CSE599d: Advanced Query Processing

Lecture 3: Optimization by Dynamic Programming

Dan Suciu

University of Washington

From SQL to Execution

- Input: text of SQL query
- Parse ($\Rightarrow$ AST), semantic analysis, etc, $\Rightarrow$ Query Plan
- Query Plan Optimization
- Optional: code generation (LLVM or other)
- Execute

Query Plan Optimization

- SQL is Declarative: specifies What, does not say How.
- Many alternative execution plans are possible.
- Goal of query optimizer: find the optimal plan.
- In practice: avoid a catastrophic plan.

Two Query Optimization Approaches

- Dynamic Programming: this and the next few lectures

- Extensible Query Optimization: will discuss Cascades later

References for Today

[Moerkotte and Neumann, 2006]

[Moerkotte et al., 2013]

[Neumann and Kemper, 2015]

[Neumann and Radke, 2018]

[Meister et al., 2018]

[Neumann, 2024]

[Neumann, 2025]

[Birler and Neumann, 2025]

Notations

Dynamic Programming

- Convert the SQL query into a *query graph* G
- Use dynamic programming to construct an optimal plan from G.
- Complications: non-reordable operators (outer joins, anti joins), nested subqueries, large join graphs. Will discuss in future lectures.

Operators

$\sigma_p(R)$

$e_1 \times e_2$ cross product

$e_1 \bowtie_p e_2$ join

$e_1 \bowtie e_2$ natural join

$e_1 \ltimes_p e_2$ semi-join

$e_1 \triangleright_p e_2$ anti-join

$e_1 \rightarrowtail_p e_2$ left outer join

$e_1 \bowtie\!\!\!\!\!\!\rightarrowtail_p e_2$ right outer join

$e_1 \rightarrowtail\!\!\!\!\!\!\rightarrowtail_p e_2$ right outer join

$\chi_{a:f}(e)$ map function (apply f, call it a)

$\Gamma_{A;a:f}(e)$ group by/aggregate

$e_1 \bowtie\!\!\!\!\!\!\blacktriangleright_p e_2$ dependent join

Notations: Attributes and Free Variables

$\mathcal{A}(e)$ = attributes produced by the expression e;

$\mathcal{F}(e)$ = free variables that occur in the expression e;

$\mathcal{A}_T(e)$, $\mathcal{F}_T(e)$ tables used by $\mathcal{A}(e)$, $\mathcal{F}(e)$.

Notations: Attributes and Free Variables

$\mathcal{A}(e)$ = attributes produced by the expression e;

$\mathcal{F}(e)$ = free variables that occur in the expression e;

$\mathcal{A}_T(e)$, $\mathcal{F}_T(e)$ tables used by $\mathcal{A}(e)$, $\mathcal{F}(e)$.

Examples:

$\mathcal{A}(R(A,B) \bowtie_{B=C} S(C,D)) = \{A,B,C,D\}$,

$\mathcal{A}(\bowtie_{B=C}) = \{B,C\}$

$\mathcal{A}_T(\bowtie_{B=C}) = \{R,S\}$

$\mathcal{F}(R(A,B) \bowtie_{B=C} S(C,D)) = \emptyset$

$\mathcal{F}(\bowtie_{B=C}) = \{B,C\}$

Dependend Join [Neumann, 2025]

$e_1 \bowtie_p e_2$

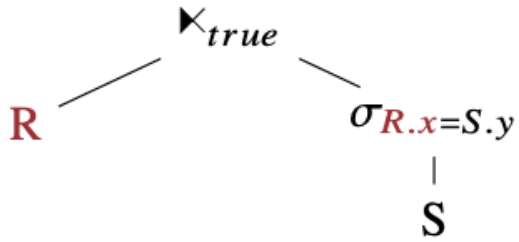
e_2 may refer to attributes from e_1: $\mathcal{F}(e_2) \subseteq \mathcal{A}(e_1)$.

Dependend Join [Neumann, 2025]

$e_1 \bowtie_p e_2$

e_2 may refer to attributes from e_1: $\mathcal{F}(e_2) \subseteq \mathcal{A}(e_1)$.

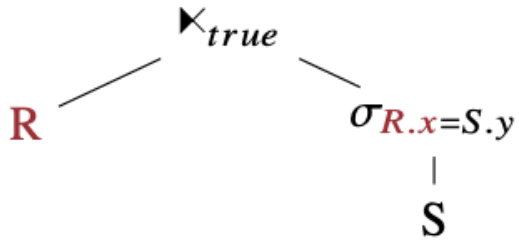
```
SELECT *
FROM R
WHERE EXISTS(SELECT *
             FROM S
             WHERE R.x=S.y)
```



Dependend Join [Neumann, 2025]

$e_1 \bowtie^{\cdot}_p e_2$ e_2 may refer to attributes from e_1: $\mathcal{F}(e_2) \subseteq \mathcal{A}(e_1)$.

```
SELECT *
FROM R
WHERE EXISTS(SELECT *
             FROM S
             WHERE R.x=S.y)
```




```
SELECT *
FROM customer
WHERE c_mktsegment='AUTOMOBILE' AND
  (SELECT COUNT(*)
   FROM orders
   WHERE o_custkey=c_custkey AND
     (SELECT SUM(l_extendedprice)
      FROM lineitem
      WHERE l_orderkey=o_orderkey)>300000)>5
```

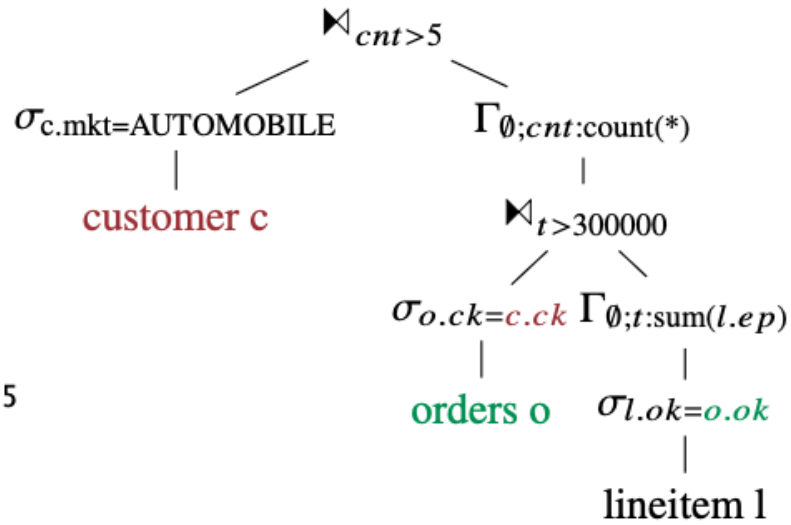


Fig. 1: Example Query with Nested Correlated Subqueries, the Correlations are Color Coded

Dependent joins are bad: nested loops. Will discuss next week.

Query Graph

Vertices = relations; Edges = represent join conditions

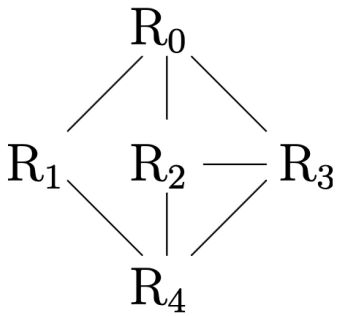
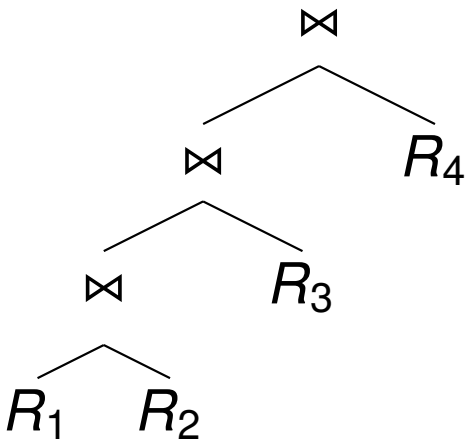


Figure 6: Sample graph to illustrate `EnumerateCsgRec`

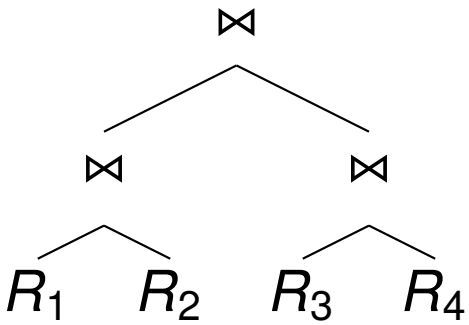
[Moerkotte and Neumann, 2006]

In class: discuss relationship to the query's hypergraph

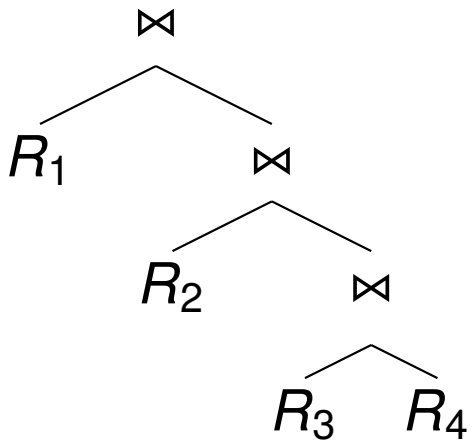
Types of Query Plans



Left deep



Bushy

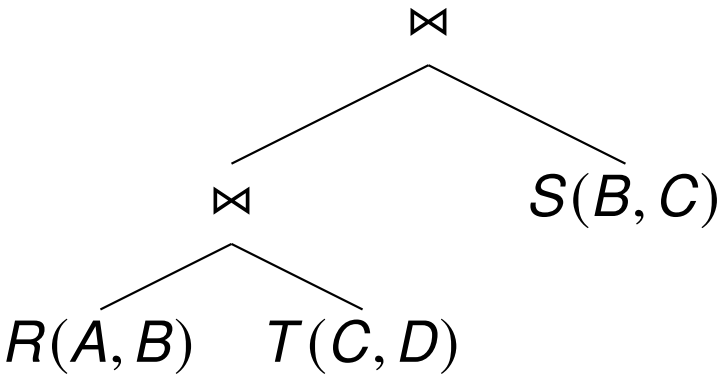


Right deep

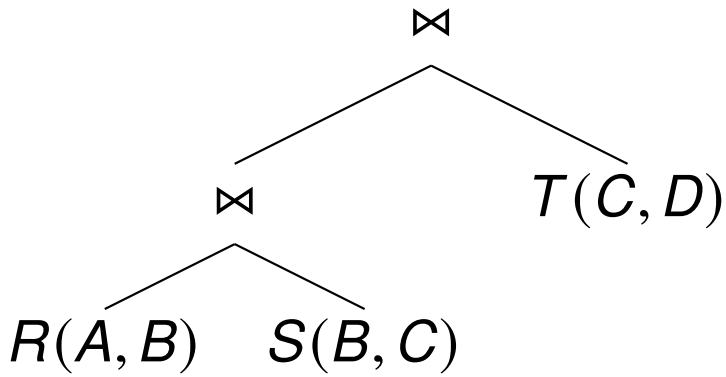
Cross Products

Optimizers avoid query plans with cross product:

Has cross a product:



No cross product:

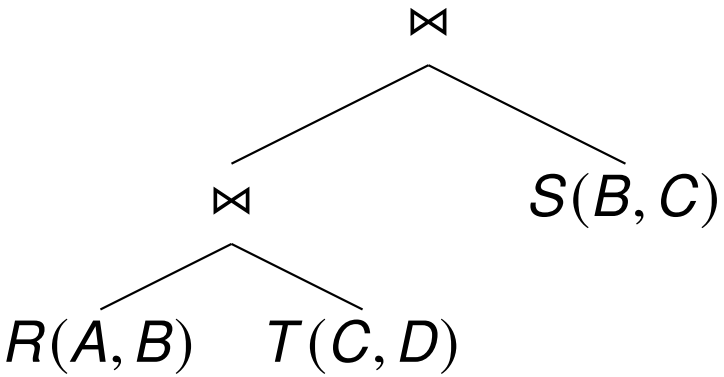


When can a cross product be beneficial?

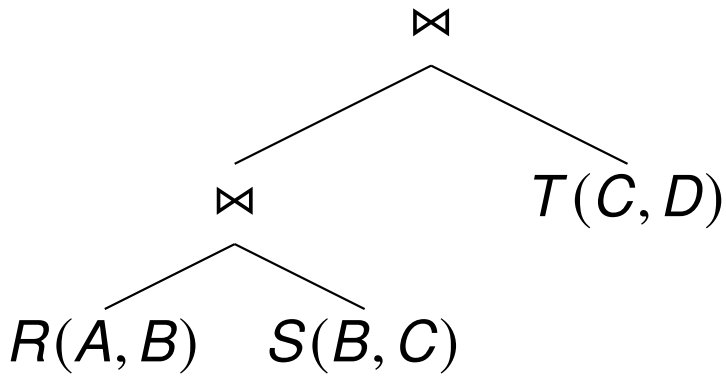
Cross Products

Optimizers avoid query plans with cross product:

Has cross a product:



No cross product:



When can a cross product be beneficial? When both tables are small

In that case, define the cross-product as a new table for the optimizer.

Cost of a Query Plan

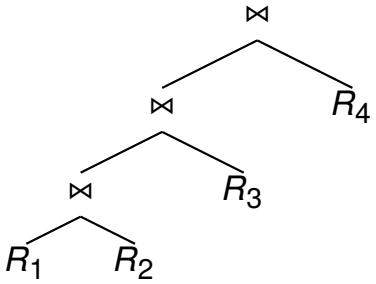
We will assume to have a function Cost(P) = the cost of plan P.

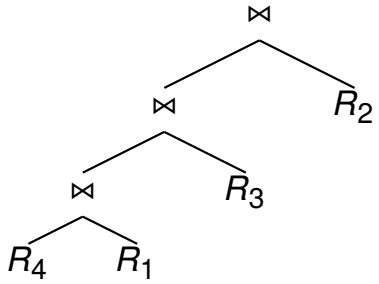
Later we will look at specific cost functions which can lead to more efficient algorithms.

Counting Plans

Counting Left Linear Plans: $R_1 \bowtie R_2 \bowtie \cdots \bowtie R_n$

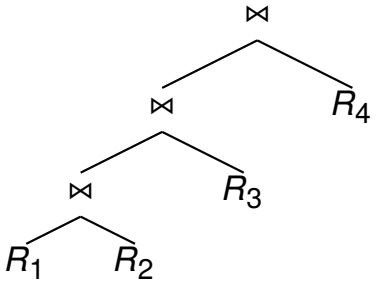
How Many Left Linear Plans?

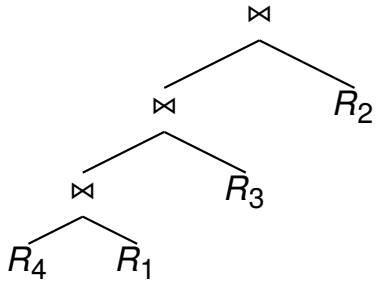




Counting Left Linear Plans: $R_1 \bowtie R_2 \bowtie \cdots \bowtie R_n$

How Many Left Linear Plans?

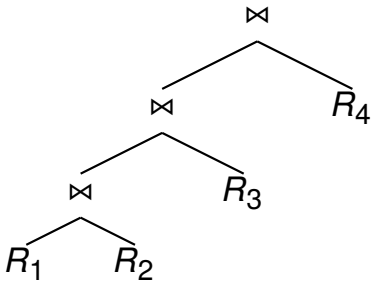


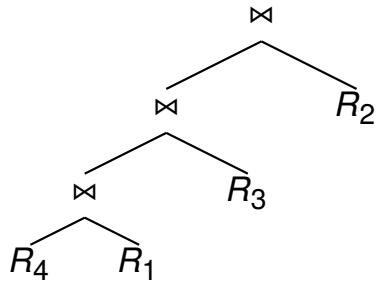


$n!$

Counting Left Linear Plans: $R_1 \bowtie R_2 \bowtie \cdots \bowtie R_n$

How Many Left Linear Plans?





$n!$

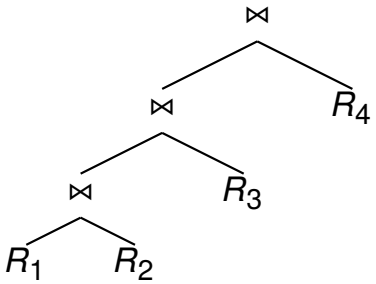
How many are without cross products?

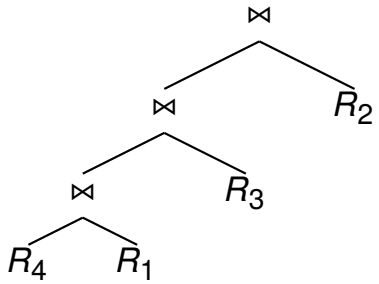
Chain query: $R_1(x_0, x_1) \bowtie R_2(x_1, x_2) \bowtie \cdots$

$$R_1 - R_2 - R_3 - \cdots - R_n$$

Counting Left Linear Plans: $R_1 \bowtie R_2 \bowtie \cdots \bowtie R_n$

How Many Left Linear Plans?





$n!$

How many are without cross products?

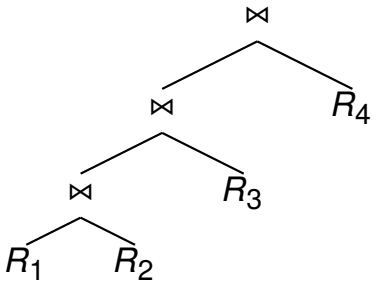
Chain query: $R_1(x_0, x_1) \bowtie R_2(x_1, x_2) \bowtie \cdots$

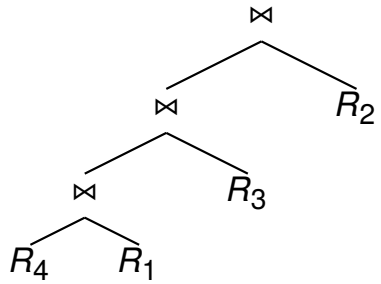
2^n

$$R_1 - R_2 - R_3 - \cdots - R_n$$

Counting Left Linear Plans: $R_1 \bowtie R_2 \bowtie \cdots \bowtie R_n$

How Many Left Linear Plans?





$n!$

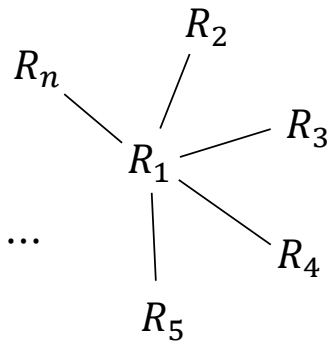
How many are without cross products?

Chain query: $R_1(x_0, x_1) \bowtie R_2(x_1, x_2) \bowtie \cdots$

2^n

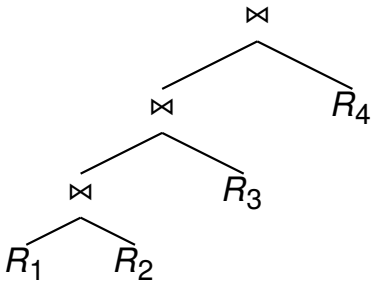
$$R_1 - R_2 - R_3 - \cdots - R_n$$

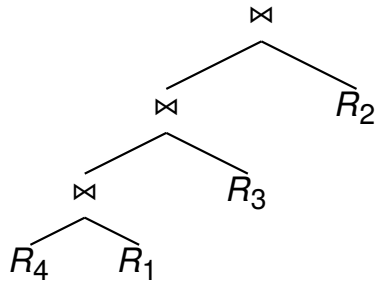
Star query: $R_1(x_2, \ldots, x_n) \bowtie R_2(x_2) \bowtie R_3(x_3) \bowtie \cdots$



Counting Left Linear Plans: $R_1 \bowtie R_2 \bowtie \cdots \bowtie R_n$

How Many Left Linear Plans?





$n!$

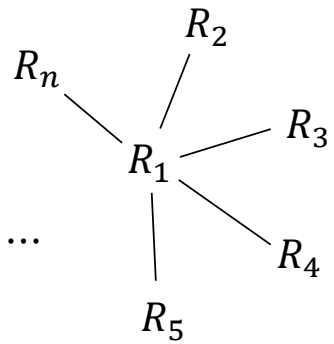
How many are without cross products?

Chain query: $R_1(x_0, x_1) \bowtie R_2(x_1, x_2) \bowtie \cdots$

$$R_1 - R_2 - R_3 - \cdots - R_n$$

2^n

Star query: $R_1(x_2, \ldots, x_n) \bowtie R_2(x_2) \bowtie R_3(x_3) \bowtie \cdots$



$2 \cdot (n-1)!$

Counting Bushy Plans: $R_1 \bowtie R_2 \bowtie \cdots \bowtie R_n$

How many are without cross products?

Chain query: $R_1(x_0, x_1) \bowtie R_2(x_1, x_2) \bowtie \cdots$

$R_1 - R_2 - R_3 - \cdots - R_n$

Counting Bushy Plans: $R_1 \bowtie R_2 \bowtie \cdots \bowtie R_n$

How many are without cross products?

Chain query: $R_1(x_0, x_1) \bowtie R_2(x_1, x_2) \bowtie \cdots$

$$R_1 - R_2 - R_3 - \cdots - R_n$$

Catalan Number:

$$C_{n-1} = \frac{1}{n}\binom{2(n-1)}{n-1}$$

$((R_1R_2)(R_3(R_4R_5)))$

$(R_1((R_2R_3)(R_4R_5)))$

Counting Bushy Plans: $R_1 \bowtie R_2 \bowtie \cdots \bowtie R_n$

How many are without cross products?

Chain query: $R_1(x_0, x_1) \bowtie R_2(x_1, x_2) \bowtie \cdots$

$R_1 - R_2 - R_3 - \cdots - R_n$

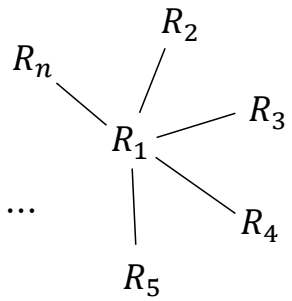
Catalan Number:

$$C_{n-1} = \frac{1}{n}\binom{2(n-1)}{n-1}$$

$((R_1R_2)(R_3(R_4R_5)))$

$(R_1((R_2R_3)(R_4R_5)))$

Star query:



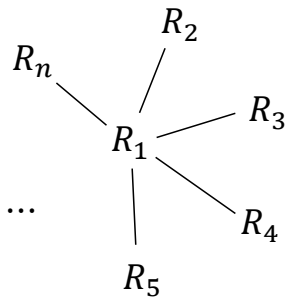
Counting Bushy Plans: $R_1 \bowtie R_2 \bowtie \cdots \bowtie R_n$

How many are without cross products?

Chain query: $R_1(x_0, x_1) \bowtie R_2(x_1, x_2) \bowtie \cdots$

$R_1 - R_2 - R_3 - \cdots - R_n$

Star query:



Catalan Number:

$$C_{n-1} = \frac{1}{n}\binom{2(n-1)}{n-1}$$

$((R_1R_2)(R_3(R_4R_5)))$

$(R_1((R_2R_3)(R_4R_5)))$

Zig-zag plans, from R_1:

$$2^{n-1}(n-1)!$$

Counting Bushy Plans: $R_1 \bowtie R_2 \bowtie \cdots \bowtie R_n$

How many are without cross products?

Chain query: $R_1(x_0, x_1) \bowtie R_2(x_1, x_2) \bowtie \cdots$

$R_1 - R_2 - R_3 - \cdots - R_n$

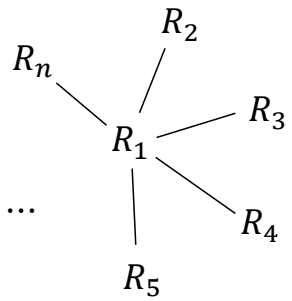
Catalan Number:

$$C_{n-1} = \frac{1}{n}\binom{2(n-1)}{n-1}$$

$((R_1R_2)(R_3(R_4R_5)))$

$(R_1((R_2R_3)(R_4R_5)))$

Star query:



Zig-zag plans, from R_1:

$$2^{n-1}(n-1)!$$

How many are with cross products?

Counting Bushy Plans: $R_1 \bowtie R_2 \bowtie \cdots \bowtie R_n$

How many are without cross products?

Chain query: $R_1(x_0, x_1) \bowtie R_2(x_1, x_2) \bowtie \cdots$

$R_1 - R_2 - R_3 - \cdots - R_n$

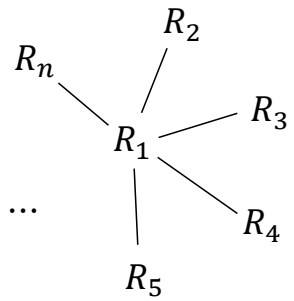
Catalan Number:

$$C_{n-1} = \frac{1}{n}\binom{2(n-1)}{n-1}$$

$((R_1R_2)(R_3(R_4R_5)))$

$(R_1((R_2R_3)(R_4R_5)))$

Star query:



Zig-zag plans, from R_1:

$$2^{n-1}(n-1)!$$

How many are with cross products?

Permutation plus parentheses

$((R_1R_2)(R_3(R_4R_5)))$

$((R_4R_1)(R_2(R_3R_5)))$

$$n! \cdot C_{n-1} = \frac{(2(n-1))!}{(n-1)!}$$

Catalan Number

$C_n = \frac{1}{n+1}\binom{2n}{n}$ # of parenthesis on $R_0 R_1 \cdots R_n$

$C_0 = 1, \quad C_n = \sum_{k=1}^{n-1} C_k C_{n-k-1}$ $((R_0 \cdots R_{k-1}) \cdot (R_k \cdots R_n))$

Generating function $f(x) = \sum_{n \geq 0} C_n x^n \quad f(x) = 1 + x f^2(x) \quad f(x) = \frac{1-\sqrt{1-4x}}{2x}$.

$C_n \approx \frac{4^n}{n^{3/2}\sqrt{\pi}}$

Growth Rate W/ and W/O Cross Product

n	Left Linear			Bushy		
	w/ CP	w/o CP		w/ CP	w/o CP	
		Chain	Star		Chain	Star
	$n!$	2^n	$2(n-1)!$	$n!C_{n-1}$	C_{n-1}	$2^{n-1}(n-1)!$
1	1	2	2	1	1	1
2	2	4	2	2	1	2
3	6	8	4	12	2	8
4	24	16	12	120	5	48
5	120	32	48	1680	14	384
6	720	64	240	30240	42	3840
7	5040	128	1440	665280	132	46080
8	40320	256	10080	17297280	429	645120
9	362880	512	80640	518918400	1430	10321920
10	3628800	1024	725760	17643225600	4862	185794560

It is important to avoid cross products!

Top-down Dynamic Programming

Two Types of Dyanmic Programming Query Optimizers

- Top-down. Easiest to describe, finds a first plan quickly.

- Bottom-up. More control of the search space.

Top-Down Algorithm

Function OptimalPlanAndCost(S) $\quad\quad$ $\boldsymbol{R} = \{R_1, \ldots, R_n\}$

if $|S| = 1$ **then**
  $(\text{plan}, \text{cost}) := (\text{scan}(R_i), \text{Cost}(\text{scan}(R_i)))$; ; $\quad\quad$ // $S = \{R_i\}$
else
  $(\text{plan}, \text{cost}) := (\text{null}, \infty)$;
  for $S_1 : \emptyset \subsetneq S_1 \subsetneq S$ **do**
    $S_2 := S - S_1$;
    $(P_1, c_1) := \text{OptimalPlanAndCost}(S_1)$;
    $(P_2, c_2) := \text{OptimalPlanAndCost}(S_2)$;
    $c := c_1 + c_2 + \text{Cost}(P_1 \bowtie P_2)$;
    if $c <$ *cost* **then** $(\text{plan}, \text{cost}) := (P_1 \bowtie P_2, c)$;
return $(\text{plan}, \text{cost})$;

Top-Down Algorithm

Function OptimalPlanAndCost(S) $\quad\quad$ $\boldsymbol{R} = \{R_1, \ldots, R_n\}$

if $|S| = 1$ **then**
  $(\text{plan}, \text{cost}) := (\text{scan}(R_i), \text{Cost}(\text{scan}(R_i)))$; ; $\quad\quad$ // $S = \{R_i\}$
else
  $(\text{plan}, \text{cost}) := (\text{null}, \infty)$;
  for $S_1 : \emptyset \subsetneq S_1 \subsetneq S$ **do**
    $S_2 := S - S_1$;
    $(P_1, c_1) := \text{OptimalPlanAndCost}(S_1)$;
    $(P_2, c_2) := \text{OptimalPlanAndCost}(S_2)$;
    $c := c_1 + c_2 + \text{Cost}(P_1 \bowtie P_2)$;
    if $c <$ *cost* **then** $(\text{plan}, \text{cost}) := (P_1 \bowtie P_2, c)$;
return $(\text{plan}, \text{cost})$;

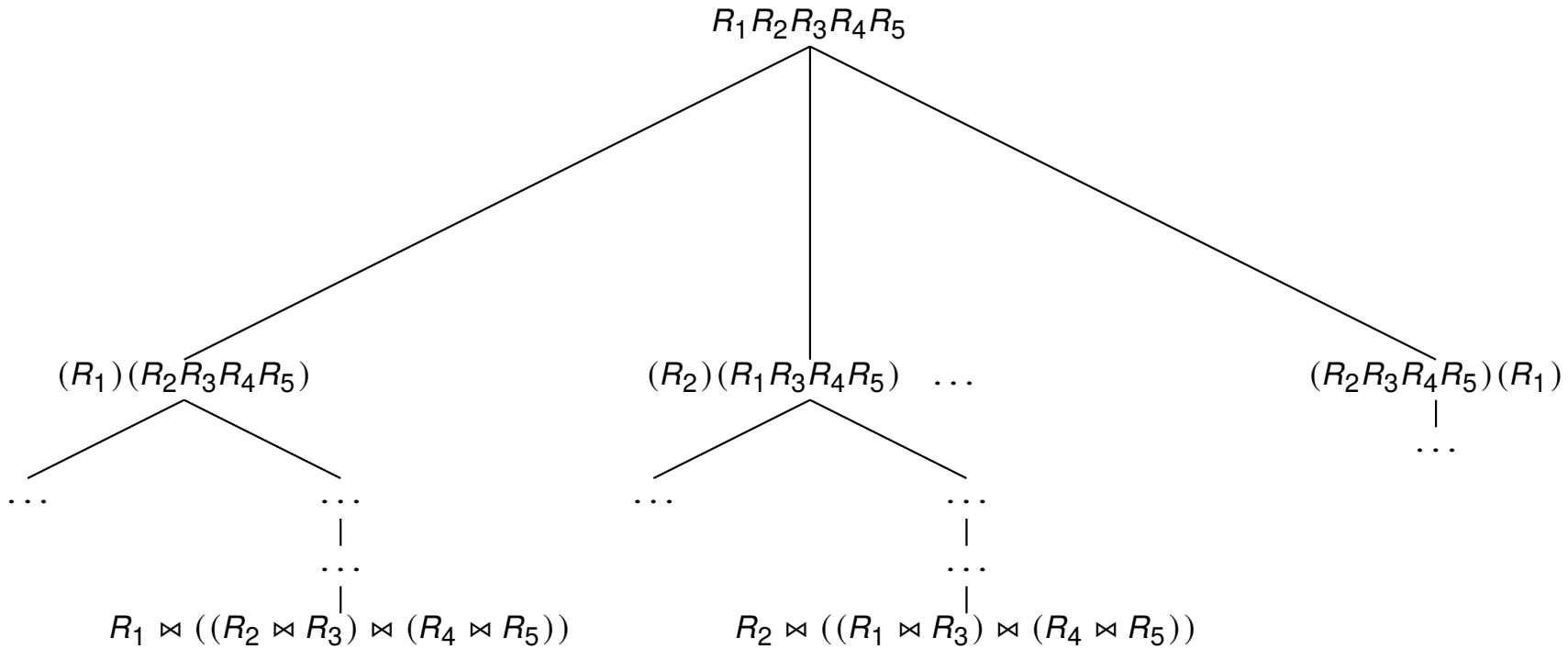
What is the runtime of the top-down algorithm?

Top-Down Algorithm

What is the runtime of the top-down algorithm?

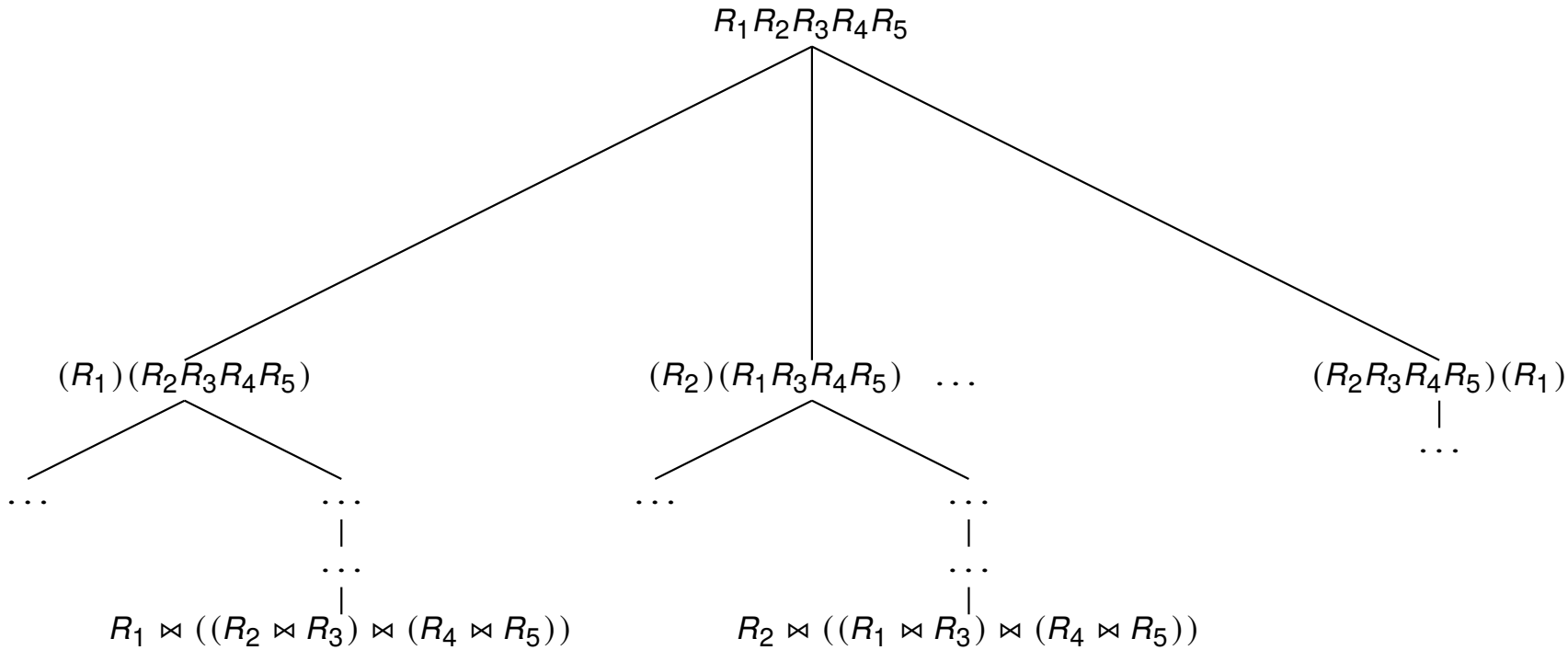
Top-Down Algorithm

What is the runtime of the top-down algorithm?



Top-Down Algorithm

What is the runtime of the top-down algorithm?



Runtime $\approx$ number of leaves $= n!C_{n-1}$.

We can improve it using memoization.

Top-Down Algorithm with Memoization

Set PlanCost[S] := null for all $\emptyset \subsetneq S \subseteq \{R_1, \ldots, R_n\}$.

Function OptimalPlanAndCost(S) $\quad\quad$ $\boldsymbol{R} = \{R_1, \ldots, R_n\}$

if *PlanCost*[S] $\neq$ *null* **then return** PlanCost[S];
if $|S| = 1$ **then**
  (plan, cost) := (scan(R_i), Cost(scan(R_i)));
else
  (plan, cost) := (null, ∞);
  for $S_1 : \emptyset \subsetneq S_1 \subsetneq S$ **do**
    $S_2 := S - S_1$;
    (P_1, c_1) := OptimalPlanAndCost(S_1);
    (P_2, c_2) := OptimalPlanAndCost(S_2);
    $c := c_1 + c_2 + \text{Cost}(P_1 \bowtie P_2)$;
    if $c <$ *cost* **then** (plan, cost) := $(P_1 \bowtie P_2, c)$;
PlanCost[S] := (plan, cost)
return PlanCost[S];

Top-Down Algorithm with Memoization

What is the runtime of the top-down algorithm with memoization?

Top-Down Algorithm with Memoization

What is the runtime of the top-down algorithm with memoization?

Number of pairs of disjoint sets $S_1, S_2 \subseteq \{R_1, \ldots, R_n\} = 3^n$.

Bottom-up Dynamic Programming

Bottom Up Algorithm: Dynamic Programming

- [Selinger et al., 1979]: left linear plans, wont discuss.
- [Ono and Lohman, 1990]: `DPSize`, $O(4^n)$.
- [Vance and Maier, 1996]: `DPSub`, $O(3^n)$.
- [Moerkotte and Neumann, 2006]: `DPccp`. Avoids cross products; runtime depends on topology.
- [Moerkotte and Neumann, 2008]: `DPhyp`. Extension to "hypergraphs" (probably wont discuss).
- [Moerkotte et al., 2013]: DPsube adds non-reordable operators.

DPSize [Ono and Lohman, 1990]

Set PlanCost$[S]$:= (null, ∞) for all $\emptyset \subsetneq S \subseteq \boldsymbol{R} := \{R_1, \ldots, R_n\}$.

Algorithm 1: DPSize

```
for s1 = 1, n − 1 do
    for s2 = 1, n − s1 do
        for S1 ⊆ R : |S1| = s1 do
            for S2 ⊆ R : |S2| = s2 do
                if S1 ∩ S2 ≠ ∅ then continue;
                (P1, c1) := PlanCost[S1];
                (P2, c2) := PlanCost[S2];
                c := c1 + c2 + Cost(P1 ⋈ P2);
                if c < PlanCost[S1 ∪ S2].cost then
                    PlanCost[S1 ∪ S2] := (P1 ⋈ P2, c);
```

There are $2^n - 2$ sets S_1 and $2^n - 2$ sets S_2. Total runtime: $O(4^n)$.

DPSub [Vance and Maier, 1996]

Iterate only over pair of disjoint sets S_1, S_2:

Algorithm 2: DBSub

for $\emptyset \subsetneq S \subseteq \boldsymbol{R}$ **do**
  for $\emptyset \subsetneq S_1 \subsetneq S$ **do**
    $S_2 := S - S_1$;
    $(P_1, c_1) :=$ PlanCost$[S_1]$
    $(P_2, c_2) :=$ PlanCost$[S_2]$
    $c := c_1 + c_2 + \text{Cost}(P_1 \bowtie P_2)$;
    if $c <$ *PlanCost*$[S_1 \cup S_2].cost$ **then**
      PlanCost$[S_1 \cup S_2] := (P_1 \bowtie P_2, c)$;

There are 3^n pairs of disjoint subsets $S_1, S_2 \subseteq \boldsymbol{R}$ and only $3^n - 2^{n+1}$ if we restrict $S_1, S_2 \neq \emptyset$.

DPccp [Moerkotte and Neumann, 2006]

Algorithm 3: DBccp

for $S_1 \subseteq \boldsymbol{R}$, S_1 *connected* **do**
  for $S_2 \subseteq S - S_1$, S_2 *connected and connected to* S_1 **do**
    $(P_1, c_1) := \text{PlanCost}[S_1]$;
    $(P_2, c_2) := \text{PlanCost}[S_2]$;
    $c := c_1 + c_2 + \text{Cost}(P_1 \bowtie P_2)$;
    if $c < \mathit{PlanCost}[S_1 \cup S_2].\mathit{cost}$ **then**
      $\text{PlanCost}[S_1 \cup S_2] := (P_1 \bowtie P_2, c)$;

Problem: how do we enumerate efficiently the pairs (S_1, S_2).

Enumerating the Pairs S_1, S_2

- `EnumerateCsg`: enumerates all connected subgraphs S_1.

- For each S_1: `EnumerateCmp`: enumerates the "complement" S_2: $S_2 \subseteq V - S_1$, S_2 is connected, and is connected to S_1.

EnumerateCsg

[Moerkotte and Neumann, 2006]

Using the definition $\mathcal{B}_i = \{v_j | j \leq i\}$, the pseudocode looks as follows:

`EnumerateCsg`
Input: a connected query graph $G = (V, E)$
Precondition: nodes in V are numbered according to a breadth-first search
Output: emits all subsets of V inducing a connected subgraph of G
for all $i \in [n-1, \ldots, 0]$ **descending** {
    emit $\{v_i\}$;
    EnumerateCsgRec(G, $\{v_i\}$, $\mathcal{B}_i$);
}

EnumerateCsgRec(G, S, X)
$N = \mathcal{N}(S) \setminus X$;
for all $S' \subseteq N$, $S' \neq \emptyset$, enumerate subsets first {
    emit $(S \cup S')$;
}
for all $S' \subseteq N$, $S' \neq \emptyset$, enumerate subsets first {
    EnumerateCsgRec(G, $(S \cup S')$, $(X \cup N)$);
}

(details on next slide)

EnumerateCsg

Algorithm 4: EnumerateCsg

Input: Connected graph $G = (V, E)$; arbitrary order on V: $v_0, v_1, \ldots, v_{n-1}$

Output: Enumerates Connected Subgraphs, in time $O(\text{csg})$; subsets first

for $i = n-1, n-2, \ldots, 1, 0$ **do**

    emit($\{v_i\}$);

    EnumerageCsgRec($G, \{v_i\}, \{v_0, \ldots, v_i\}$);

    ; // Include v_i, exclude $v_j, j \leq i$

EnumerateCsg

Algorithm 5: EnumerateCsg

Input: Connected graph $G = (V, E)$; arbitrary order on V: $v_0, v_1, \ldots, v_{n-1}$

Output: Enumerates Connected Subgraphs, in time $O(\text{csg})$; subsets first

for $i = n-1, n-2, \ldots, 1, 0$ **do**

    emit($\{v_i\}$);

    EnumerageCsgRec($G, \{v_i\}, \{v_0, \ldots, v_i\}$);

    ; // Include v_i, exclude $v_j, j \leq i$

Function EnumerageCsgRec(G, S, X)

Output: Enumerate connected subgraphs $\supseteq S$, disjoint from X

$N := \mathcal{N}(S) - X$; ; // $\mathcal{N}(S)$ = neighbors of S

for $S' \subseteq N, S' \neq \emptyset$, *subsets first* **do**

    emit($S \cup S'$);

    EnumerageCsgRec($G, (S \cup S'), (X \cup N)$);

Which set is emitted first: $\{v_4, v_5\}$ or $\{v_5, v_6\}$?

EnumerateCmp

Original algorithm had a bug. Errata in [Meister et al., 2018]:

Algorithm 1: EnumerateCmp [1]

Precondition: `nodes in` V `are numbered according to an arbitrary enumeration`
Input : a connected query graph $G = (V, E)$, a connected subset S_1
Output : emits all complements S_2 for S_1 such that (S_1, S_2) is a csg-cmp-pair

1 $X = \mathcal{B}_{min(S_1)} \cup S_1$;
2 $N = \mathcal{N}(S_1) \setminus X$;
3 **foreach** $v_i \in N$ *by descending* i **do**
4 | **emit** $\{v_i\}$;
5 | EnumerateCsgRec(G, $\{v_i\}$, $X \cup N \rightarrow X \cup \mathcal{B}_{\mathbf{i}}(\mathbf{N})$);

Algorithm 2: EnumerateCsgRec [1]

Input : a connected query graph $G = (V, E)$, a connected subset S, a set of excluded nodes X
Output: emits connected subsets

1 $N = \mathcal{N}(S) \setminus X$;
2 **foreach** $S' \subseteq N$, $S' \neq \emptyset$ **do**
3 | **emit**$(S \cup S')$;
4 **foreach** $S' \subseteq N$, $S' \neq \emptyset$ **do**
5 | EnumerateCsgRec(G, $(S \cup S')$, $(X \cup N)$);

(details on next slide)

EnumerateCmp

Algorithm 6: EnumerateCmp

Input: Connected graph $G = (V, E)$, connected subset S_1.

Output: Enumerate all connected subsets $S_2 \subseteq V - S_1$, connected to S_1.

$k := \min\{j \mid v_j \in S_1\}$;

$X := \{v_0, \ldots, v_k\}$;

$N := \mathcal{N}(S_1) - X$;

for $v_i \in N$, *by descending* i **do**

    emit($\{v_i\}$);

    EnumerateCsgRec($G, \{v_i\}, X \cup \{v_j \in N \mid j \leq i\}$);

Discussion

- DPsize, DPsub, DPccp.
- It is important to enumerate *only* the connected component pairs. Tricky!
- Enumerating efficiently $S \subseteq \{v_0, v_1, \ldots, v_{n-1}\}$:
  - Use counter $k = 0, 2^n - 1$;
  - Convert k to a set using binary representation;
  - $O(1)$ per set generation.[Vance and Maier, 1996]
- So far the algorithm handles only joins.
- Next lecture: handling non-reordable operators (outer, semi, anti joins).

Birler, A. and Neumann, T. (2025).

Efficient enumeration of the complete join search space.
In *Proceedings of the 19th International Symposium on Database Programming Languages, DBPL 2025, Berlin, Germany, June 22-27, 2025*, pages 5:1–5:12. ACM.

Meister, A., Moerkotte, G., and Saake, G. (2018).

Errata for ”analysis of two existing and one new dynamic programming algorithm for the generation of optimal bushy join trees without cross products”.
Proc. VLDB Endow., 11(10):1069–1070.

Moerkotte, G., Fender, P., and Eich, M. (2013).

On the correct and complete enumeration of the core search space.
In Ross, K. A., Srivastava, D., and Papadias, D., editors, *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2013, New York, NY, USA, June 22-27, 2013*, pages 493–504. ACM.

Moerkotte, G. and Neumann, T. (2006).

Analysis of two existing and one new dynamic programming algorithm for the generation of optimal bushy join trees without cross products.
In Dayal, U., Whang, K., Lomet, D. B., Alonso, G., Lohman, G. M., Kersten, M. L., Cha, S. K., and Kim, Y., editors, *Proceedings of the 32nd International Conference on Very Large Data Bases, Seoul, Korea, September 12-15, 2006*, pages 930–941. ACM.

Moerkotte, G. and Neumann, T. (2008).

Dynamic programming strikes back.
In Wang, J. T., editor, *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2008, Vancouver, BC, Canada, June 10-12, 2008*, pages 539–552. ACM.

Neumann, T. (2024).

A formalization of top-down unnesting.
CoRR, abs/2412.04294.

Neumann, T. (2025).

Improving unnesting of complex queries.

In Klettke, M., Schenkel, R., Henrich, A., Nicklas, D., Schüle, M. E., and Meyer-Wegener, K., editors, *Datenbanksysteme für Business, Technologie und Web (BTW 2025), 21. Fachtagung des GI-Fachbereichs "Datenbanken und Informationssysteme" (DBIS), 03.-07, März 2025, Bamberg, Germany, Proceedings*, volume P-361 of *LNI*, pages 25–47. Gesellschaft für Informatik e.V.

Neumann, T. and Kemper, A. (2015).

Unnesting arbitrary queries.
In Seidl, T., Ritter, N., Schöning, H., Sattler, K., Härder, T., Friedrich, S., and Wingerath, W., editors, *Datenbanksysteme für Business, Technologie und Web (BTW), 16. Fachtagung des GI-Fachbereichs "Datenbanken und Informationssysteme" (DBIS), 4.-6.3.2015 in Hamburg, Germany. Proceedings*, volume P-241 of *LNI*, pages 383–402. GI.

Neumann, T. and Radke, B. (2018).

Adaptive optimization of very large join queries.
In Das, G., Jermaine, C. M., and Bernstein, P. A., editors, *Proceedings of the 2018 International Conference on Management of Data, SIGMOD Conference 2018, Houston, TX, USA, June 10-15, 2018*, pages 677–692. ACM.

Ono, K. and Lohman, G. M. (1990).

Measuring the complexity of join enumeration in query optimization.
In McLeod, D., Sacks-Davis, R., and Schek, H., editors, *16th International Conference on Very Large Data Bases, August 13-16, 1990, Brisbane, Queensland, Australia, Proceedings*, pages 314–325. Morgan Kaufmann.

Selinger, P. G., Astrahan, M. M., Chamberlin, D. D., Lorie, R. A., and Price, T. G. (1979).

Access path selection in a relational database management system.
In Bernstein, P. A., editor, *Proceedings of the 1979 ACM SIGMOD International Conference on Management of Data, Boston, Massachusetts, USA, May 30 - June 1*, pages 23–34. ACM.

Vance, B. and Maier, D. (1996).

Rapid bushy join-order optimization with cartesian products.
In Jagadish, H. V. and Mumick, I. S., editors, *Proceedings of the 1996 ACM SIGMOD International Conference on Management of Data, Montreal, Quebec, Canada, June 4-6, 1996*, pages 35–46. ACM Press.