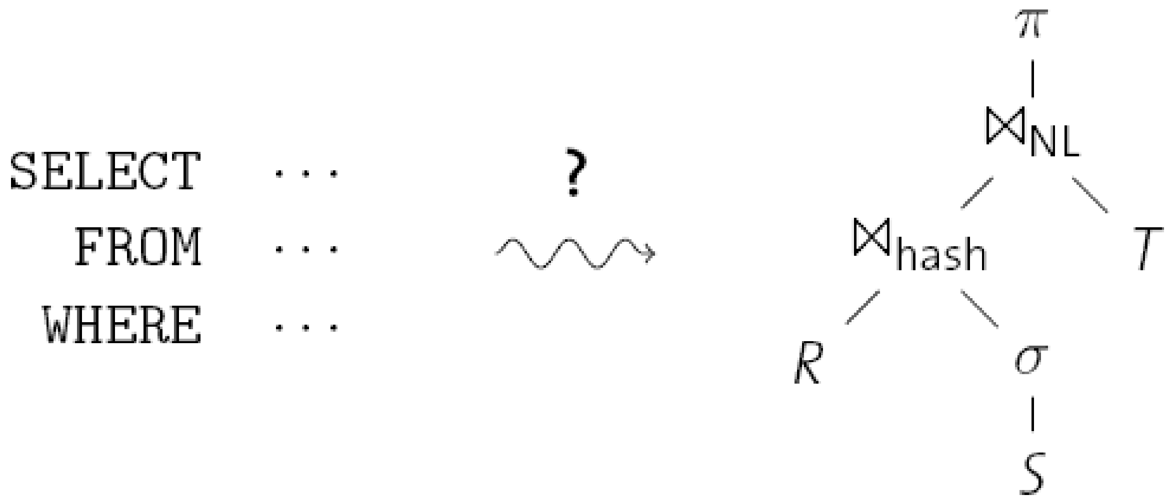
Systems Infrastructure for Data Science

Web Science Group
Uni Freiburg
WS 2013/14

Lecture V: Query Optimization

Finding the “Best” Query Plan

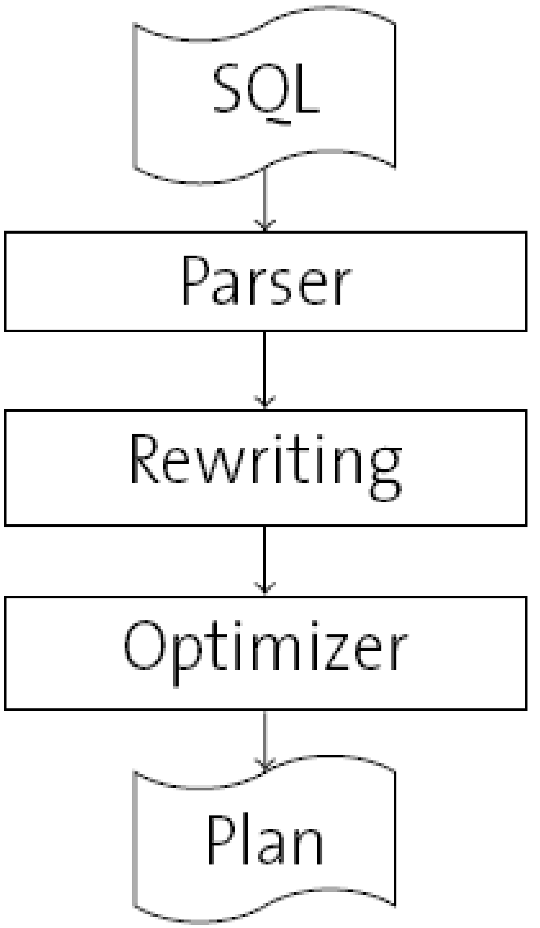


- We already saw that there may be more than one way to answer a given query.
  - Which one of the join operators should we pick? With which parameters (block size, buffer allocation, ...)?
- The task of finding the best execution plan is, in fact, the “holy grail” of any database implementation.

Query 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

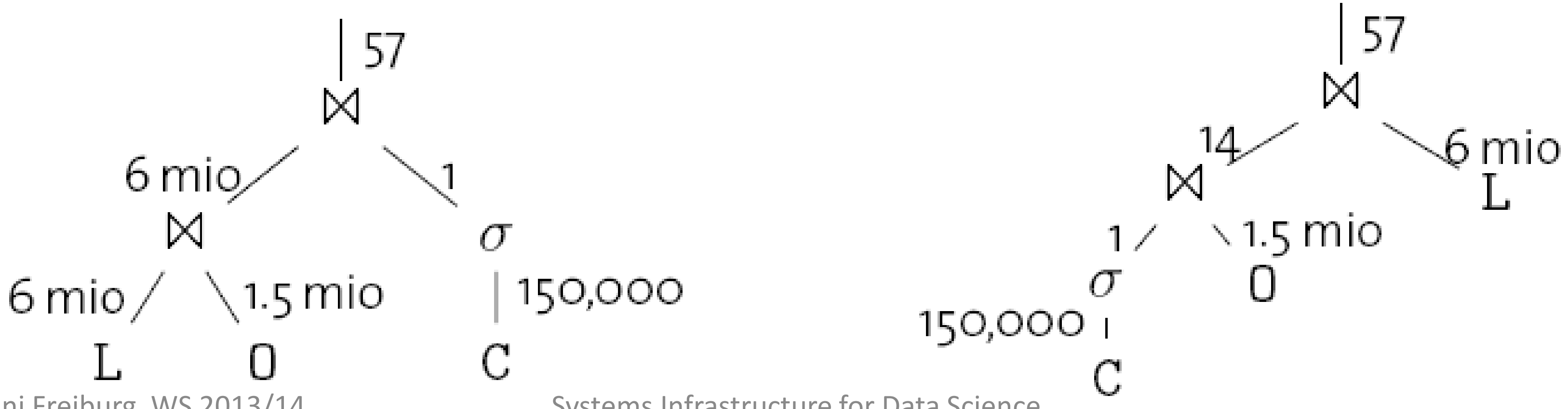
➢ The resulting plan is then evaluated by the system's **execution engine**.



Impact on Performance

- Finding the right plan can dramatically impact performance.
- In terms of execution times, these differences can easily mean “seconds vs. days”.

```
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.'
```

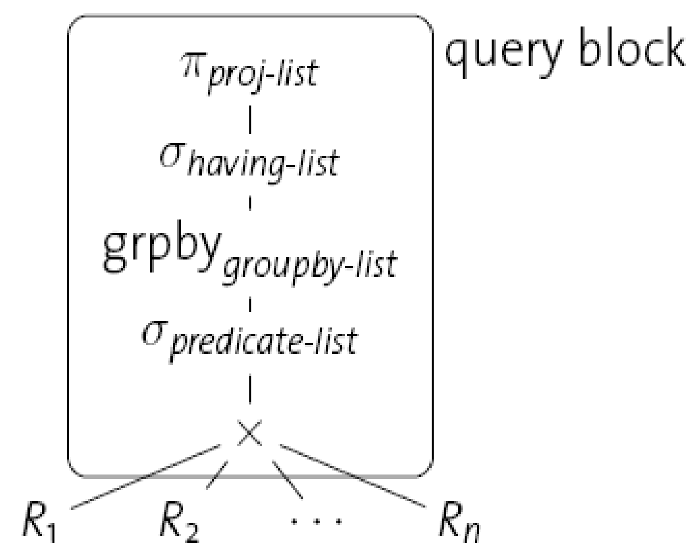


The SQL Parser

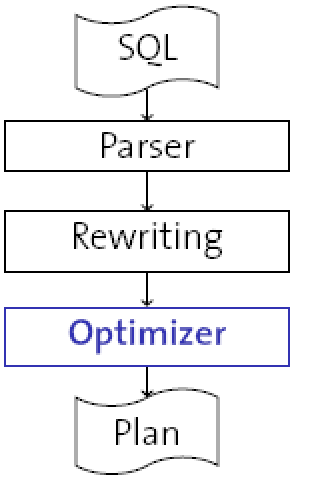
- Besides some analyses regarding the syntactical and semantical correctness of the input query, the parser creates an **internal representation** of the input query.
- This representation still resembles the original query:
  - Each SELECT-FROM-WHERE clause is translated into a **query block**.
  -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

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

$\rightarrow$



Finding the "Best" Execution Plan



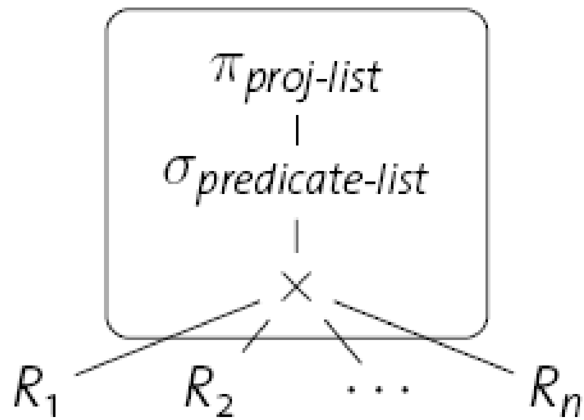
- The parser output is fed into a **rewrite engine** which, again, yields a tree of query blocks.
- It is then the optimizer's task to come up with the **optimal execution plan** for the given query.
- Essentially, the optimizer
  1. **enumerates** all possible execution plans,
  2. determines the **quality (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?

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 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 SELECT-FROM-WHERE query *Q*.



- We can estimate the result size of *Q* based on
  - the size of the input tables, $|R_1|$, …, $|R_n|$, and
  - the **selectivity** *sel()* of the predicate *predicate-list*.

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

Table Cardinalities

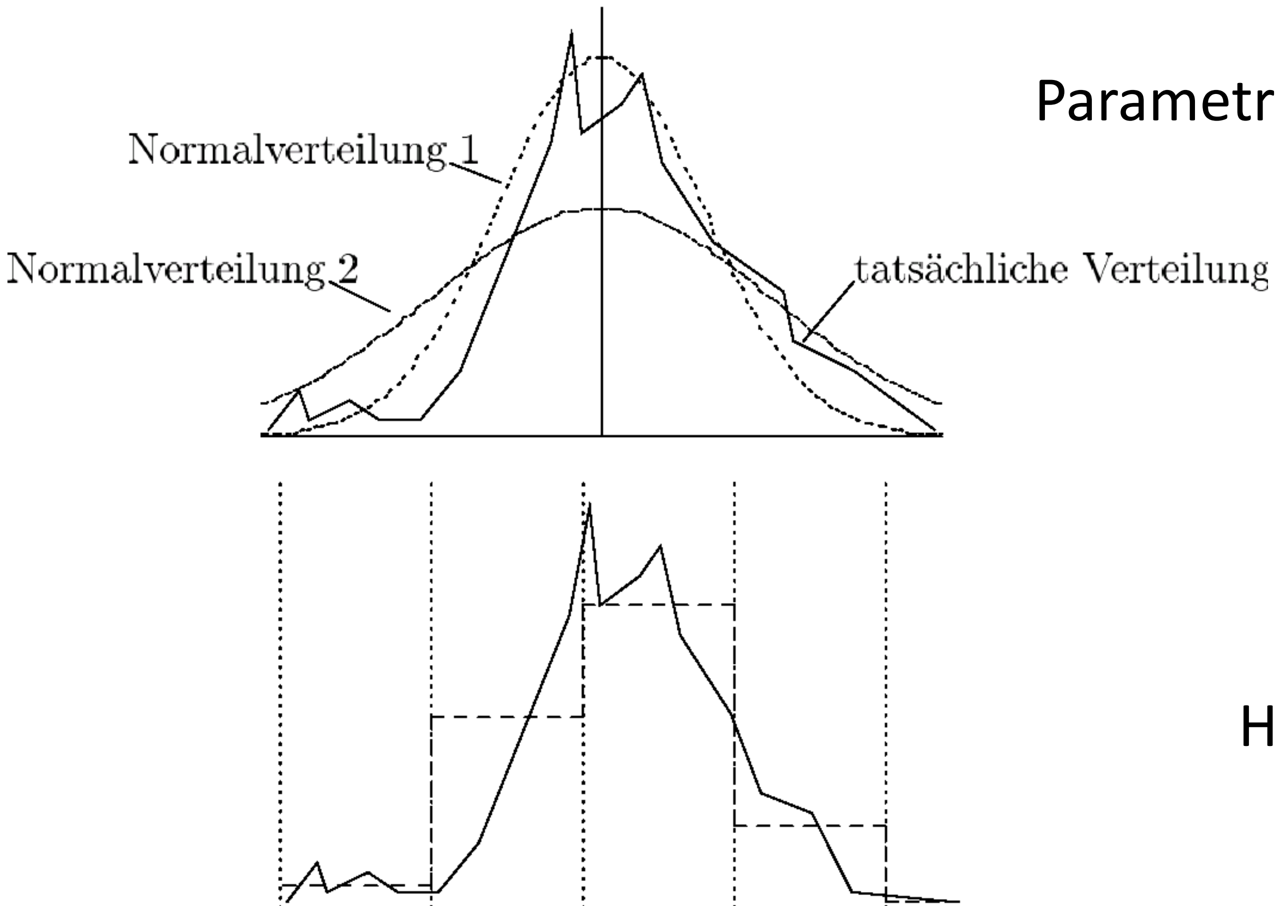
- If not coming from another query block, the size $|R|$ of an input table R is available in the DBMS's **system catalogs**.
- E.g., IBM DB2:

```
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.
```

Selectivity Estimation

- General selectivity rules 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.
    - Example: IBM DB2: **`RUNSTATS ON TABLE ...`**
  - The most popular type of statistics are **histograms**.

Describing Value Distribution



Parametric Distribution

Histogram

Figure © A. Kemper

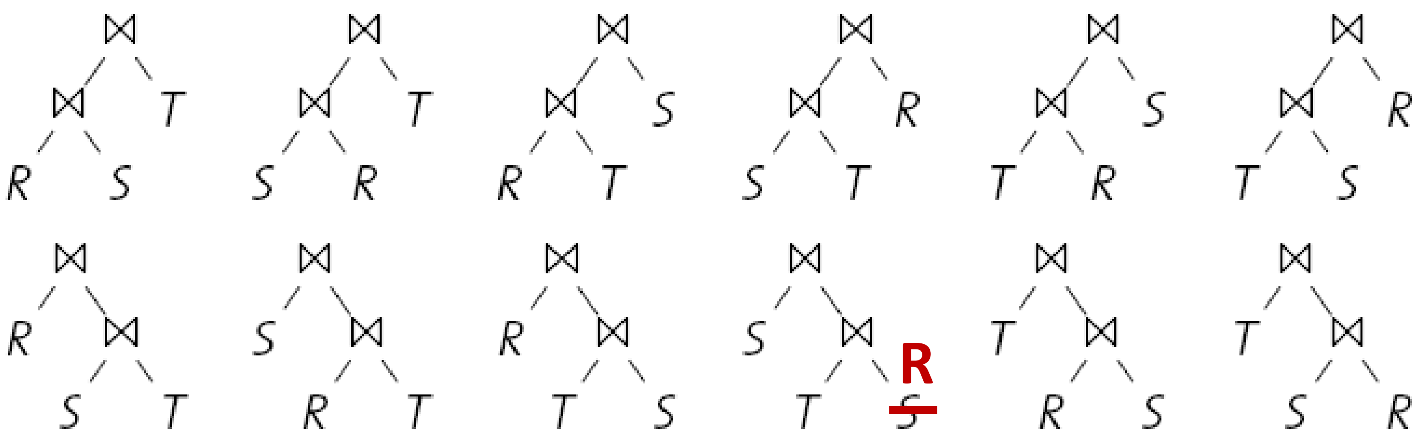
Example: Histograms in IBM DB2

- **`SYSCAT.COLDIST`** also contains information like:
  - the *n* most **frequent values** and their frequency,
  - the number of **distinct values** in each histogram bucket.
- Some explanation:
  - SEQNO: Frequency rank
  - COLVALUE is a single value
  - VALCOUNT with TYPE=Q shows the number of colums with value <= COLVALUE (Why?)

```
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
                  :
```

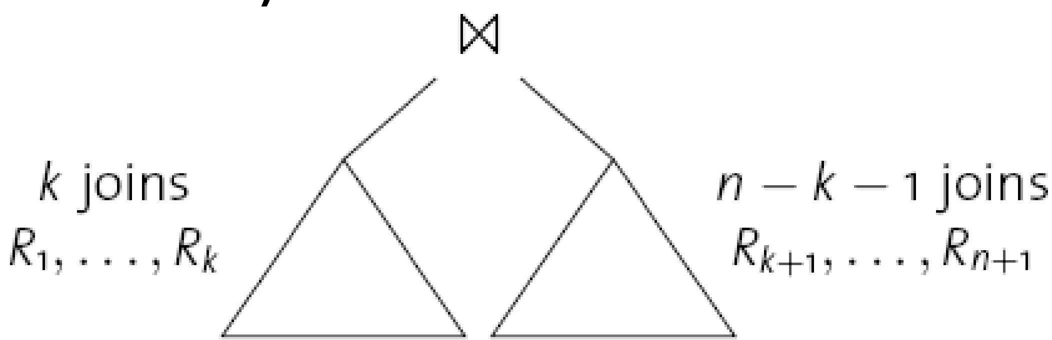
Join Optimization (R ⋈ S ⋈ T)

- We've now translated the query into a **graph of query blocks**.
  - Query blocks essentially are **multi-way Cartesian products** 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 that we saw in the previous lecture.
- We are now ready to **enumerate all possible execution plans**, i.e., all possible 3-way join combinations for each query block.



How Many 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 ($0 \le k \le n-1$):



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

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

Catalan Numbers

- This recurrence relation is satisfied by **Catalan numbers** describing the number of ordered binary trees with *n+1* leaves:

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

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

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

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

Search Space

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

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 ($1 \leq i < k$).
- Assumption:
  - To find the **optimal global plan**, it is sufficient to only consider the optimal plans of its sub-queries.

Example: Four-relation Join

- **Pass 1:** (best 1-relation plans)
  - Find the best **access path** to each of the R_i individually.
- **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 $R_j \bowtie R_i$?):

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

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:

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

24 plans to consider

Example: Four-relation Join (cont'd)

- **Pass 4:** (best 4-relation plans)
  - 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:

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

14 plans to consider

- ➢ Overall, we looked at only **50** (sub-)plans (12+24+14=50 instead of the possible **120** four-way join plans shown in slide # 16).
- ➢ All decisions required the evaluation of **simple sub-plans** only (no need to re-evaluate the interior of *optPlan*()).

Dynamic Programming Algorithm

```
Function: find_join_tree_dp (q(R1, ..., Rn))
for i = 1 to n do
    optPlan({Ri}) ← access_plans (Ri) ;
    prune_plans (optPlan({Ri})) ;
for i = 2 to n do
    foreach S ⊆ {R1, ..., Rn} such that |S| = i do
        optPlan(S) ← ∅ ;
        foreach O ⊂ S do
            optPlan(S) ← optPlan(S) ∪
                    possible_joins (optPlan(O), optPlan(S \ O));
        prune_plans (optPlan(S)) ;
return optPlan({R1, ..., Rn}) ;
```

- *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*.

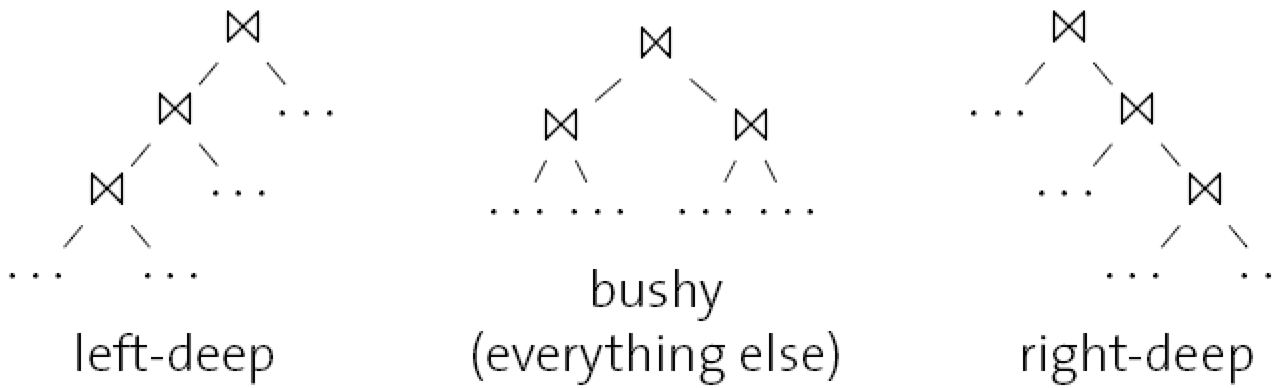
Dynamic Programming: Discussion

- *find_join_tree_dp()* draws its advantage from **filtering** plan candidates early in the process.
  - In our example, 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 the next slides).
- Such heuristics can be used as a handle to balance plan quality and optimizer runtime.
  - Example: IBM DB2:

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

Left/Right-Deep vs. Bushy Join Trees

- The dynamic programming algorithm explores all possible shapes a join tree could take:



- Actual systems often prefer **left-deep join trees** (e.g., the seminal IBM System R prototype considered only left-deep plans).
  - The **inner** relation is always a **base relation**.
  - Allows the use of **index nested loops join**.
  - Easier to implement in a **pipelined** fashion.

Joining Many Relations

- Dynamic programming still has **exponential** resource requirements:
  - time complexity: $O(3^n)$
  - space complexity: $O(2^n)$
- 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** targets solving this case.

Greedy Join Enumeration

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.

Greedy Join Enumeration: Discussion

- Greedy join enumeration:
  - The greedy algorithm has $O(n^3)$ time complexity.
    - The loop has $O(n)$ iterations.
    - Each iteration looks at all remaining pairs of plans in *worklist*: an $O(n^2)$ 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: $N_{LINEITEM}$

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 (in both DP and GJE).
- To join the two scan outputs, we now have the following choices:
  - nested loops join, or
  - 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 ~ $2(N_{ORDERS} + N_{LINEITEM})$.
  - Overall cost: $N_{ORDERS} + N_{LINEITEM} + 2(N_{ORDERS} + N_{LINEITEM})$.

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:** $\underbrace{(N_{OK\_IDX} + N_{ORDERS})}_{1} + \underbrace{2 * N_{LINEITEM}}_{2+3}$
- Although more expensive as a standalone table access plan, the use of the index pays off in the overall plan.

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.
- IBM 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 (merge join).
- In *prune_plans()*, retain
  - the cheapest "unordered" plan **and**
  - the cheapest plan for each interesting order.

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, it
  - **Pushes predicates and projections**
  - **rewrites predicates**, and
  - **unnests queries**.

Predicate/Projection Pushdown

- Applies **heuristics** to exploits **equivalence transformations** in relational algebra
- Some examples:
  1. $\sigma_{c1 \wedge c2 \wedge \ldots \wedge cn}(R) \equiv \sigma_{c1}(\sigma_{c2}(\ldots(\sigma_{cn}(R))\ldots))$
  2. $\sigma_{c1}(\sigma_{c2}((R)) \equiv \sigma_{c2}(\sigma_{c1}((R))$
  3. If $L_1 \subseteq L_2 \subseteq \ldots \subseteq L_n$:
     $\pi_{L1}(\pi_{L2}(\ldots(\pi_{Ln}(R))\ldots)) \equiv \pi_{L1}(R)$
  4. If selection only refers to attributes $A_1, \ldots, A_n$
     $\pi_{A_1, \ldots, A_n}(\sigma_c(R)) \equiv \sigma_c(\pi_{A_1, \ldots, A_n}(R))$
  5. $\times$, $\cup$, $\cap$ und $\bowtie$ are commutative
     $R \bowtie_c S \equiv S \bowtie_c R$ (we already used this)

More equivalence rules

1. If *c only accesses attributes in R*
$$\sigma_c(R \bowtie_j S) \equiv \sigma_c(R) \bowtie_j S$$
2. If c is a conjunction„$c_1 \wedge c_2$“, c_1 only accesses attribues in R, c_2 in S
$$\sigma_c(R \bowtie_j S) \equiv \sigma_c(R) \bowtie_j (\sigma_{c_2}(S))$$
3. Similar rules exist for projection

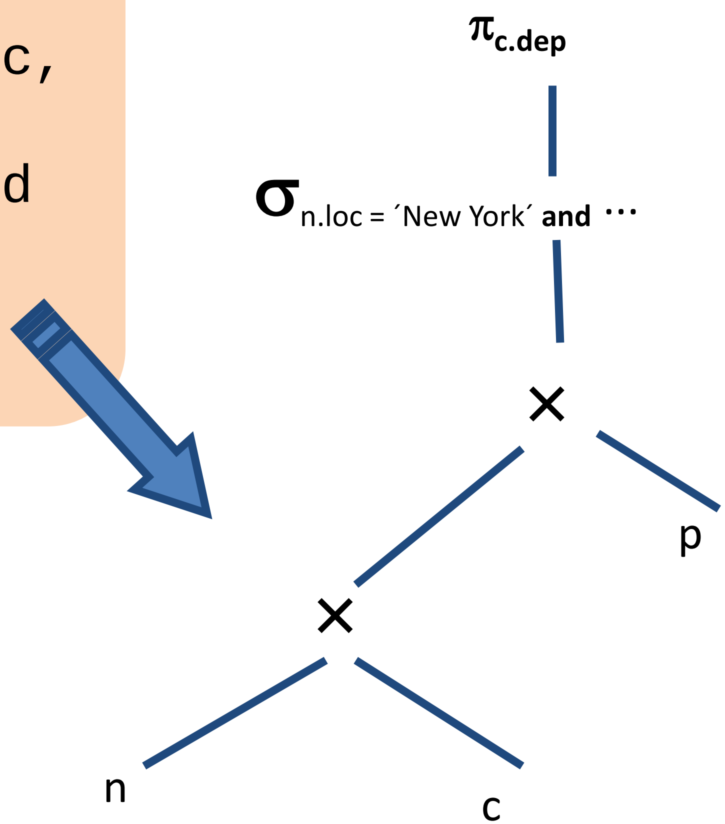
Heuristics:

- Push down predicates
- Push down projection

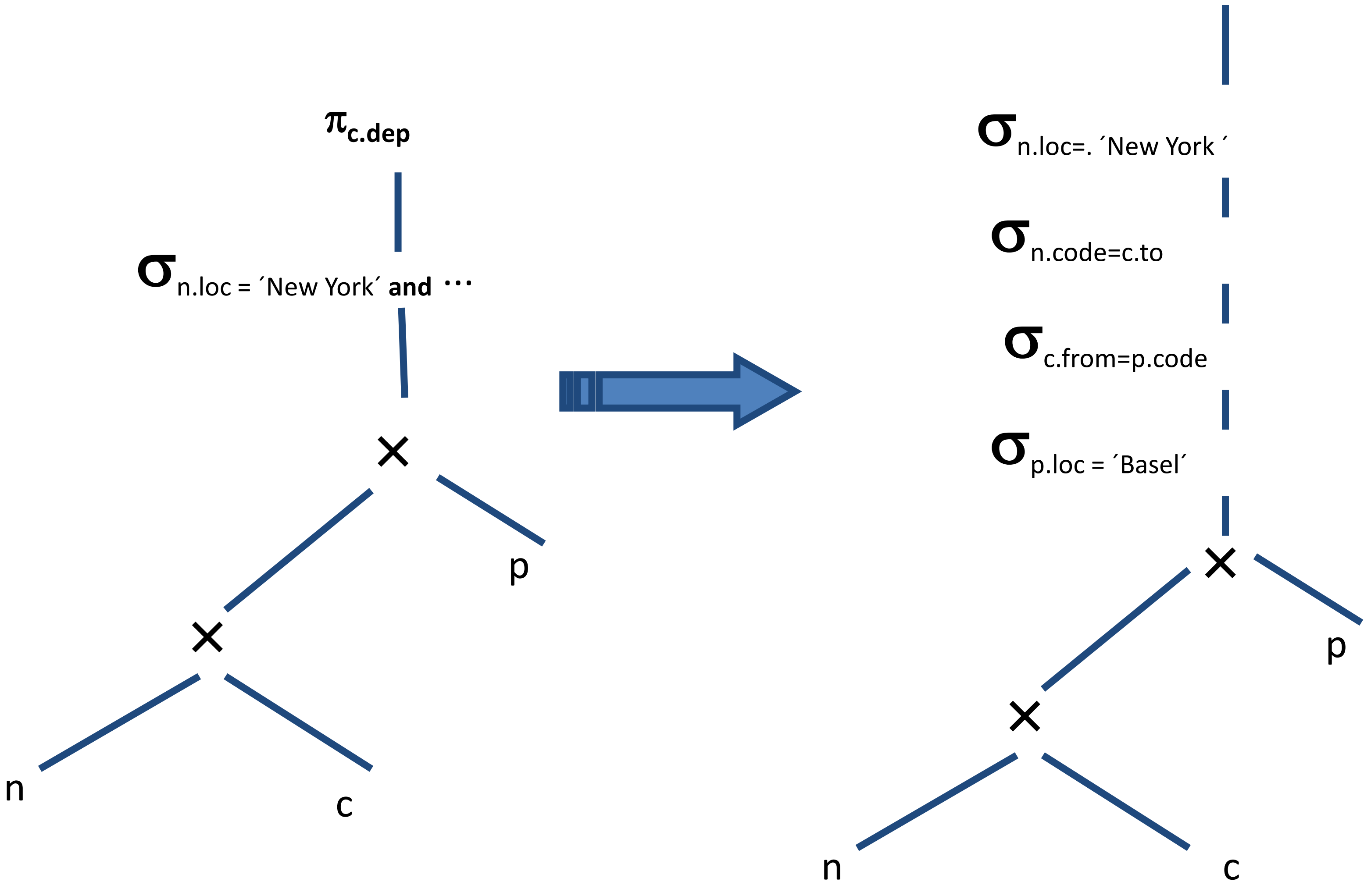
Example

- Direct flights from Basel to New York

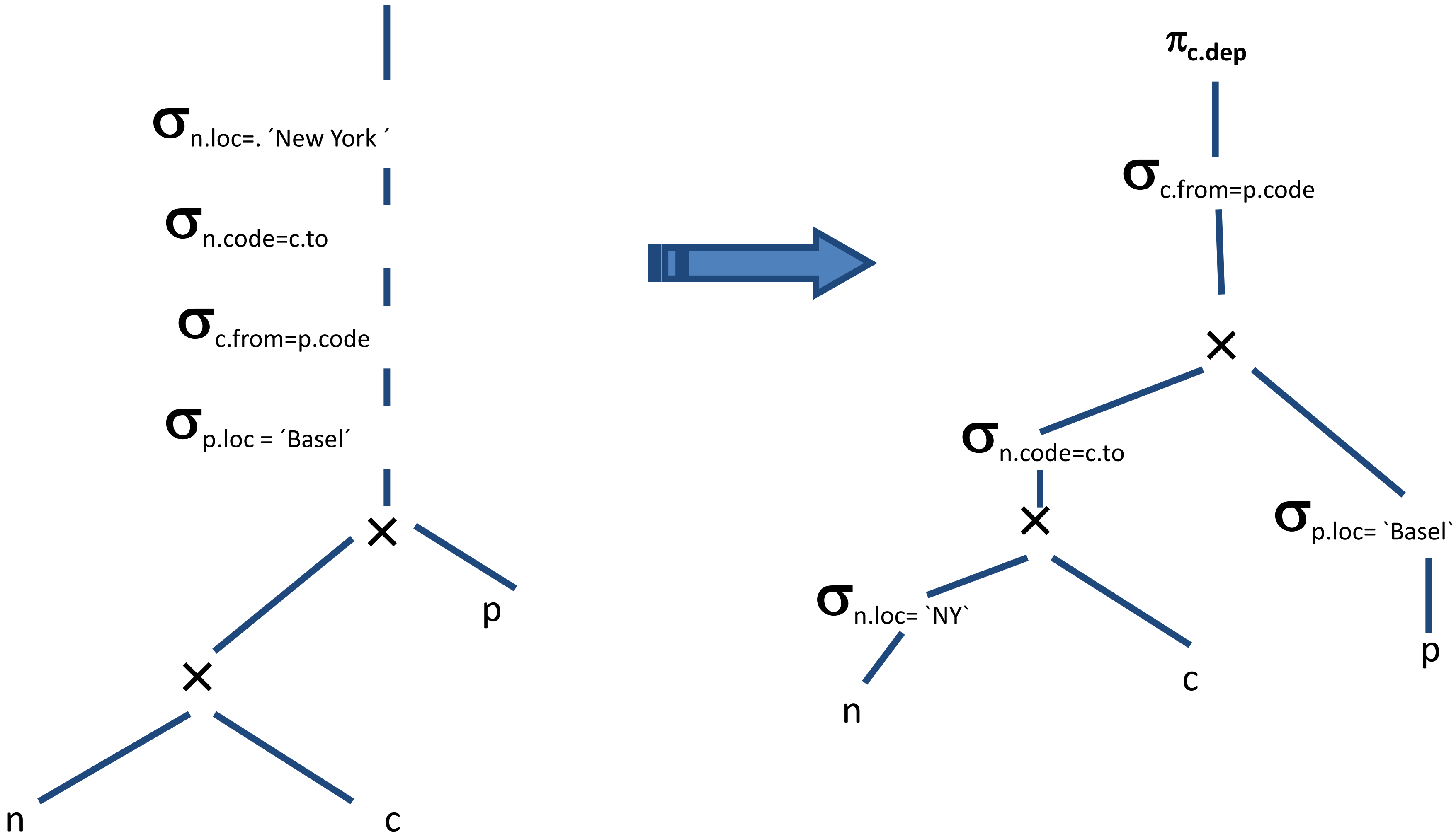
```
Select c.dep
from Airport n, Connection c,
     Airport p
where n.loc = “New York” and
      n.code = c.to and
      c.from = p.code and
      p.loc = “Basel”
```



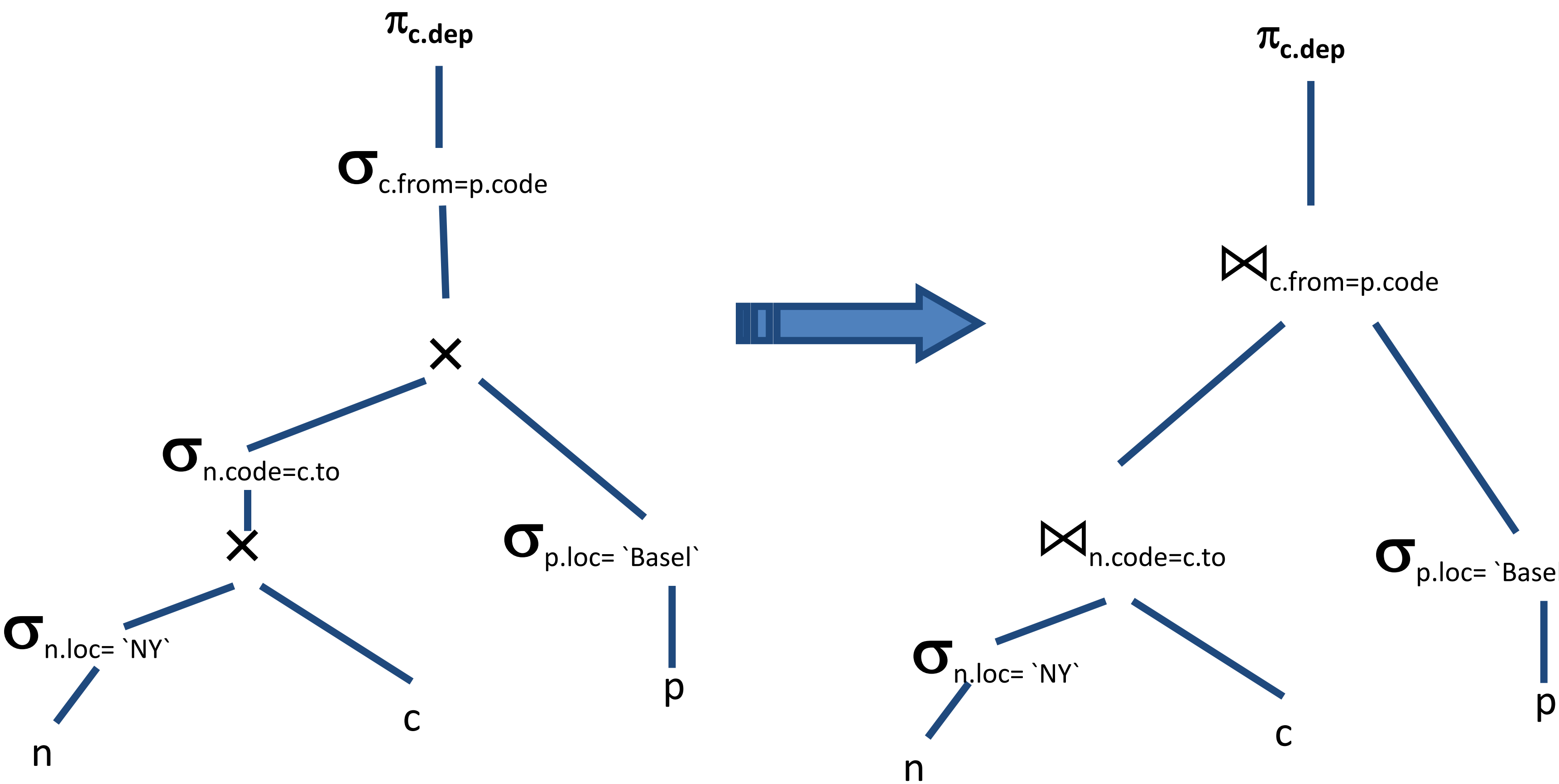
Splitting Predicates



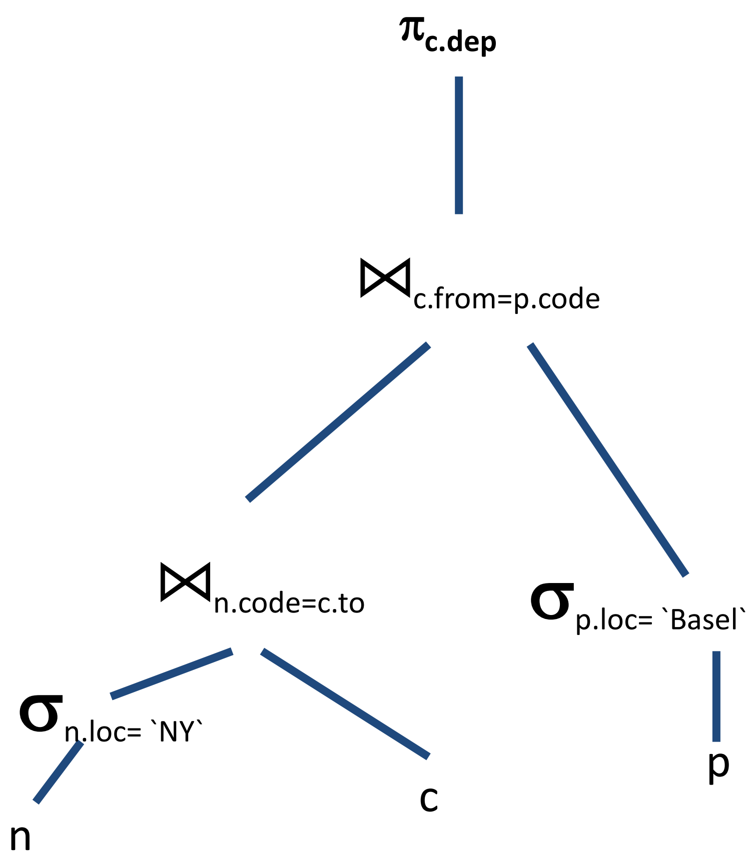
Selection Pushing



Introducing Joins



What about projections?



Predicate Simplification

- Example: Rewrite the following query

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

- into the following:

```
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: $(A \bowtie C) \bowtie B$
  - Otherwise, we would have a Cartesian product between A and C.

Nested Queries

- 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)
```

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.
- Often times, 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 7:3, 1982.

Summary

- **Query Parser**
  - Translates input query into (SFW-like) **query blocks.**
- **Query Rewriter**
  - Logical (database state-independent) optimizations
    - predicate/projection pushdown
    - predicate simplification
    - query unnesting
- **Query Optimizer** (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)