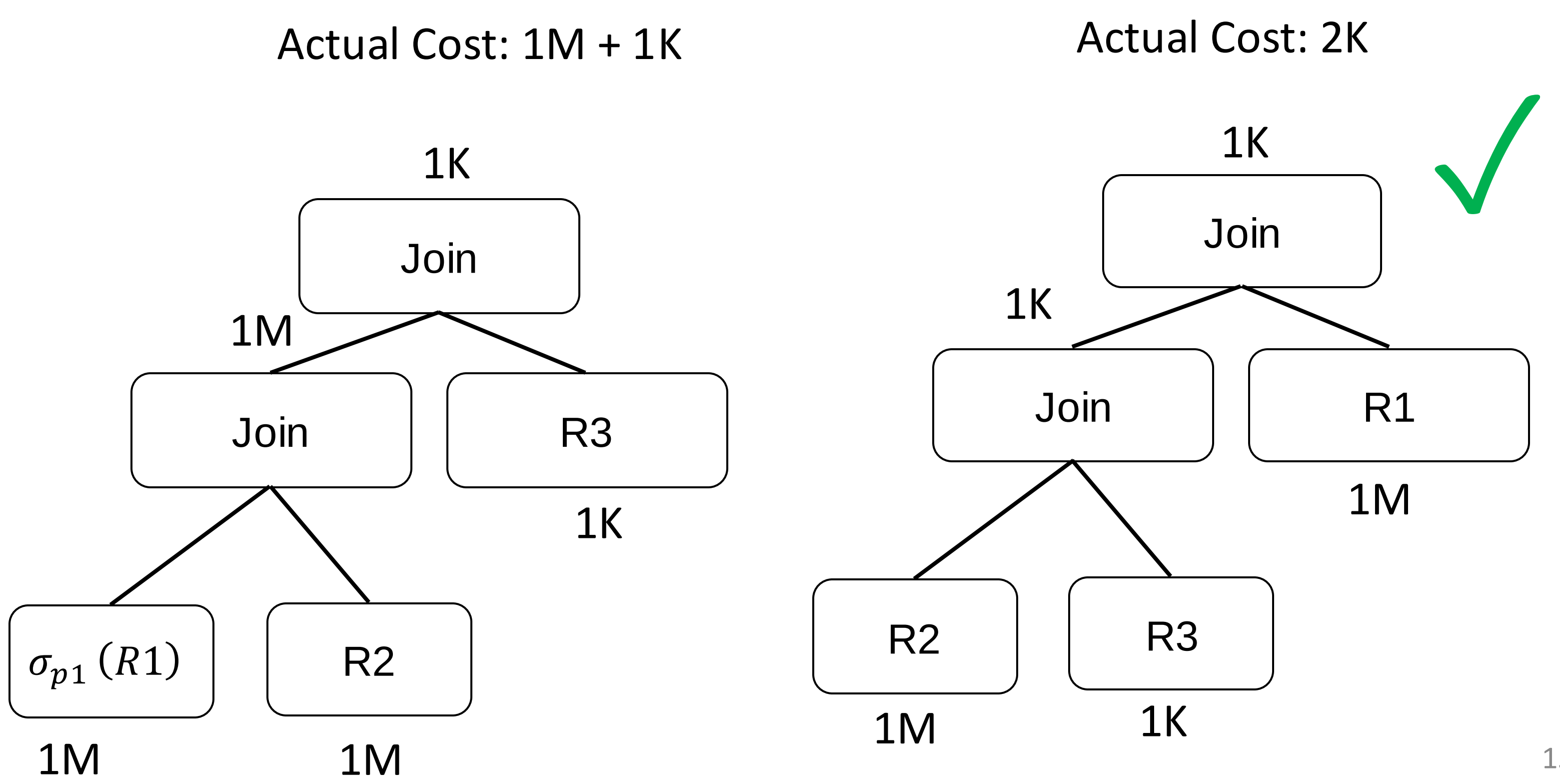
Estimated Cost: 2
(ignoring in relations)

Estimated Cost: 1001



Example Poor Optimizer Choice

- Suppose cost(o_j): # input tuples processed.
- $\sigma_{p1}(R1) \bowtie R2 \bowtie \mathrm{R3}$
- Suppose $\sigma_{p1}(R1)$ = 1M but DBMS underestimates as 1
- Suppose |R2| = 1M and |R3| = 1K
- Suppose output of join has the size of the minimum



Outline For Today

1. Goal of Query Optimization and Overview of Techniques
2. Cost-based Optimization Principles
3. Cost-based DP Logical Join Plan Optimizer
4. Cardinality Estimation Techniques
5. Rule-based Optimizations/Transformations
6. Final Remarks on Query Optimization & Query Processing

Widely Adopted Join Order Optimizer

Recall:

1. Enumerate a logical plan space (*often enumerates all join orders*)

$$L_1, L_2, \ldots, L_k$$

A widely used optimization algorithm is to use dynamic programming:

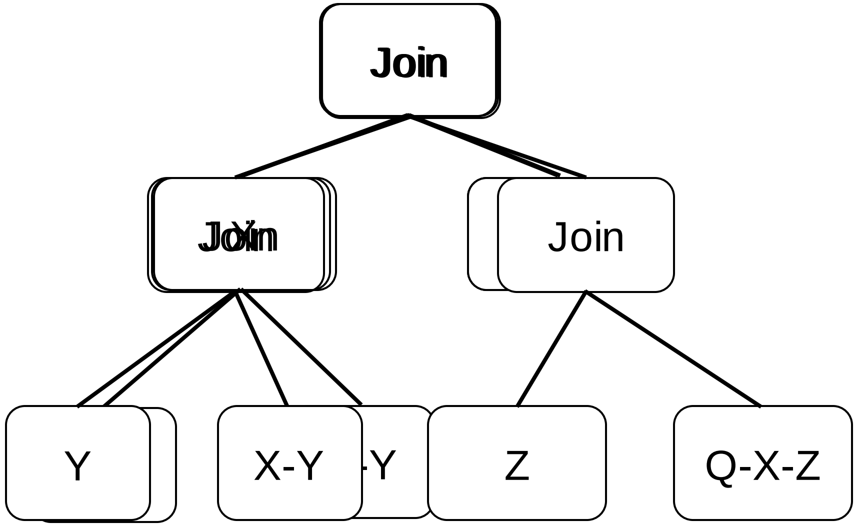
- ➢ Consider a join only query:

```
SELECT *
FROM R1 NATURAL JOIN R2 NATURAL JOIN … NATURAL JOIN Rn
```

- ➢ Q = R1 ⋈ R2 ⋈ … ⋈ Rn
- ➢ Note not-necessarily a ``chain'' query. It could be in any form, e.g:
  - ➢ R1(A, B) ⋈ R2(B, C) ⋈ R3(C, A) ⋈ R4(A, B, C)

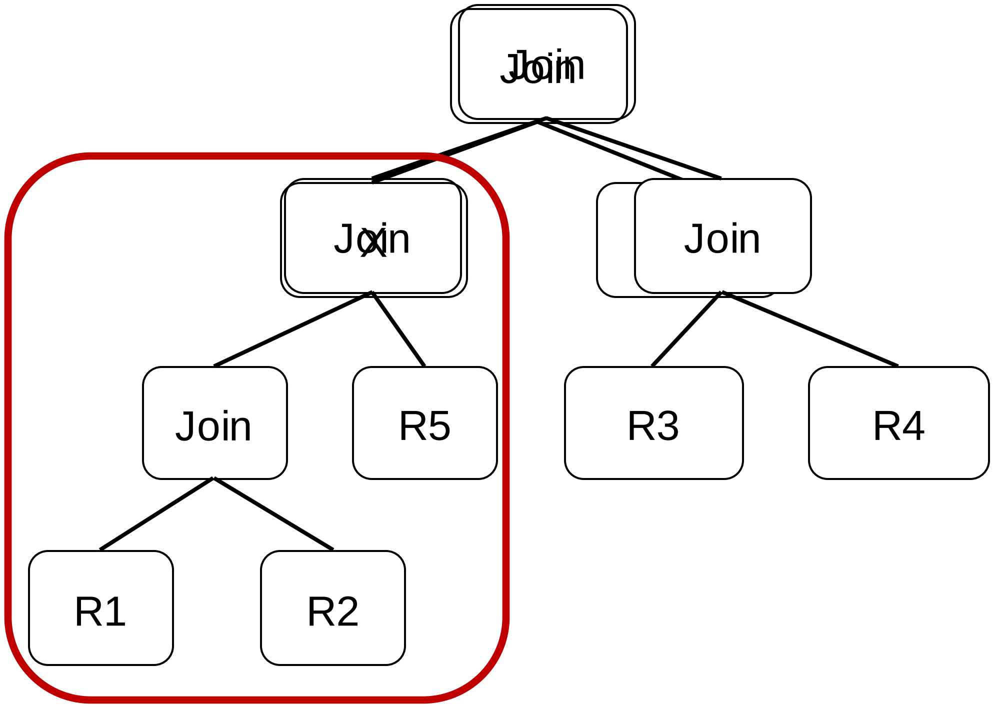
Plan Space

- In its most general form Plan Space=All possible join plan ``trees”
- In practice: If possible you’d avoid plans that do Cartesian Products
- Thought experiment: What does optimal tree L^* look like?



Optimal Sub-Join Tree Structure in L^*

- In L^*: What can we say about the sub-tree L^X starting from X?
- Must be the best plan for the sub-query $Q^X = \bowtie_{\forall Ri \in X} Ri$
  - E:g: red-box must be the best plan for R1 ⋈ R2 ⋈ R5 (o.w. just replace L^X with the best plan for Q^X: L^{X*}.)
- Therefore can use *dynamic programming algorithm to find join order.*



Cost-based DP Join Plan Optimizer

```
Input Q: R1 ⋈ R2 ⋈ … ⋈ Rn
Output Optimal Join Plan P:
OptPlans[]: a map that takes a sub-query Qt and stores the already
computed optimal plan:
for int t = 2 ... n // size of sub-queries
    for each Qt ⊆ Q with t relations
        P*Qt: // best plan found so far
        for each ``split'' X, Qt – X:
            P*x = OptPlans[X]; P*Qt-X = OptPlans[Qt-X];
            PQt: P*x ⋈ P*Qt-X; // Possible plan when split as X and Qt-X
            P*Qt = min cost of P*Qt, PQt
        OptPlans[Qt] = P*Qt
```

Optimization 1:
can enumerate over sub-queries that are ``connected" to avoid Cartesian Products

Optimization 2:
enumerate only if
X and Q_t–X have common attributes; otherwise the possible plan would Cartesian product

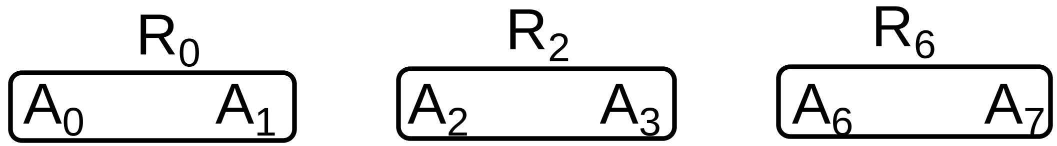
where cardinality estimation of Q_t *would happen*

Example Chain-based Join Optimizer (A3)

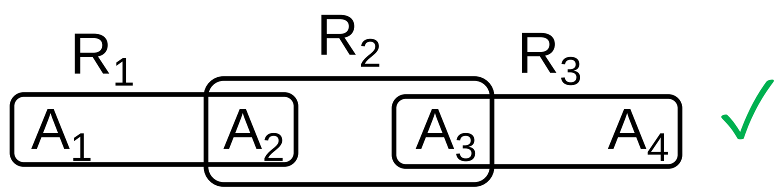
- A3: specialized version of DP Join Optimizer on ``chain queries``:

  Q: $R_0(A_0, A_1) \bowtie R_1(A_1, A_2) \bowtie \ldots \bowtie R_{n-1}(A_{n-1}, A_n)$

- Opt 1: Do not need to enumerate any dis-connected sub-query:
  - Q_{t1}: $R_0(A_0, A_1) \bowtie R_2(A_2, A_3) \bowtie R_6(A_6, A_7)$ X No common attributes



  - Q_{t2}: $R_1(A_1, A_2) \bowtie R_2(A_2, A_3) \bowtie R_3(A_3, A_4)$

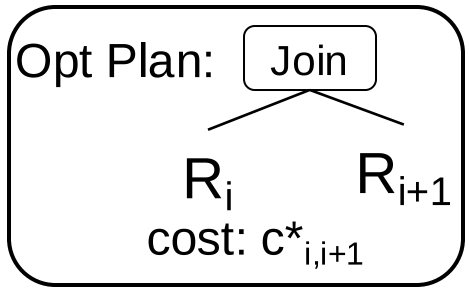


  - Enumerate plans only for "consecutive": $R_i \bowtie R_{i+1} \bowtie \ldots \bowtie R_j$
  - Enumerate only j-i``split points'' for each k: i…j-1:
    - $R_i \bowtie R_{i+1} \bowtie \ldots \bowtie R_k$ and $R_{k+1} \bowtie R_{k+2} \bowtie \ldots \bowtie R_j$

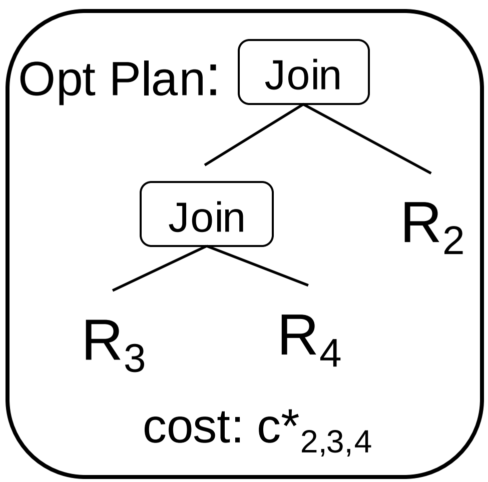
Simulation

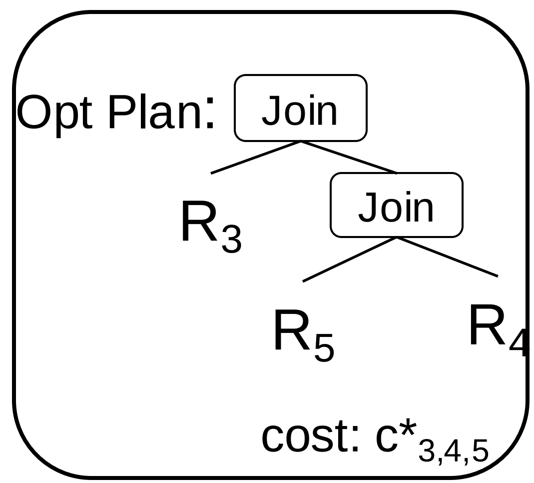
Opt Plans for 1-size sub-queries R_i:

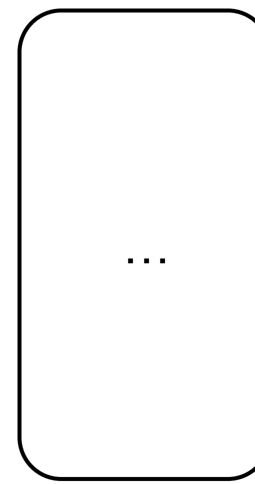
Opt Plans for 2-size sub-queries $R_i \bowtie R_{i+1}$:



Opt Plans for 3-size sub-queries (using 1- and 2-size opt. plans):

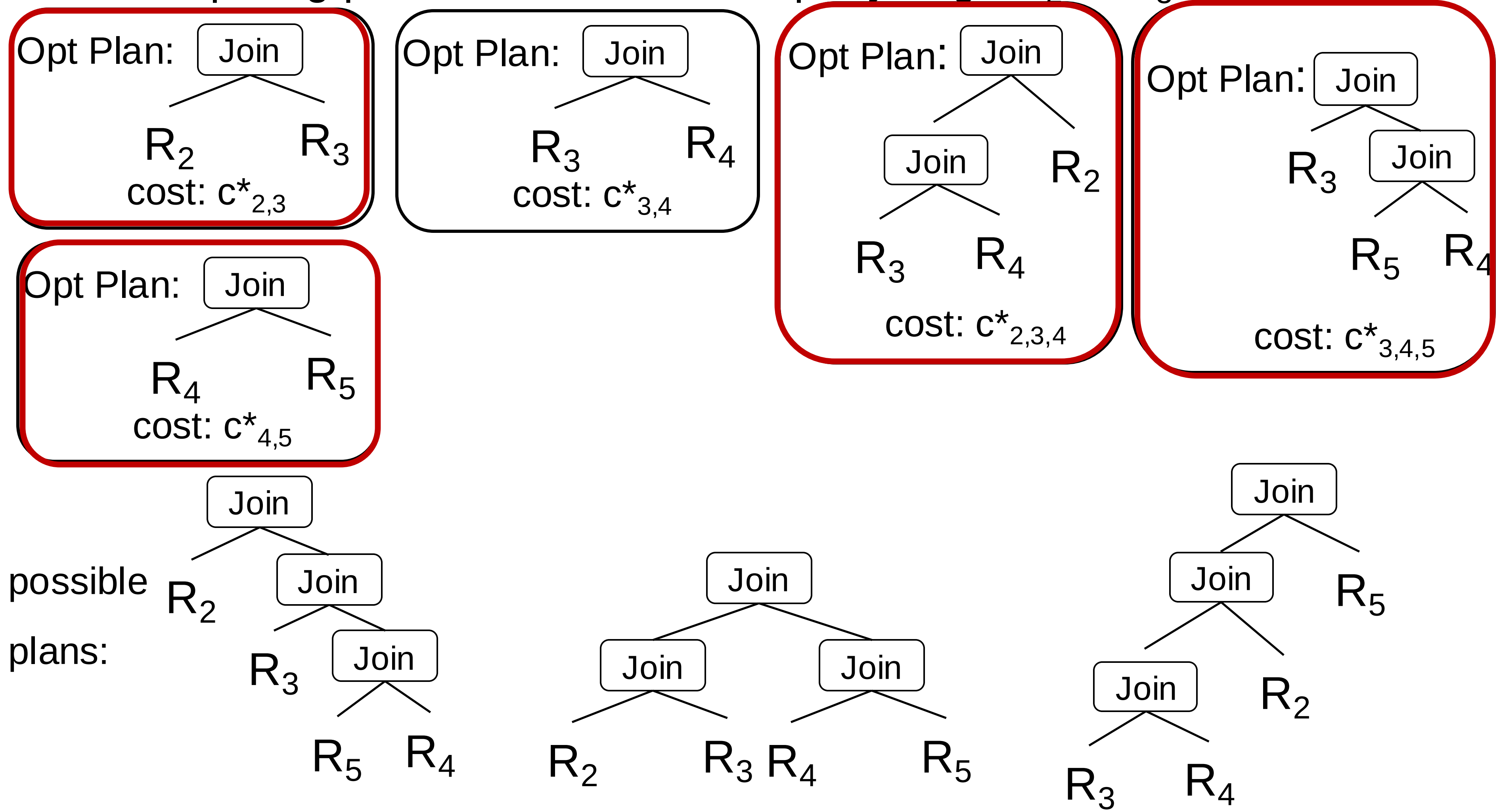






Simulation

When computing plans for a 4-size sub-query: e.g., $R_2 \bowtie R_3 \bowtie R_4 \bowtie R_5$:



R_2 as the split point R_3 as the split point R_4 as the split point

if left/right child matters compare 2x more plans

Solution (w/ wrong index accesses)

Let S[][] be an n by n array storing opt costs of to sub-queries

Note: For simplicity: we take cost($P_{i,j}$) is the cost of left & right sub-plans + card($R_i \bowtie \ldots \bowtie R_j$). Do this simplification in your solution as well.

S[i][j] is min cost of joining R_i, …, R_j

```
procedure DP-Join-Order(R_1 ... R_n):
  Base Cases: S[i][i]=|R_i|;S[i][i+1]=|R_i|+|R_{i+1}|+card(R_i⋈R_{i+1})
   for i = 1 … n
    for j = 1 … n
        min_{i,j} = +∞
        for k = i, … j-1
         min_{i,j}=min(min_{i,j},S[i][k]+S[k+1][j] + card(R_i⋈…⋈R_j)
        S[i][j] = min_{i,j}
  return S[1][n]
```

Looks wrong!

Ex: i=1, j = n, k =2

We access S[2][n] => not yet computed

Which Cells Do We Need to Access?

	1	2	3	4	5	6	7
1	$c_{1,1}$	$c_{1,2}$					
2		$c_{2,2}$	$c_{2,3}$				
3			$c_{3,3}$	$c_{3,4}$			
4				$c_{4,4}$	$c_{4,5}$		
5					$c_{5,5}$	$c_{5,6}$	
6						$c_{6,6}$	$c_{6,7}$
7							$c_{7,7}$

Which Cells Do We Need to Access?

	1	2	3	4	5	6	7
1	$C_{1,1}$	$C_{1,2}$					
2		$C_{2,2}$	$C_{2,3}$				
3			$C_{3,3}$	$C_{3,4}$			
4				$C_{4,4}$	$C_{4,5}$		
5					$C_{5,5}$	$C_{5,6}$	
6						$C_{6,6}$	$C_{6,7}$
7							$C_{7,7}$

$$S[2, 6] = \begin{cases} S[2, 2] + S[3,6] + card(R_2 \bowtie \ldots \bowtie R_6) \end{cases}$$

Which Cells Do We Need to Access?

	1	2	3	4	5	6	7
1	$c_{1,1}$	$c_{1,2}$					
2		$c_{2,2}$	$c_{2,3}$				
3			$c_{3,3}$	$c_{3,4}$			
4				$c_{4,4}$	$c_{4,5}$		
5					$c_{5,5}$	$c_{5,6}$	
6						$c_{6,6}$	$c_{6,7}$
7							$c_{7,7}$

$$S[2,6] = \begin{cases} S[2,2] + S[3,6] + card(R_2 \bowtie \ldots \bowtie R_6) \\ S[2,3] + S[4,6] + card(R_2 \bowtie \ldots \bowtie R_6) \\ \\ \end{cases}$$

Which Cells Do We Need to Access?

	1	2	3	4	5	6	7
1	$C_{1,1}$	$C_{1,2}$					
2		$C_{2,2}$	$C_{2,3}$				
3			$C_{3,3}$	$C_{3,4}$			
4				$C_{4,4}$	$C_{4,5}$		
5					$C_{5,5}$	$C_{5,6}$	
6						$C_{6,6}$	$C_{6,7}$
7							$C_{7,7}$

$$S[2,6] = \begin{cases} S[2,2] + S[3,6] + card(R_2 \bowtie \ldots \bowtie R_6) \\ S[2,3] + S[4,6] + card(R_2 \bowtie \ldots \bowtie R_6) \\ S[2,4] + S[5,6] + card(R_2 \bowtie \ldots \bowtie R_6) \end{cases}$$

Which Cells Do We Need to Access?

	1	2	3	4	5	6	7
1	$c_{1,1}$	$c_{1,2}$					
2		$c_{2,2}$	$c_{2,3}$				
3			$c_{3,3}$	$c_{3,4}$			
4				$c_{4,4}$	$c_{4,5}$		
5					$c_{5,5}$	$c_{5,6}$	
6						$c_{6,6}$	$c_{6,7}$
7							$c_{7,7}$

$$S[2,6] = \begin{cases} S[2,2] + S[3,6] + card(R_2 \bowtie \ldots \bowtie R_6) \\ S[2,3] + S[4,6] + card(R_2 \bowtie \ldots \bowtie R_6) \\ S[2,4] + S[5,6] + card(R_2 \bowtie \ldots \bowtie R_6) \\ S[2,5] + S[6,6] + card(R_2 \bowtie \ldots \bowtie R_6) \end{cases}$$

Which Cells Do We Need to Access?

	1	2	3	4	5	6	7
1	$C_{1,1}$	$C_{1,2}$					
2		$C_{2,2}$	$C_{2,3}$				
3			$C_{3,3}$	$C_{3,4}$			
4				$C_{4,4}$	$C_{4,5}$		
5					$C_{5,5}$	$C_{5,6}$	
6						$C_{6,6}$	$C_{6,7}$
7							$C_{7,7}$

$$S[2, 6] = \begin{cases} S[2, 2] + S[3,6] + card(R_2 \bowtie \ldots \bowtie R_6) \\ S[2, 3] + S[4,6] + card(R_2 \bowtie \ldots \bowtie R_6) \\ S[2, 4] + S[5,6] + card(R_2 \bowtie \ldots \bowtie R_6) \\ S[2, 5] + S[6,6] + card(R_2 \bowtie \ldots \bowtie R_6) \end{cases}$$

Which Cells Do We Need to Access?

	1	2	3	4	5	6	7
1	$c_{1,1}$	$c_{1,2}$					
2		$c_{2,2}$	$c_{2,3}$				
3			$c_{3,3}$	$c_{3,4}$			
4				$c_{4,4}$	$c_{4,5}$		
5					$c_{5,5}$	$c_{5,6}$	
6						$c_{6,6}$	$c_{6,7}$
7							$c_{7,7}$

$$S[3, 7] = \begin{cases} S[3, 3] + S[4,7] + \mathrm{card}(R_3 \bowtie \ldots \bowtie R_7) \\ \\ \end{cases}$$

Which Cells Do We Need to Access?

	1	2	3	4	5	6	7
1	$c_{1,1}$	$c_{1,2}$					
2		$c_{2,2}$	$c_{2,3}$				
3			$c_{3,3}$	$c_{3,4}$			
4				$c_{4,4}$	$c_{4,5}$		
5					$c_{5,5}$	$c_{5,6}$	
6						$c_{6,6}$	$c_{6,7}$
7							$c_{7,7}$

$$S[3, 7] = \begin{cases} S[3, 3] + S[4,7] + \text{card}(R_3 \bowtie \ldots \bowtie R_7) \\ S[3, 4] + S[5,7] + \text{card}(R_3 \bowtie \ldots \bowtie R_7) \\ \end{cases}$$

Which Cells Do We Need to Access?

	1	2	3	4	5	6	7
1	$c_{1,1}$	$c_{1,2}$					
2		$c_{2,2}$	$c_{2,3}$				
3			$c_{3,3}$	$c_{3,4}$			
4				$c_{4,4}$	$c_{4,5}$		
5					$c_{5,5}$	$c_{5,6}$	
6						$c_{6,6}$	$c_{6,7}$
7							$c_{7,7}$

$$S[3, 7] = \begin{cases} S[3, 3] + S[4,7] + card(R_3 \bowtie \ldots \bowtie R_7) \\ S[3, 4] + S[5,7] + card(R_3 \bowtie \ldots \bowtie R_7) \\ S[3, 5] + S[6,7] + card(R_3 \bowtie \ldots \bowtie R_7) \end{cases}$$

Which Cells Do We Need to Access?

	1	2	3	4	5	6	7
1	$c_{1,1}$	$c_{1,2}$					
2		$c_{2,2}$	$c_{2,3}$				
3			$c_{3,3}$	$c_{3,4}$			
4				$c_{4,4}$	$c_{4,5}$		
5					$c_{5,5}$	$c_{5,6}$	
6						$c_{6,6}$	$c_{6,7}$
7							$c_{7,7}$

$$S[3, 7] = \begin{cases} S[3, 3] + S[4,7] + card(R_3 \bowtie \ldots \bowtie R_7) \\ S[3, 4] + S[5,7] + card(R_3 \bowtie \ldots \bowtie R_7) \\ S[3, 5] + S[6,7] + card(R_3 \bowtie \ldots \bowtie R_7) \\ S[3, 6] + S[7,7] + card(R_3 \bowtie \ldots \bowtie R_7) \end{cases}$$

Which Cells Do We Need to Access?

	1	2	3	4	5	6	7
1	$c_{1,1}$	$c_{1,2}$					
2		$c_{2,2}$	$c_{2,3}$				
3			$c_{3,3}$	$c_{3,4}$			
4				$c_{4,4}$	$c_{4,5}$		
5					$c_{5,5}$	$c_{5,6}$	
6						$c_{6,6}$	$c_{6,7}$
7							$c_{7,7}$

$$S[3, 7] = \begin{cases} S[3, 3] + S[4,7] + \mathrm{card}(R_3 \bowtie \ldots \bowtie R_7) \\ S[3, 4] + S[5,7] + \mathrm{card}(R_3 \bowtie \ldots \bowtie R_7) \\ S[3, 5] + S[6,7] + \mathrm{card}(R_3 \bowtie \ldots \bowtie R_7) \\ S[3, 6] + S[7,7] + \mathrm{card}(R_3 \bowtie \ldots \bowtie R_7) \end{cases}$$

Correct Way to Traverse

	1	2	3	4	5	6	7
1	$c_{1,1}$	$c_{1,2}$					
2		$c_{2,2}$	$c_{2,3}$				
3			$c_{3,3}$	$c_{3,4}$			
4				$c_{4,4}$	$c_{4,5}$		
5					$c_{5,5}$	$c_{5,6}$	
6						$c_{6,6}$	$c_{6,7}$
7							$c_{7,7}$

The Way We Should Traverse

	1	2	3	4	5	6	7
1	$c_{1,1}$	$c_{1,2}$					
2		$c_{2,2}$	$c_{2,3}$				
3			$c_{3,3}$	$c_{3,4}$			
4				$c_{4,4}$	$c_{4,5}$		
5					$c_{5,5}$	$c_{5,6}$	
6						$c_{6,6}$	$c_{6,7}$
7							$c_{7,7}$

Correct Way to Traverse

	1	2	3	4	5	6	7
1	$c_{1,1}$	$c_{1,2}$					
2		$c_{2,2}$	$c_{2,3}$				
3			$c_{3,3}$	$c_{3,4}$			
4				$c_{4,4}$	$c_{4,5}$		
5					$c_{5,5}$	$c_{5,6}$	
6						$c_{6,6}$	$c_{6,7}$
7							$c_{7,7}$

Correct Way to Traverse

	1	2	3	4	5	6	7
1	$c_{1,1}$	$c_{1,2}$	$c_{1,3}$				
2		$c_{2,2}$	$c_{2,3}$				
3			$c_{3,3}$	$c_{3,4}$			
4				$c_{4,4}$	$c_{4,5}$		
5					$c_{5,5}$	$c_{5,6}$	
6						$c_{6,6}$	$c_{6,7}$
7							$c_{7,7}$

Correct Way to Traverse

	1	2	3	4	5	6	7
1	$c_{1,1}$	$c_{1,2}$	$c_{1,3}$				
2		$c_{2,2}$	$c_{2,3}$				
3			$c_{3,3}$	$c_{3,4}$			
4				$c_{4,4}$	$c_{4,5}$		
5					$c_{5,5}$	$c_{5,6}$	
6						$c_{6,6}$	$c_{6,7}$
7							$c_{7,7}$

Correct Way to Traverse

	1	2	3	4	5	6	7
1	$c_{1,1}$	$c_{1,2}$	$c_{1,3}$				
2		$c_{2,2}$	$c_{2,3}$				
3			$c_{3,3}$	$c_{3,4}$			
4				$c_{4,4}$	$c_{4,5}$		
5					$c_{5,5}$	$c_{5,6}$	
6						$c_{6,6}$	$c_{6,7}$
7							$c_{7,7}$

Correct Way to Traverse

	1	2	3	4	5	6	7
1	$c_{1,1}$	$c_{1,2}$	$c_{1,3}$				
2		$c_{2,2}$	$c_{2,3}$	$c_{2,4}$			
3			$c_{3,3}$	$c_{3,4}$			
4				$c_{4,4}$	$c_{4,5}$		
5					$c_{5,5}$	$c_{5,6}$	
6						$c_{6,6}$	$c_{6,7}$
7							$c_{7,7}$

Correct Way to Traverse

	1	2	3	4	5	6	7
1	$c_{1,1}$	$c_{1,2}$	$c_{1,3}$				
2		$c_{2,2}$	$c_{2,3}$	$c_{2,4}$			
3			$c_{3,3}$	$c_{3,4}$			
4				$c_{4,4}$	$c_{4,5}$		
5					$c_{5,5}$	$c_{5,6}$	
6						$c_{6,6}$	$c_{6,7}$
7							$c_{7,7}$

Correct Way to Traverse

	1	2	3	4	5	6	7
1	$c_{1,1}$	$c_{1,2}$	$c_{1,3}$				
2		$c_{2,2}$	$c_{2,3}$	$c_{2,4}$			
3			$c_{3,3}$	$c_{3,4}$			
4				$c_{4,4}$	$c_{4,5}$		
5					$c_{5,5}$	$c_{5,6}$	
6						$c_{6,6}$	$c_{6,7}$
7							$c_{7,7}$

Correct Way to Traverse

	1	2	3	4	5	6	7
1	$c_{1,1}$	$c_{1,2}$	$c_{1,3}$				
2		$c_{2,2}$	$c_{2,3}$	$c_{2,4}$			
3			$c_{3,3}$	$c_{3,4}$	$c_{3,5}$		
4				$c_{4,4}$	$c_{4,5}$		
5					$c_{5,5}$	$c_{5,6}$	
6						$c_{6,6}$	$c_{6,7}$
7							$c_{7,7}$

Correct Way to Traverse

	1	2	3	4	5	6	7
1	$c_{1,1}$	$c_{1,2}$	$c_{1,3}$				
2		$c_{2,2}$	$c_{2,3}$	$c_{2,4}$			
3			$c_{3,3}$	$c_{3,4}$	$c_{3,5}$		
4				$c_{4,4}$	$c_{4,5}$		
5					$c_{5,5}$	$c_{5,6}$	
6						$c_{6,6}$	$c_{6,7}$
7							$c_{7,7}$

Correct Way to Traverse

	1	2	3	4	5	6	7
1	$c_{1,1}$	$c_{1,2}$	$c_{1,3}$				
2		$c_{2,2}$	$c_{2,3}$	$c_{2,4}$			
3			$c_{3,3}$	$c_{3,4}$	$c_{3,5}$		
4				$c_{4,4}$	$c_{4,5}$		
5					$c_{5,5}$	$c_{5,6}$	
6						$c_{6,6}$	$c_{6,7}$
7							$c_{7,7}$

Correct Way to Traverse

	1	2	3	4	5	6	7
1	$c_{1,1}$	$c_{1,2}$	$c_{1,3}$				
2		$c_{2,2}$	$c_{2,3}$	$c_{2,4}$			
3			$c_{3,3}$	$c_{3,4}$	$c_{3,5}$		
4				$c_{4,4}$	$c_{4,5}$	$c_{4,6}$	
5					$c_{5,5}$	$c_{5,6}$	
6						$c_{6,6}$	$c_{6,7}$
7							$c_{7,7}$

Correct Way to Traverse

	1	2	3	4	5	6	7
1	$c_{1,1}$	$c_{1,2}$	$c_{1,3}$				
2		$c_{2,2}$	$c_{2,3}$	$c_{2,4}$			
3			$c_{3,3}$	$c_{3,4}$	$c_{3,5}$		
4				$c_{4,4}$	$c_{4,5}$	$c_{4,6}$	
5					$c_{5,5}$	$c_{5,6}$	
6						$c_{6,6}$	$c_{6,7}$
7							$c_{7,7}$

Correct Way to Traverse

	1	2	3	4	5	6	7
1	$c_{1,1}$	$c_{1,2}$	$c_{1,3}$				
2		$c_{2,2}$	$c_{2,3}$	$c_{2,4}$			
3			$c_{3,3}$	$c_{3,4}$	$c_{3,5}$		
4				$c_{4,4}$	$c_{4,5}$	$c_{4,6}$	
5					$c_{5,5}$	$c_{5,6}$	
6						$c_{6,6}$	$c_{6,7}$
7							$c_{7,7}$

Correct Way to Traverse

	1	2	3	4	5	6	7
1	$c_{1,1}$	$c_{1,2}$	$c_{1,3}$				
2		$c_{2,2}$	$c_{2,3}$	$c_{2,4}$			
3			$c_{3,3}$	$c_{3,4}$	$c_{3,5}$		
4				$c_{4,4}$	$c_{4,5}$	$c_{4,6}$	
5					$c_{5,5}$	$c_{5,6}$	$c_{5,7}$
6						$c_{6,6}$	$c_{6,7}$
7							$c_{7,7}$

Correct Way to Traverse

	1	2	3	4	5	6	7
1	$c_{1,1}$	$c_{1,2}$	$c_{1,3}$				
2		$c_{2,2}$	$c_{2,3}$	$c_{2,4}$			
3			$c_{3,3}$	$c_{3,4}$	$c_{3,5}$		
4				$c_{4,4}$	$c_{4,5}$	$c_{4,6}$	
5					$c_{5,5}$	$c_{5,6}$	$c_{5,7}$
6						$c_{6,6}$	$c_{6,7}$
7							$c_{7,7}$

Correct Way to Traverse

	1	2	3	4	5	6	7
1	$c_{1,1}$	$c_{1,2}$	$c_{1,3}$				
2		$c_{2,2}$	$c_{2,3}$	$c_{2,4}$			
3			$c_{3,3}$	$c_{3,4}$	$c_{3,5}$		
4				$c_{4,4}$	$c_{4,5}$	$c_{4,6}$	
5					$c_{5,5}$	$c_{5,6}$	$c_{5,7}$
6						$c_{6,6}$	$c_{6,7}$
7							$c_{7,7}$

Correct Way to Traverse

	1	2	3	4	5	6	7
1	$c_{1,1}$	$c_{1,2}$	$c_{1,3}$				
2		$c_{2,2}$	$c_{2,3}$	$c_{2,4}$			
3			$c_{3,3}$	$c_{3,4}$	$c_{3,5}$		
4				$c_{4,4}$	$c_{4,5}$	$c_{4,6}$	
5					$c_{5,5}$	$c_{5,6}$	$c_{5,7}$
6						$c_{6,6}$	$c_{6,7}$
7							$c_{7,7}$

Correct Way to Traverse

	1	2	3	4	5	6	7
1	$c_{1,1}$	$c_{1,2}$	$c_{1,3}$	$c_{1,4}$			
2		$c_{2,2}$	$c_{2,3}$	$c_{2,4}$			
3			$c_{3,3}$	$c_{3,4}$	$c_{3,5}$		
4				$c_{4,4}$	$c_{4,5}$	$c_{4,6}$	
5					$c_{5,5}$	$c_{5,6}$	$c_{5,7}$
6						$c_{6,6}$	$c_{6,7}$
7							$c_{7,7}$

Correct Way to Traverse

	1	2	3	4	5	6	7
1	$c_{1,1}$	$c_{1,2}$	$c_{1,3}$	$c_{1,4}$			
2		$c_{2,2}$	$c_{2,3}$	$c_{2,4}$	$c_{2,5}$		
3			$c_{3,3}$	$c_{3,4}$	$c_{3,5}$		
4				$c_{4,4}$	$c_{4,5}$	$c_{4,6}$	
5					$c_{5,5}$	$c_{5,6}$	$c_{5,7}$
6						$c_{6,6}$	$c_{6,7}$
7							$c_{7,7}$

Correct Way to Traverse

	1	2	3	4	5	6	7
1	$c_{1,1}$	$c_{1,2}$	$c_{1,3}$	$c_{1,4}$			
2		$c_{2,2}$	$c_{2,3}$	$c_{2,4}$	$c_{2,5}$		
3			$c_{3,3}$	$c_{3,4}$	$c_{3,5}$	$c_{3,6}$	
4				$c_{4,4}$	$c_{4,5}$	$c_{4,6}$	
5					$c_{5,5}$	$c_{5,6}$	$c_{5,7}$
6						$c_{6,6}$	$c_{6,7}$
7							$c_{7,7}$

Correct Way to Traverse

	1	2	3	4	5	6	7
1	$c_{1,1}$	$c_{1,2}$	$c_{1,3}$	$c_{1,4}$			
2		$c_{2,2}$	$c_{2,3}$	$c_{2,4}$	$c_{2,5}$		
3			$c_{3,3}$	$c_{3,4}$	$c_{3,5}$	$c_{3,6}$	
4				$c_{4,4}$	$c_{4,5}$	$c_{4,6}$	$c_{4,7}$
5					$c_{5,5}$	$c_{5,6}$	$c_{5,7}$
6						$c_{6,6}$	$c_{6,7}$
7							$c_{7,7}$

Correct Way to Traverse

	1	2	3	4	5	6	7
1	$c_{1,1}$	$c_{1,2}$	$c_{1,3}$	$c_{1,4}$	$c_{1,5}$		
2		$c_{2,2}$	$c_{2,3}$	$c_{2,4}$	$c_{2,5}$		
3			$c_{3,3}$	$c_{3,4}$	$c_{3,5}$	$c_{3,6}$	
4				$c_{4,4}$	$c_{4,5}$	$c_{4,6}$	$c_{4,7}$
5					$c_{5,5}$	$c_{5,6}$	$c_{5,7}$
6						$c_{6,6}$	$c_{6,7}$
7							$c_{7,7}$

Correct Way to Traverse

	1	2	3	4	5	6	7
1	$c_{1,1}$	$c_{1,2}$	$c_{1,3}$	$c_{1,4}$	$c_{1,5}$		
2		$c_{2,2}$	$c_{2,3}$	$c_{2,4}$	$c_{2,5}$	$c_{2,6}$	
3			$c_{3,3}$	$c_{3,4}$	$c_{3,5}$	$c_{3,6}$	
4				$c_{4,4}$	$c_{4,5}$	$c_{4,6}$	$c_{4,7}$
5					$c_{5,5}$	$c_{5,6}$	$c_{5,7}$
6						$c_{6,6}$	$c_{6,7}$
7							$c_{7,7}$

Correct Way to Traverse

	1	2	3	4	5	6	7
1	$c_{1,1}$	$c_{1,2}$	$c_{1,3}$	$c_{1,4}$	$c_{1,5}$		
2		$c_{2,2}$	$c_{2,3}$	$c_{2,4}$	$c_{2,5}$	$c_{2,6}$	
3			$c_{3,3}$	$c_{3,4}$	$c_{3,5}$	$c_{3,6}$	$c_{3,7}$
4				$c_{4,4}$	$c_{4,5}$	$c_{4,6}$	$c_{4,7}$
5					$c_{5,5}$	$c_{5,6}$	$c_{5,7}$
6						$c_{6,6}$	$c_{6,7}$
7							$c_{7,7}$

Correct Way to Traverse

	1	2	3	4	5	6	7
1	$c_{1,1}$	$c_{1,2}$	$c_{1,3}$	$c_{1,4}$	$c_{1,5}$	$c_{1,6}$	
2		$c_{2,2}$	$c_{2,3}$	$c_{2,4}$	$c_{2,5}$	$c_{2,6}$	
3			$c_{3,3}$	$c_{3,4}$	$c_{3,5}$	$c_{3,6}$	$c_{3,7}$
4				$c_{4,4}$	$c_{4,5}$	$c_{4,6}$	$c_{4,7}$
5					$c_{5,5}$	$c_{5,6}$	$c_{5,7}$
6						$c_{6,6}$	$c_{6,7}$
7							$c_{7,7}$

Correct Way to Traverse

	1	2	3	4	5	6	7
1	$c_{1,1}$	$c_{1,2}$	$c_{1,3}$	$c_{1,4}$	$c_{1,5}$	$c_{1,6}$	
2		$c_{2,2}$	$c_{2,3}$	$c_{2,4}$	$c_{2,5}$	$c_{2,6}$	$c_{2,7}$
3			$c_{3,3}$	$c_{3,4}$	$c_{3,5}$	$c_{3,6}$	$c_{3,7}$
4				$c_{4,4}$	$c_{4,5}$	$c_{4,6}$	$c_{4,7}$
5					$c_{5,5}$	$c_{5,6}$	$c_{5,7}$
6						$c_{6,6}$	$c_{6,7}$
7							$c_{7,7}$

Correct Way to Traverse

	1	2	3	4	5	6	7
1	$c_{1,1}$	$c_{1,2}$	$c_{1,3}$	$c_{1,4}$	$c_{1,5}$	$c_{1,6}$	
2		$c_{2,2}$	$c_{2,3}$	$c_{2,4}$	$c_{2,5}$	$c_{2,6}$	$c_{2,7}$
3			$c_{3,3}$	$c_{3,4}$	$c_{3,5}$	$c_{3,6}$	$c_{3,7}$
4				$c_{4,4}$	$c_{4,5}$	$c_{4,6}$	$c_{4,7}$
5					$c_{5,5}$	$c_{5,6}$	$c_{5,7}$
6						$c_{6,6}$	$c_{6,7}$
7							$c_{7,7}$

Correct Way to Traverse

	1	2	3	4	5	6	7
1	$c_{1,1}$	$c_{1,2}$	$c_{1,3}$	$c_{1,4}$	$c_{1,5}$	$c_{1,6}$	
2		$c_{2,2}$	$c_{2,3}$	$c_{2,4}$	$c_{2,5}$	$c_{2,6}$	$c_{2,7}$
3			$c_{3,3}$	$c_{3,4}$	$c_{3,5}$	$c_{3,6}$	$c_{3,7}$
4				$c_{4,4}$	$c_{4,5}$	$c_{4,6}$	$c_{4,7}$
5					$c_{5,5}$	$c_{5,6}$	$c_{5,7}$
6						$c_{6,6}$	$c_{6,7}$
7							$c_{7,7}$

Correct Way to Traverse

	1	2	3	4	5	6	7
1	$c_{1,1}$	$c_{1,2}$	$c_{1,3}$	$c_{1,4}$	$c_{1,5}$	$c_{1,6}$	
2		$c_{2,2}$	$c_{2,3}$	$c_{2,4}$	$c_{2,5}$	$c_{2,6}$	$c_{2,7}$
3			$c_{3,3}$	$c_{3,4}$	$c_{3,5}$	$c_{3,6}$	$c_{3,7}$
4				$c_{4,4}$	$c_{4,5}$	$c_{4,6}$	$c_{4,7}$
5					$c_{5,5}$	$c_{5,6}$	$c_{5,7}$
6						$c_{6,6}$	$c_{6,7}$
7							$c_{7,7}$

Correct Way to Traverse

	1	2	3	4	5	6	7
1	$c_{1,1}$	$c_{1,2}$	$c_{1,3}$	$c_{1,4}$	$c_{1,5}$	$c_{1,6}$	
2		$c_{2,2}$	$c_{2,3}$	$c_{2,4}$	$c_{2,5}$	$c_{2,6}$	$c_{2,7}$
3			$c_{3,3}$	$c_{3,4}$	$c_{3,5}$	$c_{3,6}$	$c_{3,7}$
4				$c_{4,4}$	$c_{4,5}$	$c_{4,6}$	$c_{4,7}$
5					$c_{5,5}$	$c_{5,6}$	$c_{5,7}$
6						$c_{6,6}$	$c_{6,7}$
7							$c_{7,7}$

Correct Way to Traverse

	1	2	3	4	5	6	7
1	$c_{1,1}$	$c_{1,2}$	$c_{1,3}$	$c_{1,4}$	$c_{1,5}$	$c_{1,6}$	
2		$c_{2,2}$	$c_{2,3}$	$c_{2,4}$	$c_{2,5}$	$c_{2,6}$	$c_{2,7}$
3			$c_{3,3}$	$c_{3,4}$	$c_{3,5}$	$c_{3,6}$	$c_{3,7}$
4				$c_{4,4}$	$c_{4,5}$	$c_{4,6}$	$c_{4,7}$
5					$c_{5,5}$	$c_{5,6}$	$c_{5,7}$
6						$c_{6,6}$	$c_{6,7}$
7							$c_{7,7}$

Correct Way to Traverse

	1	2	3	4	5	6	7
1	$c_{1,1}$	$c_{1,2}$	$c_{1,3}$	$c_{1,4}$	$c_{1,5}$	$c_{1,6}$	
2		$c_{2,2}$	$c_{2,3}$	$c_{2,4}$	$c_{2,5}$	$c_{2,6}$	$c_{2,7}$
3			$c_{3,3}$	$c_{3,4}$	$c_{3,5}$	$c_{3,6}$	$c_{3,7}$
4				$c_{4,4}$	$c_{4,5}$	$c_{4,6}$	$c_{4,7}$
5					$c_{5,5}$	$c_{5,6}$	$c_{5,7}$
6						$c_{6,6}$	$c_{6,7}$
7							$c_{7,7}$

Correct Way to Traverse

	1	2	3	4	5	6	7
1	$c_{1,1}$	$c_{1,2}$	$c_{1,3}$	$c_{1,4}$	$c_{1,5}$	$c_{1,6}$	$c_{1,7}$
2		$c_{2,2}$	$c_{2,3}$	$c_{2,4}$	$c_{2,5}$	$c_{2,6}$	$c_{2,7}$
3			$c_{3,3}$	$c_{3,4}$	$c_{3,5}$	$c_{3,6}$	$c_{3,7}$
4				$c_{4,4}$	$c_{4,5}$	$c_{4,6}$	$c_{4,7}$
5					$c_{5,5}$	$c_{5,6}$	$c_{5,7}$
6						$c_{6,6}$	$c_{6,7}$
7							$c_{7,7}$

Note, this is the final solution

(not the join plan but the cost of the opt plan)

Solution (w/ correct index accesses)

Let S[][] be an n by n array storing solutions to sub-queries

S[i][j] is min cost of joining R_i, …, R_j

```
procedure DP-Join-Order(R1 ... Rn):
 Base Cases:S[i][i]=|Ri|;S[i][i+1]=|Ri|+|Ri+1|+card(Ri⋈Ri+1)
  for d = 3 … n
   c = d; // column index
   for r = 1 … n-d+1 // row index
     min_r,c = +∞
     for k = r, … c-1 // different split points
       min_r,c= min(min_r,c, S[r][k]+S[k+1][c]+card(Rr⋈…⋈Rc))
     S[r][c] = min_r,c
     c++;// each time we increment r also increment c
return S[1][n]
```

Outline For Today

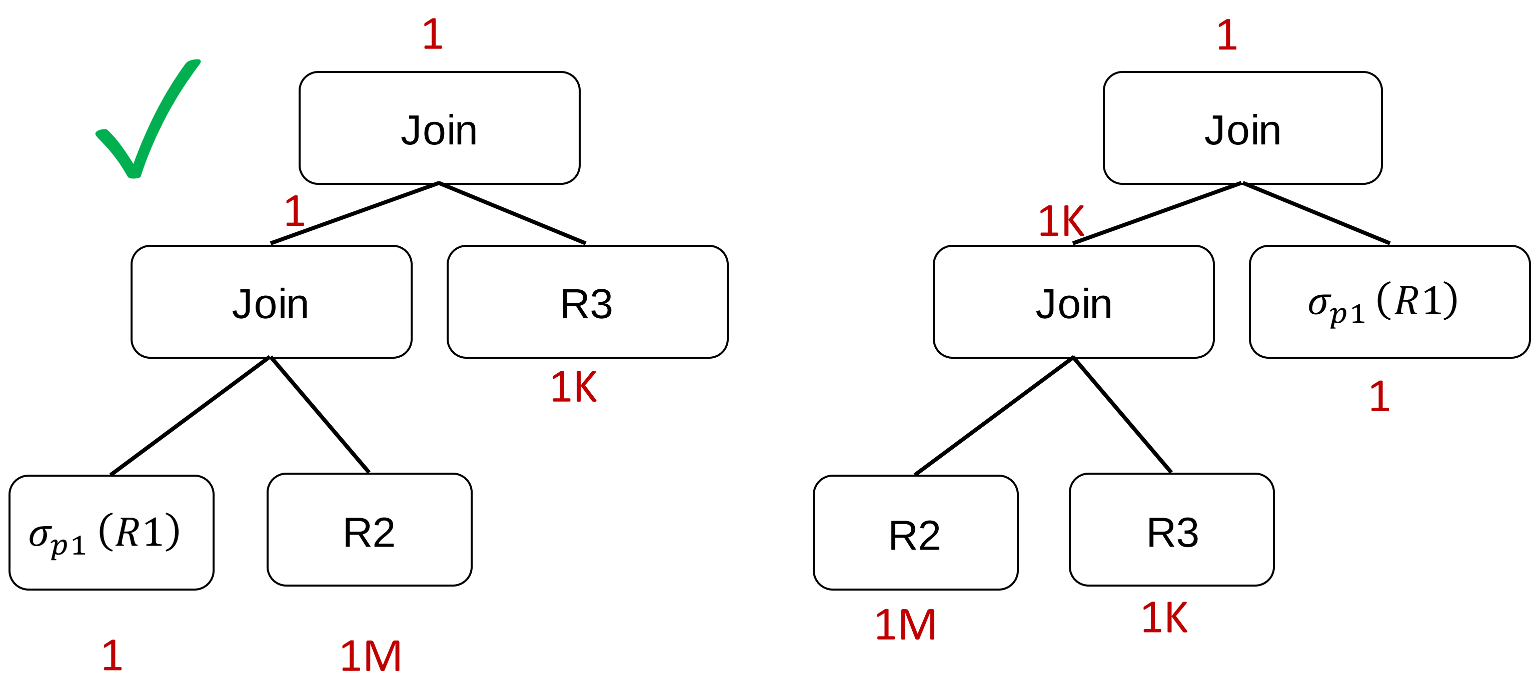
1. Goal of Query Optimization and Overview of Techniques
2. Cost-based Optimization Principles
3. Cost-based DP Logical Join Plan Optimizer
4. Cardinality Estimation Techniques
5. Rule-based Optimizations/Transformations
6. Final Remarks on Query Optimization & Query Processing

Recall Example Poor Optimizer Choice

- Suppose cost(o_j): # input tuples processed.
- $\sigma_{p1}(R1) \bowtie R2 \bowtie \mathrm{R3}$
- Suppose $\sigma_{p1}(R1)$ = 1M but DBMS underestimates as 1
- Suppose |R2| = 1M and |R3| = 1K
- Suppose output of join has the size of the minimum input relation

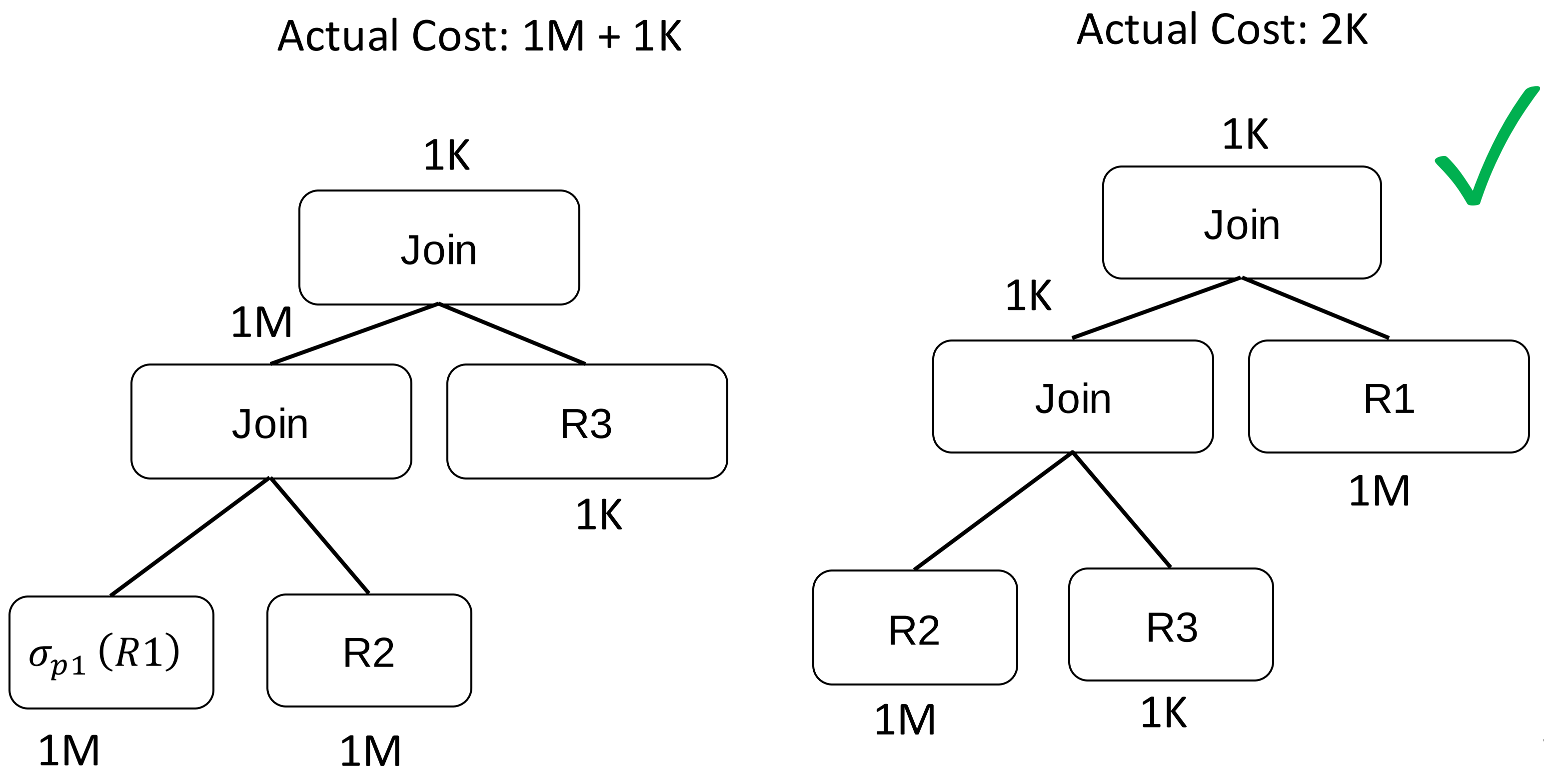
Estimated Cost: 2
(ignoring in relations)

Estimated Cost: 1001



Recall Example Poor Optimizer Choice

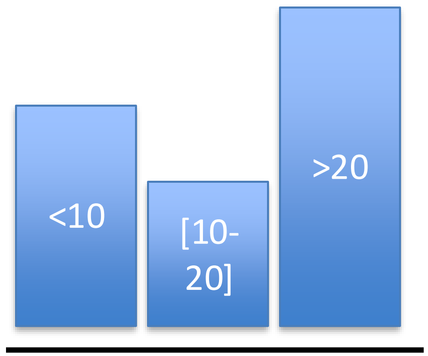
- Suppose cost(o_j): # input tuples processed.
- $\sigma_{p1}(R1) \bowtie R2 \bowtie \mathrm{R3}$
- Suppose $\sigma_{p1}(R1)$ = 1M but DBMS underestimates as 1
- Suppose |R2| = 1M and |R3| = 1K
- Suppose output of join has the size of the minimum



2 High-level Card. Estimation Techniques

1. Sampling-based:
   - While optimizing Q, sample relations to make an estimate
2. Summary/statistics-based:
   - Use statistics about D to make estimates
   - Possible statistics:
     - $|R_i|$: size of each relation
     - $|\pi_{Aj}(Ri)|$ # distinct values in column A_j
     - Histograms: Distribution of values on A_j
     - Also use known constraints:
       - E.g: FK constraint from R to S: $|R \bowtie S| = |\mathrm{R}|$
   - 2 common *simplification* assumptions (no other good reason):

     (i) uniformity; (ii) independence



Histogram on $R_i(A_j)$

Example Statistics-based Estimation Techniques

Selections with Equality Predicates

- Q: $\sigma_{A=v} R$
- Suppose the following information is available:
  - Size of R: $|R|$
  - Number of distinct A values in R: $|\pi_A R|$
- Assumptions:
  1. Values of A are *uniformly distributed* in R
  2. v ∈ $|\pi_A R|$

wild assumption, often doesn't hold

fair assumption, often holds (b/c users search things they put in the db)

- $|Q| \approx {}^{|R|}/_{|\pi_A R|}$
  - Selectivity factor of $(A = v)$ is ${}^{1}/_{|\pi_A R|}$
- Ex: $|Product| = 1000$, $|\pi_{name}(Product)| = 50$
  - $\sigma_{name=BookA} Product$: 1000/50 = 20

Conjunctive Predicates

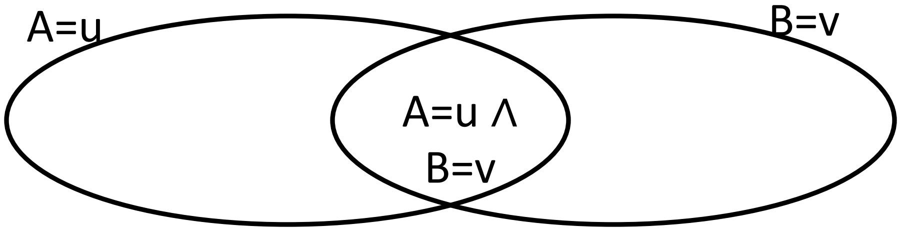
- Q: $\sigma_{A=u \wedge B=v} R$
- Additional assumption:

  3. $(A = u)$ and $(B = v)$ are *independent*

    - Counter example: age and salary
- $|Q| \approx {}^{|R|}/_{|\pi_A R| \cdot |\pi_B R|}$
  - Reduce total size by all selectivity factors
  - Directly derived from standard probability rules:
    - $Pr(E_1) = p_1$, and $Pr(E_2) = p_2$ and E_1 and E_2 are independent:
      - $Pr(E_1 \wedge E_2) = p_1 * p_2$
      - Ex: Pr(heads ∧ dice=6) = 1/2 * 1/6 = 1/12
- Ex: $|Prod| = 1000$, $|\pi_{name}(Prod)| = 50$, $|\pi_{merchant}(Prod)|$ = 4
  - $\sigma_{name=BookA \wedge \ marchant=B\&N} \ Product$: 1000/(50*4) = 5

Negated and Disjunctive Predicates

- $Q: \sigma_{A \neq v} R$
  - $|Q| \approx |R| \cdot \left(1 - {}^{1}/_{|\pi_A R|}\right)$
    - Selectivity factor of $\neg p$ is (1 − selectivity factor of p)
- $Q: \sigma_{A=u \vee B=v} R$
  - $|Q| \approx |R| \cdot \left({}^{1}/_{|\pi_A R|} + {}^{1}/_{|\pi_B R|}\right)$?
    - No! Tuples satisfying $(A = u)$ and $(B = v)$ are counted twice
    - Use only for $\sigma_{A=u \vee A=v} R$ (b/c then A=u and A=v are disjoint)
  - $|Q| \approx |R| \cdot \left({}^{1}/_{|\pi_A R|} + {}^{1}/_{|\pi_B R|} - {}^{1}/_{|\pi_A R||\pi_B R|}\right)$
    - Inclusion-exclusion principle from probability

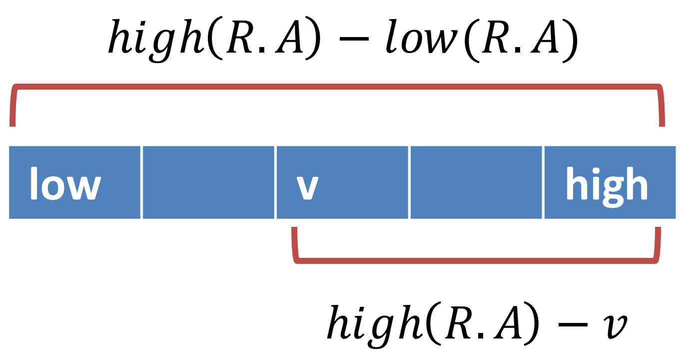


Range Predicates

- $\sigma_{A>u}\, R$?
- Case 1: Suppose the DBMS knew actual projection values:
  - Then range queries are a generalization of $\sigma_{A=u\, \vee A=v} R$
  - $\sigma_{A>u}\, R = |Q| \approx |R| \cdot \left({}^{|\#vals>u|}/_{|\pi_A R|} \right)$?
  - E.g: A was an int column and $|\pi_A R| = \{1, 2, 3, 4, 5\}$
    - $\sigma_{A>2} R = |R| * 3/5$

Case 2 of Range Predicates

- Case 2: We don't know actual values
- Not enough information!
  - Just pick a *magic constant*, e.g., $|Q| \approx |R| \cdot {}^{1}/_{3}$
- With more information
  - Largest *R.A* value: $\text{high}(R.A)$
  - Smallest *R.A* value: $\text{low}(R.A)$
  - $|Q| \approx |R| \cdot \frac{\text{high}(R.A) - v}{\text{high}(R.A) - \text{low}(R.A)}$
  - In practice: sometimes the second highest and lowest are used
    - The highest and the lowest are often used by inexperienced database designer to represent invalid values!



Equi-Join of Two Relations (1)

- Q: $R(A, B) \bowtie S(A, C)$
- Assumption: containment of value sets:
  - Every tuple in the "smaller" relation (one with fewer distinct values for the join attribute) joins with a tuple in the other relation
    - That is, if $|\pi_A R| \leq |\pi_A S|$ then $\pi_A R \subseteq \pi_A S$
  - Certainly not true in general
  - But holds in the common case of foreign key joins
- $|Q| \approx \frac{|R| \cdot |S|}{\max(|\pi_A R|, |\pi_A S|)}$
  - Selectivity factor of $R.A = S.A$ is ${}^{1}/_{\max(|\pi_A R|, |\pi_A S|)}$

Equi-Join of Two Relations (2)

- Example:

R

A	B
a_1	b_1
a_1	b_2
a_1	b_3
a_1	b_4

$\pi_A R = \{a_1\}$

S

A	C
a_1	c_1
a_1	c_2
a_2	c_3
a_2	c_4

$\pi_A S = \{a_1, a_2\}$

- $|Q| \approx \frac{|R| \cdot |S|}{\max(|\pi_A R|, |\pi_A S|)}$ = 4x4/2 = 8 (correct)
- If we had picked $\min(|\pi_A R|, |\pi_A S|)$, then we'd over-estimate
  - Intuitively a fraction of tuples from the larger-domain table will join with each tuple from smaller-domain table (not vice versa)

Other Estimations Techniques

- Using similar ideas, we can estimate the size of projection, duplicate elimination, union, difference, aggregation (with grouping)
- Lots of assumptions and very rough estimation
  - Accurate estimate is not needed
  - Maybe okay if we overestimate or underestimate consistently
- In practice: Very very difficult but very important for the optimizer.
  - B/c: ultimate goal is to help estimate costs of operators & plans
  - If we badly underestimate an expression => may lead to bad plans

Outline For Today

1. Goal of Query Optimization and Overview of Techniques
2. Cost-based Optimization Principles
3. Cost-based DP Logical Join Plan Optimizer
4. Cardinality Estimation Techniques
5. Rule-based Optimizations/Transformations
6. Final Remarks on Query Optimization & Query Processing

Rule-based Transformations

- DP-based join optimizer algorithm only considered join-only queries
- What if there was a selection, projection, group-by aggregate etc?
- When possible we consider them as we enumerate plans but often in a *rule-based manner*

Example (1)

```
SELECT *
FROM R1 NATURAL JOIN R2 NATURAL JOIN R3 NATURAL JOIN R4
WHERE R1.A1 = “foo” AND R3.A3=“bar”
```

- Intuitively instead of enumerating a plan for R1 we should enumerate a plan for relation: $\sigma_{A1=foo}(R1)$
- Similarly instead of R2, we should enumerate plans for $\sigma_{A3=bar}(R3)$
- Why?
- But not if the predicate was: `R1.A1 = “foo” OR R3.A3=“bar”`
- What to enumerate is governed by algebraic laws
  - *This is an important advantage of implementing a query language that’s based on a formal algebra: i.e., relational algebra*

Example (2)

```
SELECT *
FROM R1 NATURAL JOIN R2 NATURAL JOIN R3 NATURAL JOIN R4
WHERE R1.A1 = “foo” AND R3.A3=“bar”
```

- In relational algebra:

$$\sigma_{A1=foo \wedge A3=bar}(R1 \bowtie R2 \bowtie \mathrm{R3} \bowtie \mathrm{R4}) = (\sigma_{A1=foo}(R1) \bowtie R2 \bowtie \sigma_{A3=bar}(\mathrm{R3}) \bowtie \mathrm{R4})$$

- The expression effectively joins these smaller relations:
  - *i.* $\sigma_{A1=foo}(R1)$
  - *ii.* $R2$
  - *iii.* $\sigma_{A3=bar}(\mathrm{R3})$
  - iv. R4
- What if WHERE clause was `R1.A1 = “foo” OR R3.A3=“bar”`?
  - Apply the predicate only for sub-queries with both R1 and R3.
- The above algebraic law is called: *pushing down selections*

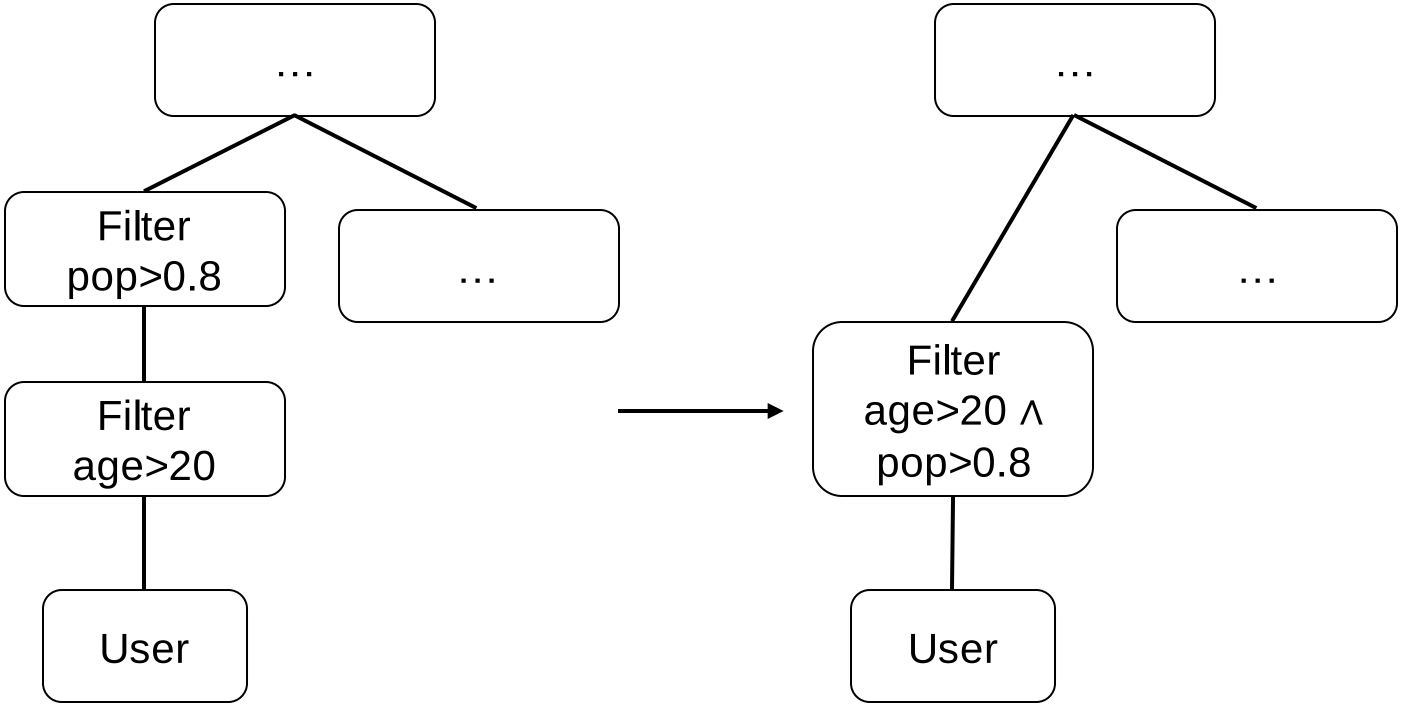
Ex Algebraic Transformation Rules (1)

Will use pure rel. algebra notation but can use our logical plan notation

- Convert σ_p-$\times$ to/from $\bowtie_p$: $\sigma_p(R \times S) = R \bowtie_p S$
  - Example: $\sigma_{User.uid=Member.uid}(User \times Member) = User \bowtie Member$
- Merge/split σ's: $\sigma_{p_1}\left(\sigma_{p_2} R\right) = \sigma_{p_1 \wedge p_2} R$
  - Example: $\sigma_{age>20}\left(\sigma_{pop=0.8} User\ \right) = \sigma_{age>20 \wedge pop=0.8} User$
- Merge/split π's: $\pi_{L_1}\left(\pi_{L_2} R\right) = \pi_{L_1} R$ (note $L_1 \subseteq L_2$ must hold for the expression to be valid)
  - Example: $\pi_{age}\left(\pi_{age,pop} User\right) = \pi_{age} User$

Example In Logical Plan Notation

- Merge/split σ's: $\sigma_{p_1}(\sigma_{p_2} R) = \sigma_{p_1 \wedge p_2} R$
  - Example: $\sigma_{age>20}(\sigma_{pop=0.8} User\ ) = \sigma_{age>20 \wedge pop=0.8} User$



Ex Algebraic Transformation Rules (2)

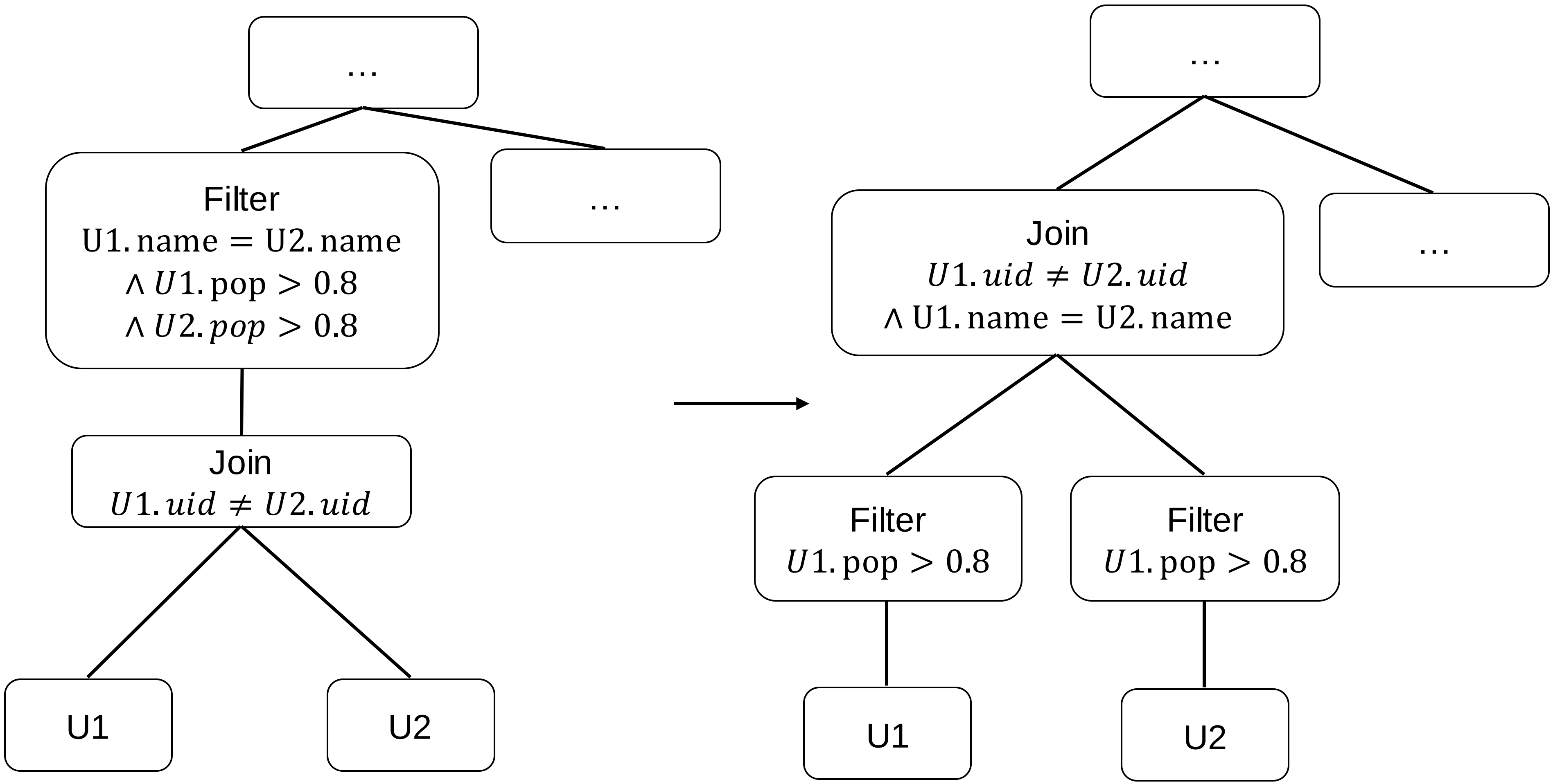
- Push down/pull up σ:

  $\sigma_{p \wedge p_r \wedge p_s}\left(R \bowtie_{p'} S\right) = \left(\sigma_{p_r} R\right) \bowtie_{p \wedge p'} \left(\sigma_{p_s} S\right)$, where

  - p_r is a predicate involving only R columns
  - p_s is a predicate involving only S columns
  - p and p' are predicates involving both R and S columns
    - i.e., p an additional join predicate
  - Example:
    $\sigma_{\text{U1.name=U2.name} \wedge U1.\text{pop}>0.8 \wedge U2.pop>0.8}(\rho_{U1} User \bowtie_{U1.uid \neq U2.uid} \rho_{U2} User)$
    $= \sigma_{pop>0.8}(\rho_{U1} User) \bowtie_{U1.uid \neq U2.uid, U1.name=U2.name} (\sigma_{pop>0.8}(\rho_{U2} User))$
  - Why should you always push selections down if possible?
    - Selections are relatively cheap (e.g., compared to joins or group-by and aggregates) and can only reduce the number tuples processed.

Example In Logical Plan Notation

$$\sigma_{\text{U1.name=U2.name} \wedge U1.pop>0.8 \wedge U2.pop>0.8}(\rho_{U1} User \bowtie_{U1.uid \neq U2.uid} \rho_{U2} User)$$
$$= \sigma_{pop>0.8}(\rho_{U1} User) \bowtie_{U1.uid \neq U2.uid, U1.name=U2.name} (\sigma_{pop>0.8}(\rho_{U2} User))$$



Ex When σ Cannot Be Pushed

- Outer Joins
- E.g: (1) $\sigma_{D>15}(\mathrm{R} ⟕_{R.B=S.C} S)$!= (2) $(\mathrm{R} ⟕_{R.B=S.C} \sigma_{D>15}(S))$
- Consider R and S instances below:
- (1)'s output is empty
- (2)'s output is (123, abc, null, null)

R	
<u>A</u>	<u>B</u>
123	abc

S	
<u>C</u>	<u>D</u>
abc	17

- Could have pushed the selection if the predicate was on A.
- You can only push when you guarantee the equality of original expression and the pushed-down expression

Ex Algebraic Transformation Rules (3)

- Push down π: $\pi_L(\sigma_p R) = \pi_L\left(\sigma_p(\pi_{LL'} R)\right)$, where
  - L' is the set of columns referenced by p that are not in L
  - Example:

    $$\pi_{age}(\sigma_{pop>0.8} User) = \pi_{age}(\sigma_{pop>0.8}(\pi_{age,pop} User))$$

  - Not as important and effective as pushing σ
- Many more equivalences can be systematically used to transform plans

Outline For Today

1. Goal of Query Optimization and Overview of Techniques
2. Cost-based Optimization Principles
3. Cost-based DP Logical Join Plan Optimizer
4. Cardinality Estimation Techniques
5. Rule-based Optimizations/Transformations
6. Final Remarks on Query Optimization & Query Processing

Final Remarks (1)

- Query Optimizer and Cardinality Estimator: Brain of the DBMS
  - *Ultimate Goal: Pick a reasonable plan (i.e,. one processing few tuples)*
- Query Processor and Storage: Skeleton
  - They do actual data searching and computation
- Several insights have emerged over the years in DBMS literature:
  - Cost model is not very critical: keep a simple model (e.g., # tuples)
  - Cardinality estimation: matters a lot
  - But! Extremely difficult to integrate a good estimator. Always a hack with wild unrealistic assumptions here and there to make it implementable: magic constants, uniformity assumptions, independence assumptions etc.
- My advice: Optimizer is important but keep it simple.
  - Do not be complacent on the query processor and storage! Work very hard on these and optimize relentlessly!

Query Optimization Through the Looking Glass, and What We Found Running the Join Order Benchmark, Leis et al., VLDBJ, 2018

Final Remarks (2)

- CS 448: Database Systems Implementation
  - Gets into many more details about the internals of query processing and optimization and other DBMS components!
  - A3's programming question is meant to give you a glimpse of CS 448 assignments.