M146 Database Systems
Spring 2021

Georgia Koutrika

Query Optimizer

- What it needs:
  1. Information about how to compute the relational operators in the tree
     - Based on the access paths and algorithms available
  2. Information about the data stored
     - System Catalog Information
  3. Formulas to compute cardinalities and costs
  4. Strategy to generate plans and select the one to be executed

System Catalog Information

- Information about the size of a file
  - n_R: number of tuples in a relation *R.*
  - b_R: number of blocks containing tuples of *R.*
  - l_R: record size of *R.*
  - bf_R: blocking factor of *R* — i.e., the number of tuples of *R* that fit into one block.

- Information about indexes and indexing attributes of a file
  - Number of levels (x) of each multilevel index
  - Number of first-level index blocks (b_{I1})
  - Number of distinct values (d) of an attribute
  - Selectivity (sl) of an attribute

Example (System Catalog)

(a)

Table_name	Column_name	Num_distinct	Low_value	High_value
PROJECT	Plocation	200	1	200
PROJECT	Pnumber	2000	1	2000
PROJECT	Dnum	50	1	50
DEPARTMENT	Dnumber	50	1	50
DEPARTMENT	Mgr_ssn	50	1	50
EMPLOYEE	Ssn	10000	1	10000
EMPLOYEE	Dno	50	1	50
EMPLOYEE	Salary	500	1	500

(b)

Table_name	Num_rows	Blocks
PROJECT	2000	100
DEPARTMENT	50	5
EMPLOYEE	10000	2000

(c)

Index_name	Uniqueness	Blevel*	Leaf_blocks	Distinct_keys
PROJ_PLOC	NONUNIQUE	1	4	200
EMP_SSN	UNIQUE	1	50	10000
EMP_SAL	NONUNIQUE	1	50	500

*Blevel is the number of levels without the leaf level.

In reality, the system catalog …

Vertica

- ACCESS_POLICY
- ALL_TABLES
- AUDIT_MANAGING_USERS_PRIVILEGES
- CATALOG_SUBSCRIPTION_CHANGES
- CATALOG_SYNC_STATE
- CATALOG_TRUNCATION_STATUS
- CLIENT_AUTH
- CLIENT_AUTH_PARAMS
- CLUSTER_LAYOUT
- COLUMNS
- COMMENTS
- CONSTRAINT_COLUMNS
- DATABASES
- DIRECTED_QUERIES
- DUAL
- ELASTIC_CLUSTER
- EPOCHS
- FAULT_GROUPS
- FOREIGN_KEYS
- GRANTS
- HCATALOG_COLUMNS
- HCATALOG_SCHEMATA
- HCATALOG_TABLES
- HCATALOG_TABLE_LIST
- KEYWORDS
- LARGE_CLUSTER_CONFIGURATION_STATUS
- LICENSE_AUDITS
- LICENSES

- LOAD_BALANCE_GROUPS
- LOG_PARAMS
- LOG_QUERIES
- LOG_TABLES
- MATERIALIZE_FLEXTABLE_COLUMNS_RESULTS
- MODELS
- NETWORK_ADDRESSES
- NODES
- NODE_SUBSCRIPTION_CHANGE_PHASES
- NODE_SUBSCRIPTIONS
- ODBC_COLUMNS
- PASSWORD_AUDITOR
- PASSWORDS
- PRIMARY_KEYS
- PROFILE_PARAMETERS
- PROFILES
- PROJECTION_CHECKPOINT_EPOCHS
- PROJECTION_COLUMNS
- PROJECTION_DELETE_CONCERNS
- PROJECTIONS
- RESOURCE_POOL_DEFAULTS
- RESOURCE_POOLS
- ROLES
- ROUTING_RULES
- SCHEMATA
- SEQUENCES
- SESSION_SUBSCRIPTIONS
- SHARDS
- STORAGE_LOCATIONS
- SYSTEM_COLUMNS
- SYSTEM_TABLES
- TABLE_CONSTRAINTS
- TABLES

- PROJECTIONS
- RESOURCE_POOL_DEFAULTS
- RESOURCE_POOLS
- ROLES
- ROUTING_RULES
- SCHEMATA
- SEQUENCES
- SESSION_SUBSCRIPTIONS
- SHARDS
- STORAGE_LOCATIONS
- SYSTEM_COLUMNS
- SYSTEM_TABLES
- TABLE_CONSTRAINTS
- TABLES
- TEXT_INDICES
- TYPES
- USER_AUDITS
- USER_CLIENT_AUTH
- USER_FUNCTION_PARAMETERS
- USER_FUNCTIONS
- USER_PROCEDURES
- USER_TRANSFORMS
- USERS
- VIEW_COLUMNS
- VIEW_TABLES
- VIEWS

How to compute the cost of an execution plan?

Let us now see a simple example. We will need:

- statistics
- formulas for the costs of operations
- formulas for cardinalities

Example

Let us assume that we have these tables:

PROJECT(Pnumber, Plocation, Dnum, PStartDate)

DEPARTMENT(Dnumber, Dname, Mgr_ssn)

EMPLOYEE(SSN, Fname, Lname, Address, Bdate)

SQL:

```
SELECT P.Pnumber, P.Dnum, E.Lname, E.Address, E.Bdate
FROM   PROJECT P, DEPARTMENT D, EMPLOYEE E
WHERE P.Dnum = D.Dnumber AND D.Mgr_ssn = E.SSN
          AND P.Plocation = ‘STAFFORD’;
```

Example

(a)

Table_name	Column_name	Num_distinct	Low_value	High_value
PROJECT	Plocation	200	1	200
PROJECT	Pnumber	2000	1	2000
PROJECT	Dnum	50	1	50
DEPARTMENT	Dnumber	50	1	50
DEPARTMENT	Mgr_ssn	50	1	50
EMPLOYEE	Ssn	10000	1	10000
EMPLOYEE	Dno	50	1	50
EMPLOYEE	Salary	500	1	500

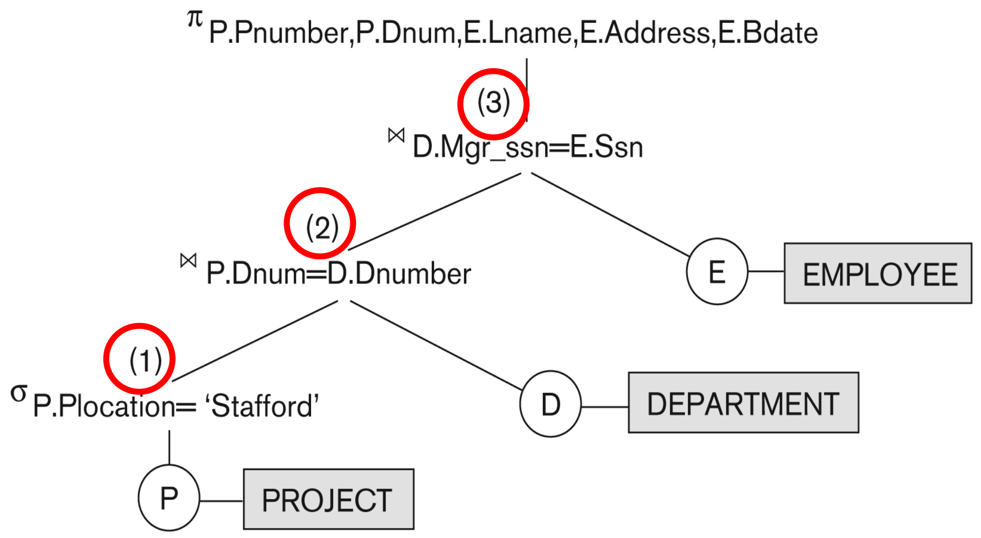
(b)

Table_name	Num_rows	Blocks
PROJECT	2000	100
DEPARTMENT	50	5
EMPLOYEE	10000	2000

(c)

Index_name	Uniqueness	Blevel*	Leaf_blocks	Distinct_keys
PROJ_PLOC	NONUNIQUE	1	4	200
EMP_SSN	UNIQUE	1	50	10000
EMP_SAL	NONUNIQUE	1	50	500

*Blevel is the number of levels without the leaf level.



Example

(a)

Table_name	Column_name	Num_distinct	Low_value	High_value
PROJECT	Plocation	200	1	200
PROJECT	Pnumber	2000	1	2000
PROJECT	Dnum	50	1	50
DEPARTMENT	Dnumber	50	1	50
DEPARTMENT	Mgr_ssn	50	1	50
EMPLOYEE	Ssn	10000	1	10000
EMPLOYEE	Dno	50	1	50
EMPLOYEE	Salary	500	1	500

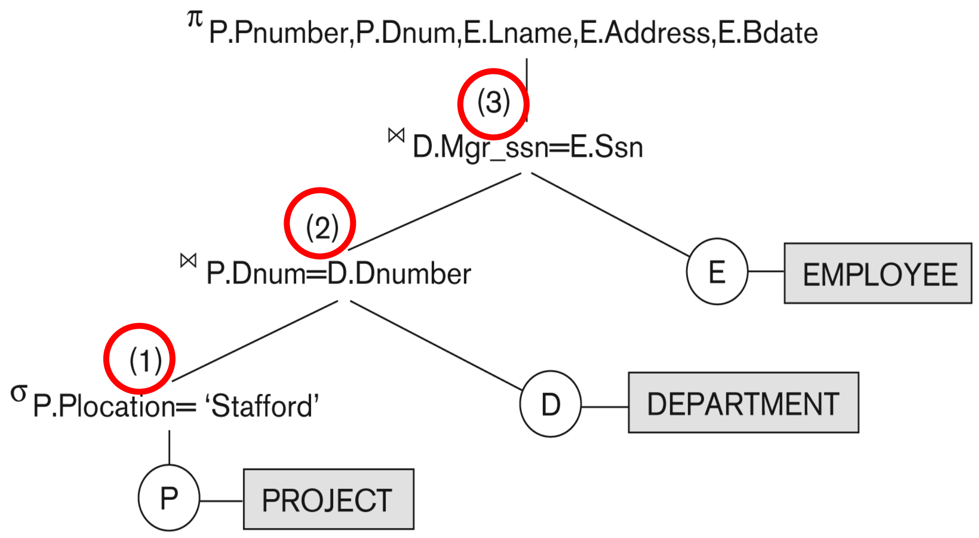
(b)

Table_name	Num_rows	Blocks
PROJECT	2000	100
DEPARTMENT	50	5
EMPLOYEE	10000	2000

(c)

Index_name	Uniqueness	Blevel*	Leaf_blocks	Distinct_keys
PROJ_PLOC	NONUNIQUE	1	4	200
EMP_SSN	UNIQUE	1	50	10000
EMP_SAL	NONUNIQUE	1	50	500

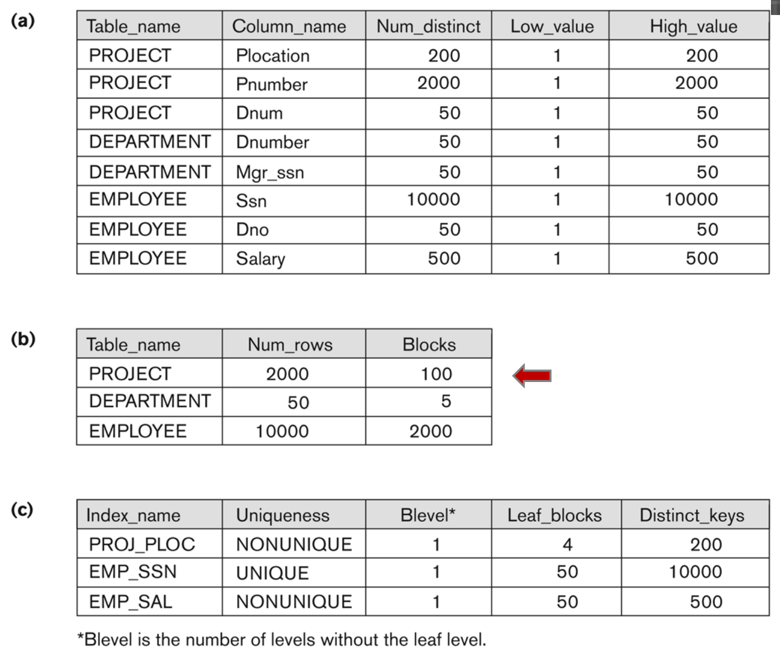
*Blevel is the number of levels without the leaf level.



1 $\sigma_{\text{Plocation = 'Stafford'}}$ (PROJECT)

- Table scan (Plocation is not primary key)
  - Cost = 100
- PROJ_PLOC Index (number of levels, x = 2)
  - Selectivity = 1/200 (assuming uniformly distributed)
  - Selection cardinality = Selectivity * Num_rows = 10 rows → 10 blocks
  - Cost = 2 + 10 = 12 √

Example



(a)

Table_name	Column_name	Num_distinct	Low_value	High_value
PROJECT	Plocation	200	1	200
PROJECT	Pnumber	2000	1	2000
PROJECT	Dnum	50	1	50
DEPARTMENT	Dnumber	50	1	50
DEPARTMENT	Mgr_ssn	50	1	50
EMPLOYEE	Ssn	10000	1	10000
EMPLOYEE	Dno	50	1	50
EMPLOYEE	Salary	500	1	500

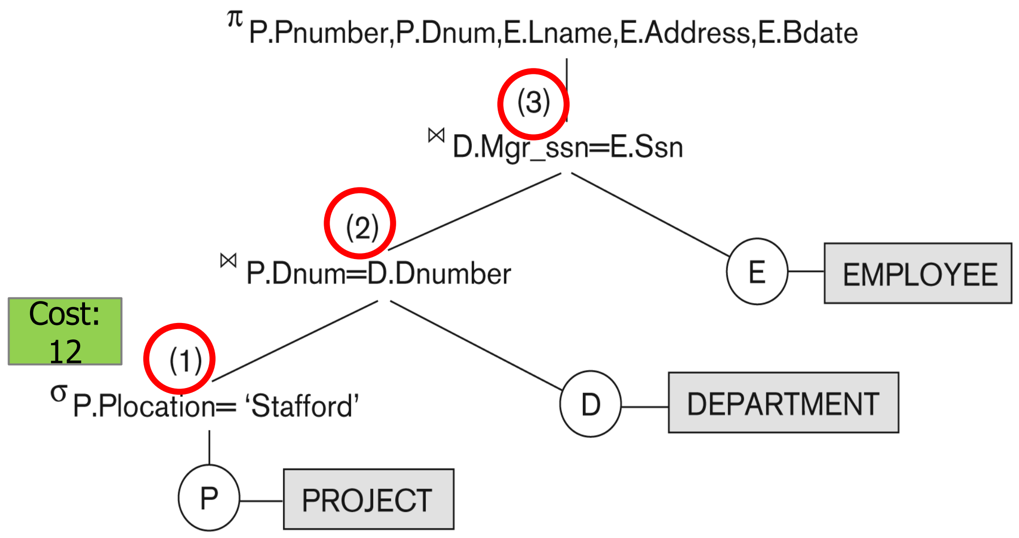
(b)

Table_name	Num_rows	Blocks
PROJECT	2000	100
DEPARTMENT	50	5
EMPLOYEE	10000	2000

(c)

Index_name	Uniqueness	Blevel*	Leaf_blocks	Distinct_keys
PROJ_PLOC	NONUNIQUE	1	4	200
EMP_SSN	UNIQUE	1	50	10000
EMP_SAL	NONUNIQUE	1	50	500

*Blevel is the number of levels without the leaf level.


π P.Pnumber,P.Dnum,E.Lname,E.Address,E.Bdate
(3)
$\bowtie$ D.Mgr_ssn=E.Ssn
(2)
$\bowtie$ P.Dnum=D.Dnumber
E
EMPLOYEE
Cost: 12
(1)
σ P.Plocation= 'Stafford'
D
DEPARTMENT
P
PROJECT

1 $\sigma_{\text{Plocation = 'Stafford'}}$ (PROJECT)

– Table scan (Plocation is not primary key)
  - Cost = 100
– PROJ_PLOC Index (number of levels, x = 2)
  - Selectivity = 1/200 (assuming uniformly distributed)
  - Selection cardinality = Selectivity * Num_rows = 10 rows → 10 blocks
  - Cost = 2 + 10 = 12 √

Example

(a)

Table_name	Column_name	Num_distinct	Low_value	High_value
PROJECT	Plocation	200	1	200
PROJECT	Pnumber	2000	1	2000
PROJECT	Dnum	50	1	50
DEPARTMENT	Dnumber	50	1	50
DEPARTMENT	Mgr_ssn	50	1	50
EMPLOYEE	Ssn	10000	1	10000
EMPLOYEE	Dno	50	1	50
EMPLOYEE	Salary	500	1	500

(b)

Table_name	Num_rows	Blocks
PROJECT	2000	100
DEPARTMENT	50	5
EMPLOYEE	10000	2000

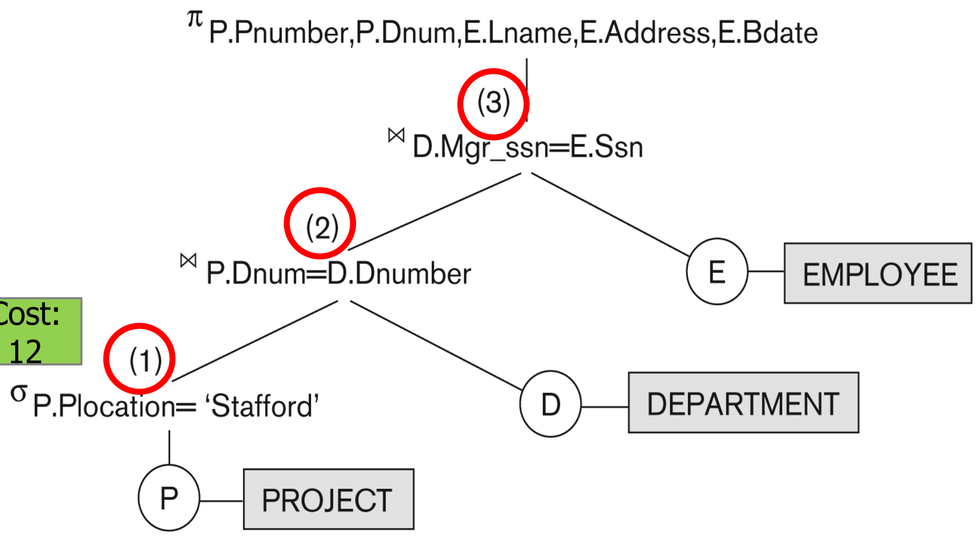
(c)

Index_name	Uniqueness	Blevel*	Leaf_blocks	Distinct_keys
PROJ_PLOC	NONUNIQUE	1	4	200
EMP_SSN	UNIQUE	1	50	10000
EMP_SAL	NONUNIQUE	1	50	500

*Blevel is the number of levels without the leaf level.

Blocks: 1

Cost: 12



1 $\sigma_{\text{Plocation = 'Stafford'}}$ (PROJECT) = TEMP1

- Estimated number of rows = 2000/200 = 10
- Blocking factor = 2000/100 = 20 tuples/block
- So, number of blocks needed = 1

Example

(a)

Table_name	Column_name	Num_distinct	Low_value	High_value
PROJECT	Plocation	200	1	200
PROJECT	Pnumber	2000	1	2000
PROJECT	Dnum	50	1	50
DEPARTMENT	Dnumber	50	1	50
DEPARTMENT	Mgr_ssn	50	1	50
EMPLOYEE	Ssn	10000	1	10000
EMPLOYEE	Dno	50	1	50
EMPLOYEE	Salary	500	1	500

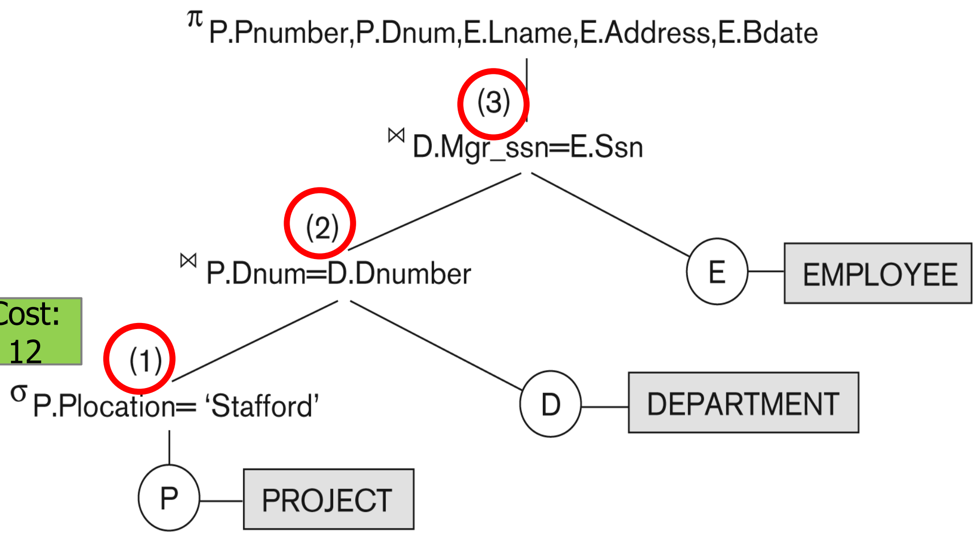
(b)

Table_name	Num_rows	Blocks
PROJECT	2000	100
DEPARTMENT	50	5
EMPLOYEE	10000	2000

(c)

Index_name	Uniqueness	Blevel*	Leaf_blocks	Distinct_keys
PROJ_PLOC	NONUNIQUE	1	4	200
EMP_SSN	UNIQUE	1	50	10000
EMP_SAL	NONUNIQUE	1	50	500

*Blevel is the number of levels without the leaf level.



Blocks: 1

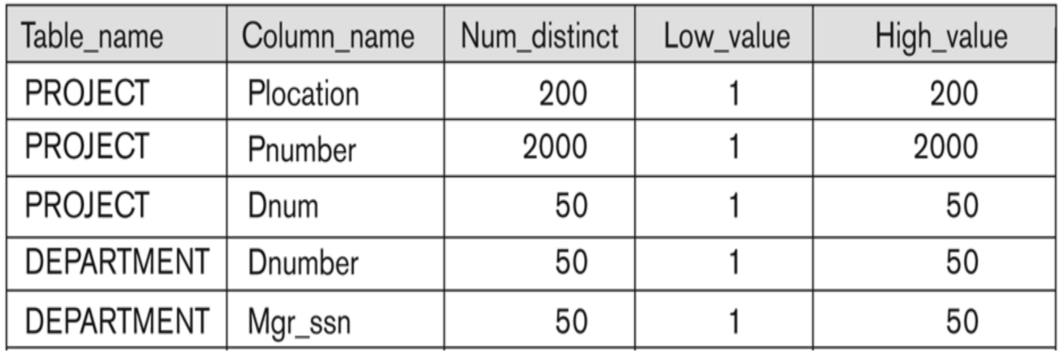
Cost: 12

2 Cost for $\overbrace{\sigma_{\text{Plocation = 'Stafford'}}(\text{PROJECT})}^{\text{TEMP1}} \bowtie \text{DEPARTMENT}$

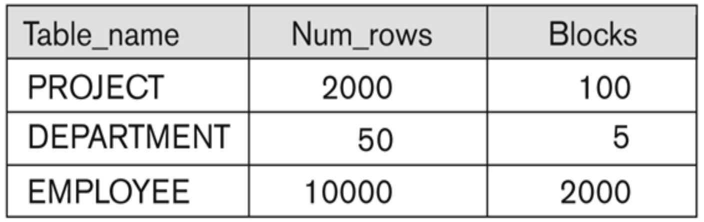
- No index available to process the join
- We use the nested loop join

P(R) + T(R)*P(S) + OUT

Example



Table_name	Column_name	Num_distinct	Low_value	High_value
PROJECT	Plocation	200	1	200
PROJECT	Pnumber	2000	1	2000
PROJECT	Dnum	50	1	50
DEPARTMENT	Dnumber	50	1	50
DEPARTMENT	Mgr_ssn	50	1	50



Table_name	Num_rows	Blocks
PROJECT	2000	100
DEPARTMENT	50	5
EMPLOYEE	10000	2000

2 Nested loop join TEMP1 $\bowtie_{\text{Dnum=Dnumber}}$ DEPARTMENT = TEMP2

- TEMP1: result of $\sigma_{\text{Plocation = 'Stafford'}}$ (PROJECT)
  - Estimated number of rows = 2000/200 = 10
  - Number of blocks needed = 1
- DEPARTMENT
  - number of blocks needed= 5

P(R) + T(R)*P(S) + OUT

Example

2 Nested loop join TEMP1 $\bowtie_{\text{Dnum=Dnumber}}$ DEPARTMENT = TEMP2

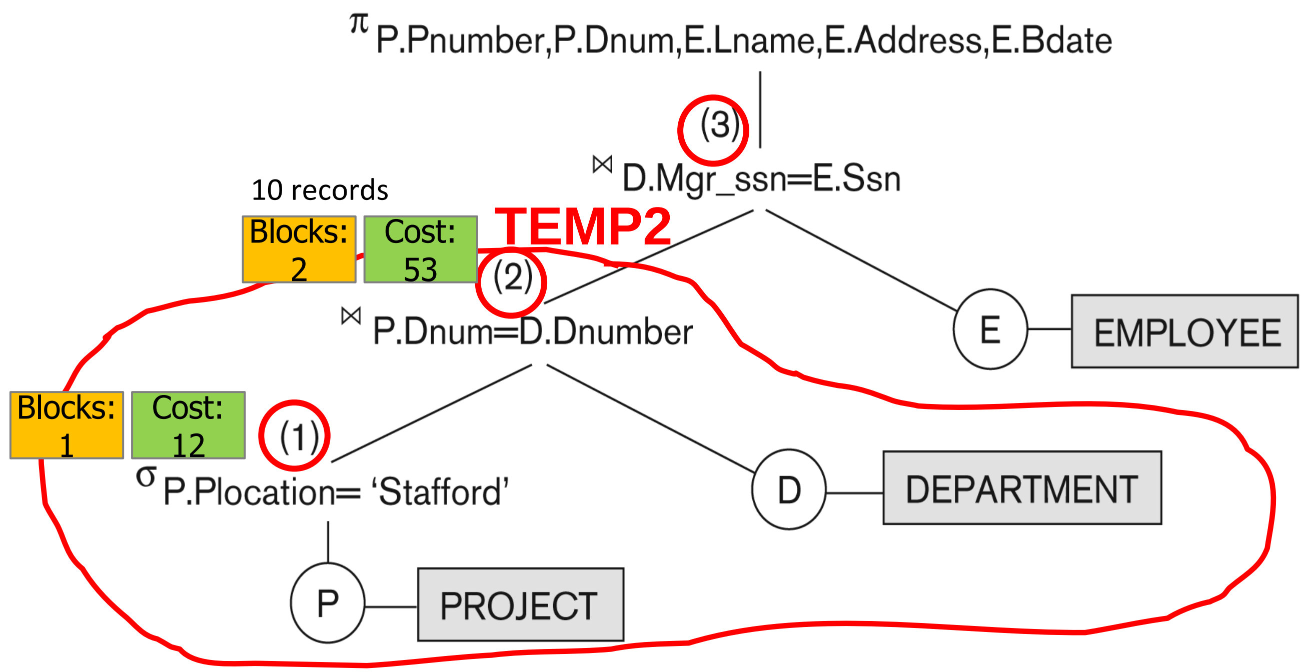
Cost: 53

P(R) + T(R)*P(S) + OUT

– Use TEMP1 in outer loop for nested-loop join
  - Cost = 1 + 10*5 + cost to write join output into TEMP2
    = 51 + cost to write join output into TEMP2

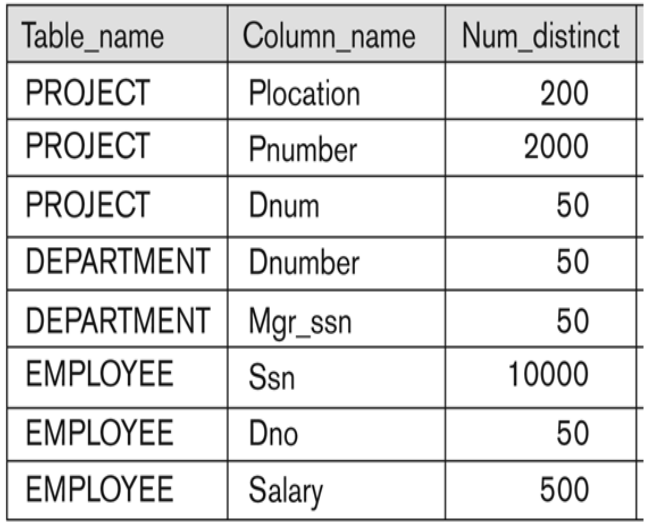
– What is the cost for writing join output?
  - Each row in TEMP1 joins exactly with 1 row in DEPARTMENT (why?)
  - Estimated number of rows in TEMP2 = 10
    join attribute Dnumber is the key of department.
    So we assume there are 10 joined records
  - Estimated blocking factor = 5 (from estimated record size)
  - Number of blocks needed = 2

Example

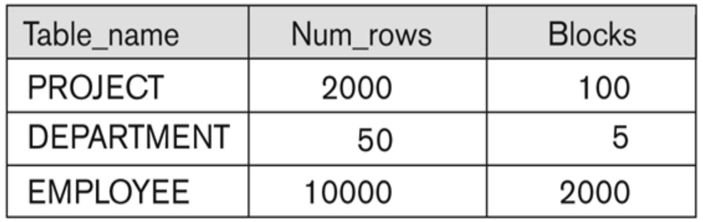


3 Cost for join TEMP2 $\bowtie_{\text{Mgr\_ssn=Ssn}}$ EMPLOYEE

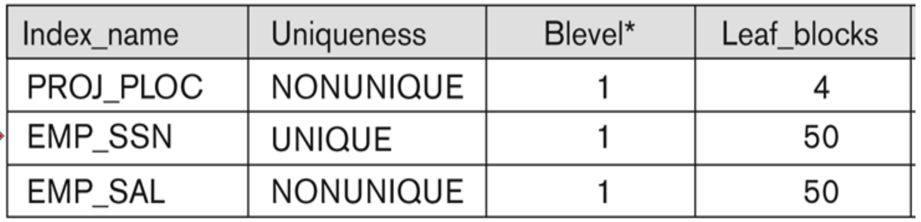
Example



Table_name	Column_name	Num_distinct
PROJECT	Plocation	200
PROJECT	Pnumber	2000
PROJECT	Dnum	50
DEPARTMENT	Dnumber	50
DEPARTMENT	Mgr_ssn	50
EMPLOYEE	Ssn	10000
EMPLOYEE	Dno	50
EMPLOYEE	Salary	500



Table_name	Num_rows	Blocks
PROJECT	2000	100
DEPARTMENT	50	5
EMPLOYEE	10000	2000



Index_name	Uniqueness	Blevel*	Leaf_blocks
PROJ_PLOC	NONUNIQUE	1	4
EMP_SSN	UNIQUE	1	50
EMP_SAL	NONUNIQUE	1	50

*Blevel is the number of levels without the leaf level.

3 Cost for join TEMP2 $\bowtie_{\text{Mgr\_ssn=Ssn}}$ EMPLOYEE

P(R) + T(R)****L*** + OUT

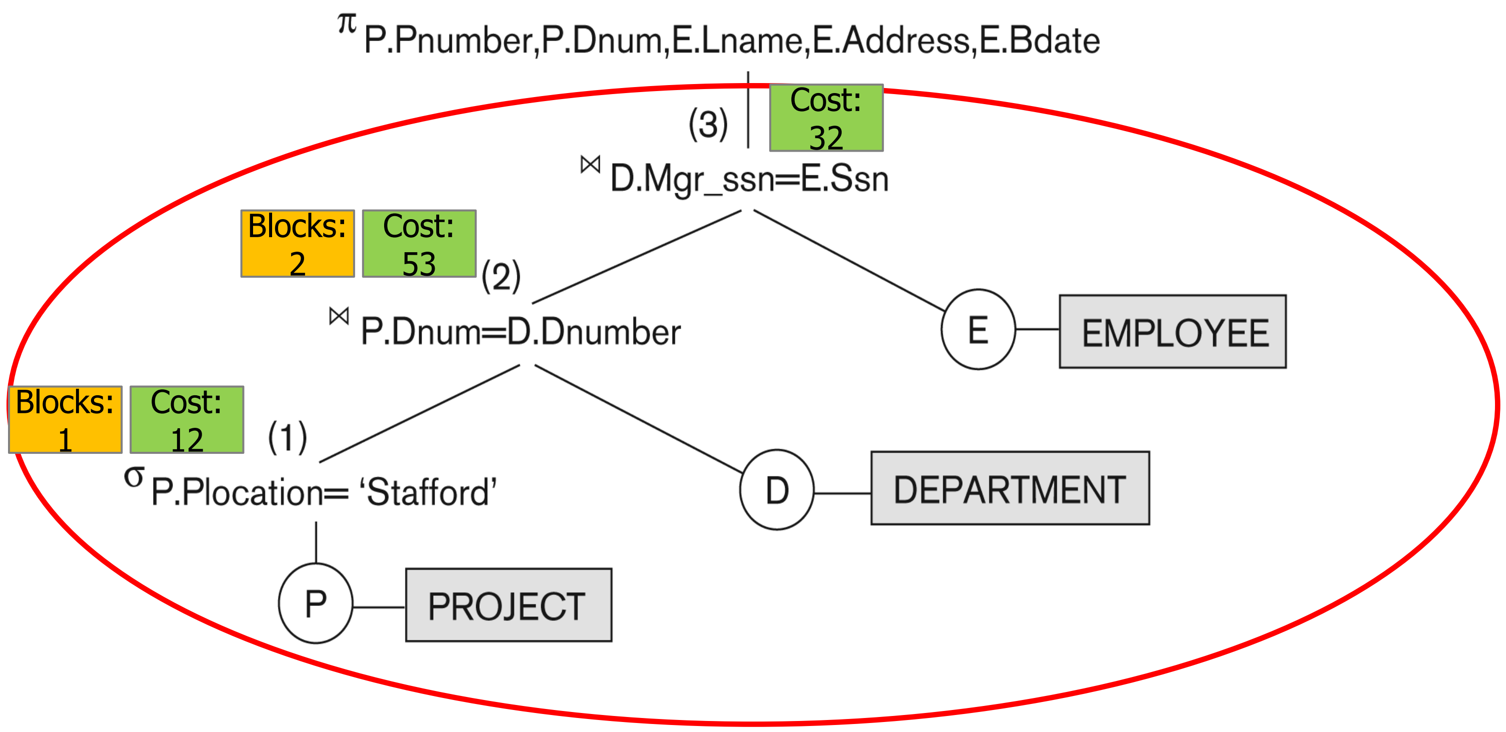
– Primary index (EMP_SSN) available for Ssn in EMPLOYEE

– Can use Index Nested Loop join on TEMP2

– For each row in TEMP2, use primary index to retrieve corresponding rows in EMPLOYEE

- Cost = 2 + 10 $\times$ (1 + 1 + 1) + cost of output = 32 + cost of output

Example



- Use pipelining to produce the final result
  - So, no additional cost for projection
  - Total cost = 12 + 53 + 32 + cost of writing final output

Query Optimizer

- What it needs:
  1. Information about how to compute the relational operators in the tree
     - Based on the access paths and algorithms available ✓
  2. Information about the data stored ✓
     - System Catalog Information
  3. **Formulas to compute costs and cardinalities**
  4. Strategy to generate plans and select the one to be executed

Selectivity

Consider a query predicate, such as WHERE last_name LIKE 'A%’ (or a combination of predicates)

Selectivity is the **percentage of rows** returned by a query predicate

- with 0 meaning no rows
- 1 meaning all rows.

A predicate becomes more selective as the selectivity value approaches 0 and less selective (or more unselective) as the value approaches 1.

Selectivity

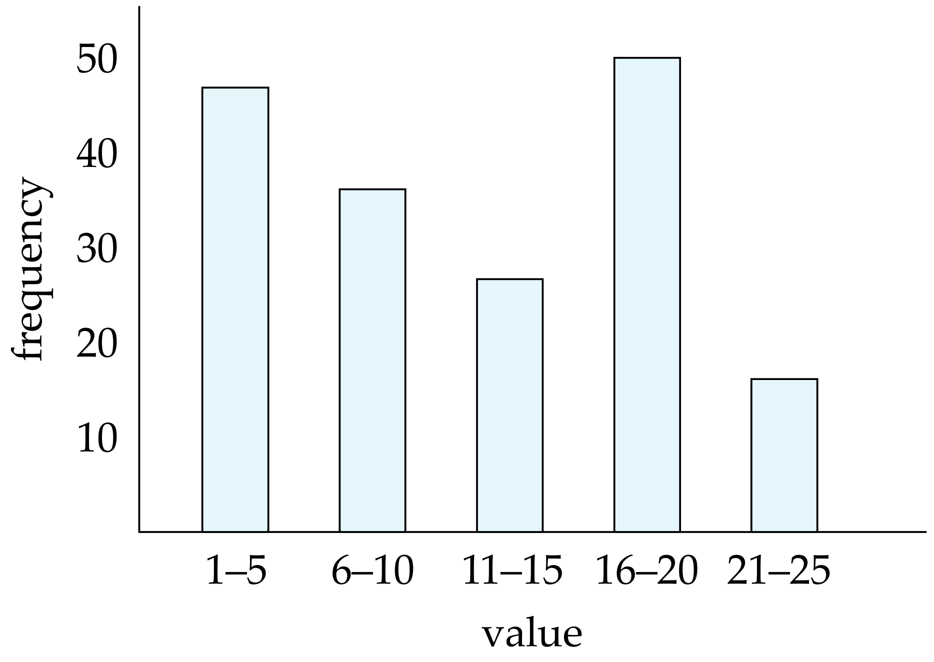
For an equality predicate :

Selectivity = **1/(number of distinct values)**

If there is a **histogram** on a column, then the estimator uses the histogram instead of the number of distinct values. The histogram captures the distribution of different values in a column, so it yields better selectivity estimates, especially for columns that have data skew.

Histograms

- Histogram on attribute *age* of relation *person*



- **Equi-width** histograms
- **Equi-depth** histograms

Cardinality

The **cardinality** is the **number of rows** returned by each operation in an execution plan.

This input, which is crucial to obtaining an optimal plan, is common to all cost functions.

The estimator can derive cardinality from:

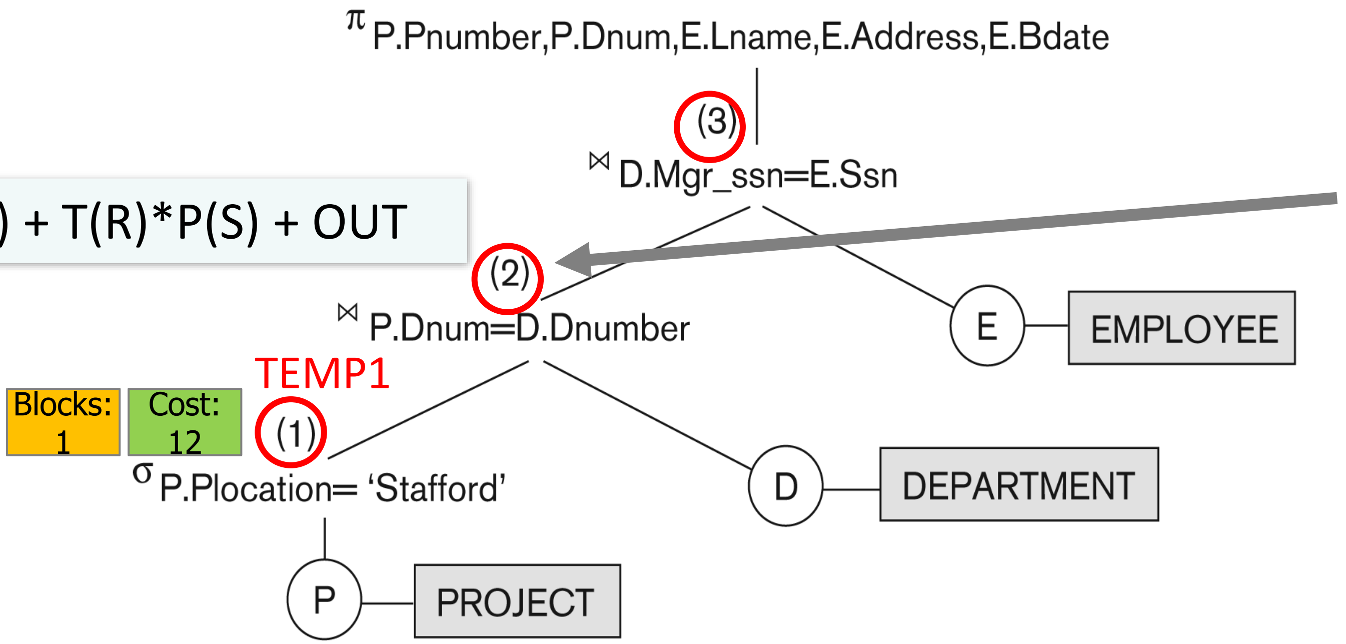
- the table statistics,
- after accounting for effects from predicates (filter, join, and so on), DISTINCT or GROUP BY operations, and so on.

Cardinality for a SQL query with 1 equality predicate = **(number of rows)/(number of distinct values)**

Cardinality

Cardinality estimates must be **as accurate as possible** because they influence all aspects of the execution plan. Cardinality is important when the optimizer determines the cost of a join, the cost of a sort.

$P(R) + T(R)*P(S) + OUT$



For example:
For operation 2 (the join),
we picked TEMP1 as the outer relation!

Cardinality Estimation of Query Result

There are two principal approaches to query cardinality estimation:

Database Profile.

- Maintain **statistical information** about numbers and sizes of tuples, distribution of attribute values for base relations, as part of the database catalog (meta information) during database updates.

- Calculate these parameters for **intermediate query results** based upon a (simple) statistical model during query optimization.

- Typically, the statistical model is based upon the **uniformity** and **independence** assumptions.
- Both are typically not valid, but they allow for simple calculations → limited accuracy.

- In order to improve accuracy, the system can record **histograms** to more closely model the actual value distributions in relations.

Cardinality Estimation of Query Result

There are two principal approaches to query cardinality estimation:

Sampling Techniques.

- Gather the necessary characteristics of a query plan (base relations and intermediate results) at **query execution time**

  - **Run query on a small sample** of the input.
  - **Extrapolate** to the full input size.
  - It is crucial to find the **right balance** between sample size and the resulting accuracy

Statistical Information

- n_R: number of tuples in a relation *R.*
- b_R: number of blocks containing tuples of *R.*
- l_R: record size of *R.*
- bf_R: blocking factor of *R* — i.e., the number of tuples of *R* that fit into one block.
- *V(A, R):* number of distinct values that appear in *R* for attribute *A*

- If tuples of *R* are stored together physically in a file, then:

  $b_R = n_R / bf_R$

Assumptions

In order to obtain tractable cardinality estimation formulae,
assume one of the following:

Uniformity & independence (simple, yet rarely realistic)
All values of an attribute **uniformly appear** with the same probability.
Values of different attributes **are independent** of each other.

Worst case (unrealistic)
No knowledge about relation contents at all.

Perfect knowledge (unrealistic)
Details about the exact distribution of values are known.
Requires huge catalog or prior knowledge of incoming queries.

Cardinality Estimation of Selection

- **$\sigma_{A=v}(R)$**
  - **cardinality =** n_R / $V(A,R)$: number of records that will satisfy the selection
  - Equality condition on a key attribute: **cardinality** = 1

Uniformity

V(A, R): number of distinct values that appear in *R* for attribute *A*

Cardinality Estimation of Selection

- $\sigma_{A=v}(R)$
    - **cardinality =** n_R / $V(A,R)$: number of records that will satisfy the selection
    - Equality condition on a key attribute: **cardinality** = 1

Uniformity

- $\sigma_{A \leq v}(R)$ (case of $\sigma_{A \geq v}(R)$ is symmetric)
    - If min(A,R) and max(A,R) are available in catalog
        - **cardinality** = 0 if v < min(A,R)
        - **cardinality** = $$n_R \cdot \frac{v - \min(A,R)}{max(A,R) - \min(A,R)}$$
    - If histograms available, we can refine above estimate
    - In absence of statistical information **cardinality** is assumed to be n_R / 2.

Cardinality of Complex Selections

The **selectivity** of a condition θ_i is the **probability** that a tuple in the relation *r* satisfies θ_i .

- If s_i is the number of satisfying tuples in *r,*
  selectivity of $\theta_i = s_i / n_r$.

Uniformity & independence

Let us recall some probability formulas:

P (A and B) = P (A) . P (B)

Conjunction: $\sigma_{\theta 1 \wedge \theta 2 \wedge \ldots \wedge \theta n}(r)$. *Assuming independence,* **cardinality** = $n_r * \frac{s_1 * s_2 * \ldots * s_n}{n_r^n}$

Cardinality of Complex Selections

The **selectivity** of a condition θ_i is the **probability** that a tuple in the relation *r* satisfies θ_i .

- If s_i is the number of satisfying tuples in *r,*
  selectivity of $\theta_i = s_i / n_r$.

Uniformity & independence

Let us recall some probability formulas:

P (A OR B) = P (A) + P (B) - P (A) . P (B)

Disjunction: $\sigma_{\theta 1 \vee \theta 2 \vee \ldots \vee \theta n}(r)$. **cardinality** = $n_r * \left(1 - (1 - \frac{s_1}{n_r}) * (1 - \frac{s_2}{n_r}) * \ldots * (1 - \frac{s_n}{n_r})\right)$

Negation: $\sigma_{\neg\theta}(r)$. **cardinality** = $n_r - size(\sigma_\theta(r))$

Cardinality of Projections

$\Pi_A(R)$

$$\textbf{cardinality} = \begin{cases} V(A, R) & \text{if A is a single attribute} \\ |R| & \text{if A contains the key of R} \\ \text{Min}(|R|, \Pi_{Ai} V(A_i, R)) & \text{otherwise} \end{cases}$$

Independence

V(A, R): number of distinct values that appear in *R* for attribute *A*

Cardinality of Joins

- If $R \cap S = \{A\}$ is not a key for R or S.
  If we assume that **every tuple *t* in *R* produces tuples in *R*⋈*S*,**
  the number of tuples in $R \bowtie S$ is estimated to be:

  $$\textbf{cardinality} = \frac{n_r * n_s}{V(A,s)}$$

  If the reverse is true, the estimate obtained will be:

  $$\textbf{cardinality} = \frac{n_r * n_s}{V(A,r)}$$

  The lower of these two estimates is probably the more accurate one.

n_R: number of tuples in a relation *R.*

V(A, R): number of distinct values that appear in *R* for attribute *A*

- Can improve on above if histograms are available
  - Use formula similar to above, for each cell of histograms on the two relations

Cardinality of Other Operations

- Aggregation : **cardinality** of ${}_{A}g_{F}(r)$ = $V(A,r)$

Cardinality of Other Operations

- Set operations
  - For unions/intersections of selections on the same relation: rewrite and use size estimate for selections
    - E.g. $\sigma_{\theta 1}(r) \cup \sigma_{\theta 2}(r)$ can be rewritten as $\sigma_{\theta 1 \vee \theta 2}(r)$
  - For operations on different relations:
    - **cardinality** of $r \cup s$ = size of r + size of s.
    - **cardinality** of $r \cap s$ = minimum size of r and size of s.
    - **cardinality** of $r - s$ = r.

    All the three estimates may be quite inaccurate, but provide upper bounds on the sizes.

Query Optimizer

- What it needs:

  1. Information about how to compute the relational operators in the tree
     - Based on the access paths and algorithms available
  2. Information about the data stored
     - System Catalog Information
  3. Formulas to compute costs and cardinalities
  4. **Strategy to generate plans and select the one to be executed**

Plan Generator

The plan generator explores various plans for a query block by trying out **different access paths**, **join methods**, and **join orders**.

Many plans are possible because of the various combinations that the database can use to produce the same result.

The optimizer picks the plan with the **lowest** cost (from the ones it examines)

Search Space Challenges

For example: Join orders
Consider **finding the best join-order** for $r_1 \quad r_2 \quad . . . r_n$.

There are $(2(n – 1))!/(n – 1)!$ different join orders for above expression.

Search space is huge!

– Many possible equivalent trees

– Many implementations for each operator

– Many access paths for each relation

- Cannot consider ALL plans
- Want a search space that includes low-cost plans

A local optimal method

- Choose the best algorithm for each operator

- The global effect may not be optimal

- Must consider the **interaction of evaluation techniques** when choosing evaluation plans
  - choosing the cheapest algorithm for each operation independently may not yield best overall cost. E.g.
    - merge-join may be costlier than hash-join, but may provide a sorted output which reduces the cost for an outer level aggregation.
    - nested-loop join may provide opportunity for pipelining

Choice of Evaluation Plans

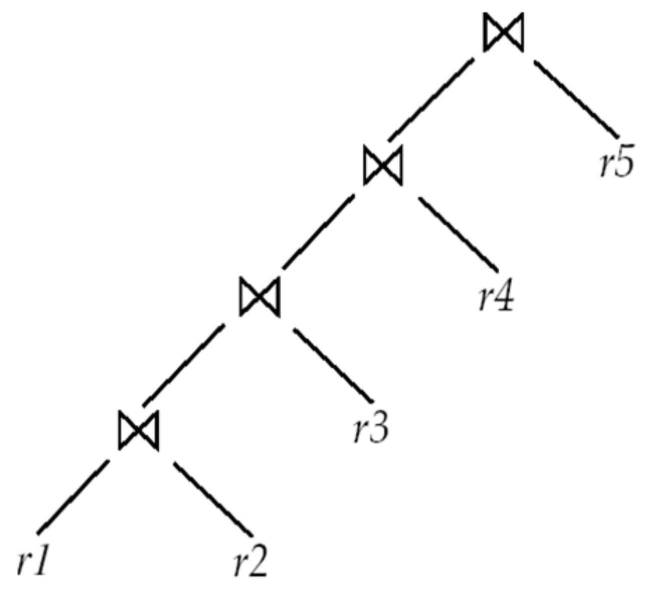
- Practical query optimizers incorporate elements of the following two broad approaches:
  1. Search all the plans and choose the best plan in a cost-based fashion.
  2. Uses heuristics to choose a plan.

System R Search Space

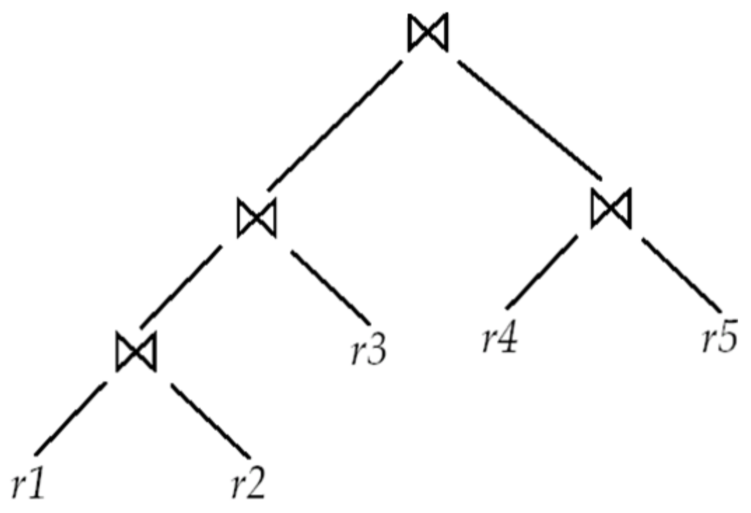
- **Only left-deep plans** ①
    - Enable **dynamic programming** ② for enumeration
    - Facilitate **tuple pipelining** from outer relation
- Consider plans with all **"interesting orders"** ③
- Perform cross-products after all other joins (heuristic)
- Only consider nested loop & sort-merge joins
- Consider both file scan and indexes
- Try to evaluate predicates early

1 Left Deep Join Trees

- In **left-deep join trees,** the right-hand-side input for each join is a relation, not the result of an intermediate join.



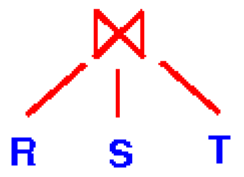
(a) Left-deep join tree



(b) Non-left-deep join tree

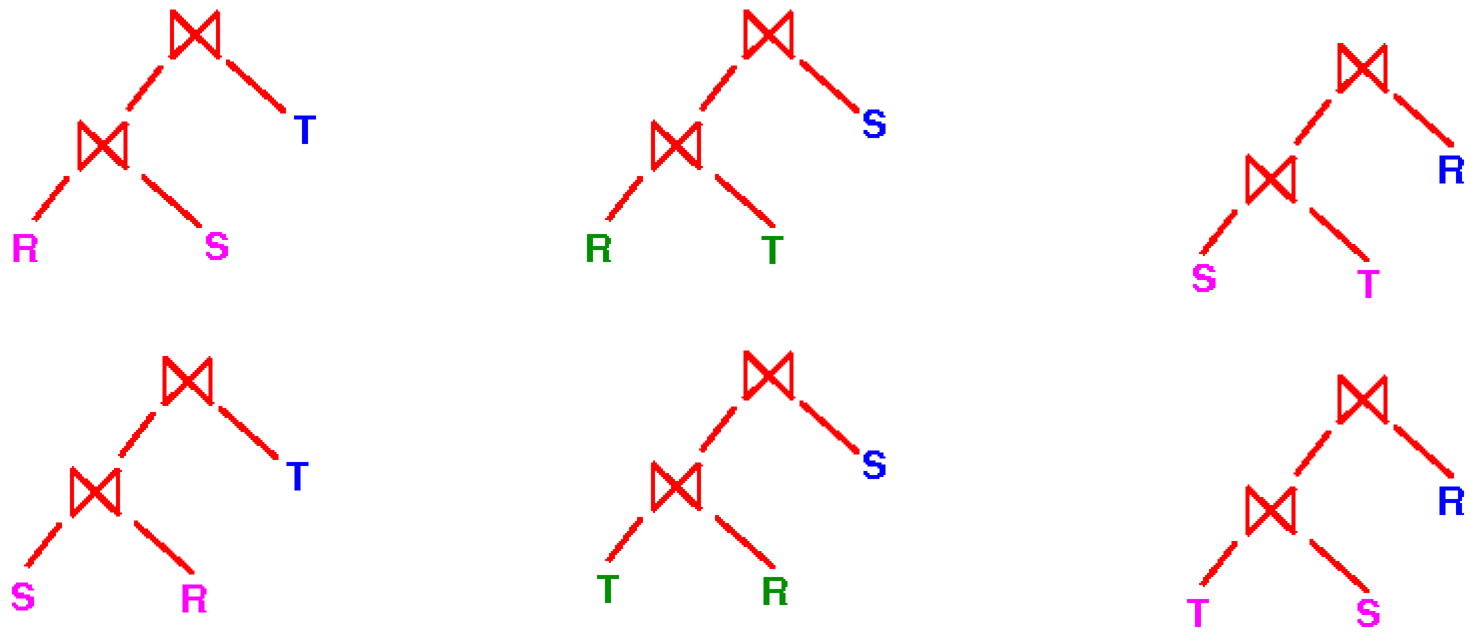
1 Left Deep Join Trees

Example: 3-way join:



Possible left-deep join trees

Number of ***left-deep*** **join trees** with ***N*** **relations** = ***N !*** **(factorial))**



1 Why left Deep Join Trees?

Query plans based on:

- **Left-deep** join trees **and**
- **Commonly used join implementations (algorithms)**

tend to be more efficient because:

Left-deep join trees **interact *very* well** with **commonly used join (implementation) algorithms**

2 Plan Enumeration Algorithm

Idea: use dynamic programming

- For each subset of {R1, …, Rn}, compute the best plan for that subset

- In increasing order of set cardinality:
  - Step 1: for {R1}, {R2}, …, {Rn}
  - Step 2: for {R1,R2}, {R1,R3}, …, {Rn-1, Rn}
  - …
  - Step n: for {R1, …, Rn}

- It is a bottom-up strategy
- A subset of {R1, …, Rn} is also called a *subquery*

2 Cost of Optimization

- **n=10 joins**
- **Finding the best join-order :** $(2(n-1))!/(n-1)!$ = 17,643,225,600

With dynamic programming:

- Time complexity of optimization with bushy trees is $\boldsymbol{O(3^n)}$ = 59000
- Space complexity is $O(2^n)$

- If only left-deep trees are considered, time complexity of finding best join order is $\boldsymbol{O(n\ 2^n)}$ = 10,240
  - Space complexity remains at $O(2^n)$

- Cost-based optimization is expensive, but worthwhile for queries on large datasets (typical queries have small n, generally < 10)

3 Interesting Sort Orders

- Consider the expression $(r_1 \bowtie r_2) \bowtie r_3$ (with A as common attribute)

- An **interesting sort order** is a particular sort order of tuples that could be useful for a later operation
  - Using merge-join to compute $r_1 \bowtie r_2$ may be costlier than hash join but generates result sorted on A
  - Which in turn may make merge-join with r_3 cheaper, which may reduce cost of join with r_3 and minimizing overall cost
  - Sort order may also be useful for order by and for grouping

Interesting Sort Orders

- Not sufficient to find the best join order for each subset of the set of *n* given relations
  - must find the best join order for each subset, **for each interesting sort order**
  - Simple extension of earlier dynamic programming algorithms
  - Usually, number of interesting orders is quite small and doesn’t affect time/space complexity significantly

Dynamic Programming Algo

- **Step 1**: Enumerate all single-relation plans
  - Consider selections on attributes of relation
  - Consider all possible access paths
  - Consider attributes that are not needed
  - Compute cost for each plan
  - Keep cheapest plan per "interesting" output order

Dynamic Programming Algo

- **Step 2**: Generate all two-relation plans
  - For each each single-relation plan from step 1
  - Consider that plan as outer relation
  - Consider every other relation as inner relation
  - Compute cost for each plan
  - Keep cheapest plan per "interesting" output order

Dynamic Programming Algo

- **Step 3**: Generate all three-relation plans
  - For each each two-relation plan from step 2
  - Consider that plan as outer relation
  - Consider every other relation as inner relation
  - Compute cost for each plan
  - Keep cheapest plan per "interesting" output order

- **Steps 4 through n**: repeat until plan contains all the relations in the query

Commercial Query Optimizers

DB2, Informix, Microsoft SQL Server, Oracle 8

- Inspired by System R

– Left-deep plans and dynamic programming

– Cost-based optimization (CPU and IO)

- Go beyond System R style of optimization

– Also consider right-deep and bushy plans (e.g., Oracle and DB2)

– Variety of additional strategies for generating plans (e.g., DB2 and SQL Server)

Other Query Optimizers

Randomized plan generation

– Genetic algorithm

– PostgreSQL uses it for queries with many joins

Rule-based

– ***Extensible*** collection of rules

– Rule = Algebraic law with a direction

– Algorithm for firing these rules

- Generate many alternative plans, in some order

- Prune by cost

– Startburst (later DB2) and Volcano (later SQL Server)

Cost Based Optimization with Equivalence Rules

- **Physical equivalence rules** allow logical query plan to be converted to physical query plan specifying what algorithms are used for each operation.

- Efficient optimizer based on equivalent rules depends on
  - A space efficient representation of expressions which avoids making multiple copies of subexpressions
  - Efficient techniques for detecting duplicate derivations of expressions
  - A form of dynamic programming based on **memoization**, which stores the best plan for a subexpression the first time it is optimized, and reuses in on repeated optimization calls on same subexpression
  - Cost-based pruning techniques that avoid generating all plans

- Pioneered by the Volcano project and implemented in the SQL Server optimizer

Structure of Query Optimizers

- Many optimizers consider only left-deep join orders.
  - Plus heuristics to push selections and projections down the query tree
  - Reduces optimization complexity and generates plans amenable to pipelined evaluation.
- Heuristic optimization used in some versions of Oracle:
  - Repeatedly pick “best” relation to join next
    - Starting from each of n starting points. Pick best among these
- Intricacies of SQL complicate query optimization
  - E.g. nested subqueries

Structure of Query Optimizers

- Some query optimizers integrate heuristic selection and the generation of alternative access plans.
  - Frequently used approach
    - heuristic rewriting of nested block structure and aggregation
    - followed by cost-based join-order optimization for each block
  - Some optimizers (e.g. SQL Server) apply transformations to entire query and do not depend on block structure
  - **Optimization cost budget** to stop optimization early (if cost of plan is less than cost of optimization)
  - **Plan caching** to reuse previously computed plan if query is resubmitted
    - Even with different constants in query

- Even with the use of heuristics, cost-based query optimization imposes a substantial overhead.
  - But is worth it for expensive queries
  - Optimizers often use simple heuristics for very cheap queries, and perform exhaustive enumeration for more expensive queries

Optimizing Nested Subqueries

- Nested query example:
  select *name*
  from *instructor*
  where exists (**select** *
  from *teaches*
  where *instructor.ID = teaches.ID* **and** *teaches.year = 2007*)

SQL conceptually treats nested subqueries in the where clause as functions that take parameters and return a single value or set of values

Parameters are variables from outer level query that are used in the nested subquery; such variables are called **correlation variables**

Optimizing Nested Subqueries

- Nested query example:
  select *name*
  from *instructor*
  where exists (**select** *
  from *teaches*
  where *instructor.ID = teaches.ID* **and** *teaches.year = 2007*)

Conceptually, a nested subquery is executed once for each tuple in the cross-product generated by the outer level **from** clause

Such evaluation is called **correlated evaluation**

Note: other conditions in where clause may be used to compute a join (instead of a cross-product) before executing the nested subquery

Optimizing Nested Subqueries

- Correlated evaluation may be quite inefficient since
  - a large number of calls may be made to the nested query
  - there may be unnecessary random I/O as a result

- SQL optimizers attempt to transform nested subqueries to joins where possible, enabling use of efficient join techniques

Optimizing Nested Subqueries

- Nested query example:
  select *name*
  from *instructor*
  where exists (**select** *
                  from *teaches*
                  where *instructor.ID = teaches.ID* **and** *teaches.year = 2007*)

can be rewritten as

select *name*
from *instructor, teaches*
where *instructor.ID = teaches.ID* **and** *teaches.year = 2007*

Optimizing Nested Subqueries

- In general, it is not possible/straightforward to move the entire nested subquery from clause into the outer level query from clause
  - A temporary relation is created instead, and used in body of outer level query

Optimizing Nested Subqueries

In general, SQL queries of the form below can be rewritten as shown

- Rewrite: **select …**
  from L_1
  where P_1 **and exists** (**select** *
  from L_2 **where** P_2)
- To: **create table** t_1 as
  select distinct V
  from L_2 **where** P_2^1

  select …
  from L_1, t_1 **where** P_1 **and** P_2^2

  - P_2^1 contains predicates in P_2 that do not involve any correlation variables
  - P_2^2 reintroduces predicates involving correlation variables, with relations renamed appropriately
  - V contains all attributes used in predicates with correlation variables

Optimizing Nested Subqueries

- In our example, the original nested query would be transformed to

```
create table t1 as
     select distinct ID
     from teaches
     where year = 2007

select name
from instructor, t1
 where t1.ID = instructor.ID
```

- The process of replacing a nested query by a query with a join (possibly with a temporary relation) is called **decorrelation**.

- Decorrelation is more complicated when
  - the nested subquery uses aggregation, or
  - when the result of the nested subquery is used to test for equality, or
  - when the condition linking the nested subquery to the other query is **not exists**,
  - and so on.