# 11. Query Execution

[Most of the content is drawn from book Elmasri/Navathe]

A query can be expressed in different way;

(1) $\pi_{fname, dname, salary}($
$\sigma_{salary >= 30000 \text{ AND } employee.dno = department.dno}$
(employee x department))

(2) $\pi_{fname, dname, salary}(\sigma_{salary >= 30000}$ (employee)
* department )

(3) $\sigma_{salary >= 30000}(\pi_{fname, dname, salary}$
(employee * department))

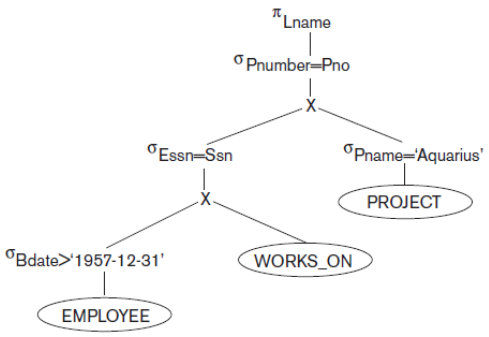
Which is more efficient to execute?

One of the idea behind logical data model is let the user express the queries that are mathematical correct without bothering about its efficient execution.

However in reality, one query expression could be more efficient than the other. Also there can be multiple strategies of processing disk files for answering a query. There is no standard answer that which strategy works better for performing a database manipulation operation. It requires considering many parameters - organization of data file, availability of indexes, file size, selectivity for a given value for an attribute, and so forth.

DBMS provides a module called *Query Optimizer*! The responsibility of query optimizer is to find out most optimal *query execution plan*. Let us attempt understanding what we mean by efficient query plan and how it is determined?

Execution tree in terms of logical (relational) operations, as shown in the figure here, is what termed as logical plan.




Each logical operation is to be performed using some *physical operations*, operations that are actually performed on disk files.

## Physical operations and Physical Plan

Following are typical physical operations-

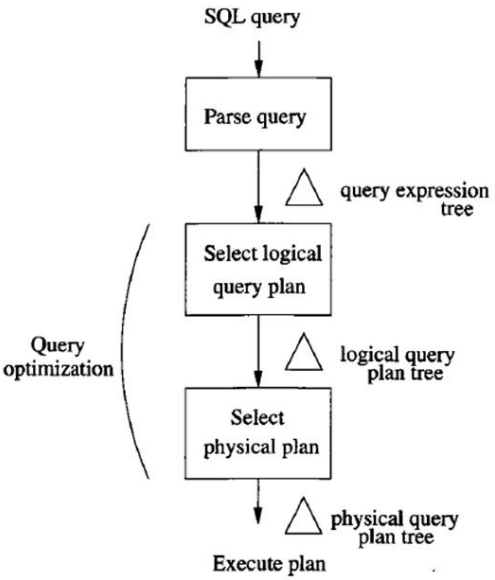
- Table Scan: performed for finding desired data records. Table scan is a sequential search operation.

- Sorted Scan: when data records are sorted and search is to be performed on attributes of sort order. Binary search can be performed in this case.
- Index Scan: searching is performed through index. Starting from root node, appropriate access path is followed and relevant leaf node is located, and reading of desired data blocks.

*Physical plan* is basically sequence of physical operations in order to execute for a logical plan.

## Query Optimization

Query optimization is done in steps showed here in the diagram drawn from book[1]




Parsing and "Execution Tree" generation:
Checks for syntax errors, checks for correct references to tables and attributes; and finally produces "query execution tree" typically in terms of relational algebra.

### Select Logical Query Plan (Query rewriting):

It is learnt that if reorder operations (like selection, join, and so) query will execute faster. Also query execution can further be improved if certain operations are split into multiple or by combining multiple operations in one. Using certain heuristic rules queries are rewritten that would be more efficient. Following are such some rules -

- If possible selection should be executed early in the order; this reduces size of operand relations in following operations, and that will reduce the overall *cost of execution*.
- If a user has submitted a query in which JOIN is expressed in terms of CROSS PRODUCT, it should be rewritten using JOIN.

Outcome of the step is an Evaluation Tree in terms of relational operations (note that order of operations also gets specified in evaluation tree)

### Select Physical Query Plan:

Unclustered index

For producing a physical plan for a given logical plan, often we have multiple options. Available options depend on file organization and availability of indexes or so. While choosing a physical plan, query optimizer uses a concept of "cost" of a plan. It chooses a plan that has lowest cost. A cost function typically should be function of available memory, number of processors, speed of disk access, number of disk blocks read and written, and so.

[1] Garcia-Molina, Hector. Database systems: the complete book. Pearson Education India, 2008.

For database processing purposes, number of data blocks processed remains most significant portion of cost. Therefore we will be using this only for our discussions here.

Final outcome of this step is "code for executing the query"

## Algorithms for query execution

Parameters, used in cost of algorithm:

- N: number of records in a relation
- Rs: record size
- B: number of blocks in a relation
- BF (Blocking factor): number of records in a block
- Selection Cardinality S = N / distinct values for an attributes
- Selectivity (fractional selection cardinality) = S / N
- H: height of B+-tree index tree
- Join Selectivity (js) describe later

## Sorting Algorithms:

Sorting is one of the common physical task done while answering this query; sorting typically required in

- In executing ORDER BY
- In duplicate removal
- In Sort-Merge technique for JOIN
- In performing various set operations like UNION, INTERSECTION

External sorting is common techniques for sorting large files. It uses sort-merge strategy. In this case sorting is done in chunks (typically of size available main memory for the sort); then sorted chunks are merged in one, and final sorted file is produced. The cost of sorting is estimated to $(2 \times B) + (2 \times B \times (log_{dM} N))$; Where dM is called as the degree of merging, that is number of sorted subfiles that can be merged in each merge step.

## Algorithms for SELECTION operation:

### Simple Selections

(S1) Linear Search: brute force search on data file; often referred as *table scan*.

Cost Estimate: *B/2* for key attributes; B for non-key attributes

(S2) Binary Search on data file: requires records in data file to be ordered.

Cost Estimate: For key attribute: $log_2(B)$; and
for non-key attribute: $log_2(B) + \lceil S/BF \rceil - 1$ (Using this formulation for key attribute; s=1, and cost comes to $log_2(B)$

(S3) Equality Search based on Key

For example: answering query $\sigma_{ssn=12345}(EMP)$; and
relation having primary index on SSN.

>> B+-tree index: Locate leaf node (H) + Read appropriate data block (1)

= H + 1

>> Hashing Index: typically 2 blocks for extendible hashing

(S4) Equality Search based on Non-Key

For example answering of queries like $\sigma_{dno=5}(EMP)$

>> B+-tree clustered index (data blocks are sorted)

Strategy: locate leaf node, and then scan in disk block for all records for given value in search key

= $H + \lceil S/BF \rceil$

(S5) Range Search on Key

For example answering of queries like $\sigma_{ssn > 56756}(EMP)$

>> B+-tree clustered index (data blocks are sorted)

Strategy: reach to the leaf node that matches with ssn=56756, and then sequentially scan the data blocks.

Cost Estimate: $H + B/2$; $H$ to reach to first/last leaf node meeting the condition. This is very rough assuming that half the records (B/2 blocks) will meet the selection criteria. A more accurate estimate is possible if the key value wise distribution of records is stored in a histogram.

>> B+-tree un-clustered index (data blocks are not sorted)

Strategy: require scanning leaf nodes as long as value of search key remains given value. Separate data block is read for each matched value in index leaf nodes. Number of data blocks to be read are S.

Cost estimate: $H + S$

(S6) Range Search on Non-Key

For example answering of queries like $\sigma_{dno>5}(EMP)$

>> B+-tree clustered index (data blocks are sorted)

Strategy: reach to the leaf node that matches with dno=5, and then sequentially scan the data blocks.

Cost Estimate: $H + B/2$; $H$ to reach to first/last leaf node meeting the condition. This is very rough assuming that half the records (B/2 blocks) will meet the selection criteria. A more accurate estimate is possible if the key value wise distribution of records is stored in a histogram.

>> B+-tree un-clustered index (data blocks are not sorted)

Strategy: Leaf nodes are scanned, and disk block is read for every index entry. Average case number of blocks to be read is estimated to N/2. Again it is very rough estimate, and half the file records by the index are accessed; cost estimate: $H + N/2$

Again N/2 is very approximate, can be replaced by selectivity for range values.

## Selections with conjunctive conditions

(S7) Conjunctive selection using an individual index:

If any of the attribute has an index; use any of the approach S2 to S6 (whichever is applicable); and do sequential search *n* accessed record on other attribute.

For example dno=5 and salary > 50000; we can perform index search on dno while linear for salary on selected records for dno=5. Considering secondary index on DNO comes out to same as of S6b.

(S8) Conjunctive selection using a composite index

If we have composite index on both; then can directly use it as any of approach S2 to S6, whichever is applicable

Other example, we may have index on ssn, pno in works_on relation

(S9) Conjunctive selection by intersection of record pointers

if we have individual indexes on multiple attributes; first, we have individual records selected using respective indexes; and then compute intersection. Restriction on remaining attributes can be done by sequential scan in result of intersection.

While performing conjunctive selections; access path on attribute with low selectivity is desirable to be executed first

### Selections with disjunctive conditions

Selections based on disjunctive conditions are hard to execute; if any of the participating attribute does not have access path (i.e. index); then we are forced to have "sequential scan".

There is hardly any optimization can be done in such cases.

If access paths for all attributes are there, then individual index based selections can be union-ed.

Having these many options available for performing selection; DBMS does some cost analysis and choose most efficient approach for evaluating the query.

Many things play role- record size, index availability, index size, number of records, selectivity, available memory, etc.

Cost formulas are defined for implemented algorithms, and used by query optimizer

## Join Algorithms

Join examples used here

OJ1: $R \overset{\bowtie}{\underset{R.A = S.B}{\boxed{}}} S$

OJ2: $EMP \overset{\bowtie}{\underset{E.DNO = D.DNO}{\boxed{}}} DEP$

OJ3: $EMP \overset{\bowtie}{\underset{ssn = mgrssn}{\boxed{}}} DEP$

Join Selectivity:

Join Selectivity (js) is defined as the ratio of number of tuples in join result with the number rows in corresponding cross product: |R JOIN S| / |R CROSS S| = |R JOIN S| / (|R| * |S|)

Its value can be anything from 0 to 1; js=1 if no join condition; it is zero if there no match at all. If A is key in R, then every tuple in S can atmost be joined with one tuple in R (i.e. say S.B is FK referring into R.A) then size of JOIN will be <= |S|; and js will be <= (|S|)/(|R|*|S|) , will <= 1/|R|; if attribute S.B has NOT NULL constraint; then size of JOIN result will be |S|, and js = 1/|R|.

Join selectivity helps in estimating size of join resultant relation.

(J1) Nested block join

This is brute-force and default technique for join.

It goes as following one for every block in a relation R, scan every block in other relation S; cases in which join condition is met, joined tuple is computed and added to the result.

Cost Estimate: $B_R + B_R \times B_S + (js \times |R| \times |S|/BF_{RS})$

Last part of cost is for writing result file.

If available memory is also accounted for, Let us say available memory is M blocks, cost of nested block comes as following-

Cost Estimate: $B_R + \lceil B_R/(M-2) \rceil \times B_S + (js \times |R| \times |S|/BF_{RS})$

*Important question, which relation should be used in outer loop?*

Normally the relation that has lesser tuples is used in outer loop; sample calculation follows.

(J2) Single loop Join

When we have index on one relation in join (on attribute of that relation in join condition); for example we having index on SSN in join operation OJ3 above.

We need to have one scan on relation without index, and have index lookup for find matched tuple from other relation. In example OJ3, we can have sequential scan on DEP and index look on EMP (ssn)

Following are cost estimates for different types of indexes; suppose R is scanned and index lookup is done in S

Secondary index: $B_R + (|R| \times (H_{BS} + S_{BS})) + (js \times |R| \times |S|/BF_{RS})$

For every value of R.A, and we perform index lookup in S.B

Clustered index (non-key): $B_R + (|R| \times (H_{BS} + S_{BS}/BF_S)) + (js \times |R| \times |S|/BF_{RS})$

Primary index (key attribute index): $B_R + (|R| \times (H_{BS} + 1)) + (js \times |R| \times |S|/BF_{RS})$

*Note: Notation $H_{BS}$ represents height of b+-index on attribute B in relation S, and $S_{BS}$ denotes Selectivity of attribute B in relation S. $BF_{RS}$ represents Blocking Factor of joined relation RS.*

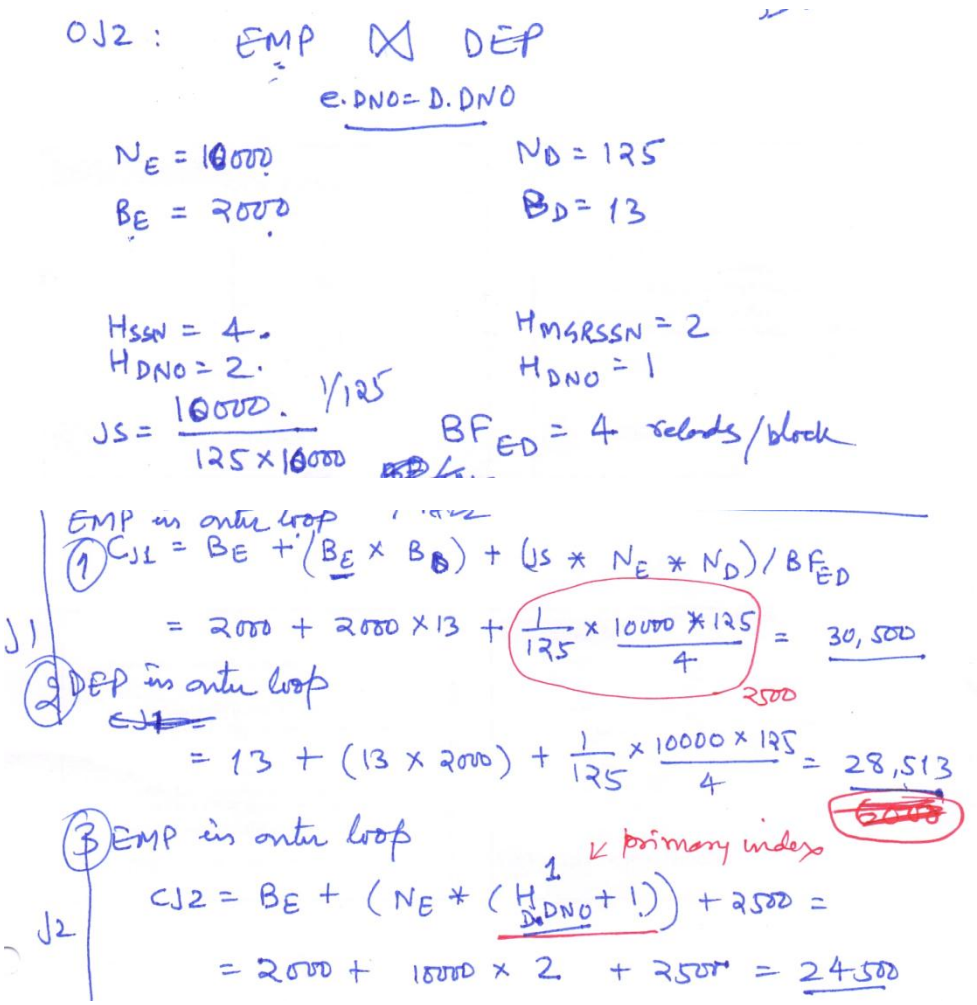
Hashing: $B_R + (|R| \times hf) + (js \times |R| \times |S|/BF_{RS})$

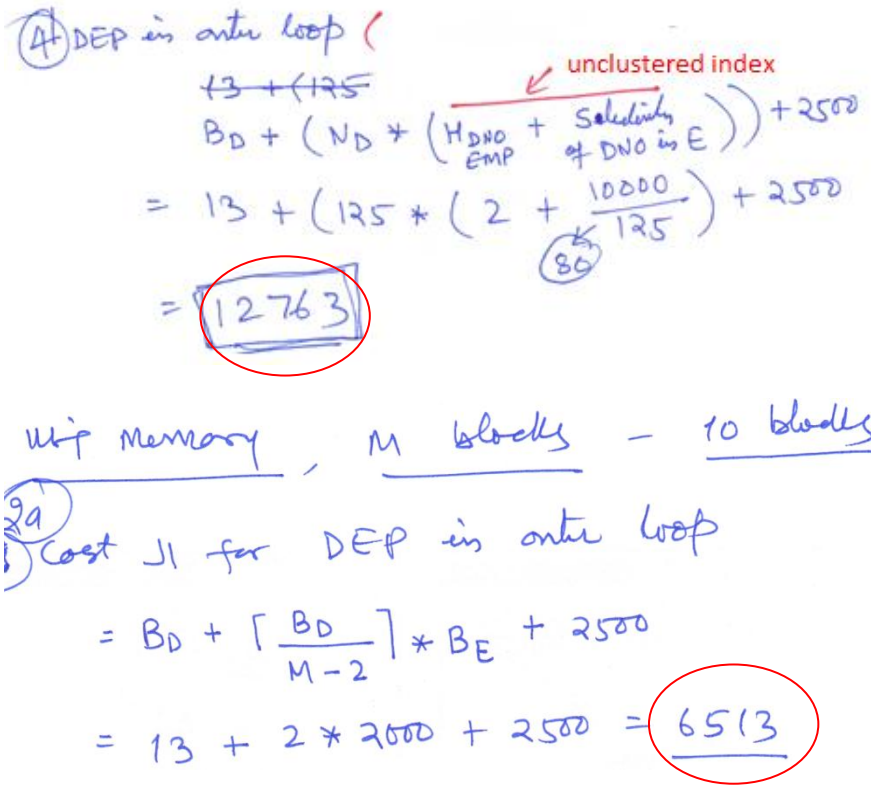
Here *hf* is number of block accesses to retrieve a record, given its hash key value. Typically 2 for extendible hashing

Below is an worked example for cost in Join, we can roughly infer following from the sample calculations –

(1) It is better to a smaller relation in outer loop – in both cases
(2) If sufficient memory is available, a nested join may turn out to be better than single loop join.

A worked example for cost for JOIN






(J3) Sort-Merge Join

If records of R and S are already sorted on A and B respectively, this could turn out to be efficient

Simultaneous scan of both files and generated combined tuples for matches yields join results

Cost Estimate (assuming that records of both relations are already sorted):
$B_R + B_S + (js \times |R| \times |S|/BF_{RS})$

Cost Estimate (with sorting):
$(2 \times B) + (2 \times B \times (log_{dM} N)) + B_R + B_S + (js \times |R| \times |S|/BF_{RS})$

(J4) Partition Hash

- Same hash function is used for both the attributes A in R and B in S.
- Done in two steps -
  - First, a single pass through the file with fewer records (say, R) and hashes its records to the various partitions of R; this is called the *partitioning phase*. Partitioning phase because, the records of R are partitioned into the hash buckets.
  - Next is *probing phase*, scan through all records in S one by one, using same hash function find out buckets in hash table of R (HR). Combine all found records in the bucket with S's record and append to the result.

## Algorithms for PROJECT

PROJECT is straight forward as subset of attributes needs to be selected.

However project might produce duplicate tuples; therefore duplicate removal might be needed. You should not that duplicate removal is not needed in SQL unless it has been asked for (DISTINCT keyword).

Duplication removal can be done by sorting or hashing.

## Algorithms for SET operations

Set operations—UNION, INTERSECTION, SET DIFFERENCE, and CARTESIAN PRODUCT are often expensive; particularly cross product.

For set operations Sort-Merge is effective technique.

Hashing is again effective technique.

## Heuristics for query [logical query] optimizations

Parser produces initial parse tree without any optimization.

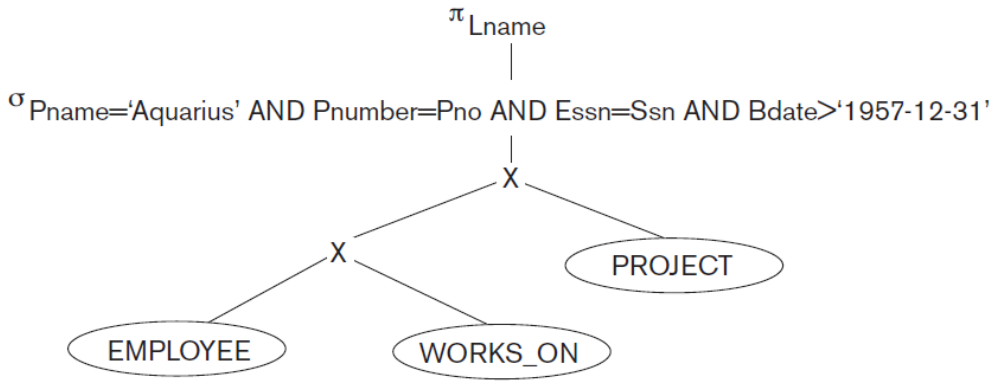
Let us say SQL Query for

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

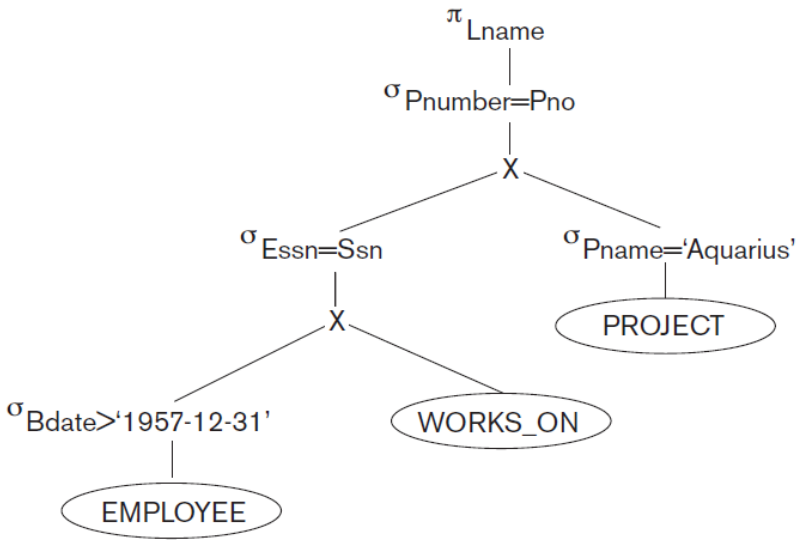
Say expressed SQL as

```
SELECT lname from employee, works_on, project
WHERE pname=‘aquarius’ and pnumber=pno and essn=ssn and
        bdate > ‘1957-12-31’;
```

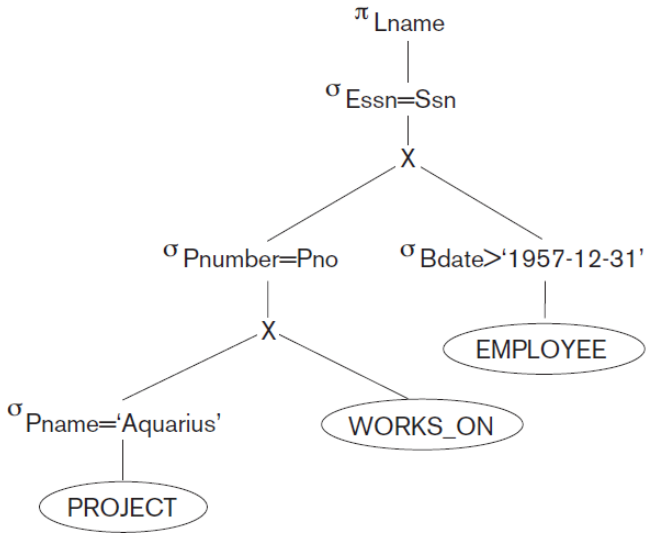
(a) Initial query tree for SQL query above: without any optimization, just translation



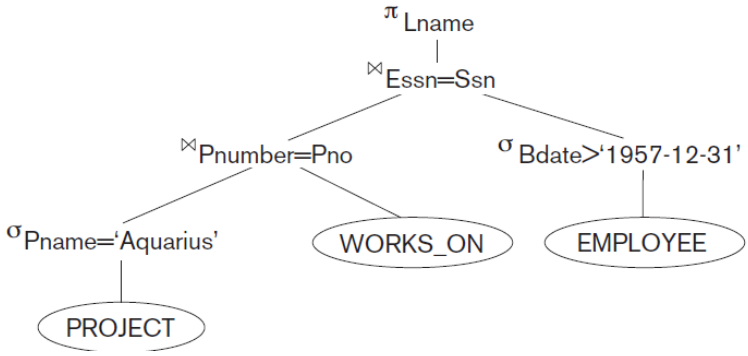
(b) Below is tree after moving SELECT operations down the query tree



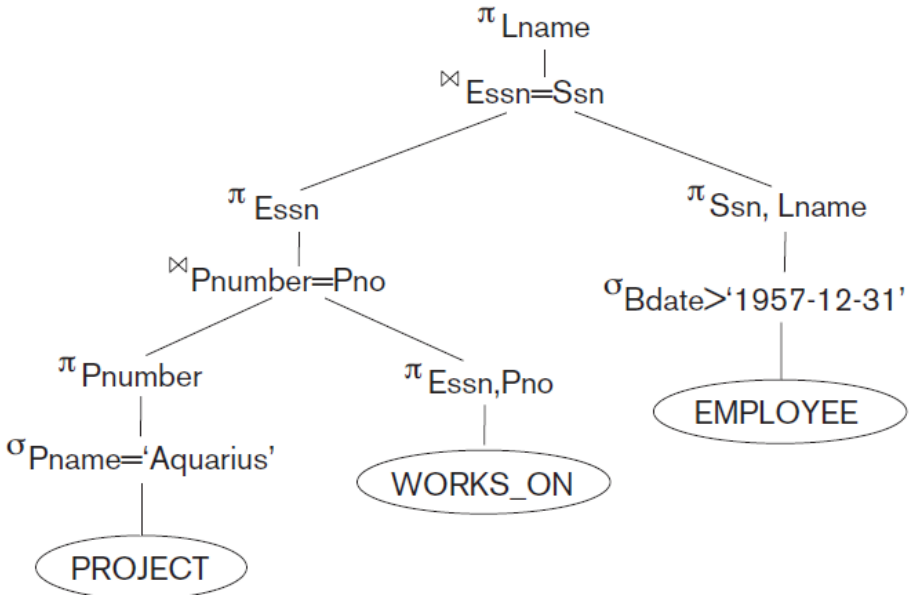

(c) Below is tree after applying the more restrictive SELECT operation first.




(d) Below is tree after replacing CARTESIAN PRODUCT and SELECT with JOIN operations.



(e) Below is tree after moving PROJECT operations down the query tree, and is final optimized tree at logical level.




General Transformation Rules for Relational Algebra Operations.
Below is representative list of rules drawn from book elmasri/navathe-

1. Cascading of σ
   $\sigma_{c_1 \text{ AND } c_2 \text{ AND } \ldots \text{ AND } c_n}(R) \equiv \sigma_{c_1}(\sigma_{c_2}(\ldots(\sigma_{c_n}(R))\ldots))$
2. Commutativity of σ
   $\sigma_{c_1}(\sigma_{c_2}(R)) \equiv \sigma_{c_2}(\sigma_{c_1}(R))$
3. Cascade of π
   $\pi_{\text{List}_1}(\pi_{\text{List}_2}(\ldots(\pi_{\text{List}_n}(R))\ldots)) \equiv \pi_{\text{List}_1}(R)$
4. Commuting σ with π
   $\pi_{A_1, A_2, \ldots, A_n}(\sigma_c(R)) \equiv \sigma_c(\pi_{A_1, A_2, \ldots, A_n}(R))$
5. Commutativity of ⋈ (and ×)
   $R \bowtie_c S \equiv S \bowtie_c R$
   $R \times S \equiv S \times R$
6. Commuting σ with ⋈ (and ×)
   $\sigma_c(R \bowtie S) \equiv (\sigma_c(R)) \bowtie S$
   Alternative if condition c can be split into c1 and c2 drawing attributes from R and S only, can be expressed as
   $\sigma_c(R \bowtie S) \equiv (\sigma_{c_1}(R)) \bowtie (\sigma_{c_2}(S))$
7. Commuting π with ⋈ (and ×)

If set of attributes L = {a1,a2, an} from R and {b1,b2,..bm} from S, and attributes in condition c is from L, then can be expressed as

$$\pi_L (R \bowtie_c S) \equiv (\pi_{A_1, \ldots, A_n} (R)) \bowtie_c (\pi_{B_1, \ldots, B_m} (S))$$

If attributes in c not there in L, then can be added to respective side, and can be rewritten

8. Commutativity of set operations. The set operations ∪ and ∩ are commutative but − is not.
9. Associativity of $\bowtie$, ×, ∪, and ∩. If θ stands for any of these, can be expressed as $(R \theta S) \theta T \equiv R \theta (S \theta T)$
10. Commuting σ with set operations. If θ stands for any one from ∪, ∩, and −, can be expressed as -

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

11. Converting a (σ, ×) sequence into $\bowtie$

$$(\sigma_c (R \times S)) \equiv (R \bowtie_c S)$$

12. The π operation commutes with ∪

$$\pi_L (R \cup S) \equiv (\pi_L (R)) \cup (\pi_L (S))$$

### Outline of a Heuristic Algebraic Optimization Algorithm

1. Move SELECT operations with conjunctive conditions into a cascade of SELECT operations. This permits a greater degree of freedom in moving SELECT operations down in different branches of the tree
2. Move each SELECT operation as far as down the query tree as is permitted by the attributes involved in the select condition.
3. Move more restrictive selects down so that intermediate results are smaller; we can say most restrictive condition is the one that has less *selectivity*
4. If a condition involves attributes from multiple relation then it is likely to be a join condition. Replace Cartesian products with JOIN

The main intuition of query optimization is order the operations such that they minimize the size intermediate results – preform selection and projection as early as possible.

## EXPLAIN command of SQL

- You can get a fair idea how query planner work by observing various execution plans by using EXPLAIN command in Postgres

```
EXPLAIN SELECT fname, dname, salary FROM employee NATURAL JOIN
department
WHERE salary > 50000;
```

Below are some screen shot for few queries – try interpreting the output

```
pmjat=> explain select * from employee natural join department;
                            QUERY PLAN
------------------------------------------------------------------
 Hash Join  (cost=1.04..2.24 rows=8 width=131)
   Hash Cond: ("outer".dno = "inner".dno)
   ->  Seq Scan on employee  (cost=0.00..1.08 rows=8 width=99)
   ->  Hash  (cost=1.03..1.03 rows=3 width=34)
         ->  Seq Scan on department  (cost=0.00..1.03 rows=3 width=34)
(5 rows)
```

```
pmjat=> explain select * from employee e, department d where e.dno = d.dno;
                            QUERY PLAN
------------------------------------------------------------------
 Hash Join  (cost=1.04..2.24 rows=8 width=133)
   Hash Cond: ("outer".dno = "inner".dno)
   ->  Seq Scan on employee e  (cost=0.00..1.08 rows=8 width=99)
   ->  Hash  (cost=1.03..1.03 rows=3 width=34)
         ->  Seq Scan on department d  (cost=0.00..1.03 rows=3 width=34)
(5 rows)
```

A blog https://use-the-index-luke.com/sql/explain-plan/postgresql/operations provides a basic introduction to postgresql “explain” operations.

```
pmjat=> explain select * from employee e, department d where e.ssn = d.mgrssn;
                            QUERY PLAN
------------------------------------------------------------------
 Merge Join  (cost=2.25..2.33 rows=3 width=133)
   Merge Cond: ("outer".ssn = "inner".mgrssn)
   ->  Sort  (cost=1.20..1.22 rows=8 width=99)
         Sort Key: e.ssn
         ->  Seq Scan on employee e  (cost=0.00..1.08 rows=8 width=99)
   ->  Sort  (cost=1.05..1.06 rows=3 width=34)
         Sort Key: d.mgrssn
         ->  Seq Scan on department d  (cost=0.00..1.03 rows=3 width=34)
(8 rows)
```

```
pmjat=> explain select ssn, fname, salary, pno,hours from employee e, works_on w
 where e.ssn = w.essn;
                            QUERY PLAN
------------------------------------------------------------------
 Merge Join  (cost=2.72..3.01 rows=17 width=46)
   Merge Cond: ("outer".ssn = "inner".essn)
   ->  Sort  (cost=1.20..1.22 rows=8 width=34)
         Sort Key: e.ssn
         ->  Seq Scan on employee e  (cost=0.00..1.08 rows=8 width=34)
   ->  Sort  (cost=1.52..1.56 rows=17 width=26)
         Sort Key: w.essn
         ->  Seq Scan on works_on w  (cost=0.00..1.17 rows=17 width=26)
(8 rows)
```

```
pmjat=> explain select ssn, fname, salary, pno,hours from employee e, (select *
from works_on) as w where e.ssn = w.essn;
                            QUERY PLAN
------------------------------------------------------------------------
 Merge Join  (cost=2.72..3.01 rows=17 width=46)
   Merge Cond: ("outer".ssn = "inner".essn)
   ->  Sort  (cost=1.20..1.22 rows=8 width=34)
         Sort Key: e.ssn
         ->  Seq Scan on employee e  (cost=0.00..1.08 rows=8 width=34)
   ->  Sort  (cost=1.52..1.56 rows=17 width=26)
         Sort Key: works_on.essn
         ->  Seq Scan on works_on  (cost=0.00..1.17 rows=17 width=26)
(8 rows)
```