# Chapter 10

## Query Optimization

### Exploring the Search Space of Alternative Query Plans

*Architecture and Implementation of Database Systems*
Summer 2014

Torsten Grust
Wilhelm-Schickard-Institut für Informatik
Universität Tübingen



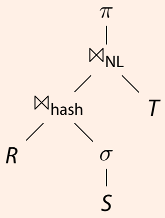
Query Optimization
- Search Space Illustration
- Dynamic Programming
  - Example: Four-Way Join
  - Algorithm
  - Discussion
  - Left/Right-Deep vs. Bushy
- Greedy join enumeration

## Finding the “Best” Query Plan

### Throttle or break?

```
SELECT ···
  FROM ···
 WHERE ···
```

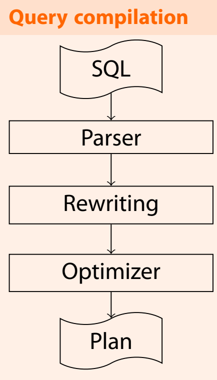
?




- We already saw that there may be more than one way to answer a given query.
  - Which one of the join operators should we pick? With which parameters (block size, buffer allocation, ...)?
- The task of **finding the best execution plan** is, in fact, the “holy grail” of any database implementation.

**Query Optimization**
Search Space Illustration
Dynamic Programming
Example: Four-Way Join
Algorithm
Discussion
Left/Right-Deep vs. Bushy
Greedy join enumeration

# Plan Generation Process




- **Parser:** syntactical/semantical analysis
- **Rewriting: heuristic** optimizations independent of the current database state (table sizes, availability of indexes, etc.). For example:
  - Apply predicates early
  - Avoid unnecessary duplicate elimination
- **Optimizer:** optimizations that rely on a **cost model** and information about the current database state
- The resulting **plan** is then evaluated by the system's **execution engine**.

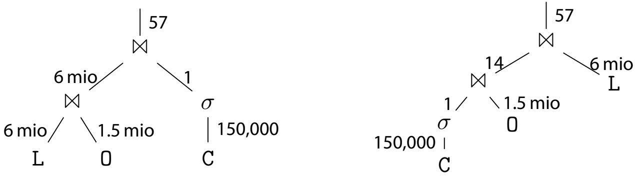
Query Optimization
Search Space Illustration
Dynamic Programming
Example: Four-Way Join
Algorithm
Discussion
Left/Right-Deep vs. Bushy
Greedy join enumeration

# Impact on Performance

Finding the right plan can dramatically impact performance.

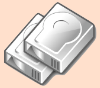
**Sample query over TPC-H tables**

```sql
SELECT L.L_PARTKEY, L.L_QUANTITY, L.L_EXTENDEDPRICE
  FROM LINEITEM L, ORDERS O, CUSTOMER C
 WHERE L.L_ORDERKEY = O.O_ORDERKEY
   AND O.O_CUSTKEY = C.C_CUSTKEY
   AND C.C_NAME = 'IBM Corp.'
```




- In terms of execution times, these differences can easily mean "seconds versus days."

Query Optimization
Search Space Illustration
Dynamic Programming
Example: Four-Way Join
Algorithm
Discussion
Left/Right-Deep vs. Bushy
Greedy join enumeration



Query Optimization
Search Space Illustration
Dynamic Programming
Example: Four-Way Join
Algorithm
Discussion
Left/Right-Deep vs. Bushy
Greedy join enumeration

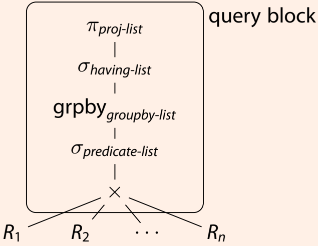
# The SQL Parser

- Besides some analyses regarding the syntactical and semantical correctness of the input query, the parser creates an **internal representation** of the input query.
- This representation still resembles the original query:
    - Each `SELECT-FROM-WHERE` clause is translated into a **query block**.

## Deriving a query block from a SQL SFW block

```
SELECT  proj-list
  FROM  R1 , R2 , ... , Rn
 WHERE  predicate-list
 GROUP BY  groupby-list
HAVING  having-list
```

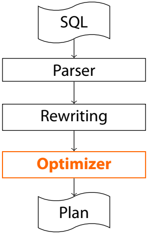
$\rightarrow$




- Each $R_i$ can be a base relation or another query block.

Query Optimization
Search Space Illustration
Dynamic Programming
Example: Four-Way Join
Algorithm
Discussion
Left/Right-Deep vs. Bushy
Greedy join enumeration

## Finding the "Best" Execution Plan




The parser output is fed into a **rewrite engine** which, again, yields a tree of query blocks.

It is then the **optimizer's** task to come up with the optimal **execution plan** for the given query.

Essentially, the optimizer

1. **enumerates** all possible execution plans,
   (if this yields too many plans, at least enumerate the "promising" plan candidates)
2. determines the **quality** (cost) of each plan, then
3. **chooses** the best one as the final execution plan.

Before we can do so, we need to answer the question

- What is a "good" execution plan at all?

## Cost Metrics

Database systems judge the quality of an execution plan based on a number of **cost factors**, *e.g.*,

- the number of **disk I/Os** required to evaluate the plan,
- the plan's **CPU cost**,
- the overall **response time** observable by the database client as well as the total **execution time**.

A cost-based optimizer tries to **anticipate** these costs and find the cheapest plan before actually running it.

- All of the above factors depend on one critical piece of information: the **size of (intermediate) query results**.
- Database systems, therefore, spend considerable effort into accurate **result size estimates**.

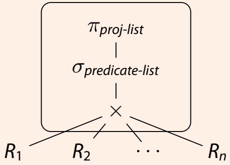


Query Optimization
Search Space Illustration
Dynamic Programming
Example: Four-Way Join
Algorithm
Discussion
Left/Right-Deep vs. Bushy
Greedy join enumeration

## Result Size Estimation

Consider a query block corresponding to a simple SFW query $Q$.

### SFW query block




We can estimate the result size of $Q$ based on

- the size of the input tables, $|R_1|, \ldots, |R_n|$, and
- the **selectivity** $sel(p)$ of the predicate *predicate-list*:

$$|Q| \approx |R_1| \cdot |R_2| \cdots |R_n| \cdot sel(\textit{predicate-list}) \enspace .$$

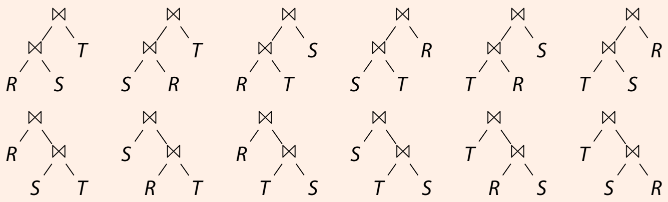
**Query Optimization**
Search Space Illustration
Dynamic Programming
Example: Four-Way Join
Algorithm
Discussion
Left/Right-Deep vs. Bushy
Greedy join enumeration

# Join Optimization

- We've now translated the query into a graph of **query blocks**.
  - Query blocks essentially are a **multi-way** Cartesian product with a number of selection predicates on top.
- We can estimate the **cost** of a given **execution plan**.
  - Use result size estimates in combination with the cost for individual join algorithms discussed in previous chapters.

We are now ready to **enumerate** all possible execution plans, *i.e.*, all possible **2-way** join combinations for each query block.

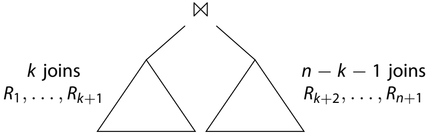
## Ways of building a 3-way join from two 2-way joins




**Query Optimization**
Search Space Illustration
Dynamic Programming
Example: Four-Way Join
Algorithm
Discussion
Left/Right-Deep vs. Bushy
Greedy join enumeration

# How Many Such Combinations Are There?

- A join over $n+1$ relations $R_1, \ldots, R_{n+1}$ requires $n$ **binary joins**.
- Its **root-level operator** joins sub-plans of $k$ and $n-k-1$ join operators ($0 \leqslant k \leqslant n-1$):




- Let $C_i$ be the **number of possibilities** to construct a binary tree of $i$ inner nodes (join operators):

$$C_n = \sum_{k=0}^{n-1} C_k \cdot C_{n-k-1} \quad .$$




Query Optimization
Search Space Illustration
Dynamic Programming
Example: Four-Way Join
Algorithm
Discussion
Left/Right-Deep vs. Bushy
Greedy join enumeration

# Catalan Numbers

This recurrence relation is satisfied by **Catalan numbers**:

$$C_n = \sum_{k=0}^{n-1} C_k \cdot C_{n-k-1} = \frac{(2n)!}{(n+1)!n!} \quad ,$$

describing the number of ordered binary trees with $n+1$ leaves.

For **each** of these trees, we can **permute** the input relations (why?) $R_1, \dots, R_{n+1}$, leading to:

**Number of possible join trees for an $(n+1)$-way relational join**

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



Query Optimization
Search Space Illustration
Dynamic Programming
Example: Four-Way Join
Algorithm
Discussion
Left/Right-Deep vs. Bushy
Greedy join enumeration

# Search Space

The resulting search space is **enormous**:

**Possible bushy join trees joining $n$ relations**

| number of relations $n$ | $C_{n-1}$ | join trees |
|---|---|---|
| 2 | 1 | 2 |
| 3 | 2 | 12 |
| 4 | 5 | 120 |
| 5 | 14 | 1,680 |
| 6 | 42 | 30,240 |
| 7 | 132 | 665,280 |
| 8 | 429 | 17,297,280 |
| 10 | 4,862 | 17,643,225,600 |

- And we haven't yet even considered the use of $k$ **different join algorithms** (yielding another factor of $k^{(n-1)}$)!



Query Optimization
Search Space Illustration
Dynamic Programming
Example: Four-Way Join
Algorithm
Discussion
Left/Right-Deep vs. Bushy
Greedy join enumeration

Query Optimization
Search Space Illustration
Dynamic Programming
Example: Four-Way Join
Algorithm
Discussion
Left/Right-Deep vs. Bushy
Greedy join enumeration

# Dynamic Programming

The traditional approach to master this search space is the use of **dynamic programming**.

**Idea:**

- Find the cheapest plan for an $n$-way join in $n$ **passes**.
- In each pass $k$, find the best plans for all $k$-relation **sub-queries**.
- **Construct** the plans in pass $k$ from best $i$-relation and $(k - i)$-relation sub-plans found in **earlier passes** ($1 \leqslant i < k$).

**Assumption:**

- To find the optimal **global plan**, it is sufficient to only consider the optimal plans of its **sub-queries** ("Principle of optimality").

## Dynamic Programming

### Example (Four-way join of tables $R_{1,\dots,4}$)

**Pass 1** (best 1-relation plans)
Find the best **access path** to each of the $R_i$ individually (considers index scans, full table scans).

**Pass 2** (best 2-relation plans)
For each **pair** of tables $R_i$ and $R_j$, determine the best order to join $R_i$ and $R_j$ (use $R_i \bowtie R_j$ or $R_j \bowtie R_i$ ?):

$$optPlan(\{R_i, R_j\}) \leftarrow \text{ best of } R_i \bowtie R_j \text{ and } R_j \bowtie R_i \ .$$

$\rightarrow$ 12 plans to consider.

**Pass 3** (best 3-relation plans)
For each **triple** of tables $R_i$, $R_j$, and $R_k$, determine the best three-table join plan, using sub-plans obtained so far:

$$optPlan(\{R_i, R_j, R_k\}) \leftarrow \text{ best of } R_i \bowtie optPlan(\{R_j, R_k\}), \\ optPlan(\{R_j, R_k\}) \bowtie R_i, \quad R_j \bowtie optPlan(\{R_i, R_k\}), \dots \ .$$

$\rightarrow$ 24 plans to consider.



Query Optimization
Search Space Illustration
Dynamic Programming
Example: Four-Way Join
Algorithm
Discussion
Left/Right-Deep vs. Bushy
Greedy join enumeration

## Dynamic Programming

**Example (Four-way join of tables $R_{1,\dots,4}$ (cont'd))**

**Pass 4** (best 4-relation plan)

For each set of **four** tables $R_i$, $R_j$, $R_k$, and $R_l$, determine the best four-table join plan, using sub-plans obtained so far:

$$
\begin{aligned}
&\mathit{optPlan}(\{R_i, R_j, R_k, R_l\}) \leftarrow \text{ best of } R_i \bowtie \mathit{optPlan}(\{R_j, R_k, R_l\}),\\
&\quad \mathit{optPlan}(\{R_j, R_k, R_l\}) \bowtie R_i,\quad R_j \bowtie \mathit{optPlan}(\{R_i, R_k, R_l\}), \dots,\\
&\quad \mathit{optPlan}(\{R_i, R_j\}) \bowtie \mathit{optPlan}(\{R_k, R_l\}), \dots \quad .
\end{aligned}
$$

$\longrightarrow$ 14 plans to consider.

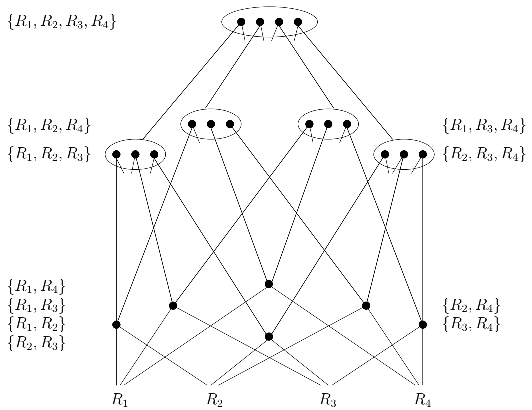
- Overall, we looked at only **50** (sub-)plans (instead of the possible 120 four-way join plans; $\nearrow$ slide 12).
- All decisions required the evaluation of **simple** sub-plans only (**no need to re-evaluate** *optPlan*$(\cdot)$ for already known relation combinations $\Rightarrow$ use lookup table).



Query Optimization
Search Space Illustration
Dynamic Programming
Example: Four-Way Join
Algorithm
Discussion
Left/Right-Deep vs. Bushy
Greedy join enumeration

# Sharing Under the Optimality Principle

## Sharing optimal sub-plans




Drawing by Guido Moerkotte, U Mannheim

Query Optimization
Search Space Illustration
Dynamic Programming
Example: Four-Way Join
Algorithm
Discussion
Left/Right-Deep vs. Bushy
Greedy join enumeration

Query Optimization
Search Space Illustration
Dynamic Programming
Example: Four-Way Join
Algorithm
Discussion
Left/Right-Deep vs. Bushy
Greedy join enumeration

# Dynamic Programming Algorithm

**Find optimal $n$-way bushy join tree via dynamic programming**

**Function:** `find_join_tree_dp` $(q(R_1, \ldots, R_n))$

**for** $i = 1$ **to** $n$ **do**
  $\mathit{optPlan}(\{R_i\}) \leftarrow$ `access_plans` $(R_i)$ ;
  `prune_plans` $(\mathit{optPlan}(\{R_i\}))$ ;

**for** $i = 2$ **to** $n$ **do**
  **foreach** $S \subseteq \{R_1, \ldots, R_n\}$ **such that** $|S| = i$ **do**
    $\mathit{optPlan}(S) \leftarrow \varnothing$ ;
    **foreach** $O \subset S$ with $O \neq \varnothing$ **do**
      $\mathit{optPlan}(S) \leftarrow \mathit{optPlan}(S) \cup$
        `possible_joins` $\left[ \begin{array}{ccc} & \bowtie & \\ \mathit{optPlan}(O) & & \mathit{optPlan}(S \setminus O) \end{array} \right]$ ;
    `prune_plans` $(\mathit{optPlan}(S))$ ;

**return** $\mathit{optPlan}(\{R_1, \ldots, R_n\})$ ;

- `possible_joins` $[R \bowtie S]$ enumerates the possible joins between $R$ and $S$ (nested loops join, merge join, etc.).
- `prune_plans` $(\mathit{set})$ discards all but the best plan from $\mathit{set}$.

## Dynamic Programming—Discussion

- Enumerate all non-empty true subsets of *S* (using C):

```
O = S & -S;
do {
  /* perform operation on O */
  O = S & (O - S);
} while (O != S);
```

- `find_join_tree_dp ()` draws its advantage from **filtering** plan candidates early in the process.
  - In our example on slide 14, pruning in Pass 2 reduced the search space by a factor of 2, and another factor of 6 in Pass 3.
- Some **heuristics** can be used to prune even more plans:
  - Try to avoid **Cartesian products**.
  - Produce **left-deep plans** only (see next slides).
- Such heuristics can be used as a handle to balance plan quality and optimizer runtime.

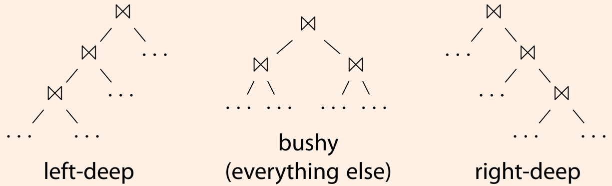
**DB2 Control optimizer investment**

```
SET CURRENT QUERY OPTIMIZATION = n
```

## Left/Right-Deep vs. Bushy Join Trees

The algorithm on slide 17 explores all possible shapes a join tree could take:

### Join tree shapes




Actual systems often prefer **left-deep** join trees.[1]

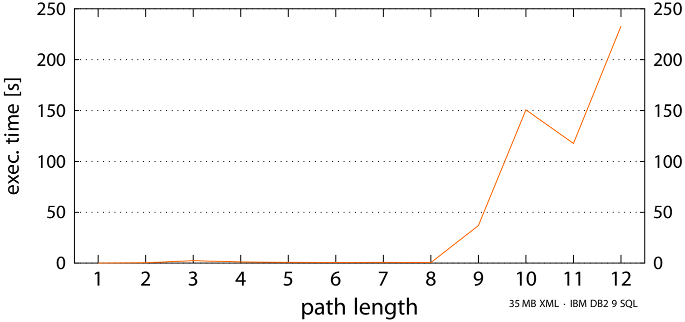
- The **inner** (rhs) relation always is a **base relation**.
- Allows the use of **index nested loops join**.
- Easier to implement in a **pipelined** fashion.

[1] The seminal **System R** prototype, *e.g.*, considered only left-deep plans.

Query Optimization
Search Space Illustration
Dynamic Programming
Example: Four-Way Join
Algorithm
Discussion
Left/Right-Deep vs. Bushy
Greedy join enumeration

## Join Order Makes a Difference

- XPath location step evaluation over relationally encoded XML data.[2]
- *n*-way self-join with a range predicate.



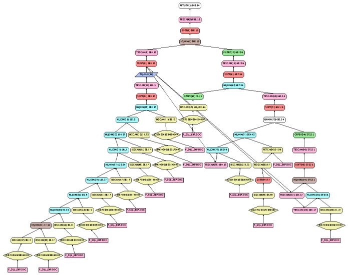

[2] ↗ Grust *et al.* Accelerating XPath Evaluation in Any RDBMS. *TODS 2004*. `http://www.pathfinder-xquery.org/`

**Query Optimization**
Search Space Illustration
Dynamic Programming
Example: Four-Way Join
Algorithm
Discussion
Left/Right-Deep vs. Bushy
Greedy join enumeration

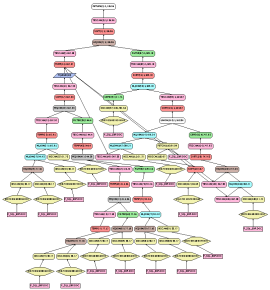
# Join Order Makes a Difference

Contrast the execution plans for a path of 8 and 9 XPath location steps:

## DB2 Join plans



**left-deep join tree**



**bushy join tree**

⇒ DB2's optimizer essentially gave up in the face of 9+ joins.

## Joining Many Relations

Dynamic programming still has **exponential** resource requirements:
(↗ K. Ono, G.M. Lohman, *Measuring the Complexity of Join Enumeration in Query Optimization*, VLDB 1990)

- time complexity: $\mathcal{O}(3^n)$
- space complexity: $\mathcal{O}(2^n)$

This may still be too expensive

- for joins involving many relations ($\sim$ 10–20 and more),
- for simple queries over well-indexed data (where the right plan choice should be easy to make).

The **greedy join enumeration** algorithm jumps into this gap.



Query Optimization
Search Space Illustration
Dynamic Programming
Example: Four-Way Join
Algorithm
Discussion
Left/Right-Deep vs. Bushy
Greedy join enumeration

# Greedy Join Enumeration



Query Optimization
Search Space Illustration
Dynamic Programming
Example: Four-Way Join
Algorithm
Discussion
Left/Right-Deep vs. Bushy
Greedy join enumeration

## Greedy join enumeration for $n$-way join

**Function:** `find_join_tree_greedy` $(q(R_1, \ldots, R_n))$

*worklist* $\leftarrow \varnothing$ ;

**for** $i = 1$ **to** $n$ **do**

  *worklist* $\leftarrow$ *worklist* $\cup$ `best_access_plan` $(R_i)$ ;

**for** $i = n$ **downto** 2 **do**

  // *worklist* $= \{P_1, \ldots, P_i\}$

  find $P_j, P_k \in$ *worklist* and $\bowtie_{\ldots}$ such that $cost(P_j \bowtie_{\ldots} P_k)$ is minimal ;

  *worklist* $\leftarrow$ *worklist* $\setminus \{P_j, P_k\} \cup \{(P_j \bowtie_{\ldots} P_k)\}$ ;

// *worklist* $= \{P_1\}$

**return** single plan left in *worklist* ;

- In each iteration, choose the **cheapest** join that can be made over the remaining sub-plans at that time (this is the “greedy” part).
- Observe that `find_join_tree_greedy ()` operates similar to finding the optimum binary tree for **Huffman coding**.

# Join Enumeration—Discussion

**Greedy join enumeration:**

- The greedy algorithm has $\mathcal{O}(n^3)$ time complexity:
  - The loop has $\mathcal{O}(n)$ iterations.
  - Each iteration looks at all remaining pairs of plans in *worklist*. An $\mathcal{O}(n^2)$ task.

**Other join enumeration techniques:**

- **Randomized algorithms:** randomly rewrite the join tree one rewrite at a time; use **hill-climbing** or **simulated annealing** strategy to find optimal plan.
- **Genetic algorithms:** explore plan space by **combining** plans ("creating offspring") and **altering** some plans randomly ("mutations").



Query Optimization
Search Space Illustration
Dynamic Programming
Example: Four-Way Join
Algorithm
Discussion
Left/Right-Deep vs. Bushy
Greedy join enumeration

# Physical Plan Properties

Consider the simple equi-join query

**Join query over TPC-H tables**

```
SELECT O.O_ORDERKEY
  FROM ORDERS O, LINEITEM L
 WHERE O.O_ORDERKEY = L.L_ORDERKEY
```

where table `ORDERS` is indexed with a **unclustered index** `OK_IDX` on column `O_ORDERKEY`.

Possible table access plans (1-relation plans) are:

`ORDERS`
- **full table scan**: estimated I/Os: $N_{\texttt{ORDERS}}$
- **index scan**: estimated I/Os: $N_{\texttt{OK\_IDX}} + N_{\texttt{ORDERS}}$.

`LINEITEM`
- **full table scan**: estimated I/Os: $N_{\texttt{LINEITEM}}$.



Query Optimization
Search Space Illustration
Dynamic Programming
Example: Four-Way Join
Algorithm
Discussion
Left/Right-Deep vs. Bushy
Greedy join enumeration

## Physical Plan Properties

- Since the **full table scan** is the cheapest access method for both tables, our join algorithms will select them as the best 1-relation plans in Pass 1.[3]

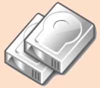
  To **join** the two scan outputs, we now have the choices
  - **nested loops join**,
  - **hash join**, or
  - **sort** both inputs, then use **merge join**.

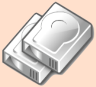
  Let us assume that sort-merge join is the preferable candidate, incurring a cost of $\approx 2 \cdot (N_{\texttt{ORDERS}} + N_{\texttt{LINEITEM}})$.

  $\Rightarrow$ **Overall cost:**
  $N_{\texttt{ORDERS}} + N_{\texttt{LINEITEM}} + 2 \cdot (N_{\texttt{ORDERS}} + N_{\texttt{LINEITEM}})$.

[3] Dynamic programming and the greedy algorithm happen to do the same in this example.







Query Optimization
Search Space Illustration
Dynamic Programming
Example: Four-Way Join
Algorithm
Discussion
Left/Right-Deep vs. Bushy
Greedy join enumeration

## Physical Plan Properties—A Better Plan

It is easy to see, however, that there is a better way to evaluate the query:

1. Use an **index scan** to access ORDERS. This guarantees that the scan output is already **in** O_ORDERKEY **order**.
2. Then only **sort** LINEITEM and
3. join using **merge join**.

⇒ **Overall cost:** $\underbrace{(N_{\texttt{OK\_IDX}} + N_{\texttt{ORDERS}})}_{1} + \underbrace{2 \cdot N_{\texttt{LINEITEM}}}_{2\,/\,3}$.

Although more expensive as a standalone table access plan, the **use of the index (order enforcement) pays off later on** in the overall plan.

# Physical Plan Properties: Interesting Orders

- The advantage of the index-based access to `ORDERS` is that it provides beneficial **physical properties**.
- Optimizers, therefore, keep track of such properties by **annotating** candidate plans.
- System R introduced the concept of **interesting orders**, determined by
  - `ORDER BY` or `GROUP BY` clauses in the input query, or
  - join attributes of subsequent joins ($\rightsquigarrow$ merge join).

$\Rightarrow$ In `prune_plans ()`, retain

- the cheapest "unordered" plan **and**
- the cheapest plan for each interesting order.



Query Optimization
Search Space Illustration
Dynamic Programming
Example: Four-Way Join
Algorithm
Discussion
Left/Right-Deep vs. Bushy
Greedy join enumeration

# Query Rewriting

- Join optimization essentially takes a set of relations and a set of join predicates to find the best join order.
- By **rewriting** query graphs beforehand, we can improve the effectiveness of this procedure.
- The **query rewriter** applies **heuristic rules**, without looking into the actual database state (no information about cardinalities, indexes, etc.).
  In particular, the optimizer
  - **relocates predicates** (predicate pushdown),
  - **rewrites predicates**, and
  - **unnests queries**.



Query Optimization
Search Space Illustration
Dynamic Programming
Example: Four-Way Join
Algorithm
Discussion
Left/Right-Deep vs. Bushy
Greedy join enumeration

# Predicate Simplification

Rewrite

### Example (Query against TPC-H table)

```sql
    SELECT *
      FROM LINEITEM L
     WHERE L.L_TAX * 100 < 5
```

into

### Example (Query after predicate simplification)

```sql
    SELECT *
      FROM LINEITEM L
     WHERE L.L_TAX  < 0.05
```

### ✎ In which sense is the rewritten predicate simpler?

Why would a RDBMS query optimizer rewrite the selection predicate as shown above?



Query Optimization
Search Space Illustration
Dynamic Programming
Example: Four-Way Join
Algorithm
Discussion
Left/Right-Deep vs. Bushy
Greedy join enumeration

# Introducing Additional Join Predicates

**Implicit join predicates** as in

**Implicit join predicate through transitivity**

```sql
SELECT *
  FROM A, B, C
 WHERE A.a = B.b AND B.b = C.c
```

can be turned into explicit ones:

**Explicit join predicate**

```sql
SELECT *
  FROM A, B, C
 WHERE A.a = B.b AND B.b = C.c
   AND A.a = C.c
```

This makes the following join tree feasible:

$$(\mathtt{A} \bowtie \mathtt{C}) \bowtie \mathtt{B} \ .$$

(Note: $(\mathtt{A} \bowtie \mathtt{C})$ would have been a Cartesian product before.)



Query Optimization
Search Space Illustration
Dynamic Programming
Example: Four-Way Join
Algorithm
Discussion
Left/Right-Deep vs. Bushy
Greedy join enumeration

# Nested Queries and Correlation

SQL provides a number of ways to write **nested queries**.

- **Uncorrelated** sub-query:

**No free variables in subquery**

```
SELECT *
  FROM ORDERS O
 WHERE O_CUSTKEY IN (SELECT C_CUSTKEY
                       FROM CUSTOMER
                      WHERE C_NAME = 'IBM␣Corp.')
```

- **Correlated** sub-query:

**Row variable O occurs free in subquery**

```
SELECT *
  FROM ORDERS O
 WHERE O.O_CUSTKEY IN
         (SELECT C.C_CUSTKEY
            FROM CUSTOMER C
           WHERE C.C_ACCTBAL < O.O_TOTALPRICE)
```

Query Optimization
Search Space Illustration
Dynamic Programming
Example: Four-Way Join
Algorithm
Discussion
Left/Right-Deep vs. Bushy
Greedy join enumeration

# Query Unnesting

- Taking query nesting literally might be **expensive**.
  - An uncorrelated query, *e.g.*, need not be re-evaluated for every tuple in the outer query.
- Oftentimes, sub-queries are only used as a syntactical way to express a **join** (or a semi-join).
- The query rewriter tries to detect such situations and **make the join explicit**.
- This way, the sub-query can become part of the regular **join order optimization**.

## Turning correlation into joins

Reformulate the correlated query of slide 32 (use SQL syntax or relational algebra) to remove the correlation (and introduce a join).

↗ Won Kim. On Optimizing an SQL-like Nested Query. *ACM TODS*, vol. 7, no. 3, September 1982.



Query Optimization
Search Space Illustration
Dynamic Programming
Example: Four-Way Join
Algorithm
Discussion
Left/Right-Deep vs. Bushy
Greedy join enumeration

## Summary

### Query Parser

Translates input query into (SFW-like) **query blocks**.

### Rewriter

Logical (database state-independent) optimizations; predicate simplification; query unnesting.

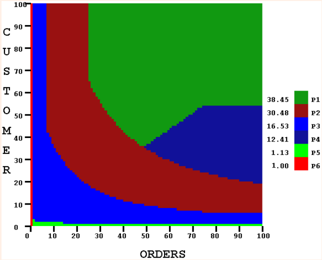
### (Join) Optimization

Find “best” query execution plan based on a **cost model** (considering I/O cost, CPU cost, . . . ); data statistics (histograms); dynamic programming, greedy join enumeration; physical plan properties (interesting orders).

Database optimizers still are true pieces of art. . .

## “Picasso” Plan Diagrams

**Generated by “Picasso”: SQL join query with filters of parameterizable selectivities (0 . . . 100) against both join inputs**




↗ Naveen Reddy and Jayant Haritsa. Analyzing Plan Diagrams of Database Query Optimizers. *VLDB 2005*.




Query Optimization

Search Space Illustration

Dynamic Programming

Example: Four-Way Join

Algorithm

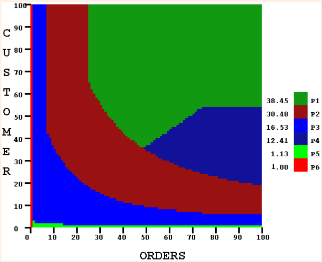
Discussion

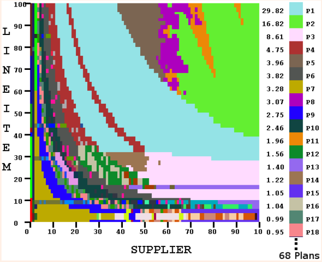
Left/Right-Deep vs. Bushy

Greedy join enumeration

# "Picasso" Plan Diagrams

**Generated by "Picasso": SQL join query with filters of parameterizable selectivities (0 . . . 100) against both join inputs**



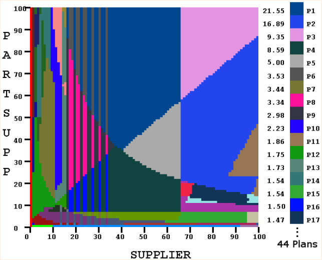

↗ Naveen Reddy and Jayant Haritsa. Analyzing Plan Diagrams of Database Query Optimizers. *VLDB 2005*.



Query Optimization
Search Space Illustration
Dynamic Programming
Example: Four-Way Join
Algorithm
Discussion
Left/Right-Deep vs. Bushy
Greedy join enumeration

# “Picasso” Plan Diagrams

**Generated by “Picasso”: each distinct color represent a distinct plan considered by the DBMS**

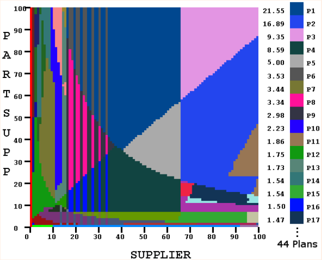


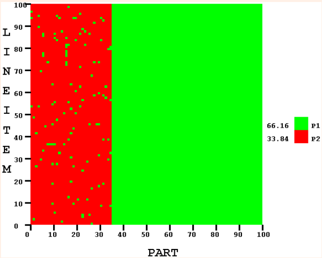



Query Optimization
Search Space Illustration
Dynamic Programming
Example: Four-Way Join
Algorithm
Discussion
Left/Right-Deep vs. Bushy
Greedy join enumeration

# "Picasso" Plan Diagrams

**Generated by "Picasso": each distinct color represent a distinct plan considered by the DBMS**




Download Picasso at

`http://dsl.serc.iisc.ernet.in/projects/PICASSO/index.html.`