

Architecture and Implementation of Database Systems (Winter 2013/14)

Jens Teubner, DBIS Group
`jens.teubner@cs.tu-dortmund.de`

Winter 2013/14

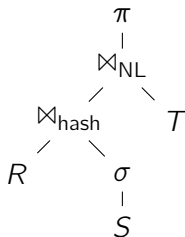
Part VI

Query Optimization

Finding the “Best” Query Plan

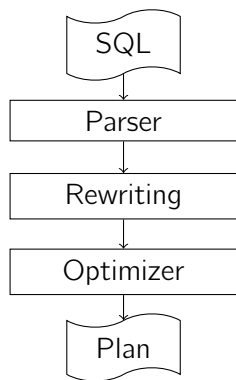
SELECT ...
FROM ...
WHERE ...

?



- 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.

Plan Generation Process

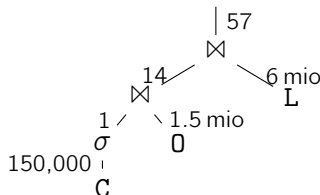
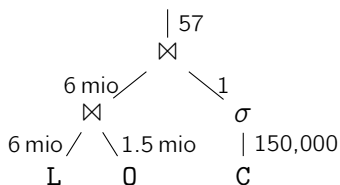


- **Parser:** syntactical/semantical analysis
- **Rewriting:** optimizations **independent** of the current database state (table sizes, availability of indexes, etc.)
- **Optimizer:** optimizations that rely on a **cost model** and information about the current database state
- The resulting **plan**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

Impact on Performance

Finding the right plan can dramatically impact performance.

```
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.'
```



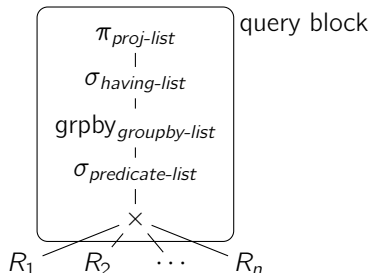
- In terms of execution times, these differences can easily mean “seconds versus days.”

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**.

```
SELECT proj-list
  FROM  $R_1, R_2, \dots, R_n$ 
 WHERE predicate-list
  GROUP BY groupby-list
  HAVING having-list
```

→



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

Finding the “Best” Execution Plan

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

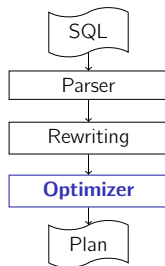
It is then the **optimizer’s** task to come up with the optimal **execution plan** for the given query.

Essentially, the optimizer

- 1 **enumerates** all possible execution plans,
- 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?



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

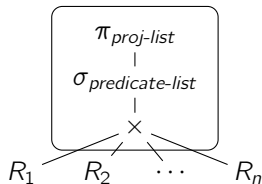
- the number of **disk I/Os** required to evaluate the plan,
- the plan's **CPU cost**,
- the overall **response time** observable by the user as well as the total **execution time**.

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**.

Result Size Estimation

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



We can estimate the result size of Q based on

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

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

Table Cardinalities

If not coming from another query block, the size $|R|$ of an input table R is available in the DBMS's **system catalogs**.

E.g., IBM DB2:

```
db2 => SELECT TABNAME, CARD, NPAGES  
db2 (cont.) => FROM SYSCAT.TABLES  
db2 (cont.) => WHERE TABSCHEMA = 'TPCH';
```

TABNAME	CARD	NPAGES
ORDERS	1500000	44331
CUSTOMER	150000	6747
NATION	25	2
REGION	5	1
PART	200000	7578
SUPPLIER	10000	406
PARTSUPP	800000	31679
LINEITEM	6001215	207888

8 record(s) selected.

Estimating Selectivities

To estimate the selectivity of a predicate, we look at its structure.

column = value

$$sel(\cdot) = \begin{cases} 1/|I| & \text{if there is an index } I \text{ on } column \\ 1/10 & \text{otherwise} \end{cases}$$

column₁ = column₂

$$sel(\cdot) = \begin{cases} \frac{1}{\max\{|I_1|, |I_2|\}} & \text{if there are indexes on **both** cols.} \\ \frac{1}{|I_k|} & \text{if there is an index only on col. } k \\ 1/10 & \text{otherwise} \end{cases}$$

p₁ AND p₂

$$sel(\cdot) = sel(p_1) \cdot sel(p_2)$$

p₁ OR p₂

$$sel(\cdot) = sel(p_1) + sel(p_2) - sel(p_1) \cdot sel(p_2)$$

Improving Selectivity Estimation

The selectivity rules we saw make a fair amount of assumptions:

- **uniform distribution** of data values within a column,
- **independence** between individual predicates.

Since these assumptions aren't generally met, systems try to improve selectivity estimation by gathering **data statistics**.

- These statistics are collected offline and stored in the system catalog.

 IBM DB2: RUNSTATS ON TABLE ...

- The most popular type of statistics are **histograms**.

```
SELECT SEQNO, COLVALUE, VALCOUNT
FROM SYSCAT.COLDIST
WHERE TABNAME = 'LINEITEM'
AND COLNAME = 'L_EXTENDEDPRICE'
AND TYPE = 'Q';
```

SEQNO	COLVALUE	VALCOUNT
1	+0000000000996.01	3001
2	+0000000004513.26	315064
3	+0000000007367.60	633128
4	+0000000011861.82	948192
5	+0000000015921.28	1263256
6	+0000000019922.76	1578320
7	+0000000024103.20	1896384
8	+0000000027733.58	2211448
9	+0000000031961.80	2526512
10	+0000000035584.72	2841576
11	+0000000039772.92	3159640
12	+0000000043395.75	3474704
13	+0000000047013.98	3789768
	⋮	

SYSCAT.COLDIST also contains information like

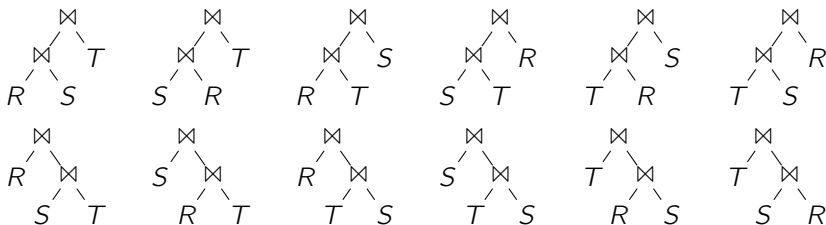
- the n most frequent values (and their frequency),
- the number of **distinct** values in each histogram bucket.

Histograms may even be manipulated **manually** to tweak the query optimizer.

Join Optimization

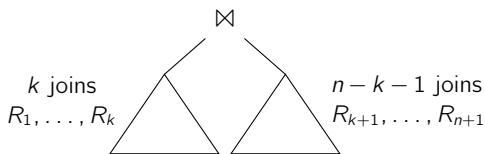
- We've now translated the query into a graph of **query blocks**.
 - Query blocks essentially are a **multi-way** Cartesian product with a number of selection predicates on top.
- We can estimate the **cost** of a given **execution plan**.
 - Use result size estimates in combination with the cost for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

We are now ready to **enumerate** all possible execution plans, *e.g.*, all possible **3-way** join combinations for a query block.



How Many Such Combinations Are There?

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



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

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

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

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

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

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

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

possibilities to evaluate an $(n + 1)$ -way join.

The resulting search space is **enormous**:

number of relations n	C_{n-1}	join trees
2	1	2
3	5	12
4	14	120
5	42	1,680
6	132	30,240
7	429	665,280
8	1,430	17,297,280
10	16,796	17,643,225,600

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

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** ($1 \leq i < k$).

Assumption:

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

Example: Four-Way Join

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 ($R_i \bowtie R_j$ or $R_j \bowtie R_i$):

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

→ 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:

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

→ 24 plans to consider.

Example (cont.)

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:

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


→ 14 plans to consider.

- Overall, we looked at only **50** (sub-)plans (instead of the possible 120 four-way join plans; ↗ slide 187).
- All decisions required the evaluation of **simple** sub-plans only (no need to re-evaluate the interior of $\text{optPlan}(\cdot)$).

Dynamic Programming Algorithm

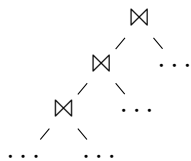
```
1 Function: find_join_tree_dp ( $q(R_1, \dots, R_n)$ )
2 for  $i = 1$  to  $n$  do
3    $optPlan(\{R_i\}) \leftarrow access\_plans(R_i)$  ;
4    $prune\_plans(optPlan(\{R_i\}))$  ;
5 for  $i = 2$  to  $n$  do
6   foreach  $S \subseteq \{R_1, \dots, R_n\}$  such that  $|S| = i$  do
7      $optPlan(S) \leftarrow \emptyset$  ;
8     foreach  $O \subset S$  do
9        $optPlan(S) \leftarrow optPlan(S) \cup$ 
10         $possible\_joins(optPlan(O), optPlan(S \setminus O))$ ;
11      $prune\_plans(optPlan(S))$  ;
12 return  $optPlan(\{R_1, \dots, R_n\})$  ;
```

- $possible_joins(R, S)$ enumerates the possible joins between R and S (nested loops join, merge join, etc.).
- $prune_plans(set)$ discards all but the best plan from set .

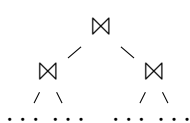
- `find_join_tree_dp()` draws its advantage from **filtering** plan candidates early in the process.
 - In our example on slide 189, pruning in Pass 2 reduced the search space by a factor of 2, and another factor of 6 in Pass 3.
- Some **heuristics** can be used to prune even more plans:
 - Try to avoid **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 UDB: SET CURRENT QUERY OPTIMIZATION = n**

Left/Right-Deep vs. Bushy Join Trees

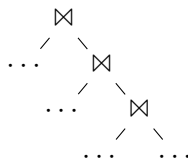
The algorithm on slide 191 explores all possible shapes a join tree could take:



left-deep



bushy
(everything else)



right-deep

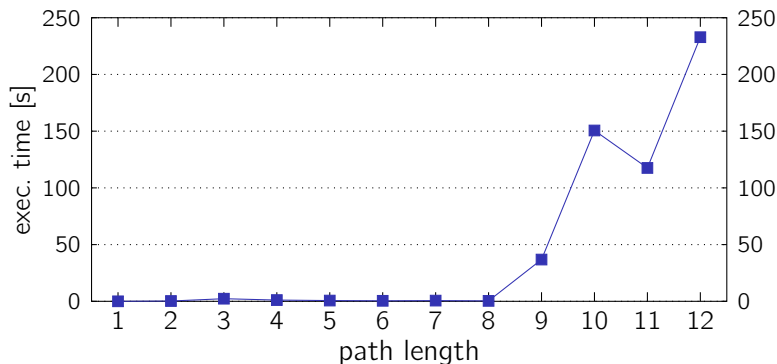
Actual systems often prefer **left-deep** join trees.¹⁴

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

¹⁴The seminal **System R** prototype, *e.g.*, considered only left-deep plans.

Join Order Makes a Difference

- XPath evaluation over relationally encoded XML data¹⁵
- n -way self-join with a range predicate.

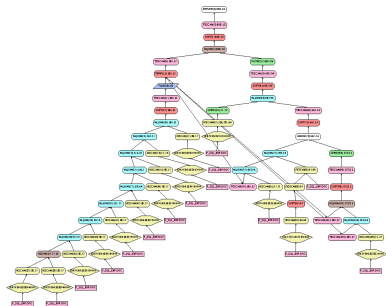


35 MB XML · IBM DB2 9 SQL

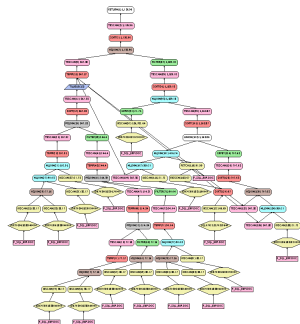
¹⁵ ↗ Grust *et al.* Accelerating XPath Evaluation in Any RDBMS. *TODS 2004*.
<http://www.pathfinder-xquery.org/>

Join Order Makes a Difference

Contrast the execution plans for a 8- and a 9-step path.



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:

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

This may still be too expensive

- for joins involving many relations (~ 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.

Greedy Join Enumeration

```
1 Function: find_join_tree_greedy ( $q(R_1, \dots, R_n)$ )
2 worklist  $\leftarrow \emptyset$  ;
3 for  $i = 1$  to  $n$  do
4    $\left[ \textit{worklist} \leftarrow \textit{worklist} \cup \text{best\_access\_plan}(R_i) ; \right.$ 
5 for  $i = n$  downto  $2$  do
6    $\left[ \begin{array}{l} // \textit{worklist} = \{P_1, \dots, P_i\} \\ \text{find } P_j, P_k \in \textit{worklist} \text{ and } \bowtie \dots \text{ such that } \text{cost}(P_j \bowtie \dots P_k) \text{ is minimal ;} \\ \textit{worklist} \leftarrow \textit{worklist} \setminus \{P_j, P_k\} \cup \{(P_j \bowtie \dots P_k)\} ; \end{array} \right.$ 
7    $\left. // \textit{worklist} = \{P_1\} \right.$ 
8 return single plan left in worklist ;
```

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

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”).

Physical Plan Properties

Consider the query

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

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

Possible table access plans are:

`ORDERS` ■ **full table scan**: estimated I/Os: N_{ORDERS}
 ■ **index scan**: estimated I/Os: $N_{OK_IDX} + N_{ORDERS}$.

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

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 1.¹⁶

To **join** the two scan outputs, we now have the choices

- **nested loops join**,
- **hash join**, or
- **sort** both inputs, then use **merge join**.

Hash join or sort-merge join are probably the preferable candidates here, incurring a cost of $\approx 2(N_{\text{ORDERS}} + N_{\text{LINEITEM}})$.

→ **overall cost:** $N_{\text{ORDERS}} + N_{\text{LINEITEM}} + 2(N_{\text{ORDERS}} + N_{\text{LINEITEM}})$.

¹⁶Dynamic programming and the greedy algorithm happen to do the same in this example.

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**.

$$\rightarrow \text{overall cost: } \underbrace{(N_{\text{OK_IDX}} + N_{\text{ORDERS}})}_{1.} + \underbrace{2 \cdot N_{\text{LINEITEM}}}_{2./3.}$$

Although more expensive as a standalone table access plan, the use of the index pays off in the overall plan.

- 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).
- In `prune_plans ()`, retain
 - the cheapest “unordered” plan **and**
 - the cheapest plan for each interesting order.

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, it

- **rewrites predicates** and
- **unnests queries.**

Predicate Simplification

Example: rewrite

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

into

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

- Predicate simplification may enable the use of **indexes** and simplify the detection of opportunities for join algorithms.

Additional Join Predicates

Implicit join predicates as in

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

can be turned into explicit ones:

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

This enables plans like

$$(A \bowtie C) \bowtie B .$$

((A \bowtie C) would have been a Cartesian product before.)

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

- **Uncorrelated** sub-query:

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

- **Correlated** sub-query:

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

- 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**.

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

Query Parser

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

Rewriter

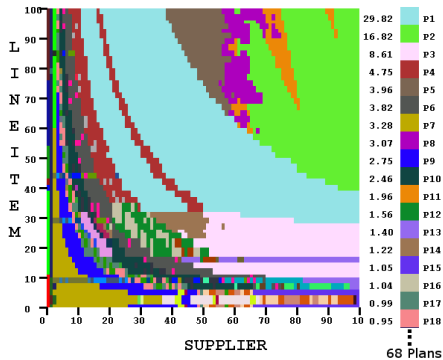
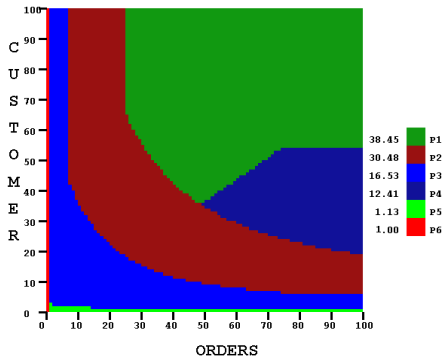
Logical (database state-independent) optimizations; predicate simplification; query unnesting.

(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).

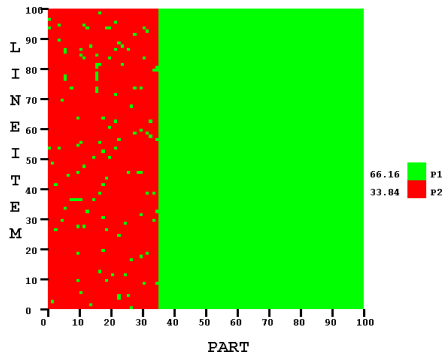
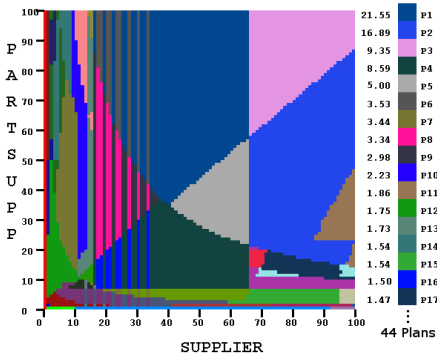
Database optimizers still are true pieces of art. . .

“Picasso” Plan Diagrams



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

“Picasso” Plan Diagrams



Download Picasso at

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