Carnegie Mellon University

OPTIMIZE!

Database Query Optimization

Join Ordering: Top-Down

SPRING 2025 » SPECIAL TOPICS IN DATABASES » PROF. ANDY PAVLO

ADMINISTRIVIA

Paper Reviews resume this Wednesday Feb 5th

Project #1 is due Friday Feb 28th

UPCOMING DATABASE TALKS

Convex (DB Seminar)
→ Monday Feb 10th @ 4:30pm ET
→ Zoom

The Germans (DB Seminar)
→ Monday Feb 17th @ 4:30pm ET
→ Zoom

Pinot (DB Seminar)
→ Monday Feb 24th @ 4:30pm ET
→ Zoom

ERRATA

Clarification of the Dynamic Programming with Hypergraph Algorithm (DPHyp).

Send Corrections: **db-mistakes@cs.cmu.edu**

DYNAMIC PROGRAMMING HYPERGRAPH (DPHYP)

Model the query as a hypergraph and then incrementally expand to enumerate new plans.

Algorithm Overview:

- → Iterate connected sub-graphs and incrementally add new edges to other nodes to complete query plan.
- → Use rules to determine which nodes the traversal is allowed to visit and expand.

DYNAMIC PROGRAMMING STRIKES BACK
SIGMOD 2008

DPHYP: HYPERGRAPHS

A hypergraph is a pair $H=(V,E)$ such that:

- → V is a non-empty set of nodes.
- → E is a set of hyperedges, where a hyperedge is an unordered pair (u,v) of non-empty subsets of V ($u \subset V$, $v \subset V$) with the additional condition that $u \cap v = \emptyset$.

Allows search algorithm to consider node groupings instead of each individual node.

```
SELECT * FROM R1, R2, R3, R4, R5, R6
WHERE R1.a = R2.a
  AND R2.b = R3.c
  AND R4.d = R5.d
  AND R5.e = R6.e
  AND abs(R1.f + R3.f) =
      abs(R4.g + R6.g)
```

DYNAMIC PROGRAMMING STRIKES BACK
SIGMOD 2008

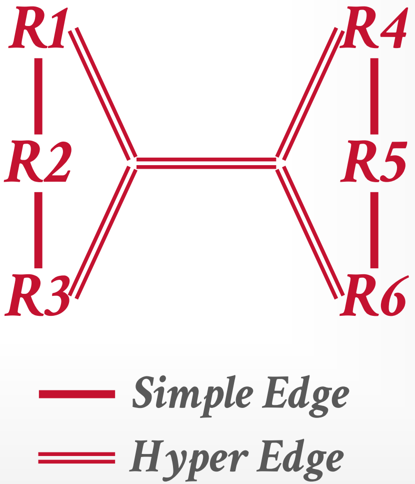
DPHYP: HYPERGRAPHS

A hypergraph is a pair $H=(V,E)$ such that:

- → V is a non-empty set of nodes.
- → E is a set of hyperedges, where a hyperedge is an unordered pair (u,v) of non-empty subsets of V ($u \subset V$, $v \subset V$) with the additional condition that $u \cap v = \emptyset$.

Allows search algorithm to consider node groupings instead of each individual node.

```
SELECT * FROM R1, R2, R3, R4, R5, R6
WHERE R1.a = R2.a
  AND R2.b = R3.c
  AND R4.d = R5.d
  AND R5.e = R6.e
  AND abs(R1.f + R3.f) =
      abs(R4.g + R6.g)
```

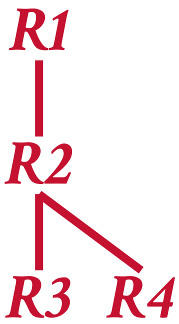



DYNAMIC PROGRAMMING STRIKES BACK
SIGMOD 2008

DHYP: BASIC ALGORITHM

Enumerate all connected subgraphs of the query graph.

For each subgraph, enumerate all other connected subgraphs that are disjoint but connected to it.
→ Start with one node and expand recursively by following edges.

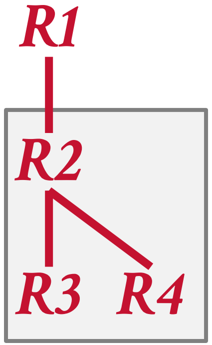


Source: Thomas Neumann

DHYP: BASIC ALGORITHM

Enumerate all connected subgraphs of the query graph.

For each subgraph, enumerate all other connected subgraphs that are disjoint but connected to it.
→ Start with one node and expand recursively by following edges.



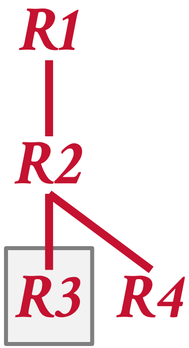
Source: Thomas Neumann

DHYP: BASIC ALGORITHM

Enumerate all connected subgraphs of the query graph.

For each subgraph, enumerate all other connected subgraphs that are disjoint but connected to it.

→ Start with one node and expand recursively by following edges.



Source: Thomas Neumann

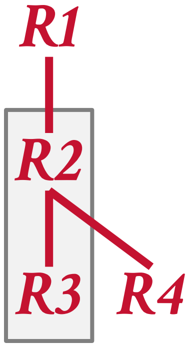
R3

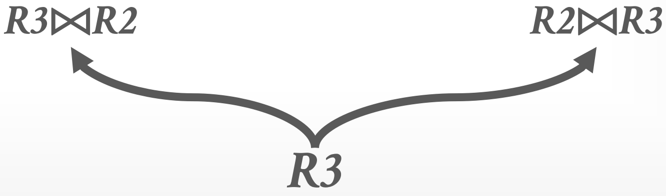
DHYP: BASIC ALGORITHM

Enumerate all connected subgraphs of the query graph.

For each subgraph, enumerate all other connected subgraphs that are disjoint but connected to it.

→ Start with one node and expand recursively by following edges.





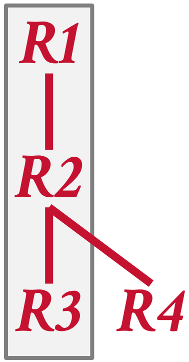
Source: Thomas Neumann

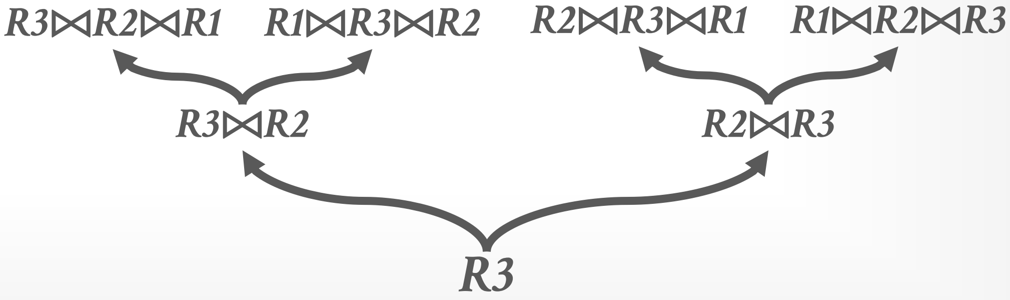
DHYP: BASIC ALGORITHM

Enumerate all connected subgraphs of the query graph.

For each subgraph, enumerate all other connected subgraphs that are disjoint but connected to it.

→ Start with one node and expand recursively by following edges.





$R3 \bowtie R2 \bowtie R1$ $R1 \bowtie R3 \bowtie R2$ $R2 \bowtie R3 \bowtie R1$ $R1 \bowtie R2 \bowtie R3$

$R3 \bowtie R2$ $R2 \bowtie R3$

$R3$

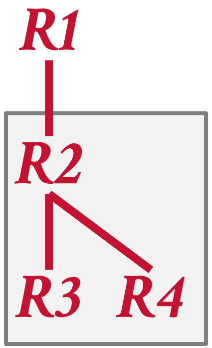
Source: Thomas Neumann

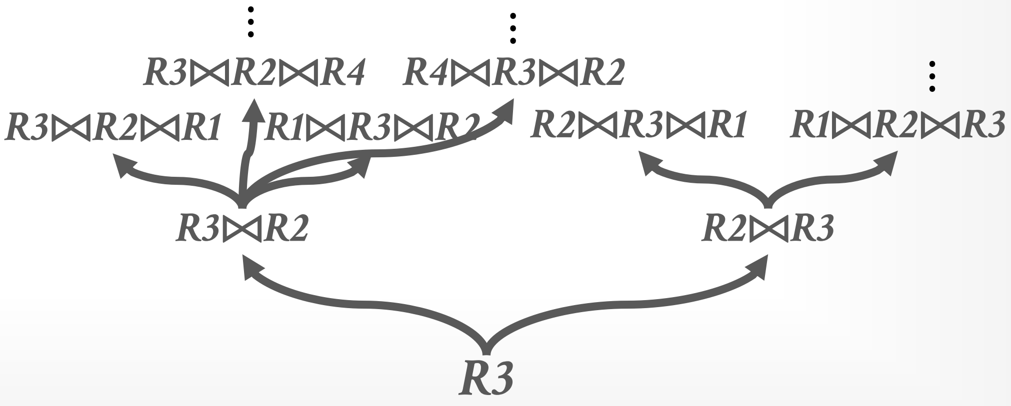
DHYP: BASIC ALGORITHM

Enumerate all connected subgraphs of the query graph.

For each subgraph, enumerate all other connected subgraphs that are disjoint but connected to it.

→ Start with one node and expand recursively by following edges.





Source: Thomas Neumann

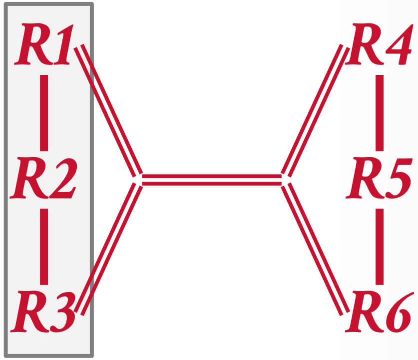
DHYP: NOW WITH HYPERGRAPHS

Since hyperedges are ***n*:*m*** edges, adding them to a subgraph connects additional nodes.

Where to expand to and from {**R1**,**R2**,**R3**} while still guaranteeing DP order?

→ Adding **R4** causes **R6** to be disconnected from the new graph.

Recursively expand subgraph to cover all nodes in a hyperedge.



Source: Thomas Neumann

LAST CLASS

Defining a query's complexity based on the structure of its join graph rather than the number of relations that it references.

Bottom-Up Join Enumeration

- → Adapting search strategy based on query complexity.
- → Using approximations and simplifications to initialize search algorithm.

OBSERVATION

Top-down search enables enhancements that are not compatible with bottom-up DP algorithms:

→ Demand-driven interesting orders
→ Branch-and-bound pruning
→ Exploiting partial plan information

But top-down search has other problems:

→ Must store all generated plans and not just optimal ones.
→ No optimal enumeration method that generate plans for any query without Cartesian products.

This is what today's paper solves!

TODAY'S AGENDA

Partition-based Top-Down Join Enumeration

Branch-and-Bound Pruning Strategies

Top-Down Hypergraph Join Enumeration

OPTIMAL TOP-DOWN PARTITIONING (OTDP)

```
SELECT * FROM A, B, C, D, E
 WHERE A.a_id = B.a_id
   AND B.c_id = C.c_id
   AND B.d_id = D.d_id
   AND D.e_id = E.e_id;
```

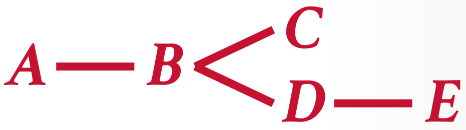
Recursively split the join graph into smaller partitions. Then choose the optimal ordering for progressively larger partitions.

Query plan quality is highly dependent on partitioning scheme.

The algorithm's optimality is not based on the query plan…

OPTIMAL TOP-DOWN JOIN ENUMERATION
SIGMOD 2007

OPTIMAL TOP-DOWN PARTITIONING (OTDP)



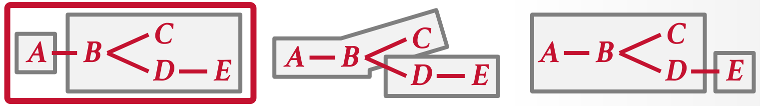
Recursively split the join graph into smaller partitions. Then choose the optimal ordering for progressively larger partitions.

Query plan quality is highly dependent on partitioning scheme.

The algorithm's optimality is <u>not</u> based on the query plan…

OPTIMAL TOP-DOWN JOIN ENUMERATION
SIGMOD 2007

OPTIMAL TOP-DOWN PARTITIONING (OTDP)



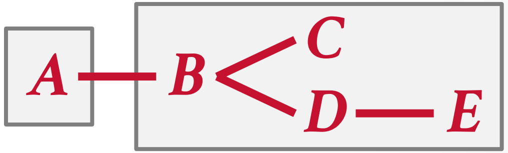
Recursively split the join graph into smaller partitions. Then choose the optimal ordering for progressively larger partitions.

Query plan quality is highly dependent on partitioning scheme.

The algorithm's optimality is not based on the query plan…

OPTIMAL TOP-DOWN JOIN ENUMERATION
SIGMOD 2007

OPTIMAL TOP-DOWN PARTITIONING (OTDP)



Recursively split the join graph into smaller partitions. Then choose the optimal ordering for progressively larger partitions.

Query plan quality is highly dependent on partitioning scheme.

The algorithm's optimality is <u>not</u> based on the query plan…

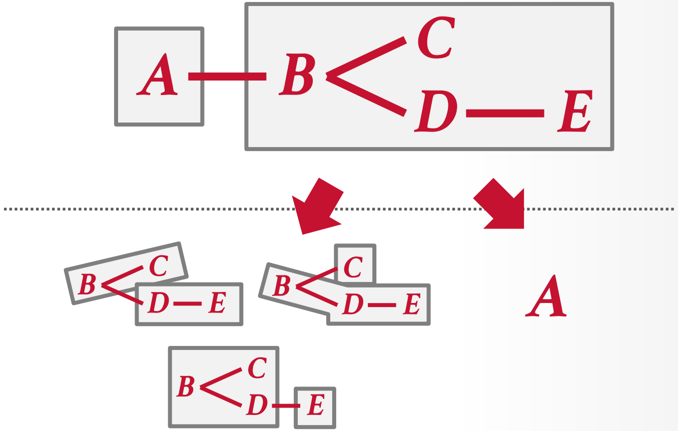
OPTIMAL TOP-DOWN JOIN ENUMERATION
SIGMOD 2007

OPTIMAL TOP-DOWN PARTITIONING (OTDP)

Recursively split the join graph into smaller partitions. Then choose the optimal ordering for progressively larger partitions.

Query plan quality is highly dependent on partitioning scheme.

The algorithm's optimality is not based on the query plan…



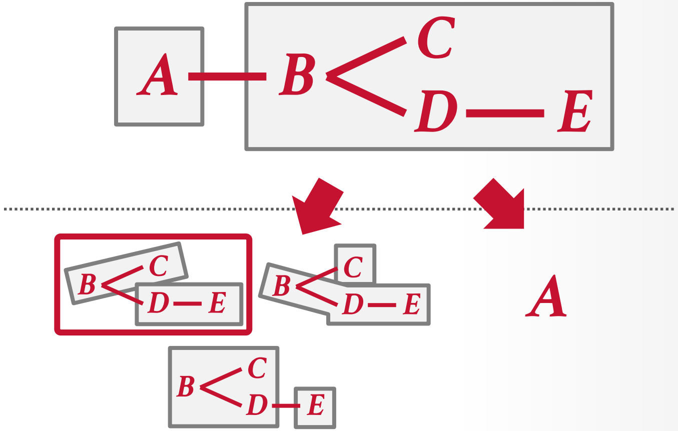
OPTIMAL TOP-DOWN JOIN ENUMERATION
SIGMOD 2007

OPTIMAL TOP-DOWN PARTITIONING (OTDP)

Recursively split the join graph into smaller partitions. Then choose the optimal ordering for progressively larger partitions.

Query plan quality is highly dependent on partitioning scheme.

The algorithm's optimality is <u>not</u> based on the query plan…



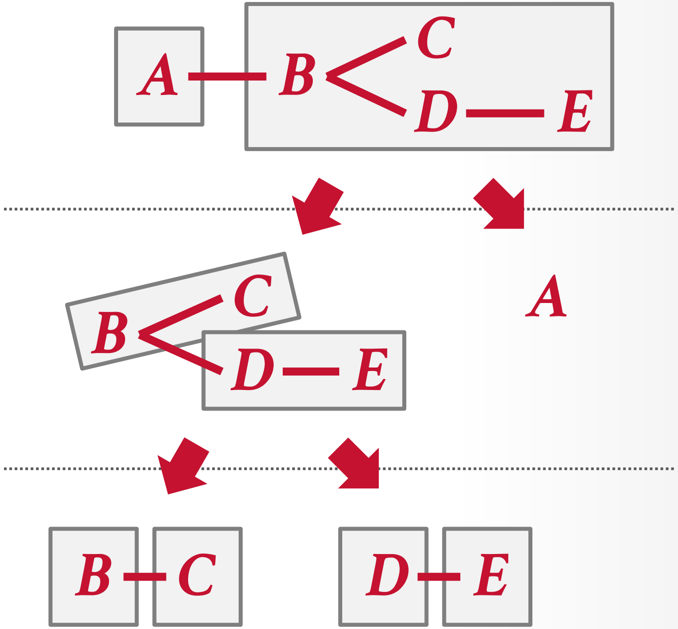
OPTIMAL TOP-DOWN JOIN ENUMERATION
SIGMOD 2007

OPTIMAL TOP-DOWN PARTITIONING (OTDP)

Recursively split the join graph into smaller partitions. Then choose the optimal ordering for progressively larger partitions.

Query plan quality is highly dependent on partitioning scheme.

The algorithm's optimality is <u>not</u> based on the query plan…



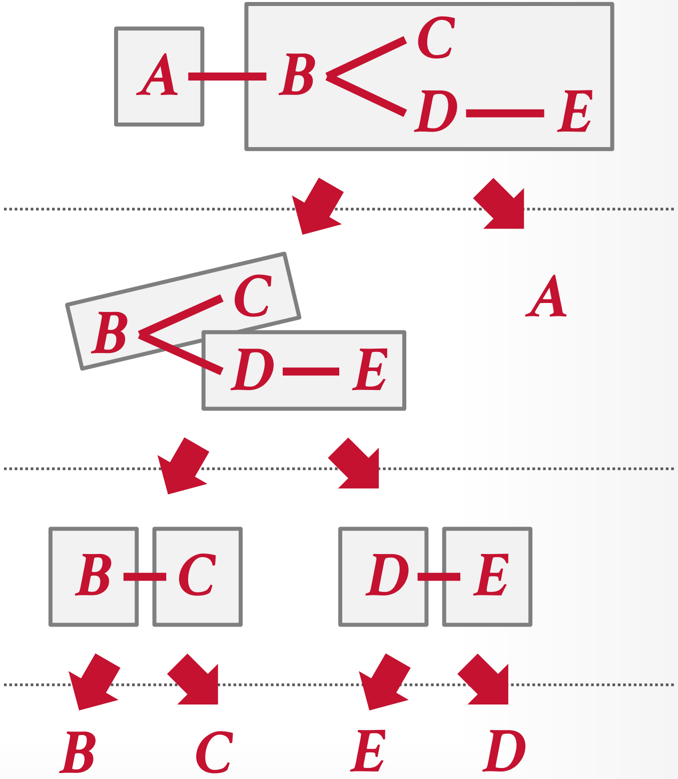

OPTIMAL TOP-DOWN JOIN ENUMERATION
SIGMOD 2007

OPTIMAL TOP-DOWN PARTITIONING (OTDP)

Recursively split the join graph into smaller partitions. Then choose the optimal ordering for progressively larger partitions.

Query plan quality is highly dependent on partitioning scheme.

The algorithm's optimality is not based on the query plan…



OPTIMAL TOP-DOWN JOIN ENUMERATION
SIGMOD 2007

OPTIMALITY

1990 Definition:

→ A join enumeration algorithm only enumerates the minimum number of join operators.

2006 Definition:

→ A join enumeration algorithm incurs no more than linear time overhead between enumerated join operators for any join graph.

MEASURING THE COMPLEXITY OF JOIN ENUMERATION IN QUERY OPTIMIZATION
VLDB 1990

ANALYSIS OF TWO EXISTING AND ONE NEW DYNAMIC PROGRAMMING ALGORITHM FOR THE GENERATION OF OPTIMAL BUSHY JOIN TREES WITHOUT CROSS PRODUCTS
VLDB 2006

OTDP: GRAPH ANALYSIS COST

The enumeration algorithm will repeatedly perform set operations on the join graph during its search.

→ Example: Check whether edge ***e*** exists in graph ***G***.

The computational cost of analyzing the join graph depends on how the optimizer encodes the graph and the efficiency of those operations.

GetBestPlan(G,o)

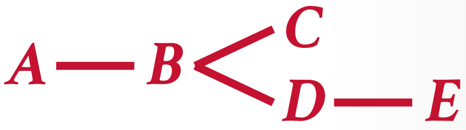
```
Input: join graph $G=(V,E)$
Input: interesting order $o$
Output: best plan satisfying $o$

bestPlan←∅
for partition $(G_L,G_R) \in$ Partition(G):
  for operator $G_L \bowtie_i G_R$ satisfying $o$:
    $o_L$←order for $G_L$ required by $\bowtie_i$
    $p_L$←GetBestPlan($G_L$,$o_L$)
    $o_R$←order for $G_R$ required by $\bowtie_i$
    $p_R$←GetBestPlan($G_R$,$o_R$)
    curPlan←($p_L \bowtie_i p_R$)
    if Cost(curPlan) < Cost(bestPlan):
      bestPlan←curPlan
return bestPlan
```

OTDP: GRAPH ENCODING

Option #1: Edge-List Encoding

→ Maintain a list of vertex pairs to represent the edges in the graph ***G***.

→ Set operations execute in constant time.



Edge List

[(*A*,*B*), (*B*,*C*), (*B*,*D*), (*D*,*E*)]

Option #2: Array of Bitmaps

→ For each vertex in ***G***, maintain a bitmap where a bit is set to true if that vertex is connected to another vertex by an edge.

→ Enables the use of bit-wise machine instructions for fast set operations.

Edge Bitmaps

	A	B	C	D	E
A:	0	1	0	0	0
B:	1	0	1	1	0
C:	0	1	0	0	0
D:	0	1	0	0	1
E:	0	0	0	1	0

NAÏVE PARTITIONING ALGORITHM

#1: Left-Deep with Cart. Products

→ Partition graph by removing each vertex on at a time.

LeftDeepPartition(G)

```
Input: join graph G=(V,E)
Output: partitions of G

for v ∈ V:
  output (G|(V\{v}), G|{v})
```

NAÏVE PARTITIONING ALGORITHM

#1: Left-Deep with Cart. Products

→ Partition graph by removing each vertex on at a time.

#2: Left-Deep w/o Cart. Products

→ Check whether removing a vertex in #1 would cause a Cartesian product join.

LeftDeepPartition(G)

```
Input: join graph $G=(V,E)$
Output: partitions of $G$

for $v \in V$:
    output $(G|_{(V\setminus\{v\})}, G|_{\{v\}})$
```

NAÏVE PARTITIONING ALGORITHM

#1: Left-Deep with Cart. Products

→ Partition graph by removing each vertex on at a time.

#2: Left-Deep w/o Cart. Products

→ Check whether removing a vertex in #1 would cause a Cartesian product join.

LeftDeepPartition(G)

```
Input: join graph $G=(V,E)$
Output: partitions of $G$

for $v \in V$:
  if $G|_{(V\setminus\{v\}}$ is connected:
    output $(G|_{(V\setminus\{v\})}, G|_{\{v\}})$
```

NAÏVE PARTITIONING ALGORITHM

#1: Left-Deep with Cart. Products

→ Partition graph by removing each vertex on at a time.

#2: Left-Deep w/o Cart. Products

→ Check whether removing a vertex in #1 would cause a Cartesian product join.

#

→ Partition graph on non-empty, strict subsets ***S*** of ***V***.

LeftDeepPartition(G)

```
Input: join graph $G=(V,E)$
Output: partitions of $G$

for $v \in V$:
  if $G|_{(V\setminus\{v\})}$ is connected:
    output $(G|_{(V\setminus\{v\})}, G|_{\{v\}})$
```

BushyPartition(G)

```
Input: join graph $G=(V,E)$
Output: partitions of $G$

for non-empty subsets $S \in V$:
  output $(G|_{(V\setminus S)}, G|_{S})$
```

NAÏVE PARTITIONING ALGORITHM

#1: Left-Deep with Cart. Products

→ Partition graph by removing each vertex on at a time.

#2: Left-Deep w/o Cart. Products

→ Check whether removing a vertex in #1 would cause a Cartesian product join.

#

→ Partition graph on non-empty, strict subsets ***S*** of ***V***.

LeftDeepPartition(G)

```
Input: join graph G=(V,E)
Output: partitions of G

for v ∈ V:
  if $G|_{(V\setminus\{v\}}$ is connected:
    output ($G|_{(V\setminus\{v\})}$, $G|_{\{v\}}$)
```

BushyPartition(G)

```
Input: join graph G=(V,E)
Output: partitions of G

for non-empty subsets S ∈ V:
  output ($G|_{(V\setminus S)}$, $G|_{S}$)
```

NAÏVE PARTITIONING ALGORITHM

#1: Left-Deep with Cart. Products

→ Partition graph by removing each vertex on at a time.

#2: Left-Deep w/o Cart. Products

→ Check whether removing a vertex in #1 would cause a Cartesian product join.

#

→ Partition graph on non-empty, strict subsets ***S*** of ***V***.

#4: Bushy Plans w/o Cart. Products

→ Check whether the two subsets from #3 will cause a Cartesian products.

LeftDeepPartition(G)

```
Input: join graph G=(V,E)
Output: partitions of G

for v ∈ V:
  if G|(V\{v}) is connected:
    output (G|(V\{v}), G|{v})
```

BushyPartition(G)

```
Input: join graph G=(V,E)
Output: partitions of G

for non-empty subsets S ∈ V:
  output (G|(V\S), G|S)
```

NAÏVE PARTITIONING ALGORITHM

#1: Left-Deep with Cart. Products

→ Partition graph by removing each vertex on at a time.

#2: Left-Deep w/o Cart. Products

→ Check whether removing a vertex in #1 would cause a Cartesian product join.

#

→ Partition graph on non-empty, strict subsets ***S*** of ***V***.

#4: Bushy Plans w/o Cart. Products

→ Check whether the two subsets from #3 will cause a Cartesian products.

LeftDeepPartition(G)

```
Input: join graph $G=(V,E)$
Output: partitions of $G$

for $v \in V$:
  if $G|_{(V\setminus\{v\})}$ is connected:
    output $(G|_{(V\setminus\{v\})}, G|_{\{v\}})$
```

BushyPartition(G)

```
Input: join graph $G=(V,E)$
Output: partitions of $G$

for non-empty subsets $S \in V$:
  output $(G|_{(V\setminus S)}, G|_{S})$
```

NAÏVE PARTITIONING ALGORITHM

#1: Left-Deep with Cart. Products

→ Partition graph by removing each vertex on at a time.

#2: Left-Deep w/o Cart. Products

→ Check whether removing a vertex in #1 would cause a Cartesian product join.

#

→ Partition graph on non-empty, strict subsets ***S*** of ***V***.

#4: Bushy Plans w/o Cart. Products

→ Check whether the two subsets from #3 will cause a Cartesian products.

LeftDeepPartition(G)

```
Input: join graph $G=(V,E)$
Output: partitions of $G$

for $v \in V$:
  if $G|_{(V\setminus\{v\}}$ is connected:
    output $(G|_{(V\setminus\{v\})}, G|_{\{v\}})$
```

BushyPartition(G)

```
Input: join graph $G=(V,E)$
Output: partitions of $G$

for non-empty subsets $S \in V$:
  if $G|_S$ is connected &&
   ↳ $G|_{(V\setminus S)}$ is connected:
    output $(G|_{(V\setminus S)}, G|_S)$
```

OBSERVATION

The previous methods for avoiding Cartesian products in the naïve partitioning algorithms does not exploit the join graph's structure.

OBSERVATION

The previous methods for avoiding Cartesian products in the naïve partitioning algorithms does not exploit the join graph's structure.

A better approach is to identify bad choices upfront and then avoid them in the selection process.

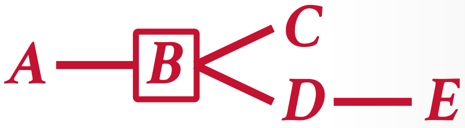
- → L
- → Need to consider edges not vertexes for bushy plans…
- → exes in G and then avoid them in the partitioning algorithm.

MIN-CUT PARTITIONING ALGORITHM

Generate partitions by selecting an edge set to remove from a graph ***G*** to divide ***G*** into two or more connected sub-graphs.

→ Start with a random vertex

→ Lazily build a **biconnection tree** to quickly identify edges to remove.



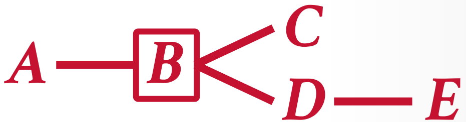
Explore the tree in a depth-first fashion by choosing edges to remove to partition the graph.

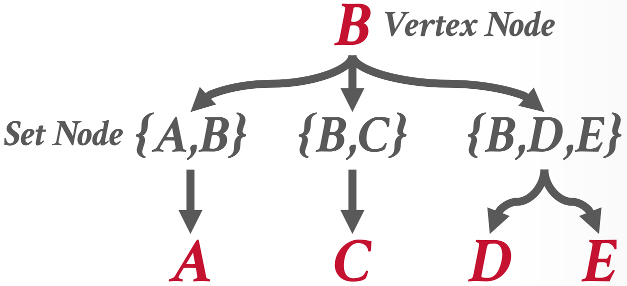
MIN-CUT PARTITIONING ALGORITHM

Generate partitions by selecting an edge set to remove from a graph ***G*** to divide ***G*** into two or more connected sub-graphs.

→ Start with a random vertex

→ Lazily build a **biconnection tree** to quickly identify edges to remove.





Explore the tree in a depth-first fashion by choosing edges to remove to partition the graph.

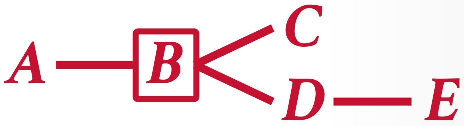
MIN-CUT PARTITIONING ALGORITHM

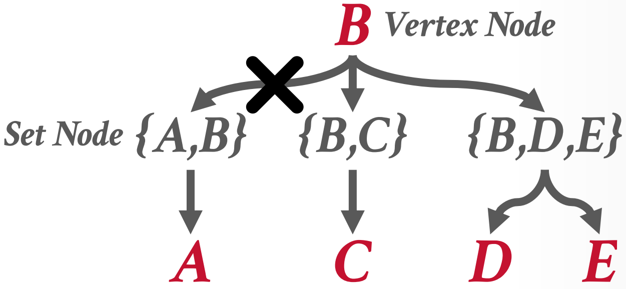
Generate partitions by selecting an edge set to remove from a graph ***G*** to divide ***G*** into two or more connected sub-graphs.

→ Start with a random vertex

→ Lazily build a **biconnection tree** to quickly identify edges to remove.

Explore the tree in a depth-first fashion by choosing edges to remove to partition the graph.





BRANCH-AND-BOUND PRUNING

Another important consideration in top-down enumeration is how to prune branches that will produce a query plan that is worse than the best plan found so far.

→ Good pruning reduces wasted computation.
→ Bad pruning prevents escaping local minimums.

Option #1: Accumulated-cost Bounding

Option #2: Predicted-cost Bounding

OPTIMAL TOP-DOWN JOIN ENUMERATION
SIGMOD 2007

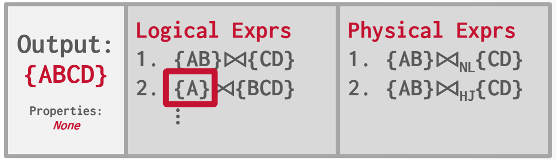
ACCUMULATED-COST BOUNDING

The upper-bound (***U***) is the cost of the best <u>complete</u> physical plan found so far in the entire search tree.

The lower-bound (***L***) is the summation of the physical operators as the optimizer traverses down the search tree.

Upper-Bound: 100

Cost: 50



Output:
{ABCD}
Properties:
None

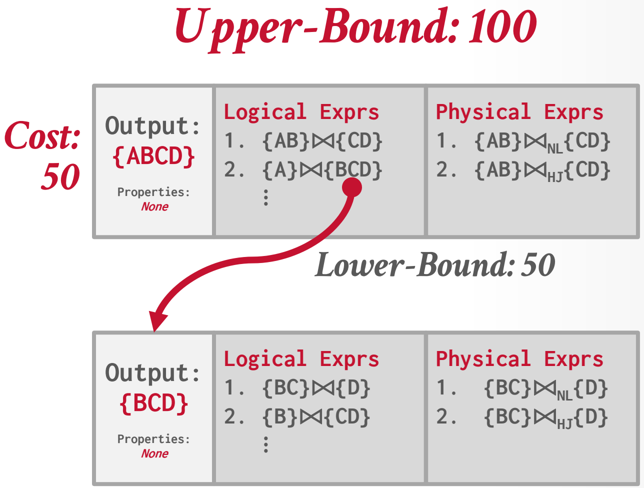
Logical Exprs
1. {AB}⋈{CD}
2. {A}⋈{BCD}
⋮

Physical Exprs
1. $\{AB\}\bowtie_{NL}\{CD\}$
2. $\{AB\}\bowtie_{HJ}\{CD\}$

ACCUMULATED-COST BOUNDING

The upper-bound (***U***) is the cost of the best complete physical plan found so far in the entire search tree.

The lower-bound (***L***) is the summation of the physical operators as the optimizer traverses down the search tree.



Upper-Bound: 100

Cost: 50

Output:
{ABCD}
Properties:
None

Logical Exprs
1. $\{AB\}\bowtie\{CD\}$
2. $\{A\}\bowtie\{BCD\}$
⋮

Physical Exprs
1. $\{AB\}\bowtie_{NL}\{CD\}$
2. $\{AB\}\bowtie_{HJ}\{CD\}$

Lower-Bound: 50

Output:
{BCD}
Properties:
None

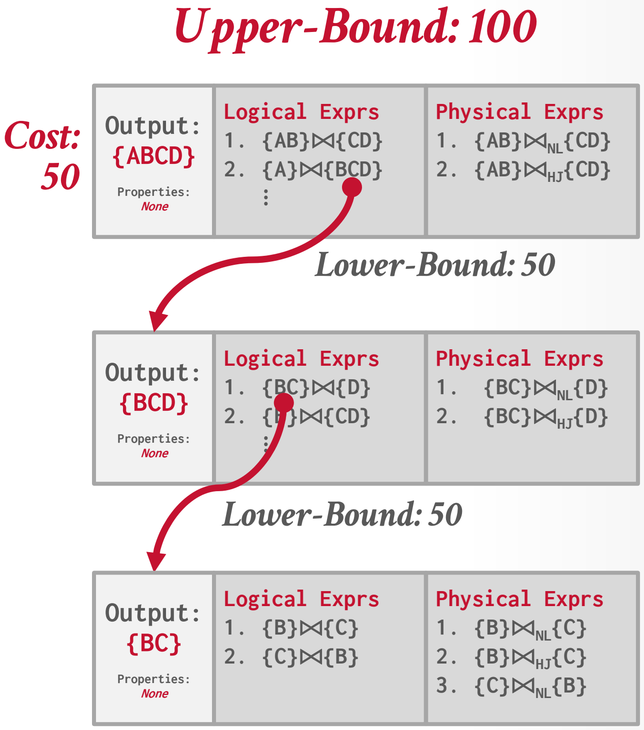
Logical Exprs
1. $\{BC\}\bowtie\{D\}$
2. $\{B\}\bowtie\{CD\}$
⋮

Physical Exprs
1. $\{BC\}\bowtie_{NL}\{D\}$
2. $\{BC\}\bowtie_{HJ}\{D\}$

ACCUMULATED-COST BOUNDING

The upper-bound (***U***) is the cost of the best complete physical plan found so far in the entire search tree.

The lower-bound (***L***) is the summation of the physical operators as the optimizer traverses down the search tree.



Upper-Bound: 100

Cost: 50

Output: {ABCD}
Properties: *None*

Logical Exprs
1. $\{AB\}\bowtie\{CD\}$
2. $\{A\}\bowtie\{BCD\}$
⋮

Physical Exprs
1. $\{AB\}\bowtie_{NL}\{CD\}$
2. $\{AB\}\bowtie_{HJ}\{CD\}$

Lower-Bound: 50

Output: {BCD}
Properties: *None*

Logical Exprs
1. $\{BC\}\bowtie\{D\}$
2. $\{B\}\bowtie\{CD\}$
⋮

Physical Exprs
1. $\{BC\}\bowtie_{NL}\{D\}$
2. $\{BC\}\bowtie_{HJ}\{D\}$

Lower-Bound: 50

Output: {BC}
Properties: *None*

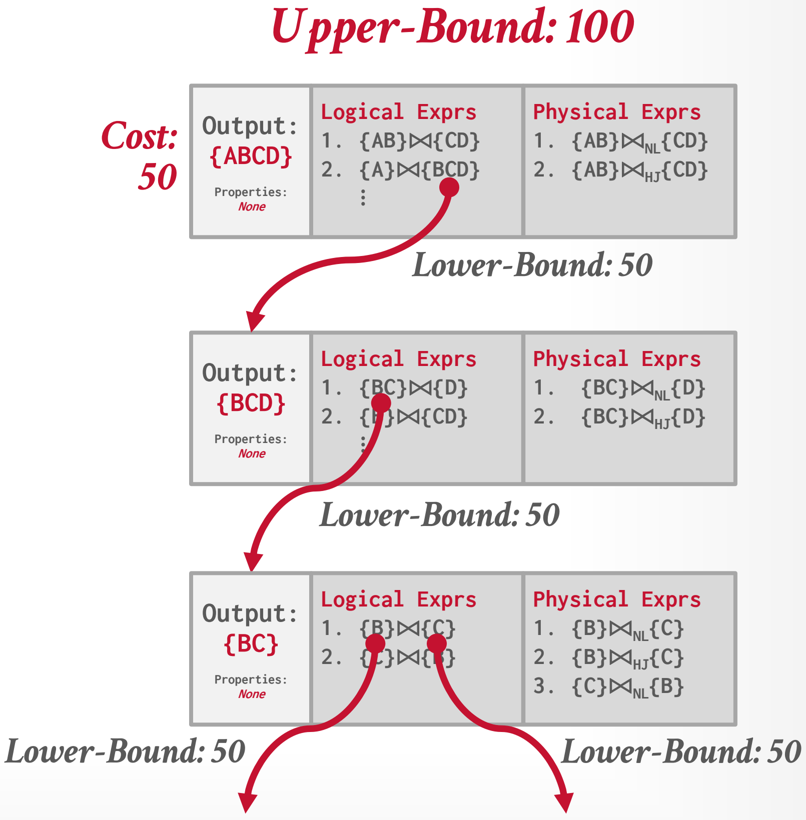
Logical Exprs
1. $\{B\}\bowtie\{C\}$
2. $\{C\}\bowtie\{B\}$

Physical Exprs
1. $\{B\}\bowtie_{NL}\{C\}$
2. $\{B\}\bowtie_{HJ}\{C\}$
3. $\{C\}\bowtie_{NL}\{B\}$

ACCUMULATED-COST BOUNDING

The upper-bound (***U***) is the cost of the best complete physical plan found so far in the entire search tree.

The lower-bound (***L***) is the summation of the physical operators as the optimizer traverses down the search tree.



Upper-Bound: 100

Cost: 50

Output: {ABCD}
Properties: *None*

Logical Exprs
1. $\{AB\} \bowtie \{CD\}$
2. $\{A\} \bowtie \{BCD\}$

⋮

Physical Exprs
1. $\{AB\} \bowtie_{NL} \{CD\}$
2. $\{AB\} \bowtie_{HJ} \{CD\}$

Lower-Bound: 50

Output: {BCD}
Properties: *None*

Logical Exprs
1. $\{BC\} \bowtie \{D\}$
2. $\{B\} \bowtie \{CD\}$

⋮

Physical Exprs
1. $\{BC\} \bowtie_{NL} \{D\}$
2. $\{BC\} \bowtie_{HJ} \{D\}$

Lower-Bound: 50

Output: {BC}
Properties: *None*

Logical Exprs
1. $\{B\} \bowtie \{C\}$
2. $\{C\} \bowtie \{B\}$

Physical Exprs
1. $\{B\} \bowtie_{NL} \{C\}$
2. $\{B\} \bowtie_{HJ} \{C\}$
3. $\{C\} \bowtie_{NL} \{B\}$

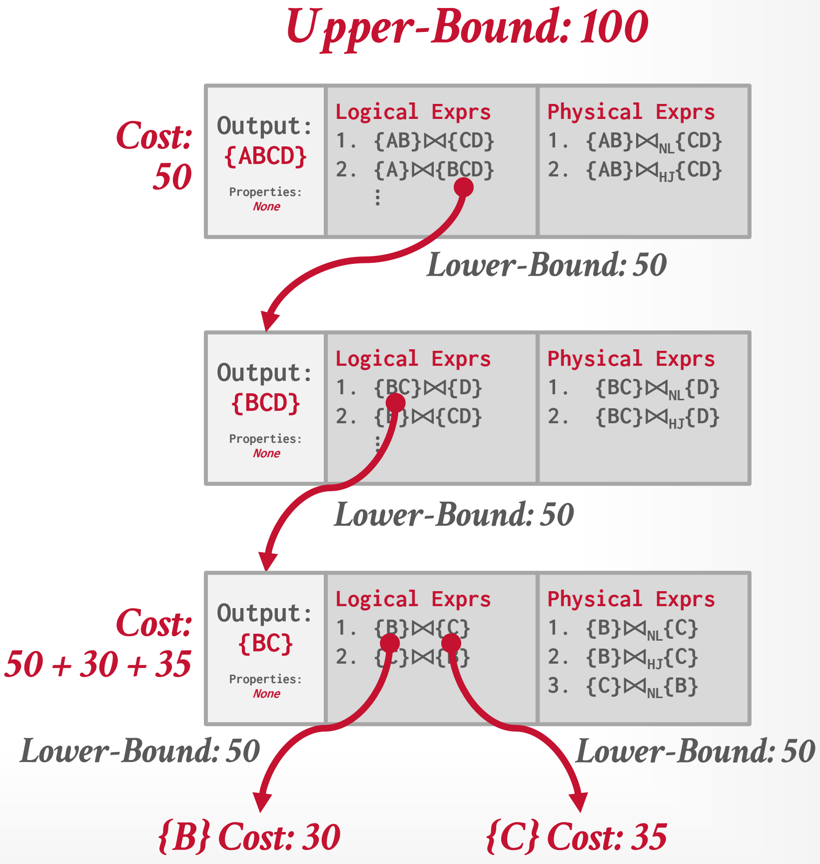
Lower-Bound: 50

Lower-Bound: 50

ACCUMULATED-COST BOUNDING

The upper-bound (***U***) is the cost of the best complete physical plan found so far in the entire search tree.

The lower-bound (***L***) is the summation of the physical operators as the optimizer traverses down the search tree.



Upper-Bound: 100

Cost: 50

Output: {ABCD}
Properties: *None*

Logical Exprs
1. $\{AB\}\bowtie\{CD\}$
2. $\{A\}\bowtie\{BCD\}$
⋮

Physical Exprs
1. $\{AB\}\bowtie_{NL}\{CD\}$
2. $\{AB\}\bowtie_{HJ}\{CD\}$

Lower-Bound: 50

Output: {BCD}
Properties: *None*

Logical Exprs
1. $\{BC\}\bowtie\{D\}$
2. $\{B\}\bowtie\{CD\}$
⋮

Physical Exprs
1. $\{BC\}\bowtie_{NL}\{D\}$
2. $\{BC\}\bowtie_{HJ}\{D\}$

Lower-Bound: 50

Cost: 50 + 30 + 35

Output: {BC}
Properties: *None*

Logical Exprs
1. $\{B\}\bowtie\{C\}$
2. $\{C\}\bowtie\{B\}$

Physical Exprs
1. $\{B\}\bowtie_{NL}\{C\}$
2. $\{B\}\bowtie_{HJ}\{C\}$
3. $\{C\}\bowtie_{NL}\{B\}$

Lower-Bound: 50

Lower-Bound: 50

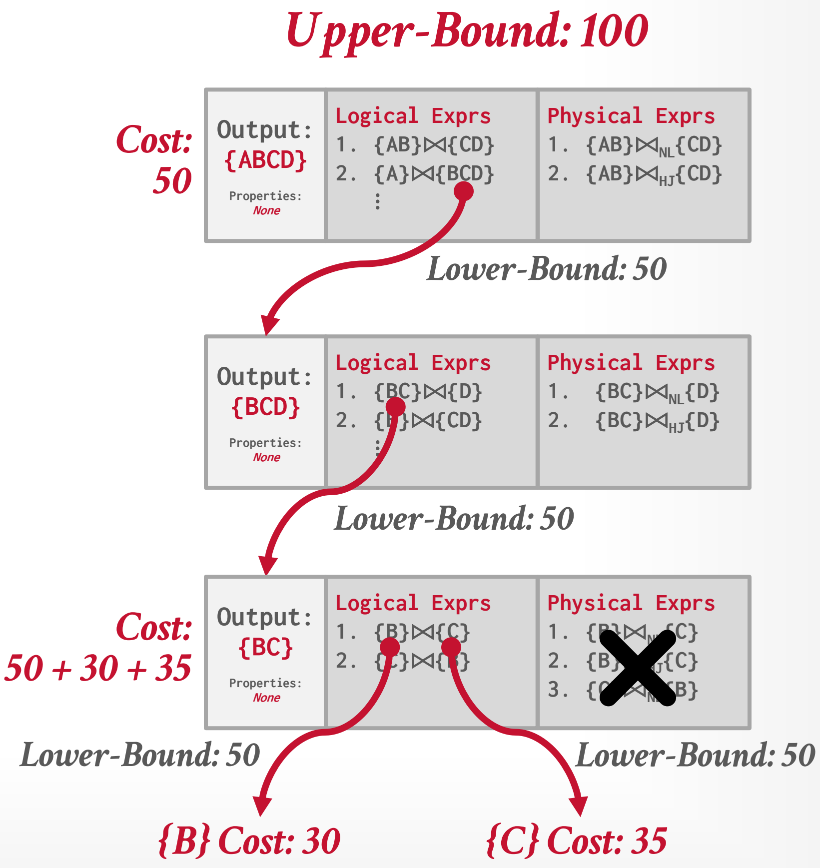
{B} Cost: 30

{C} Cost: 35

ACCUMULATED-COST BOUNDING

The upper-bound (***U***) is the cost of the best <u>complete</u> physical plan found so far in the entire search tree.

The lower-bound (***L***) is the summation of the physical operators as the optimizer traverses down the search tree.



Upper-Bound: 100

Cost: 50

Output: {ABCD}
Properties: *None*

Logical Exprs
1. $\{AB\} \bowtie \{CD\}$
2. $\{A\} \bowtie \{BCD\}$
⋮

Physical Exprs
1. $\{AB\} \bowtie_{NL} \{CD\}$
2. $\{AB\} \bowtie_{HJ} \{CD\}$

Lower-Bound: 50

Output: {BCD}
Properties: *None*

Logical Exprs
1. $\{BC\} \bowtie \{D\}$
2. $\{B\} \bowtie \{CD\}$
⋮

Physical Exprs
1. $\{BC\} \bowtie_{NL} \{D\}$
2. $\{BC\} \bowtie_{HJ} \{D\}$

Lower-Bound: 50

Cost: 50 + 30 + 35

Output: {BC}
Properties: *None*

Logical Exprs
1. $\{B\} \bowtie \{C\}$
2. $\{C\} \bowtie \{B\}$

Physical Exprs
1. $\{B\} \bowtie_{NL} \{C\}$
2. $\{B\} \bowtie_{HJ} \{C\}$
3. $\{C\} \bowtie_{NL} \{B\}$

Lower-Bound: 50

Lower-Bound: 50

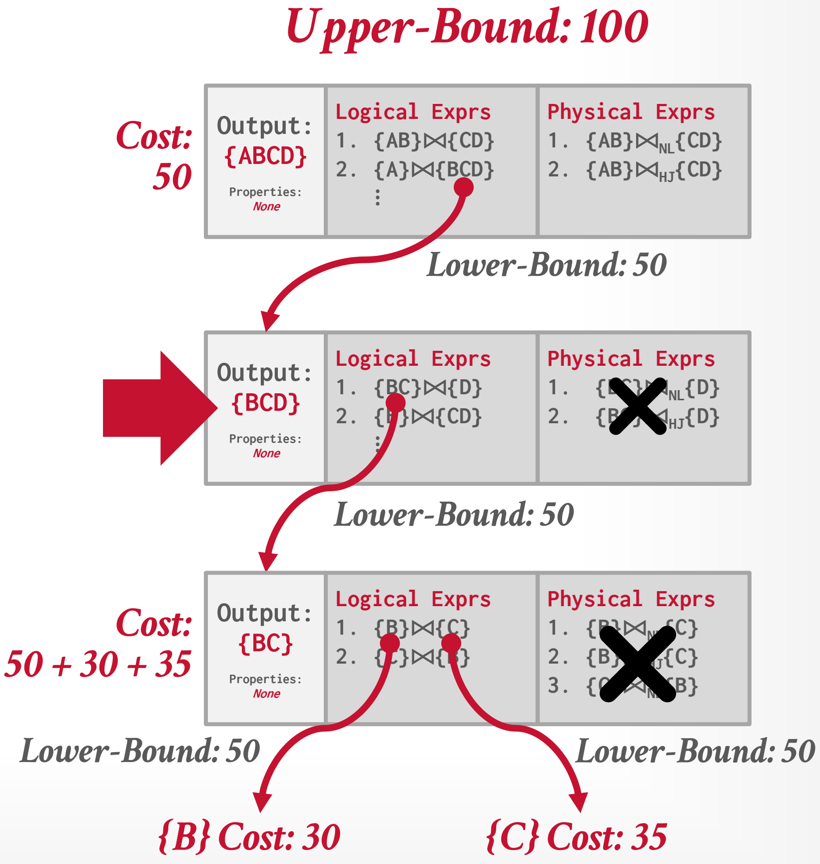
{B} Cost: 30

{C} Cost: 35

ACCUMULATED-COST BOUNDING

The upper-bound (***U***) is the cost of the best <u>complete</u> physical plan found so far in the entire search tree.

The lower-bound (***L***) is the summation of the physical operators as the optimizer traverses down the search tree.



Upper-Bound: 100

Cost: 50

Output: {ABCD}
Properties: *None*

Logical Exprs
1. {AB}⋈{CD}
2. {A}⋈{BCD}
⋮

Physical Exprs
1. {AB}⋈$_{NL}${CD}
2. {AB}⋈$_{HJ}${CD}

Lower-Bound: 50

Output: {BCD}
Properties: *None*

Logical Exprs
1. {BC}⋈{D}
2. {B}⋈{CD}
⋮

Physical Exprs
1. {BC}⋈$_{NL}${D}
2. {BC}⋈$_{HJ}${D}

Lower-Bound: 50

Cost: 50 + 30 + 35

Output: {BC}
Properties: *None*

Logical Exprs
1. {B}⋈{C}
2. {C}⋈{B}

Physical Exprs
1. {B}⋈$_{NL}${C}
2. {B}⋈$_{HJ}${C}
3. {C}⋈$_{NL}${B}

Lower-Bound: 50

Lower-Bound: 50

{B} Cost: 30

{C} Cost: 35

PREDICTED-COST BOUNDING

The upper-bound (***U***) is the cost of the best plan found for current logical expression.

- → As the optimizer traverses down to a new logical expression, reset ***U*** to **∞**.

The lower-bound (***L***) is predicted for each possible branch and the optimizer only explores best ones.

- → Without exploring a sub-tree, costs are only based on logical properties.

PREDICTED-COST BOUNDING

The upper-bound (***U***) is the cost of the best plan found for current logical expression.

- → As the optimizer traverses down to a new logical expression, reset ***U*** to **∞**.

The lower-bound (***L***) is predicted for each possible branch and the optimizer only explores best ones.

- → Without exploring a sub-tree, costs are only based on logical properties.

Predicted Costs:

{AB}⋈{CD}: 100 + 300
{A}⋈{BCD}: 0 + 600
{B}⋈{ACD}: 0 + 500

PREDICTED-COST BOUNDING

The upper-bound (***U***) is the cost of the best plan found for current logical expression.

→ As the optimizer traverses down to a new logical expression, reset ***U*** to **∞**.

The lower-bound (***L***) is predicted for each possible branch and the optimizer only explores best ones.

→ Without exploring a sub-tree, costs are only based on logical properties.

Predicted Costs:

{AB}⋈{CD}: 100 + 300

{A}⋈{BCD}: 0 + 600

{B}⋈{ACD}: 0 + 500

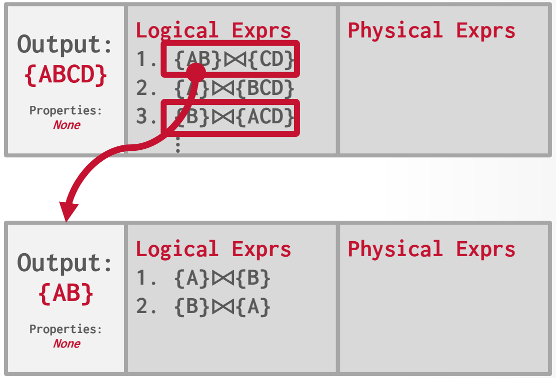
PREDICTED-COST BOUNDING

The upper-bound (***U***) is the cost of the best plan found for current logical expression.

→ As the optimizer traverses down to a new logical expression, reset ***U*** to **∞**.

The lower-bound (***L***) is predicted for each possible branch and the optimizer only explores best ones.

→ Without exploring a sub-tree, costs are only based on logical properties.



Output:
{ABCD}
Properties:
None

Logical Exprs
1. {AB}⋈{CD}
2. {A}⋈{BCD}
3. {B}⋈{ACD}
⋮

Physical Exprs

Output:
{AB}
Properties:
None

Logical Exprs
1. {A}⋈{B}
2. {B}⋈{A}

Physical Exprs

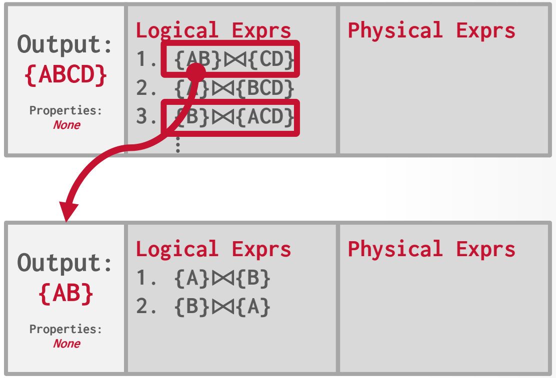
PREDICTED-COST BOUNDING

The upper-bound (***U***) is the cost of the best plan found for current logical expression.

→ As the optimizer traverses down to a new logical expression, reset ***U*** to **∞**.

The lower-bound (***L***) is predicted for each possible branch and the optimizer only explores best ones.

→ Without exploring a sub-tree, costs are only based on logical properties.



Output: {ABCD}
Properties: *None*
Logical Exprs
1. {AB}⋈{CD}
2. {A}⋈{BCD}
3. {B}⋈{ACD}
⋮
Physical Exprs

Output: {AB}
Properties: *None*
Logical Exprs
1. {A}⋈{B}
2. {B}⋈{A}
Physical Exprs

Predicted Costs:

{A}⋈{B} : 0 + 200
{B}⋈{A} : 0 + 300

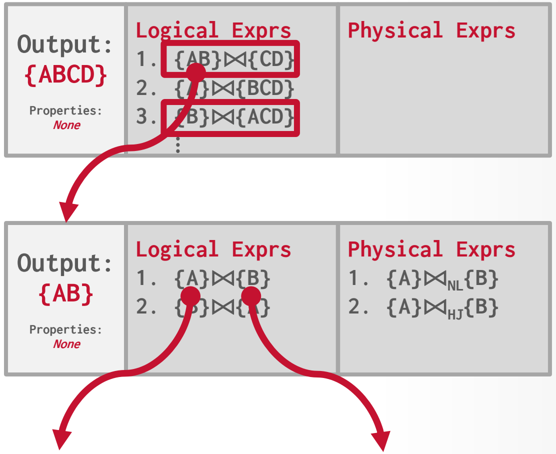
PREDICTED-COST BOUNDING

The upper-bound (***U***) is the cost of the best plan found for current logical expression.

- As the optimizer traverses down to a new logical expression, reset ***U*** to **∞**.

The lower-bound (***L***) is predicted for each possible branch and the optimizer only explores best ones.

- Without exploring a sub-tree, costs are only based on logical properties.



Output: {ABCD}
Properties: *None*

Logical Exprs
1. {AB}⋈{CD}
2. {A}⋈{BCD}
3. {B}⋈{ACD}
⋮

Physical Exprs

Output: {AB}
Properties: *None*

Logical Exprs
1. {A}⋈{B}
2. {B}⋈{A}

Physical Exprs
1. $\{A\}\bowtie_{NL}\{B\}$
2. $\{A\}\bowtie_{HJ}\{B\}$

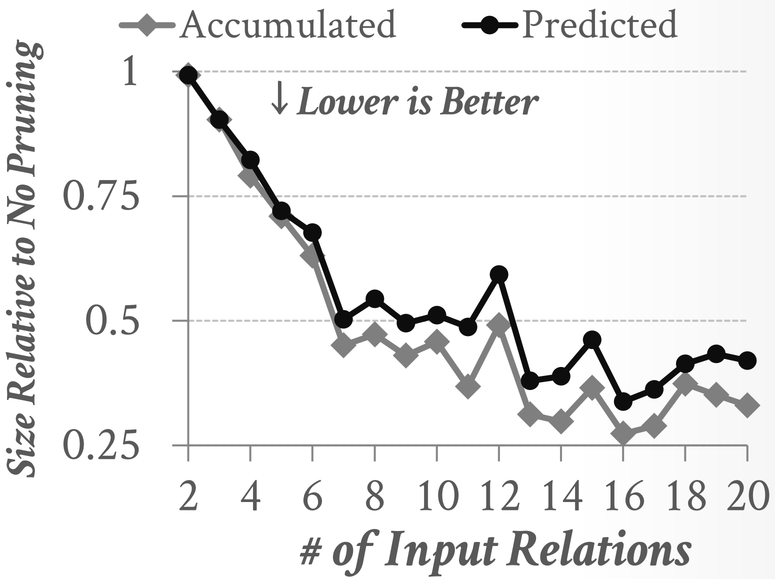
BRANCH-AND-BOUND PRUNING

Comparison of how well the pruning strategies remove branches from search tree.

Synthetic Query Graphs

→ Bushy Plans w/o Cartesian Products

→ Cannot compare plan quality.



Source: David DeHaan

BRANCH-AND-BOUND PRUNING

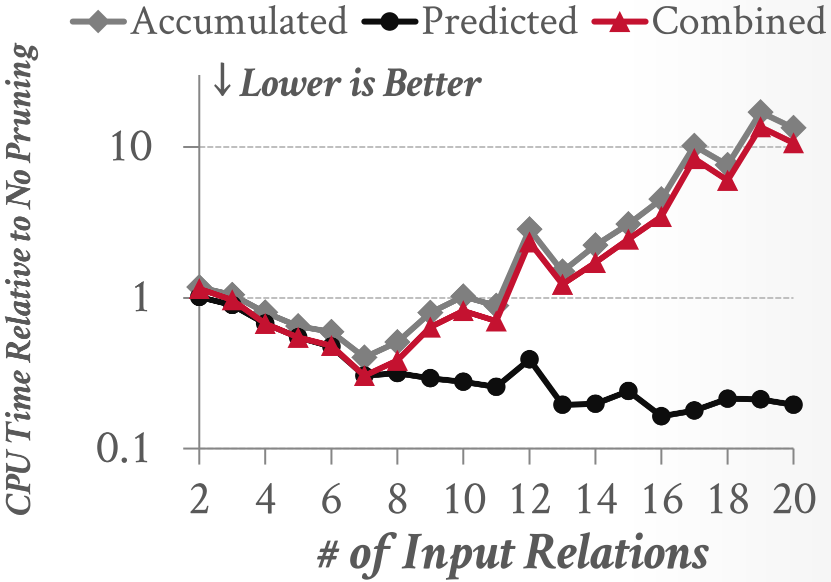
Comparison of how much computational work the optimizer consumes during search.

→ Combined is using both the accumulated and predicted pruning strategies together.

Synthetic Query Graphs

→ Bushy Plans w/o Cartesian Products

→ Cannot compare plan quality.



Source: David DeHaan

OBSERVATION

The previous join enumeration algorithm can only handle simple (binary) join predicates and inner joins.

The optimizer needs to support complex join predicates and outer / non-inner joins.

TOP-DOWN MINCUT + HYPERGRAPHS

Adaptation of the DP hypergraph algorithm from the original author for top-down join enumeration.

→ Convert hypergraphs into simple graphs to avoid excessive exploration of search space.
→ Relies on the min-cut partitioning approach discussed earlier.

We will go over this in more detail next week when we discuss search parallelization.

COUNTER STRIKE: GENERIC TOP-DOWN JOIN ENUMERATION FOR HYPERGRAPHS
VLDB 2013

TOP-DOWN MINCUT + HYPERGRAPHS

Adaptation of the DP hypergraph algorithm from the original author for top-down join enumeration.

→ Convert hypergraphs into simple graphs to avoid excessive exploration of search space.

→ Relies on the min-cut partitioning approach discussed earlier.

We will go over this in more detail next week when we discuss search parallelization.

COUNTER STRIKE: GENERIC TOP-DOWN JOIN ENUMERATION FOR HYPERGRAPHS
VLDB 2013

PARTING THOUGHTS

Andy still thinks top-down optimization is easier to understand but that does not mean it is the best approach.

→ What is good for humans can be bad for computers.

The adaptivity methods from last class could be modified to support top-down search.

→ Use approximations to preseed memo table.

NEXT CLASS

Parallelization: Bottom-Up