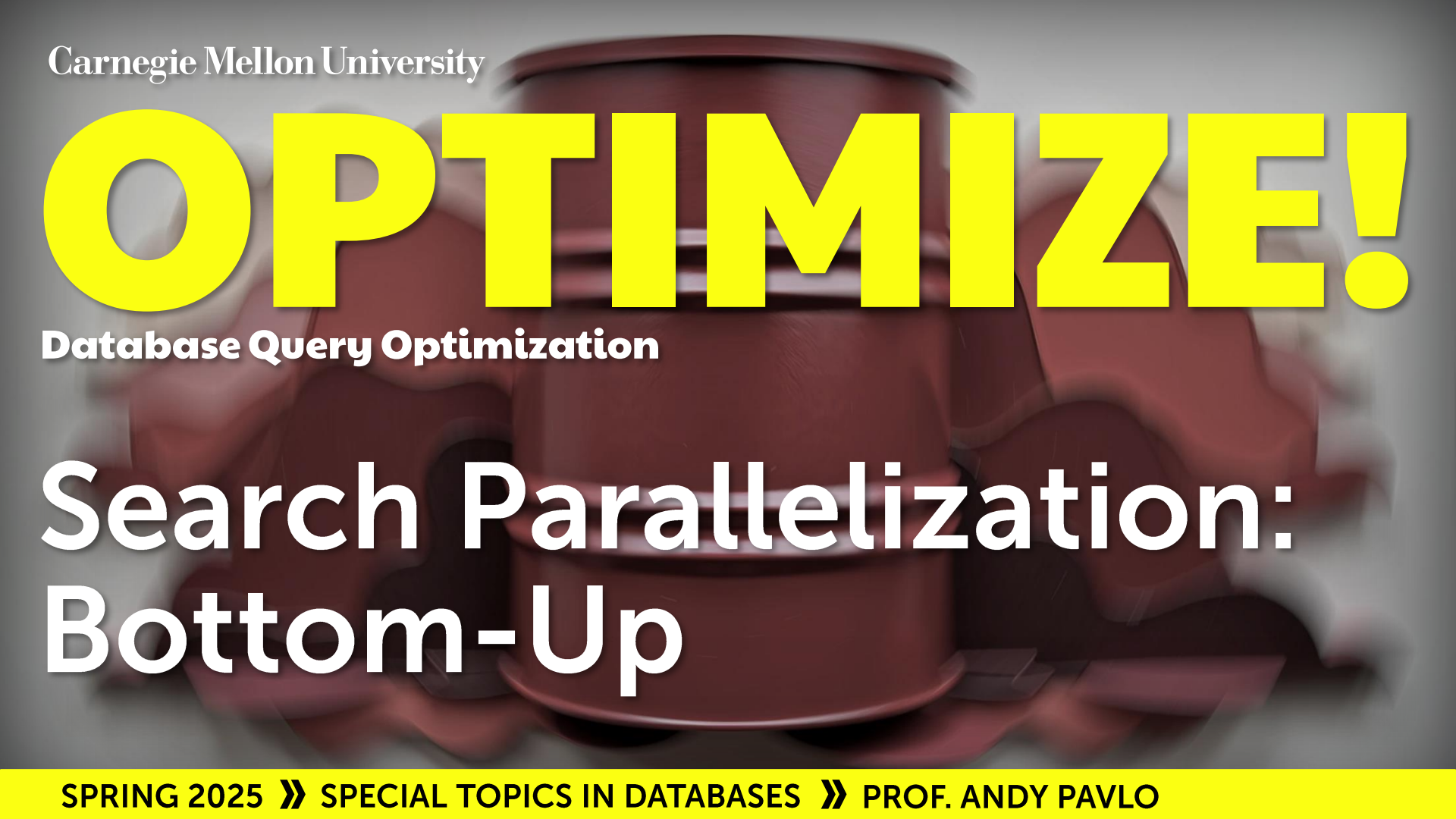

Carnegie Mellon University
OPTIMIZE!
Database Query Optimization
Search Parallelization: Bottom-Up
SPRING 2025 » SPECIAL TOPICS IN DATABASES » PROF. ANDY PAVLO

ADMINISTRIVIA

No Andy office hours today (See @22)
→ Email me to schedule a Zoom call later this week.

Project #1 is due Friday Feb 28th.

We will discuss potential Project #2 topics / directions next week.
→ Start figuring out who you want to be in your group.

LAST CLASS

Partitioning the query graph allows the optimizer to divide the search into smaller tasks to find better join orderings.

There can be dependencies that require parts of the solution space to be explored first depending on the search strategy.

→ Accumulated-cost vs. Predicted-cost Bounding

OBSERVATION

The quality of the plans that an optimizer generates is mostly based on three factors:

→ Search Algorithm
→ Cost Model
→ Transformations

All the methods and algorithms we've discussed so far have been single-threaded…

PARALLEL QUERY OPTIMIZATION

The optimizer employs multiple workers to search for better plans for a single query.

→ Ideally the optimizer's search time should improve linearly relative to the number of workers it uses.

Most DBMSs do not use a parallel optimizer implementation.

→ It takes away resources that could otherwise be used for query processing.

→ It is non-trivial to parallelize the optimizer's algorithms.

DP ALGORITHMS

The DP algorithms used in join enumeration belong to the non-serial polyadic DP class.

→ **Non-serial:** The computations in one phase depend on results from multiple previous phases, not just the immediately preceding one.

→ **Polyadic:** The recursive formulation involves more than one term on the right-hand side of the recurrence relation.

PARALLELIZATION STRATEGIES

Partition the search space evenly among workers.
→ All workers have tasks to compute at nearly all times.

Each worker processes a partition independently from the outcome of other workers.
→ No data dependencies between workers.

TODAY'S AGENDA

DP Subsets (DPSUB)

Massively Parallel DP (MPDP)

Multi-Plan Join DP (MPJ)

DYNAMIC PROGRAMMING SUBSET (DPSUB)

DPSUB enumerates all subsets (subgraphs) of relations ***R*** in increasing size based on number of vertexes.

For each subset ***S***, consider all possible partitions as two disjoint subsets S_{left} and S_{right}, where there exists a valid join predicate exists between them.

→ Evaluate the cost of joining plans for S_{left} and S_{right}.

→ Update ***BestPlan(S)*** if a cheaper plan is found.

EFFICIENT MASSIVELY PARALLEL JOIN OPTIMIZATION FOR LARGE QUERIES
SIGMOD 2022

DPSUB: OVERVIEW

```
Input: QI: Query Information
Output: Best Plan
1:  for all R_i ∈ QI.baseRelations do
2:      BestPlan({R_i}) = R_i
3:  end for
4:  for i := 2 to QI.querySize do
5:      S_i = {S | S ⊆ R and |S| = i and S is connected}
6:      for all S ∈ S_i do
7:          //the following is done in parallel
8:          for all S_left ⊆ S do
9:              EvaluatedCounter ++
10:             S_right = S \ S_left
11:             /*Begin CCP Block */
12:             if S_right == ∅ or S_left == ∅ continue
13:             if not S_left is connected continue
14:             if not S_right is connected continue
15:             if not S_right ∩ S_left = ∅ continue
16:             if not S_right is connected to S_left continue
17:             /* End CCP Block */
18:             CCP-Counter ++
19:             CurrPlan = CreatePlan(S_left, S_right)
20:             if CurrPlan < BestPlan(S) then
21:                 BestPlan(S) = CurrPlan
22:             end if
23:         end for
24:     end for
25: end for
26: return BestPlan(QI.baseRelations) //best plan for the query
```

Subsets (Size=2)

{A,B} {A,C} {A,D} {A,E}
{B,C} {B,D} {B,E}
{C,D} {C,E} {D,E}

Source: Riccardo Mancini

DPSUB: OVERVIEW

```
Input: QI: Query Information
Output: Best Plan
1:  for all R_i ∈ QI.baseRelations do
2:      BestPlan({R_i}) = R_i
3:  end for
4:  for i := 2 to QI.querySize do
5:      S_i = {S | S ⊆ R and |S| = i and S is connected}
6:      for all S ∈ S_i do
7:          //the following is done in parallel
8:          for all S_left ⊆ S do
9:              EvaluatedCounter ++
10:             S_right = S \ S_left
11:             /*Begin CCP Block */
12:             if S_right == ∅ or S_left == ∅ continue
13:             if not S_left is connected continue
14:             if not S_right is connected continue
15:             if not S_right ∩ S_left = ∅ continue
16:             if not S_right is connected to S_left continue
17:             /* End CCP Block */
18:             CCP-Counter ++
19:             CurrPlan = CreatePlan(S_left, S_right)
20:             if CurrPlan < BestPlan(S) then
21:                 BestPlan(S) = CurrPlan
22:             end if
23:         end for
24:     end for
25: end for
26: return BestPlan(QI.baseRelations) //best plan for the query
```

Subsets (Size=2)

{A,B} {A,C} {A,D} {A,E}
{B,C} {B,D} {B,E}
{C,D} {C,E} {D,E}

Source: Riccardo Mancini

DPSUB: OVERVIEW

```
Input: QI: Query Information
Output: Best Plan
 1: for all R_i ∈ QI.baseRelations do
 2:     BestPlan({R_i}) = R_i
 3: end for
 4: for i := 2 to QI.querySize do
 5:     S_i = {S | S ⊆ R and |S| = i and S is connected}
 6:     for all S ∈ S_i do
 7:         //the following is done in parallel
 8:         for all S_left ⊆ S do
 9:             EvaluatedCounter ++
10:             S_right = S \ S_left
11:             /*Begin CCP Block */
12:             if S_right == ∅ or S_left == ∅ continue
13:             if not S_left is connected continue
14:             if not S_right is connected continue
15:             if not S_right ∩ S_left = ∅ continue
16:             if not S_right is connected to S_left continue
17:             /* End CCP Block */
18:             CCP-Counter ++
19:             CurrPlan = CreatePlan(S_left, S_right)
20:             if CurrPlan < BestPlan(S) then
21:                 BestPlan(S) = CurrPlan
22:             end if
23:         end for
24:     end for
25: end for
26: return BestPlan(QI.baseRelations) //best plan for the query
```

Subsets (Size=2)

{A,B} {A,C} {A,D} {A,E}
{B,C} {B,D} {B,E}
{C,D} {C,E} {D,E}

Source: Riccardo Mancini

DPSUB: OVERVIEW

```
Input: QI: Query Information
Output: Best Plan
 1: for all R_i ∈ QI.baseRelations do
 2:    BestPlan({R_i}) = R_i
 3: end for
 4: for i := 2 to QI.querySize do
 5:    S_i = {S | S ⊆ R and |S| = i and S is connected}
 6:    for all S ∈ S_i do
 7:       //the following is done in parallel
 8:       for all S_left ⊆ S do
 9:          EvaluatedCounter ++
10:          S_right = S \ S_left
11:          /*Begin CCP Block */
12:          if S_right == ∅ or S_left == ∅ continue
13:          if not S_left is connected  continue
14:          if not S_right is connected  continue
15:          if not S_right ∩ S_left = ∅ continue
16:          if not S_right is connected to S_left  continue
17:          /* End CCP Block */
18:          CCP-Counter ++
19:          CurrPlan = CreatePlan(S_left, S_right)
20:          if CurrPlan < BestPlan(S) then
21:             BestPlan(S) = CurrPlan
22:          end if
23:       end for
24:    end for
25: end for
26: return BestPlan(QI.baseRelations) //best plan for the query
```

Subsets (Size=2)

{A,B} {A,C} {A,D} {A,E}
{B,C} {B,D} {B,E}
{C,D} {C,E} {D,E}

AB AC AD ⋯

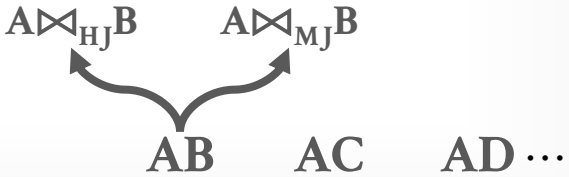
Source: Riccardo Mancini

DPSUB: OVERVIEW

```
Input: QI: Query Information
Output: Best Plan
 1: for all R_i ∈ QI.baseRelations do
 2:    BestPlan({R_i}) = R_i
 3: end for
 4: for i := 2 to QI.querySize do
 5:    S_i = {S | S ⊆ R and |S| = i and S is connected}
 6:    for all S ∈ S_i do
 7:       //the following is done in parallel
 8:       for all S_left ⊆ S do
 9:          EvaluatedCounter ++
10:          S_right = S \ S_left
11:          /*Begin CCP Block */
12:          if S_right == ∅ or S_left == ∅ continue
13:          if not S_left is connected continue
14:          if not S_right is connected continue
15:          if not S_right ∩ S_left = ∅ continue
16:          if not S_right is connected to S_left continue
17:          /* End CCP Block */
18:          CCP-Counter ++
19:          CurrPlan = CreatePlan(S_left, S_right)
20:          if CurrPlan < BestPlan(S) then
21:             BestPlan(S) = CurrPlan
22:          end if
23:       end for
24:    end for
25: end for
26: return BestPlan(QI.baseRelations) //best plan for the query
```

Subsets (Size=2)

{A,B} {A,C} {A,D} {A,E}
{B,C} {B,D} {B,E}
{C,D} {C,E} {D,E}



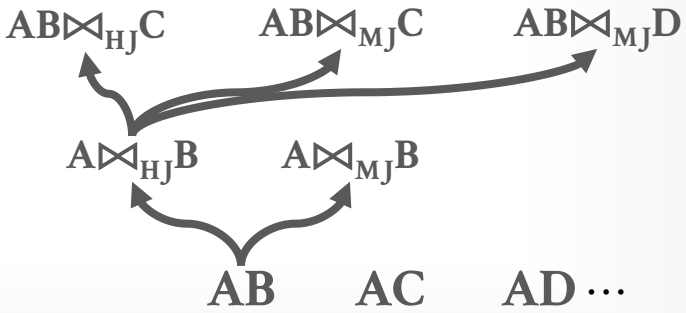
Source: Riccardo Mancini

DPSUB: OVERVIEW

```
Input: QI: Query Information
Output: Best Plan
 1: for all R_i ∈ QI.baseRelations do
 2:    BestPlan({R_i}) = R_i
 3: end for
 4: for i := 2 to QI.querySize do
 5:    S_i = {S | S ⊆ R and |S| = i and S is connected}
 6:    for all S ∈ S_i do
 7:       //the following is done in parallel
 8:       for all S_left ⊆ S do
 9:          EvaluatedCounter ++
10:          S_right = S \ S_left
11:          /*Begin CCP Block */
12:          if S_right == ∅ or S_left == ∅ continue
13:          if not S_left is connected continue
14:          if not S_right is connected continue
15:          if not S_right ∩ S_left = ∅ continue
16:          if not S_right is connected to S_left continue
17:          /* End CCP Block */
18:          CCP-Counter ++
19:          CurrPlan = CreatePlan(S_left, S_right)
20:          if CurrPlan < BestPlan(S) then
21:             BestPlan(S) = CurrPlan
22:          end if
23:       end for
24:    end for
25: end for
26: return BestPlan(QI.baseRelations) //best plan for the query
```

Subsets (Size=2)

{A,B} {A,C} {A,D} {A,E}
{B,C} {B,D} {B,E}
{C,D} {C,E} {D,E}



Source: Riccardo Mancini

MASSIVELY PARALLEL DP (MPDP)

Adaptive join enumeration that changes search strategy based on query complexity.

→ For simple queries, use the parallel version of DPSUB.

→ For larger queries, use a combination of heuristics and DP.

Combines vertex-based and edge-based enumeration and attempts to minimize the number of invalid join-pairs it evaluates.

Algorithm works on both CPUs + GPUs.

EFFICIENT MASSIVELY PARALLEL JOIN OPTIMIZATION FOR LARGE QUERIES
SIGMOD 2022

VERTEX VS. EDGE ENUMERATION

Vertex-Based Enumeration (DPSIZE, DPSUB):

→ Explore the search space by considering subsets of relations (vertices) and increasing sub-relation sizes.

→ These algorithms tend to evaluate many invalid join pairs.

Edge-Based Enumeration (DPCCP,DPHYP):

→ Enumerate join pairs based on the join graph dependencies (edges).

→ Reduces the number of invalid join pairs evaluated but can be difficult to parallelize.

UNIONDP

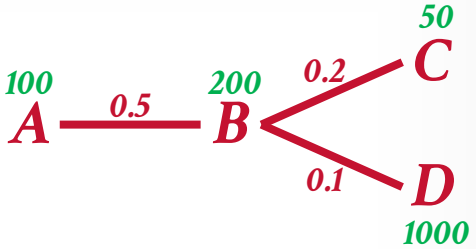
Divide join graph into smaller subgraphs of size ***k***.

→ Partition cuts are determined based on the estimated cardinalities.

→ Joins with larger cardinalities will be grouped together in the same subgraph.

Use MPDP to find the optimal ordering for each individual subgraph with ***k*** or less relations.

Construct plan by combining the best plans for each subgraph.



$A,B = 100 \times 200 \times 0.5 = 10000$

$B,C = 200 \times 50 \times 0.2 = 2000$

$B,D = 200 \times 1000 \times 0.1 = 20000$

MPDP ON GPU

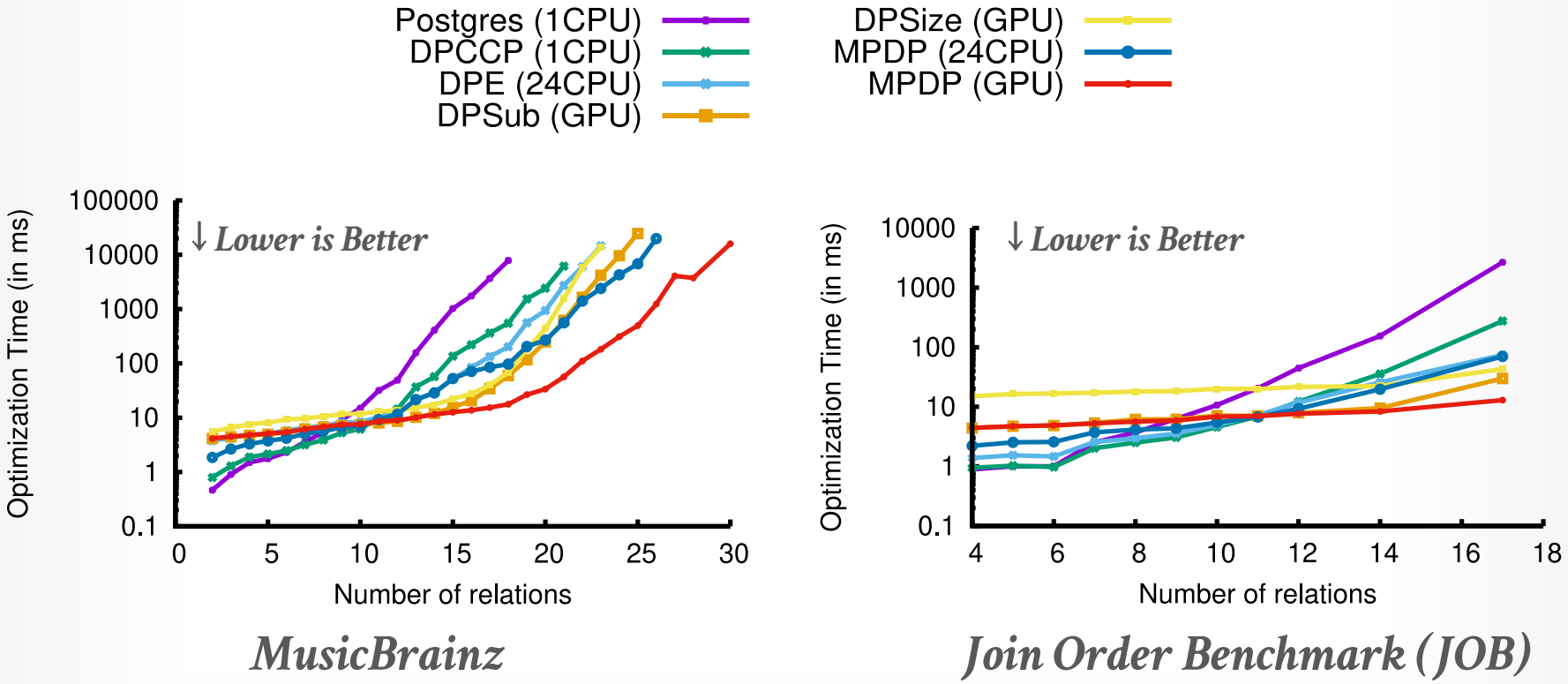
Warp-level parallelism to enumerate join orderings across multiple GPU cores.

→ Represent graph as array of fixed-width bitmaps.
→ Remove conditionals to minimize branch divergence.

All cost model estimates must be included in the data sent to GPU.

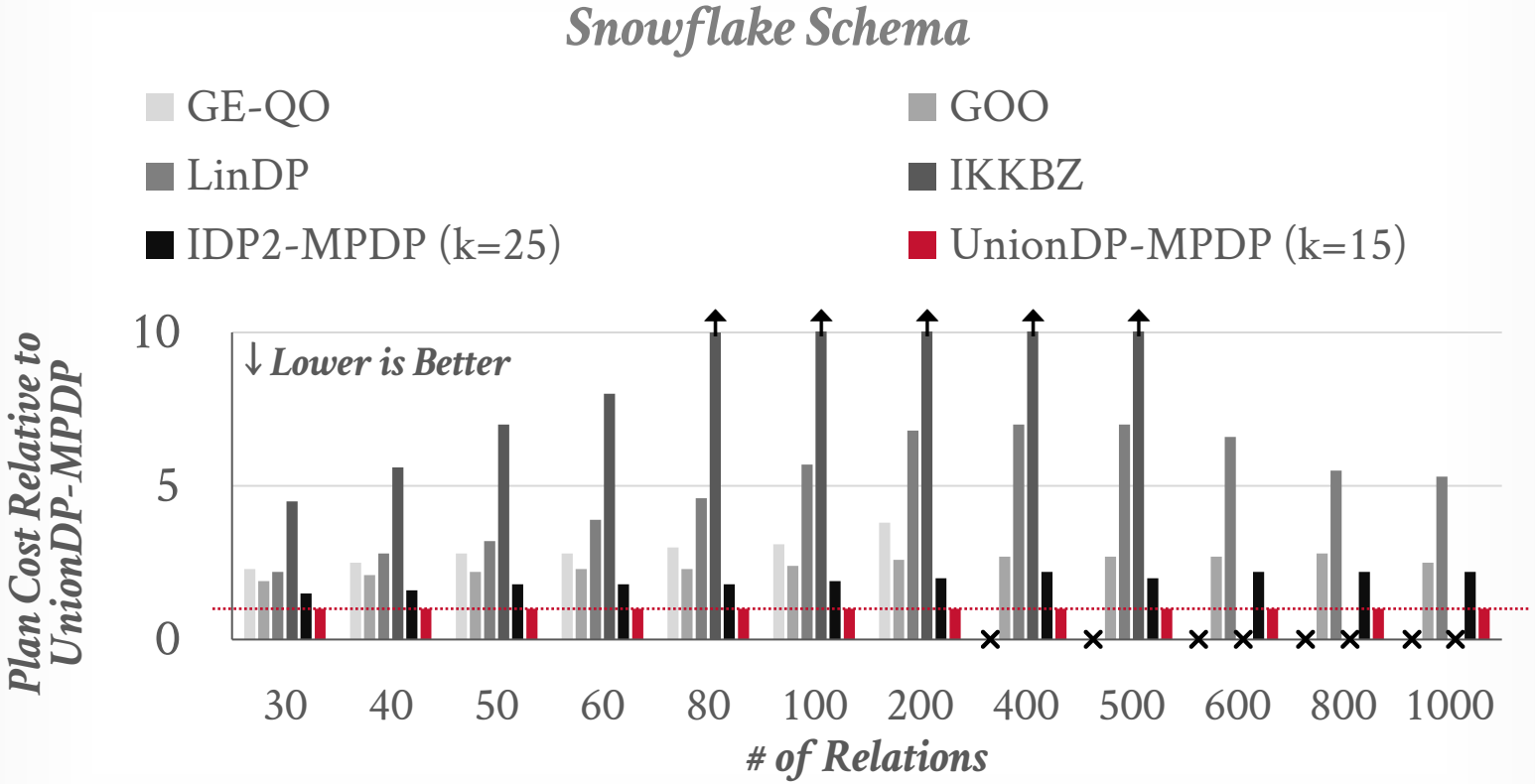
→ Don't want to block GPU threads by asking the CPU to compute a cardinality estimate.

EXPERIMENTAL RESULTS: SEARCH TIME



Source: Riccardo Mancini

EXPERIMENTAL RESULTS: PLAN QUALITY



Source: Riccardo Mancini

MULTIPLE PLAN JOIN (MPJ)

Perform join enumeration as a series of self-joins on an internal relation (i.e., Memo Table) that contains plans for subsets of query's target relations.

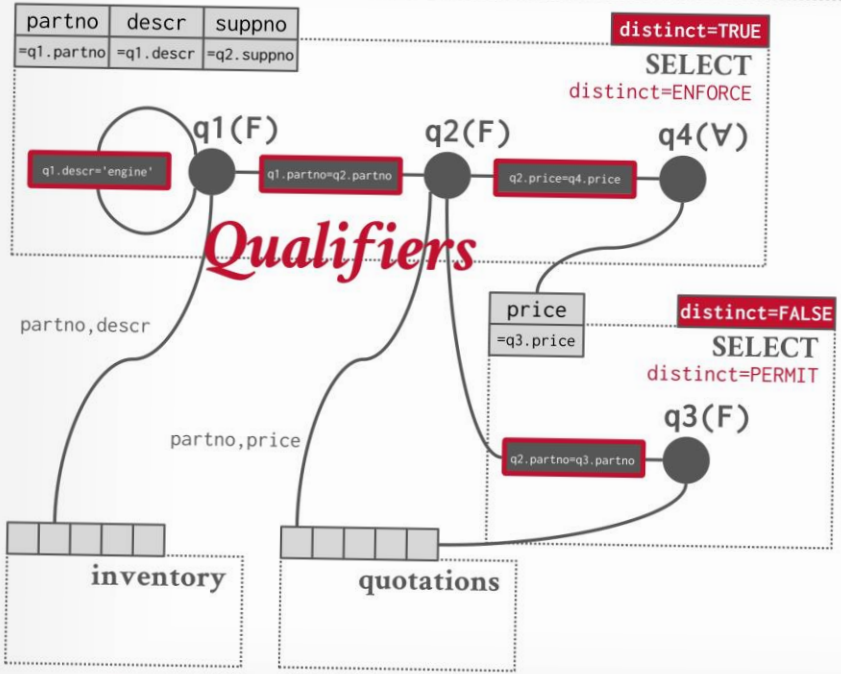
→ Each tuple contains a quantifier set and a list of query plans that generate the quantifiers.

Partition the search space into independent sub-problems based on quantifier set sizes.

MULTIPLE PLAN JOIN (MPJ)

QUERY GRAPH MODEL

18



Get the suppliers and parts information for which the supplier's price is less than that of all other suppliers.

```
SELECT DISTINCT q1.partno, q1.descr, q2.suppno
  FROM inventory AS q1, quotations AS q2
 WHERE q1.partno = q2.partno
   AND q1.descr = 'engine'
   AND q1.price <= ALL(
     SELECT q3.price
       FROM quotations AS q3
      WHERE q2.partno = q3.partno );
```

Iterators

→ SetFormers: **F**

→ Quantifiers: **∀**, **∃**

Source: Hamid Pirahesh

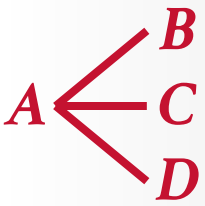
PARALLELIZING QUERY OPTIMIZATION
VLDB 2008

MPJ: OVERVIEW

Generate the set of plan tuples in memo table.

QuantifierSet	PlanList
A	_
B	_
C	_
D	_
AB	_
AC	_
AD	_
ABC	_
ABD	_
ACD	_
ABCD	_

```
SELECT * FROM A, B, C, D
 WHERE A.b_id = B.id
   AND A.c_id = C.id
   AND A.d_id = D.id;
```



Source: Wook-Shin Han

MPJ: OVERVIEW

Generate the set of plan tuples in memo table.
Logically partition tuples based on quantifier set sizes.

QuantifierSet	PlanList
A	_
B	_
C	_
D	_
AB	_
AC	_
AD	_
ABC	_
ABD	_
ACD	_
ABCD	_

P_1

A	_
B	_
C	_
D	_

P_2

AB	_
AC	_
AD	_

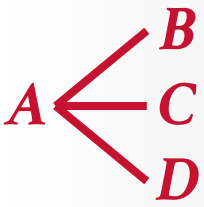
P_3

ABC	_
ABD	_
ACD	_

P_4

ABCD	_

```
SELECT * FROM A, B, C, D
 WHERE A.b_id = B.id
   AND A.c_id = C.id
   AND A.d_id = D.id;
```



Source: Wook-Shin Han

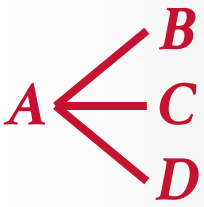
MPJ: OVERVIEW

Generate the set of plan tuples in memo table.
Logically partition tuples based on quantifier set sizes.
Explore each partition.

QuantifierSet	PlanList
A	_
B	_
C	_
D	_
AB	_
AC	_
AD	_
ABC	_
ABD	_
ACD	_
ABCD	_

P_1	A	_
	B	_
	C	_
	D	_
P_2	AB	_
	AC	_
	AD	_
P_3	ABC	_
	ABD	_
	ACD	_
P_4	ABCD	_

```
SELECT * FROM A, B, C, D
 WHERE A.b_id = B.id
   AND A.c_id = C.id
   AND A.d_id = D.id;
```



Source: Wook-Shin Han

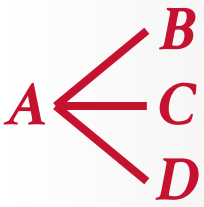
MPJ: OVERVIEW

Generate the set of plan tuples in memo table.
Logically partition tuples based on quantifier set sizes.
Explore each partition.

QuantifierSet	PlanList
A	_
B	_
C	_
D	_
AB	_
AC	_
AD	_
ABC	_
ABD	_
ACD	_
ABCD	_

P_1	A	QP_1, QP_2
	B	QP_3
	C	QP_4, QP_8
	D	QP_9, Q_{10}
P_2	AB	_
	AC	_
	AD	_
P_3	ABC	_
	ABD	_
	ACD	_
P_4	ABCD	_

```
SELECT * FROM A, B, C, D
 WHERE A.b_id = B.id
   AND A.c_id = C.id
   AND A.d_id = D.id;
```

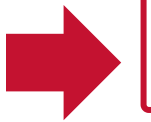


Source: Wook-Shin Han

MPJ: OVERVIEW

Generate the set of plan tuples in memo table.
Logically partition tuples based on quantifier set sizes.
Explore each partition.

QuantifierSet	PlanList
A	_
B	_
C	_
D	_
AB	_
AC	_
AD	_
ABC	_
ABD	_
ACD	_
ABCD	_



P_1

A	QP_1, QP_2
B	QP_3
C	QP_4, QP_8
D	QP_9, Q_{10}

P_2

AB	QP_{11}
AC	QP_{17}
AD	QP_{21}

$P_1 \bowtie P_1$

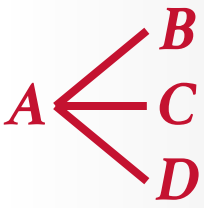
P_3

ABC	_
ABD	_
ACD	_

P_4

ABCD	_

```
SELECT * FROM A, B, C, D
 WHERE A.b_id = B.id
   AND A.c_id = C.id
   AND A.d_id = D.id;
```



Source: Wook-Shin Han

MPJ: OVERVIEW

Generate the set of plan tuples in memo table.
Logically partition tuples based on quantifier set sizes.
Explore each partition.

QuantifierSet	PlanList
A	–
B	–
C	–
D	–
AB	–
AC	–
AD	–
ABC	–
ABD	–
ACD	–
ABCD	–

P_1

A	QP_1, QP_2
B	QP_3
C	QP_4, QP_8
D	QP_9, Q_{10}

P_2 $P_1 \bowtie P_1$

AB	QP_{11}
AC	QP_{17}
AD	QP_{21}

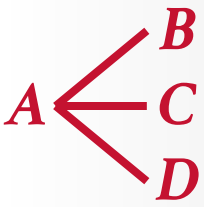
P_3 $P_1 \bowtie P_2$

ABC	QP_{27}, QP_{30}
ABD	QP_{31}
ACD	QP_{32}, QP_{33}

P_4

ABCD	–

```
SELECT * FROM A, B, C, D
 WHERE A.b_id = B.id
   AND A.c_id = C.id
   AND A.d_id = D.id;
```



Source: Wook-Shin Han

MPJ: OVERVIEW

Generate the set of plan tuples in memo table.
Logically partition tuples based on quantifier set sizes.
Explore each partition.

QuantifierSet	PlanList
A	–
B	–
C	–
D	–
AB	–
AC	–
AD	–
ABC	–
ABD	–
ACD	–
ABCD	–

P_1

A	QP_1, QP_2
B	QP_3
C	QP_4, QP_8
D	QP_9, Q_{10}

P_2

AB	QP_{11}
AC	QP_{17}
AD	QP_{21}

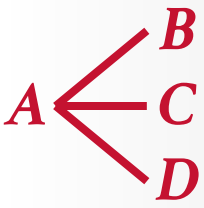
P_3

ABC	QP_{27}, QP_{30}
ABD	QP_{31}
ACD	QP_{32}, QP_{33}

P_4

ABCD	–

```
SELECT * FROM A, B, C, D
 WHERE A.b_id = B.id
   AND A.c_id = C.id
   AND A.d_id = D.id;
```



Source: Wook-Shin Han

MPJ: OVERVIEW

Generate the set of plan tuples in memo table.
Logically partition tuples based on quantifier set sizes.
Explore each partition.

QuantifierSet	PlanList
A	_
B	_
C	_
D	_
AB	_
AC	_
AD	_
ABC	_
ABD	_
ACD	_
ABCD	_

P_1	A	QP_1, QP_2
	B	QP_3
	C	QP_4, QP_8
	D	QP_9, Q_{10}
P_2	AB	QP_{11}
	AC	QP_{17}
	AD	QP_{21}
P_3	ABC	QP_{27}, QP_{30}
	ABD	QP_{31}
	ACD	QP_{32}, QP_{33}
P_4	ABCD	_

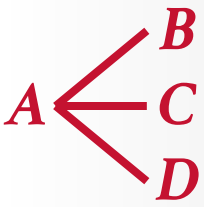
$P_1 \bowtie P_3$

(A,ABC) (A,ABD) (A,ACD)
(B,ABC) (B,ABD) (B,ACD)
(C,ABC) (C,ABD) (C,ACD)
(D,ABC) (D,ABD) (D,ACD)

$P_2 \bowtie P_2$

(AB,AB) (AB,AC) (AB,AD)
(AC,AB) (AC,AC) (AC,AD)
(AD,AB) (AD,AC) (AD,AD)

```
SELECT * FROM A, B, C, D
 WHERE A.b_id = B.id
   AND A.c_id = C.id
   AND A.d_id = D.id;
```



Source: Wook-Shin Han

MPJ: OVERVIEW

Generate the set of plan tuples in memo table.
Logically partition tuples based on quantifier set sizes.
Explore each partition.

QuantifierSet	PlanList
A	_
B	_
C	_
D	_
AB	_
AC	_
AD	_
ABC	_
ABD	_
ACD	_
ABCD	_

P_1

A	QP_1, QP_2
B	QP_3
C	QP_4, QP_8
D	QP_9, Q_{10}

P_2

AB	QP_{11}
AC	QP_{17}
AD	QP_{21}

P_3

ABC	QP_{27}, QP_{30}
ABD	QP_{31}
ACD	QP_{32}, QP_{33}

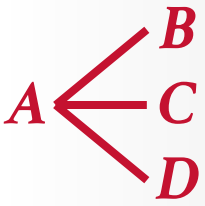
P_4

ABCD	–

$P_1 \bowtie P_3$

$P_2 \bowtie P_2$

```
SELECT * FROM A, B, C, D
 WHERE A.b_id = B.id
   AND A.c_id = C.id
   AND A.d_id = D.id;
```



(A,ABC) (A,ABD) (A,ACD)
(B,ABC) (B,ABD) (B,ACD)
(C,ABC) (C,ABD) (C,ACD)
(D,ABC) (D,ABD) (D,ACD)

(AB,AB) (AB,AD)
(AC,AB) (AC,AD)
(AD,AB) (AD,AD)

Source: Wook-Shin Han

MPJ: SEARCH SPACE ALLOCATION

Choice #1: Total Sum Allocation

→ Compute the total number of plan joins and then divide evenly among workers.
→ Easiest to implement.

Choice #2: Stratified Allocation

→ Divides the enumeration space into multiple strata (subsets), where each stratum corresponds to a plan join.

MPJ: STRATIFIED ALLOCATION

Equi-Depth Allocation:

→ Divide the whole range of the outer loop into smaller contiguous ranges of equal size.

Round-Robin Outer Allocation:

→ Iterate through each outer loop item and assign it to workers in round-robin.

Round-Robin Inner Allocation:

→ Iterate through each inner loop item and assign it to workers in round-robin.

```
for size in 2 to N:
  searchSpace ← AllocateSpace()
  for i in 1 to ⌊size/2⌋:
    j ← size - i
    for k in 1 to #threads:
      ExecuteSearch(searchSpace[k])
```

PARTING THOUGHTS

Dividing the tasks up to parallelize the search of a query plan is non-trivial.

Andy's Opinion: Using a dedicated GPU just for query optimization is not practical. But integrated accelerators (APUs) may be an attractive design option in the future.

PARTING THOUGHTS

AMD RYZEN™ 8000 G-SERIES PROCESSORS

FOR ENTRY GAMING AND PRODUCTIVITY SYSTEMS

AMD Ryzen™ 7 8700G with AMD Ryzen™ AI

8-Core 16-Thread	UP TO 5.1 GHz MAX BOOST	24MB Cache	65W TDP	AMD Radeon™ 780M Graphics

AMD Ryzen™ 5 8600G with AMD Ryzen™ AI

6-Core 12-Thread	UP TO 5.0 GHz MAX BOOST	22MB Cache	65W TDP	AMD Radeon™ 760M Graphics

AMD Ryzen™ 5 8500G

6-Core 12-Thread	UP TO 5.0 GHz MAX BOOST	22MB Cache	65W TDP	AMD Radeon™ 740M Graphics

AMD Ryzen™ 3 8300G (only in partner systems)

4-Core 8-Thread	UP TO 4.9 GHz MAX BOOST	12MB Cache	65W TDP	AMD Radeon™ 740M Graphics

(System Builder Only, available by the end of Q1 2024)

AVAILABLE JANUARY 31st

*Images are for illustrative purposes only SEE ENDNOTE: GD-150.

21 | AMD Consumer Retail Deck | NDA Confidential | Spring 2023

AMD
together we advance_

NEXT CLASS

Parallelization: Top-Down