Database Systems

Luiz Jonatã Pires de Araújo
l.araujo@innopolis.university

6. Query Optimization

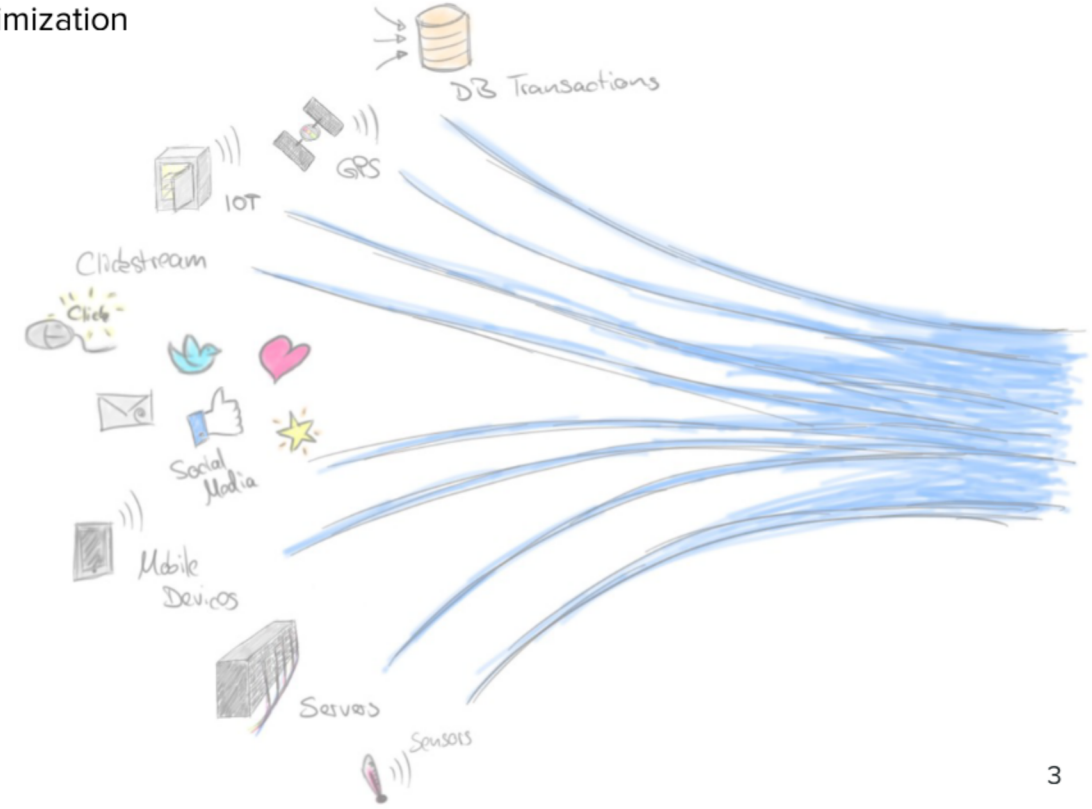
Giving credit where it's due:

- This material is based on the 7th edition of 'Fundamentals of Database Systems" by Elmasri and Navathe

  https://www.pearson.com/us/higher-education/program/Elmasri-Fundamentals-of-Database-Systems-7th-Edition/PGM189052.html
- Specialized Course "Query Optimization" - Saarland University, Germany
  http://resources.mpi-inf.mpg.de/departments/d5/teaching/ws06_07/queryoptimization/

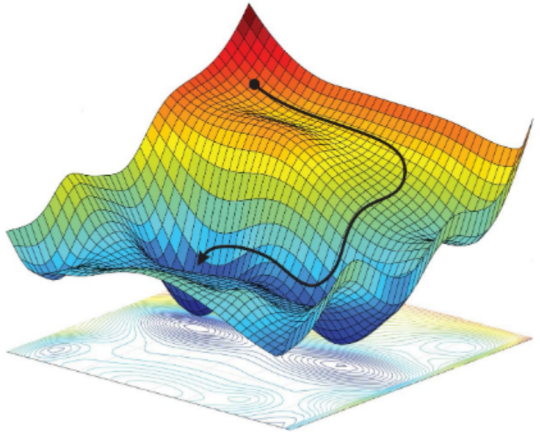
Today

- Query Trees and Heuristics for Query Optimization
- Choice of Query Execution Plans
- Cost Functions for SELECT Operation



Query Optimization

- The goal of query optimization is to select the best possible strategy for query evaluation
- The primary goal is to arrive at the most efficient and cost-effective plan using the available information about the schema and the content of relations involved, and to do so in a reasonable amount of time
- Thus a proper way to describe query optimization would be that it is an activity conducted by a query optimizer in a DBMS to select the best available strategy for executing the query



Definition

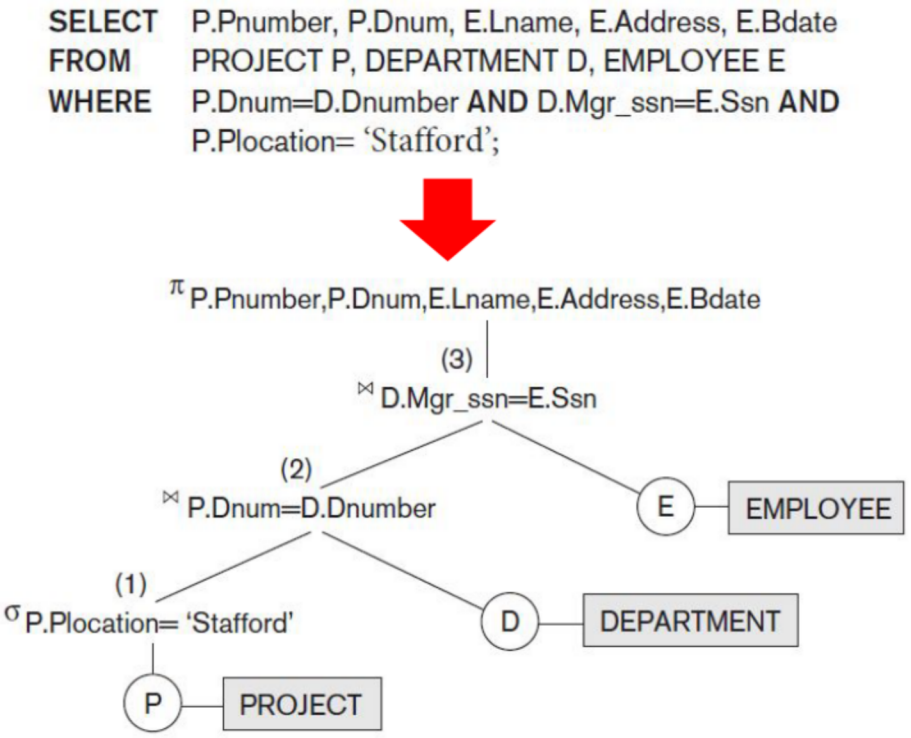
Query Trees and Heuristics for Query Optimization

Query Trees and Heuristics for Query Optimization

- There are optimization techniques that apply heuristic rules to modify the internal representation of a query, which is usually in the form of a query tree
  - One of the main heuristic rules is to apply SELECT and PROJECT operations before applying the JOIN or other binary operations
  - The size of the file resulting from a binary operation - such as JOIN - is usually a multiplicative function of the sizes of the input files
  - The SELECT and PROJECT operations reduce the size of a file and hence should be applied before a join or other binary operation
- Then, a query execution plan is generated to execute groups of operations based on the access paths available on the files involved in the query

Notation for Query Trees

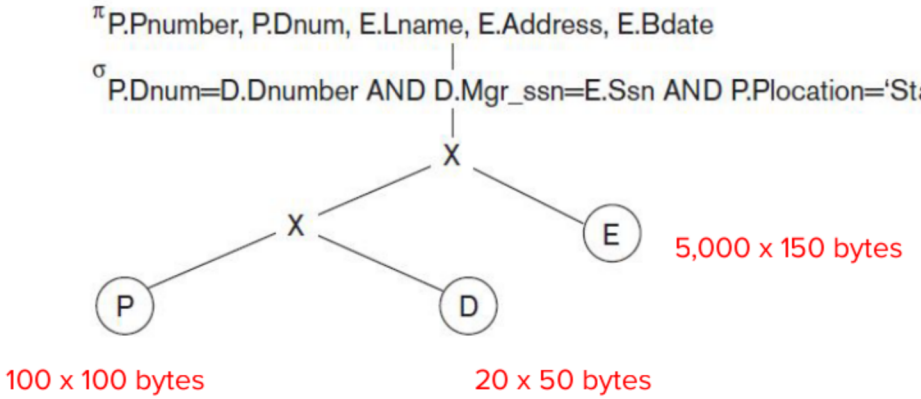
- A query tree is a tree data structure that corresponds to an extended relational algebra expression
- It represents the input relations of the query as leaf nodes of the tree, and it represents the relational algebra operations as internal nodes
- The query tree represents a specific order of operations for executing a query.
  - The order of execution of operations starts at the leaf nodes and ends at the root node.



Heuristic Optimization of Query Trees

```
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';
```

The heuristic query optimizer will transform this initial query tree into an equivalent final query tree that is efficient to execute.



10,000,000 x 300 bytes

5,000 x 150 bytes

100 x 100 bytes

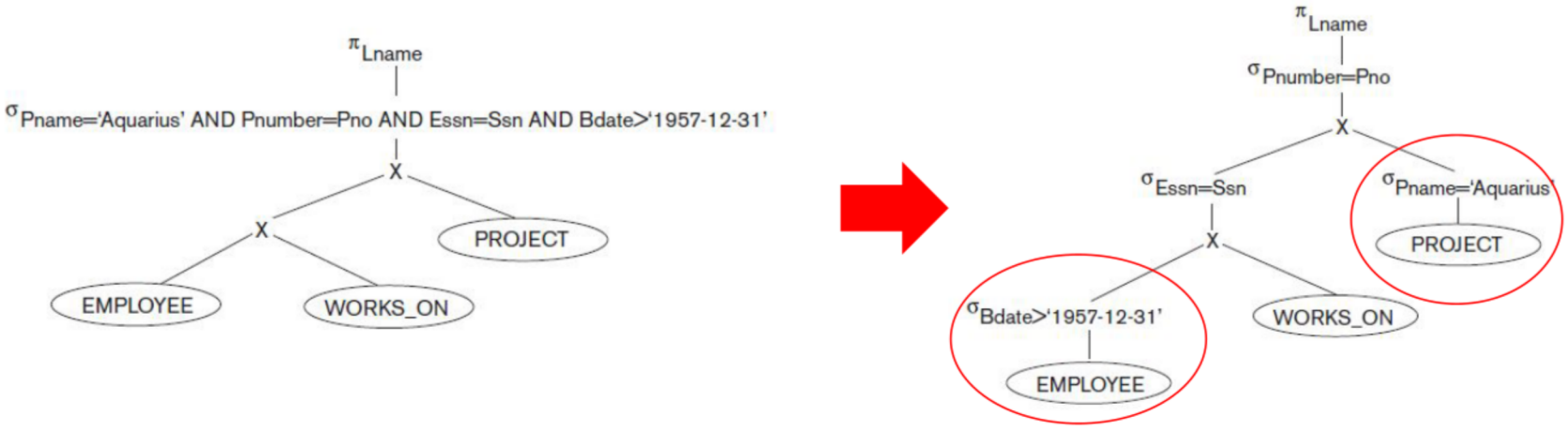
20 x 50 bytes

Example of Transforming a Query 1/4

Query: Find the last names of employees born after 1957 who work on a project named ‘Aquarius’.

```
SELECT E.Lname
FROM EMPLOYEE E, WORKS_ON W, PROJECT P
WHERE P.Pname=‘Aquarius’ AND P.Pnumber=W.Pno AND E.Essn=W.Ssn
        AND E.Bdate > ‘1957-12-31’;
```

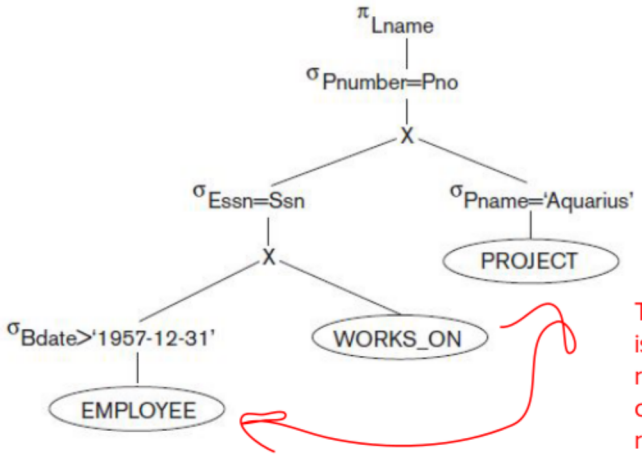
This particular query needs only one record from the PROJECT relation - for the ‘Aquarius’ project - and only the EMPLOYEE records for those whose date of birth is after ‘1957-12-31’



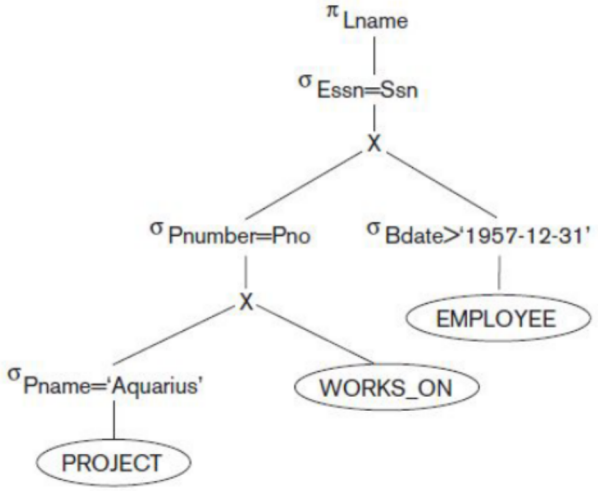
Example of Transforming a Query 2/4

Query: Find the last names of employees born after 1957 who work on a project named ‘Aquarius’.

```
SELECT E.Lname
FROM EMPLOYEE E, WORKS_ON W, PROJECT P
WHERE P.Pname=‘Aquarius’ AND P.Pnumber=W.Pno AND E.Essn=W.Ssn
        AND E.Bdate > ‘1957-12-31’;
```



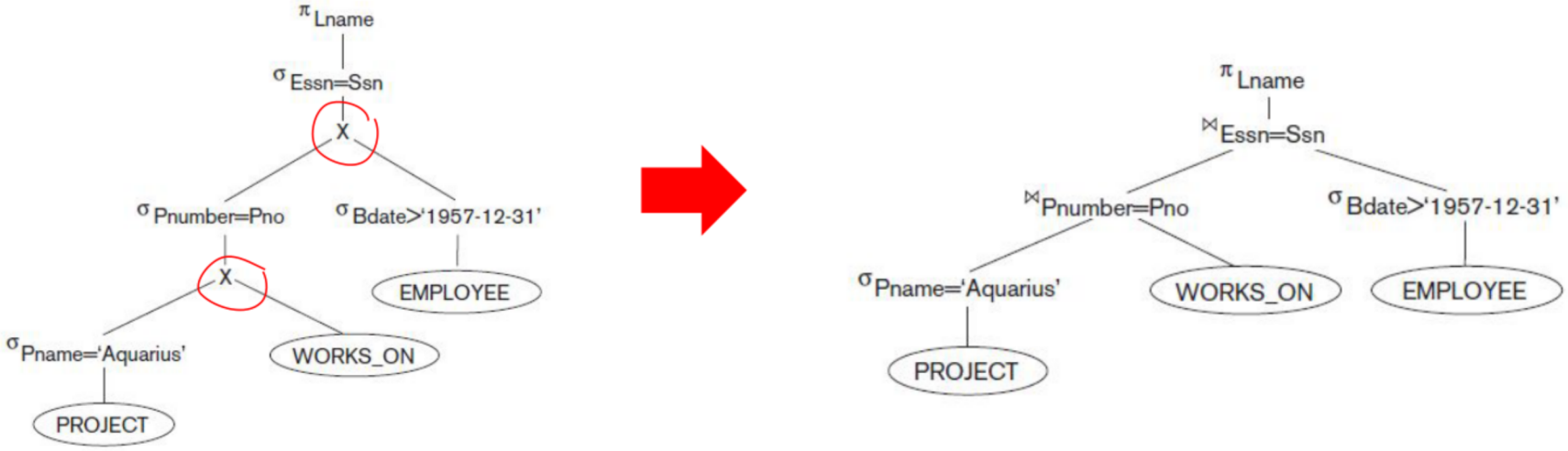
This uses the information that Pnumber is a key attribute of the PROJECT relation, and hence the SELECT operation on the PROJECT relation will retrieve a single record only.



Example of Transforming a Query 3/4

Query: Find the last names of employees born after 1957 who work on a project named ‘Aquarius’.

```
SELECT E.Lname
FROM EMPLOYEE E, WORKS_ON W, PROJECT P
WHERE P.Pname=‘Aquarius’ AND P.Pnumber=W.Pno AND E.Essn=W.Ssn
        AND E.Bdate > ‘1957-12-31’;
```

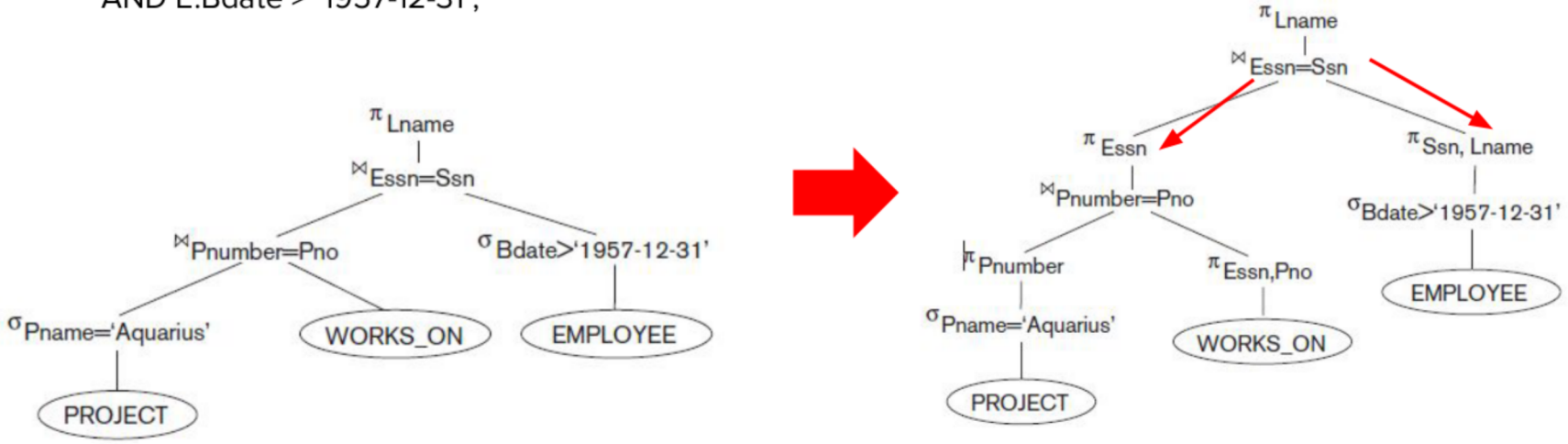


Example of Transforming a Query 4/4

Query: Find the last names of employees born after 1957 who work on a project named ‘Aquarius’.

```
SELECT E.Lname
FROM EMPLOYEE E, WORKS_ON W, PROJECT P
WHERE P.Pname=‘Aquarius’ AND P.Pnumber=W.Pno AND E.Essn=W.Ssn
        AND E.Bdate > ‘1957-12-31’;
```

This reduces the attributes (columns) of the intermediate relations, whereas the SELECT operations reduce the number of tuples (records).



General Transformation Rules for RA Operations

There are many rules for transforming relational algebra operations into equivalent ones.

1. **Cascade of σ.** A conjunctive selection condition can be broken up into a cascade (that is, a sequence) of individual σ operations:

$$\sigma_{c1\ \text{AND}\ c2\ \text{AND}\ \ldots\ \text{AND}\ cn}(R) \equiv \sigma_{c1}(\sigma_{c2}(\ldots(\sigma_{cn}(R))\ldots))$$

2. **Commutativity of σ.** The σ operation is commutative:

$$\sigma_{c1}(\sigma_{c2}(R)) \equiv \sigma_{c2}(\sigma_{c1}(R))$$

Not all the rules are shown here. For more, consult the textbook.

3. **Cascade of π.** In a cascade (sequence) of π operations, all but the last one can be ignored:

$$\pi_{List1}(\pi_{List2}(\ldots(\pi_{Listn}(R))\ldots)) \equiv \pi_{List1}(R)$$

4. **Commuting σ with π.** If the selection condition c involves only those attributes A1, ... , An in the projection list, the two operations can be commuted:

$$\pi_{A1,\ A2,\ \ldots\ ,\ An}(\sigma_{c}(R)) \equiv \sigma_{c}(\pi_{A1,\ A2,\ \ldots\ ,\ An}(R))$$

General Transformation Rules for RA Operations

Not all the rules are shown here. For more, consult the textbook.
Example: De Morgan's laws

5. **Commutativity of ⋈ (and ×).** The join operation is commutative, as is the × operation:

$$R \bowtie_c S \equiv S \bowtie_c R$$

$$R \times S \equiv S \times R$$

Notice that although the order of attributes may not be the same in the relations resulting from the two joins (or two Cartesian products), the meaning is the same because the order of attributes is not important.

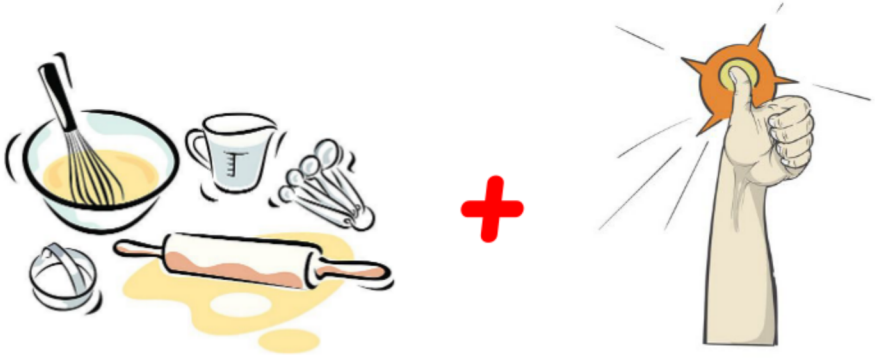
6. **Commuting σ with ⋈ (or ×).** If all the attributes in the selection condition c involve only the attributes of one of the relations being joined - say, R - the two operations can be commuted as follows:

$$\sigma_c (R \bowtie S) \equiv (\sigma_c (R)) \bowtie S$$

Alternatively, if the selection condition c can be written as (c1 AND c2), where condition c1 involves only the attributes of R and condition c2 involves only the attributes of S, the operations commute as follows:

$$\sigma_c (R \bowtie S) \equiv (\sigma_{c1} (R)) \bowtie (\sigma_{c2} (S))$$

The same rules apply if the ⋈ is replaced by a × operation



RULE OF THUMB

A Heuristic Algebraic Optimization Algorithm

1. Using Rule 1 (cascade of σ), break up any SELECT operations with conjunctive conditions into a cascade of SELECT operations. This permits a greater degree of freedom in moving SELECT operations down different branches of the tree.
2. Using rules concerning the commutativity of SELECT with other operations, move each SELECT operation as far down the query tree as is permitted by the attributes involved in the select condition. If the condition involves attributes from only one table, which means that it represents a selection condition, the operation is moved all the way to the leaf node that represents this table. If the condition involves attributes from two tables, which means that it represents a join condition, the condition is moved to a location down the tree after the two tables are combined.

A Heuristic Algebraic Optimization Algorithm

3. **Using rules concerning commutativity and associativity of binary operations**, rearrange the leaf nodes of the tree using the following criteria.
    a. First, position the leaf node relations with the most restrictive SELECT operations so they are executed first in the query tree representation. The definition of most restrictive SELECT can mean either the ones that produce a relation with the fewest tuples or with the smallest absolute size. Another possibility is to define the most restrictive SELECT as the one with the smallest selectivity; this is more practical because estimates of selectivities are often available in the DBMS catalog.
    b. Second, make sure that the ordering of leaf nodes does not cause CARTESIAN PRODUCT operations; for example, if the two relations with the most restrictive SELECT do not have a direct join condition between them, it may be desirable to change the order of leaf nodes to avoid Cartesian products.

A Heuristic Algebraic Optimization Algorithm

4. Using rule of conversion of (σ, ×) sequence into ⋈, combine a CARTESIAN PRODUCT operation with a subsequent SELECT operation in the tree into a JOIN operation, if the condition represents a join condition.
5. Using rules concerning the cascading of PROJECT and the commuting of PROJECT with other operations, break down and move lists of projection attributes down the tree as far as possible by creating new PROJECT operations as needed. Only those attributes needed in the query result and in subsequent operations in the query tree should be kept after each PROJECT operation.
6. Identify subtrees that represent groups of operations that can be executed by a single algorithm.

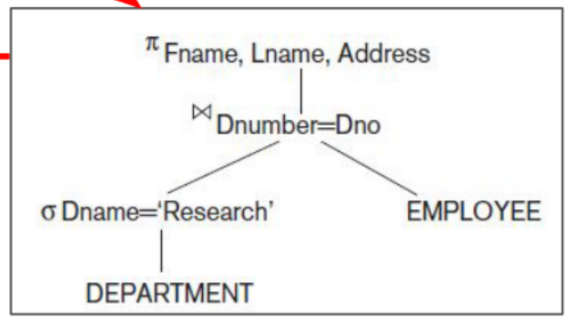
Summary of Heuristics for Algebraic Optimization

- The main heuristic is to apply first the operations that reduce the size of intermediate results
- This includes performing as early as possible SELECT operations to reduce the number of tuples and PROJECT operations to reduce the number of attributes - by moving SELECT and PROJECT operations as far down the tree as possible
- Additionally, the SELECT and JOIN operations that are most restrictive - that is, result in relations with the fewest tuples or with the smallest absolute size - should be executed before other similar operations
- The latter rule is accomplished through reordering the leaf nodes of the tree among themselves while avoiding Cartesian products, and adjusting the rest of the tree appropriately

Choice of Query Execution Plans

Alternatives for Query Evaluation 1/2

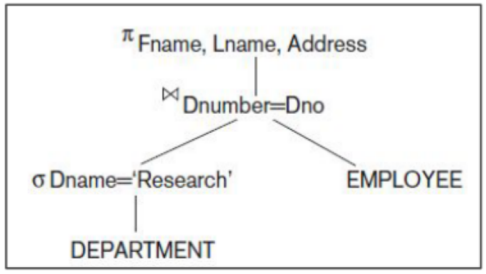
- An **execution plan** for a relational algebra expression represented as a query tree includes information about the access methods (e.g. indexes) available for each relation as well as the algorithms to be used in computing the relational operators represented in the tree
- Example: $\pi_{Fname,\ Lname,\ Address}(\sigma_{Dname='Research'}(DEPARTMENT) \bowtie_{Dnumber=Dno} EMPLOYEE)$

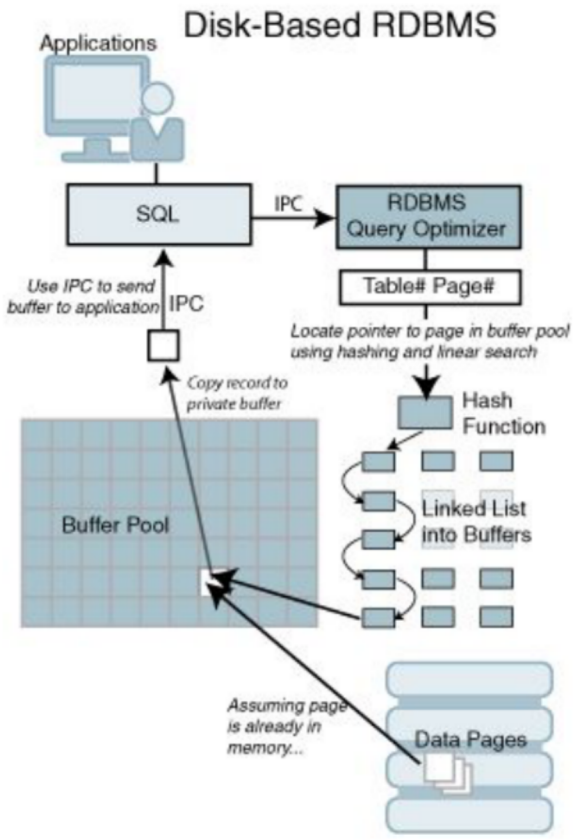


- To convert this into an execution plan, the optimizer:
  - Choose an index search for the SELECT operation on DEPARTMENT (assuming one exists)
  - An index-based nested-loop join algorithm that loops over the records in the result of the SELECT operation on DEPARTMENT for the join operation (assuming an index exists on the Dno attribute of EMPLOYEE)
  - Scan of the JOIN result for input to the PROJECT operator
  - Specify a materialized or a pipelined evaluation, although in general a pipelined evaluation is preferred whenever feasible

Alternatives for Query Evaluation 2/2

- With **materialized evaluation**, the result of an operation is stored as a temporary relation (that is, the result is physically materialized)
- With **pipelined evaluation**, the resulting tuples of an operation are forwarded directly to the next operation in the query sequence
  - The resulting tuples are placed in a buffer
  - The advantage of pipelining is the cost savings in not having to write the intermediate results to disk and not having to read them back for the next operation





Nested Subquery Optimization 1/2

- Example:

```
SELECT E1.Fname, E1.Lname
FROM EMLOYEE E1
WHERE E1.Salary =    ( SELECT MAX (Salary)
                FROM EMPLOYEE E2)
```

query block

- Evaluation of this query involves executing the nested query first, which yields a single value of the maximum salary M in the EMPLOYEE relation
- The outer block is simply executed with the selection condition Salary = M. The maximum salary could be obtained just from the highest value in the index on salary (if one exists) or from the catalog if it is up-to-date. The outer query is evaluated based on the same index.
- If no index exists, then linear search would be needed for both

If there is no index...

Nested Subquery Optimization 2/2

- Example:

```
SELECT Fname, Lname, Salary
FROM EMPLOYEE E
WHERE EXISTS   ( SELECT * FROM DEPARTMENT D
               WHERE D.Dnumber = E.Dno AND D.Zipcode=30332);
```

nested correlated subquery

- The naive strategy for evaluating the query is to evaluate the inner nested subquery for every tuple of the outer relation, which is inefficient
- Wherever possible, SQL optimizer tries to convert queries with nested subqueries into a join operation
  - The process is called unnesting or decorrelation
- The above query would be converted to:

```
SELECT Fname, Lname, Salary
FROM EMPLOYEE E, DEPARTMENT D
WHERE WHERE D.Dnumber = E.Dno AND D.Zipcode=30332
```

Subquery (View) Merging Transformation

- Inline View: when a subquery appears in the FROM clause of a query, including a derived relation
- Sometimes, an actual view defined earlier as a separate query is used as one of the argument relations in a new query
- Example:

```
SELECT E.Ln, V.Addr, V.Phone
FROM EMP E,     (SELECT D.Dno, D.Dname, B.Addr, B.Phone
                 FROM DEPT D, BLDG B WHERE D.Bldg_id = B.Bldg_id) V
WHERE V.Dno = E.Dno AND E.Fn = "John";
```

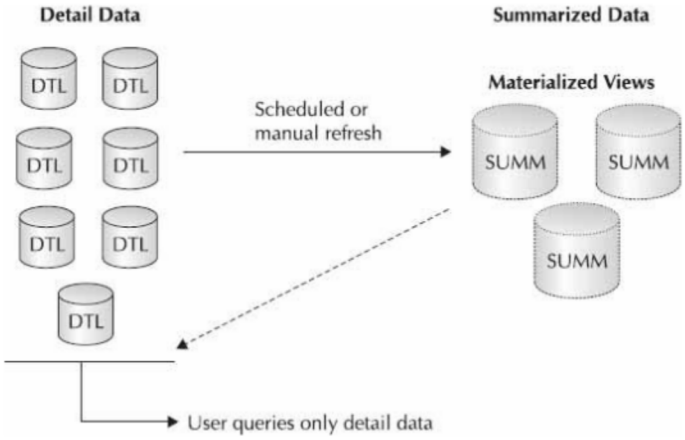
- This query may be executed by first temporarily materializing the view and then joining it with the EMP table
- The view-merging operation merges the tables in the view with the tables from the outer query block and produces the following query:

```
SELECT E.Ln, B.Addr, B.Phone
FROM EMP E, DEPT D, BLDG B
WHERE D.Bldg_id = B.Bldg_id
      AND D.Dno = E.Dno AND E.Fn = "John";
```

View-merging may be invalid under certain conditions where the view is more complex and involves DISTINCT, OUTER JOIN, AGGREGATION, GROUP BY set operations, and so forth.

Materialized Views 1/2

- A view is defined in the database as a query, and a materialized view stores the results of that query
- Using materialized views to avoid some of the computation involved in a query is another query optimization technique
- A materialized view may be stored temporarily to allow more queries to be processed against it or permanently, as is common in data warehouses
- The main idea behind materialization is that it is much cheaper to read it when needed and query against it than to recompute it from scratch
- The savings can be significant when the view involves costly operations like join, aggregation, and so forth



Materialized Views 2/2

- Update (also known as refresh) strategies for updating the view:
  - **Immediate update**, which updates the view as soon as any of the relations participating in the view are updated
  - **Lazy update**, which recomputes the view only upon demand
  - **Periodic update** (or deferred update), which updates the view later, possibly with some regular frequency
- The straightforward and naive approach is to recompute the entire view for every update to any base table and is prohibitively costly
- Hence incremental view maintenance is done in most RDBMSs today

Use of Selectivities in Cost-Based Optimization

A query optimizer does not depend solely on **heuristic rules** or query transformations;
it also **estimates and compares the costs** of executing a query using different execution strategies and algorithms, and it then chooses the strategy with the lowest cost estimate.

"If you can't measure it, you can't manage it" -
W. Edwards Deming

This approach is generally referred to as **cost-based query optimization**.

Cost Components for Query Execution

1. **Access cost to secondary storage**
   a. The cost of reading and/or writing data blocks between secondary disk storage and main memory buffers
   b. This is also known as disk I/O (input/output) cost
2. **Disk storage cost**
   a. The cost of storing on disk any intermediate files for the query
3. **Computation cost**
   a. The cost of performing in-memory operations on the records within the data buffers during query execution.
   b. Example: searching for and sorting records, merging records for a join or a sort operation, and performing computations on field values
   c. This is also known as CPU (central processing unit) cost
4. **Memory usage cost**: the number of main memory buffers needed during query execution
5. **Communication cost**: cost of transferring tables and results among various computers

Cost in PostgreSQL - Example 1/3

cost = (#blocks * seq_page_cost) + (#records * cpu_tuple_cost) + (#records * cpu_filter_cost)

- The calculate the cost, PostgreSQL first looks at the size of your table in bytes. Let's find out the size of the users table.

```
db # select pg_relation_size('users');
```

```
block_size = 8192 # block size in bytes
relation_size = 44285952
blocks = relation_size / block_size # => 5406
```

- Now, that we know the number of block, let's find out how many points will PostgreSQL allocate for each block read.

```
db # SHOW seq_page_cost;
 seq_page_cost
---------------
1
```

Cost in PostgreSQL - Example 2/3

cost = (#blocks * seq_page_cost) + (#records * cpu_tuple_cost) + (#records * cpu_filter_cost)

- Reading values from the disk is not everything that PostgreSQL needs to do. It has to send those values to the CPU and to apply a WHERE filter. Two values are interesting for this calculation.

```
db # SHOW cpu_tuple_cost;
 cpu_tuple_cost
----------------
 0.01
```

```
db # SHOW cpu_operator_cost;
 cpu_operator_cost
-------------------
 0.0025
```

Cost in PostgreSQL - Example 3/3

cost = (#blocks * seq_page_cost) + (#records * cpu_tuple_cost) + (#records * cpu_filter_cost)

- Now, we have all the values to calculate the value that we got in our explain clause.

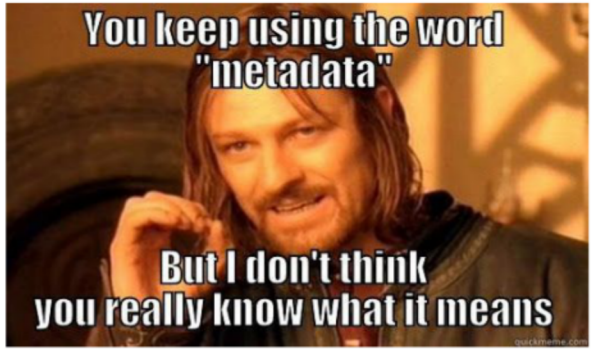
```
number_of_records = 1000000
block_size     = 8192     # block size in bytes
relation_size = 44285952
blocks = relation_size / block_size # => 5406
seq_page_cost   = 1
cpu_tuple_cost  = 0.01
cpu_filter_cost = 0.0025;
cost = blocks * seq_page_cost +
      number_of_records * cpu_tuple_cost +
      number_of_records * cpu_filter_cost

cost # => 17546
```

Catalog Information Used in Cost Functions

Information stored in the DBMS catalog, where it is accessed by the query optimizer

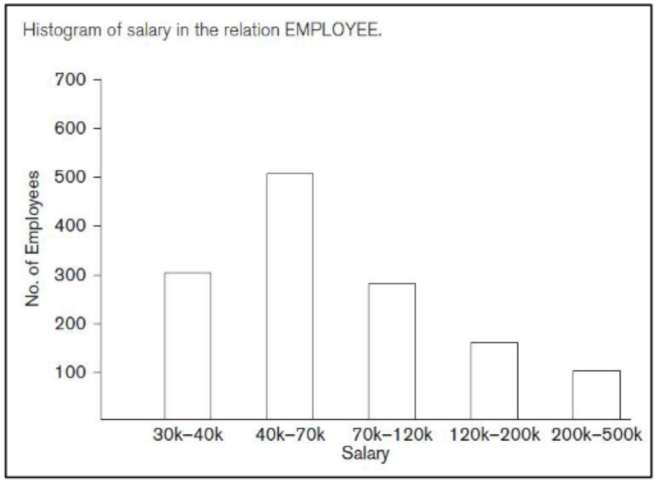
- **The size of each file**. For a file whose records are all of the same type:
  - The **number of records** (tuples) (r)
  - The **(average) record size** (R), and
  - The **number of file blocks** (b) (or close estimates of them) are needed
  - The **blocking factor** (bfr) for the file may also be needed
- The **primary file organization** for each file



- Information is also kept on all primary, secondary, or clustering **indexes** and their **indexing attributes**
- The **number of levels** (x) of each multilevel index (primary, secondary, or clustering) is needed for cost functions that estimate the number of block accesses that occur during query execution
- In some cost functions the **number of first-level index blocks** (bl1) is needed

Catalog Information Used in Cost Functions

- Another important parameter is the number of distinct values **NDV (A, R)** of an attribute in relation R and the attribute selectivity (sl), which is the fraction of records satisfying an equality condition on the attribute
  - This allows estimation of the selection cardinality (s = sl*r) of an attribute, which is the average number of records that will satisfy an equality selection condition on that attribute
- To help with estimating the size of the results of queries, it is important to have as good an **estimate of the distribution of values** as possible
  - To that end, most systems store a **histogram**



Cost Functions for SELECT Operation

Cost Functions for SELECT Operation

S1 - Linear search (brute force) approach

- We search all the file blocks to retrieve all records satisfying the selection condition
  - C_{Si}: Cost for method Si in block accesses → $C_{S1a} = b$
- For an equality condition on a key attribute, only half the file blocks are searched on the average before finding the record
  - So a rough estimate for $C_{S1b} = (b/2)$ if the record is found
  - If no record is found that satisfies the condition, $C_{S1b} = b$

Expected number of comparisons

$$E[X] = \sum_{x=1}^{n} xp(x) = \sum_{x=1}^{n} \frac{x}{n} = \frac{1}{n} + \frac{2}{n} + \cdots + \frac{n}{n} = \frac{n+1}{2}$$

Cost Functions for SELECT Operation

S2 - Binary search

- This search accesses approximately

  $C_{S2} = \log_2 b + \lceil (s/bfr) \rceil - 1$ file blocks

- This reduces to $\log_2 b$ if the equality condition is on a unique (key)

  attribute, because s = 1 in this case

C_{Si}: Cost for method Si in block accesses
sA: Selection cardinality of the attribute being selected (= $sl_A * r$)
bfr_X: Blocking factor (i.e., number of records per block) in relation X

Cost Functions for SELECT Operation

C_{Si}: Cost for method Si in block accesses
sA: Selection cardinality of the attribute being selected (= sl_A * r)
bfr_X: Blocking factor (i.e., number of records per block) in relation X

S3a - Using a primary index to retrieve a single record

- For a primary index, retrieve one disk block at each index level, plus one disk block from the data file
- Hence, the cost is one more disk block than the number of index levels: $C_{S3a} = x + 1$.

S3b - Using a hash key to retrieve a single record

- Only one disk block needs to be accessed in most cases
- The cost function is approximately $C_{S3b} = 1$ for static hashing or linear hashing, and it is 2 disk block accesses for extendible hashing

Cost Functions for SELECT Operation

C_{Si}: Cost for method Si in block accesses
sA: Selection cardinality of the attribute being selected (= sl_A * r)
bfr_X: Blocking factor (i.e., number of records per block) in relation X
xA: Number of levels of the index for attribute A

S6 - Using a secondary (B+-tree) index

- For a secondary index on a key (unique) attribute, with an equality (i.e., <attribute = value>) selection condition, the cost is x + 1 disk block accesses
- For a secondary index on a non-key (nonunique) attribute, s records will satisfy an equality condition, where s is the selection cardinality of the indexing attribute
  - Because the index is non-clustering, each of the records may reside on a different disk block, so the (worst case) cost estimate is $C_{S6a} = x + 1 + s$
  - The additional 1 is to account for the disk block that contains the record pointers after the index is searched

Cost Functions for SELECT Operation

C_{Si}: Cost for method Si in block accesses
sA: Selection cardinality of the attribute being selected (= sl_A * r)
b_{I1}A: Number of first-level blocks of the index on attribute A
xA: Number of levels of the index for attribute A
r_X: Number of records (tuples) in a relation X

S6 - Using a secondary (B+-tree) index

- For range queries, if the comparison condition is >, >=, <, or <= and half the file records are assumed to satisfy the condition, then (very roughly) half the first-level index blocks are accessed, plus half the file records via the index
- The cost estimate for this case, approximately, is $C_{S6b} = x + (b_{I1}/2) + (r/2)$
- The r/2 factor can be refined if better selectivity estimates are available through a histogram
- The latter method C_{S6b} can be very costly
- For a range condition such as v1 < A < v2, the selection cardinality s must be computed from the histogram or as a default, under the uniform distribution assumption; then the cost would be computed based on whether or not A is a key or non-key with a B+-tree index on A