11. Query Execution

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

Query Execution

Relational is a logical model. Allows expressing queries "logically".

Queries are expected to be logically (mathematically) correct, and a user writing queries need not worry about their efficient execution.

A DBMS implementation should ensure the efficient execution of queries.

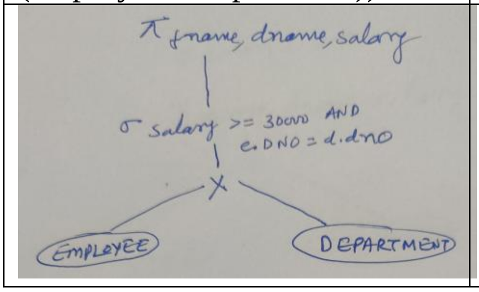
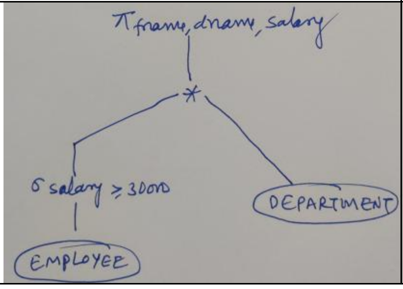
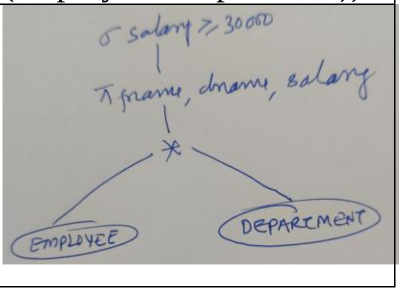
DBMS attempts executing queries in most efficient way, regardless of how they have been expressed by the user. DBMS provides a dedicated module, "Query Optimizer", that figures out most "efficient way" for executing a query.

"Most efficient way", referred as "most optimal plan", is derived as following:

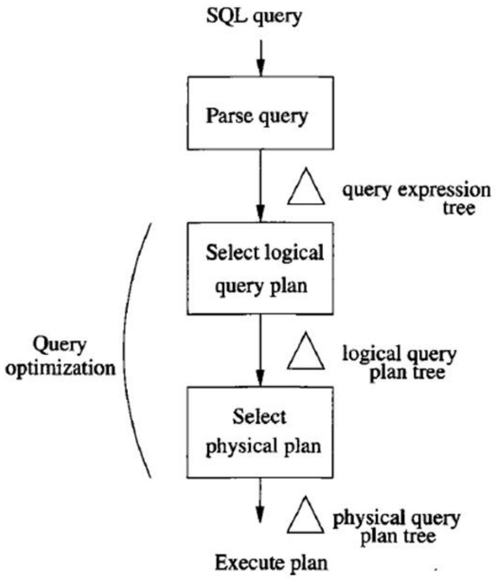
1. "Query operations" are "re-organized", that is in what order "select", "project" "join" etc. are to be performed. Here query optimizer uses various heuristic (and equivalence) rules. This is called query re-writing for most "optimal logical plan".
2. Logical operations are performed using "various physical" operations like file scan, index scan, hashing, sorting, etc. Most efficient "physical plan" that is identification of physical operations and their order is determined. This is often done by "cost estimation". While cost estimation considers various factors like availability of index, number of tuples in a table, selectivity of the query predicate etc.

Outcome of optimization is a sequence of physical operations to be executed in order to execute the query!

Let us say a query can be expressed in the following different ways; which is most efficient to execute?

$\pi_{\text{fname, dname, salary}}($ $\sigma_{\text{salary >= 30000 AND e.dno = d.dno}}$ (employee x department))	$\pi_{\text{fname, dname, salary}}($ $\sigma_{\text{salary >= 30000}}$ (employee) * department)	$\sigma_{\text{salary >= 30000}}($ $\pi_{\text{fname, dname, salary}}$ (employee * department))
 π fname, dname, salary σ salary >= 30000 AND e.DNO = d.dno X EMPLOYEE DEPARTMENT	 π fname, dname, salary * σ salary ≥ 30000 DEPARTMENT EMPLOYEE	 σ salary ≥ 30000 π fname, dname, salary * EMPLOYEE DEPARTMENT

Query Optimization



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 to execute than the other.

DBMS provides a module called *Query Optimizer*! The responsibility of the query optimizer is to find out the most *optimal query execution plan*.

The figure here sketches out query optimization as the activity. Query optimization is done in steps shown here in the diagram drawn from the 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 observed and proved that reordering of logical operations (like selection, join, and so) helps in finding executing the query faster. Also, query execution can further be improved if certain operations are split into multiple or by combining multiple operations in one.

Therefore, by using certain heuristic and equivalence rules, queries are modified ("rewritten") to the most optimal logical plan.

Following are a few such rules -

- If possible selection should be executed early in the order; this reduces the 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, this should be rewritten using JOIN.

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

Suppose, for the query on the previous page, the second tree is the most optimal. It does not matter what the user expresses, the query shall be converted to the second one.

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

Select Physical Query Plan:

Each logical operation is to be performed using some *physical operations*, operations that are performed on disk files. A *physical plan* is a sequence of physical operations to execute a logical plan.

Physical Operations and Physical Plan

The following are typical physical operations-

- Table Scan: performed to find desired data records. A table scan is a sequential search operation.
- Sorted Scan: when data records are sorted and a search is to be performed on attributes of sort order. A binary search can be performed in this case.
- Index Scan: Searching is performed through an index. Starting from the root node, the appropriate access path is followed, a relevant leaf node is located, and then the desired data block is read.

For producing a physical plan for a given logical plan, we often have multiple options.

For choosing a physical plan, the query optimizer uses the concept of the “cost” of a plan. It chooses a plan that has the lowest cost. A cost function typically considers various parameters - organization of data file, availability of indexes, file size, selectivity for a given value for an attribute, and so forth. It also uses available memory, number of processors, speed of disk access, number of disk blocks read and written, and so.

Algorithms for query execution

Here, we study various “**physical algorithms**” for performing logical operations. Also, give a brief explanation of the “cost” for each physical technique. For database processing purposes, the number of data blocks processed remains the most significant portion of the cost. Therefore, we will be using this only for our discussions here.

Here are the parameters used in the cost:

- 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

Note: Cost Shown here are average case

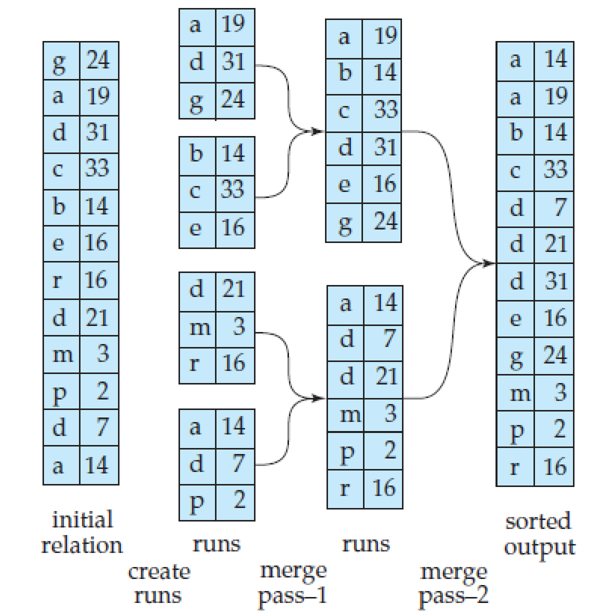
Sorting Algorithms:

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 a common technique for sorting large files. It uses a 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 a number of sorted sub-files that can be merged in each merge step. 2xB, because 1B for read and 1B for write.

Diagram below from book[2] captures the intuition of sort-merge algorithm-



```
Sort-Phase
 $i = 0$;
 repeat
     read $M$ blocks of the relation, or the rest of the relation,
         whichever is smaller;
     sort the in-memory part of the relation;
     write the sorted data to run file $R_i$;
     $i = i + 1$;
 until the end of the relation

Merge-Phase
read one block of each of the $N$ files $R_i$ into a buffer block in memory;
repeat
    choose the first tuple (in sort order) among all buffer blocks;
    write the tuple to the output, and delete it from the buffer block;
    if the buffer block of any run $R_i$ is empty and not end-of-file($R_i$)
        then read the next block of $R_i$ into the buffer block;
until all input buffer blocks are empty
```

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: $log_2(B)$; [10]
For non-key attribute: $log_2(B) + \lceil S/BF \rceil - 1$
[10+200/10-1=19]
(Using this formulation for key attribute; s=1, and cost comes to $log_2(B)$

Employee:
N=10000
BF=10
B=1000
H(eno)=3
S(dno)=200
H(dno)=5

(S3) Equality Search based on Key (when we have B+-tree Index)

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

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

= H + 1 [3+1]

>> Hashing Index: typically, 2 blocks

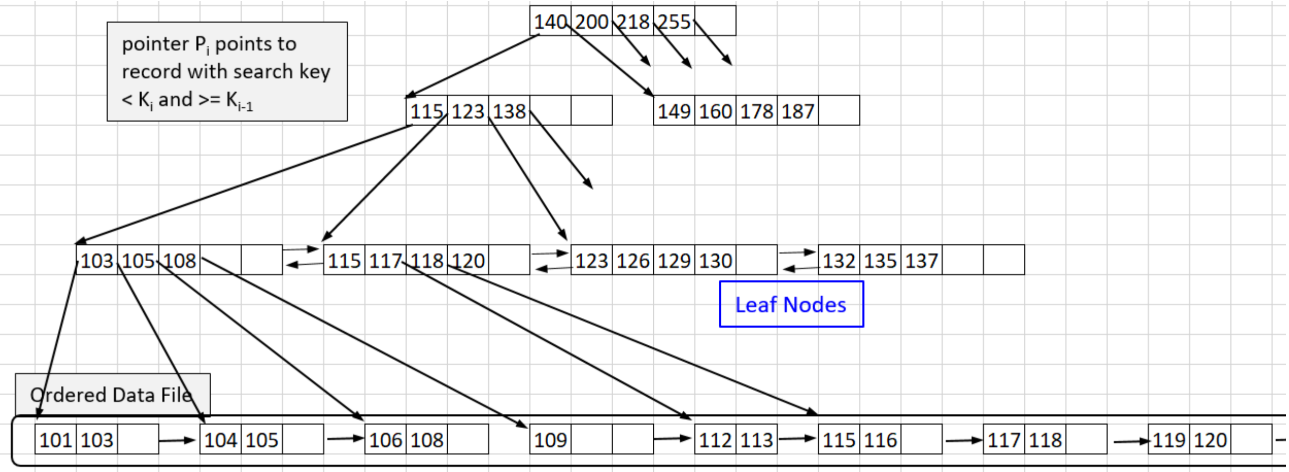
(S4) Equality Search based on Non-Key (when we have B+-tree Index)

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

$= H + \lceil S/BF \rceil$ [5+200/10=25]



(S5) Range Search on Key (when we have B+-tree Index)

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

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

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

Let us say 50% of records meet the criteria (a very rough estimate), then Cost Estimate: $H + B/2$; [3+500]

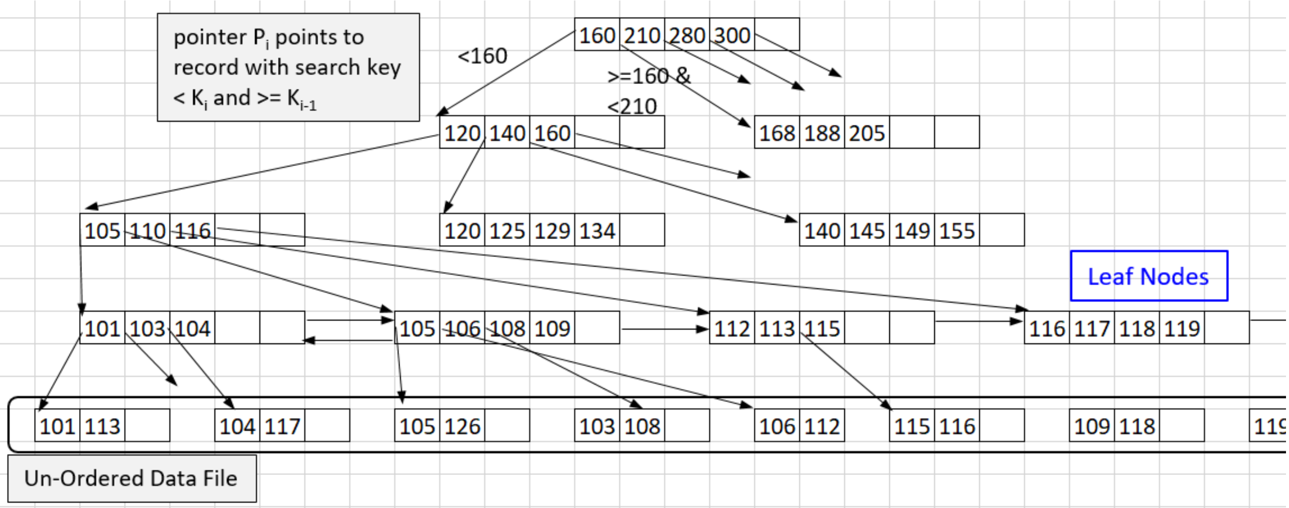
H to reach to first/last leaf node meeting the condition. This is very rough assuming that half the records (B/2 blocks) will meet the selection criteria.

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. If number of data blocks that are found to be within the range are M (say 50).

Cost estimate: $H + M$ [5+50]



(S6) Range Search on Non-Key (when we have B+-tree Index)

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

(a) 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: $H + B/2$ (roughly assuming 50% records fall in the range)

(b) 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, and 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, we can use it for searching for that attribute and then 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, technique S6b comes out to be appropriate in this case; that is : $H + N/2$

(S8) Conjunctive selection using a composite index

We can directly use the index.

For example, works_on table has index on {ENO, PNO}; we can use it.
Cost: H+1

(S9) Conjunctive selection by intersection of record pointers

if we have individual indexes on multiple attributes;

We can first search based individual indexes and finally compute the intersection.

Restriction on remaining attributes can be done by sequential scan in result of intersection.

Selections with disjunctive conditions

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

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

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

Summary Note:

We have various alternatives for performing “selection operation”; DBMS does some cost analysis and choose most efficient approach for performing the operations!

As seen, 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 are used by query optimizer

Join Algorithms

Join examples used here

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

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

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

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| x |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 be joined with at-most 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|x|S|); i.e. <= 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 the size of the join resultant relation.

Following are some approaches for performing joins:

(J1) Nested block join

This is a brute-force and default technique for joining.

It goes as follows: for every block in a relation R, scan every block in another relation S; when the join condition is met, the 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 the outer loop?

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

(J2) Single loop Join

When we have an index on one of the relation in join (that is, on the attribute of an operand relation in join condition); for example, we have an index on ENO in join operation OJ3 above.

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

Suppose R is scanned and index lookup is done in S - for every value of R.A, and we perform index lookup for S.B

The following are cost estimates for different types of indexes.

Primary index (key attribute index) and Clustered:

Read B_R blocks, index lookup of S |R| time, and we have cost, as follows
$B_R + (|R| \times (H_{BS} + 1)) + (js \times |R| \times |S|/BF_{RS})$

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.

Clustered index (non-key attribute):

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

<u>Unclustered secondary index (non-key attribute)</u>- :

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

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

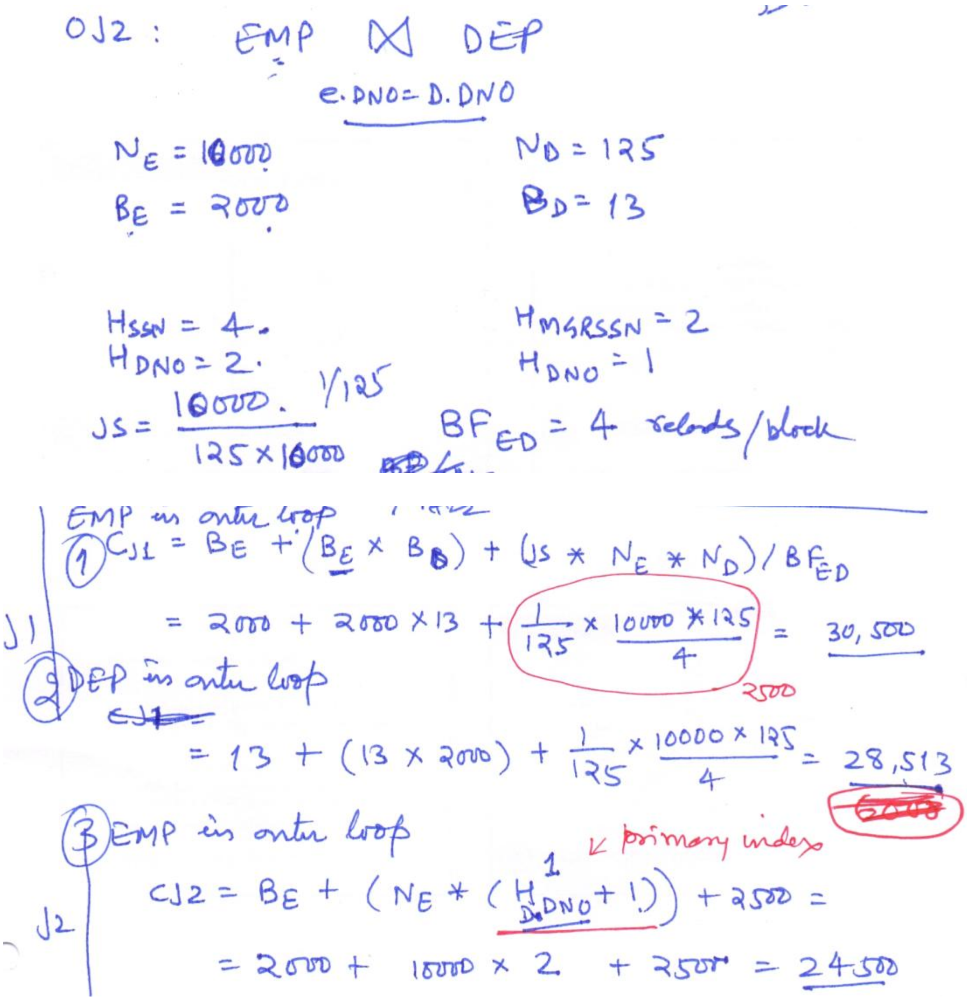
Here *hf* is number of block accesses to retrieve a record, given its hash key value.

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

(1) It is better to have a smaller relation in the outer loop – through both approaches (J1 and J2)

(2) If sufficient memory is available, a nested join may turn out to be better than a single loop join.

<u>A worked example for cost for JOIN</u>



OJ2 : EMP $\bowtie_{\underline{e.DNO = D.DNO}}$ DEP

$N_E = 10000$ $N_D = 125$

$B_E = 2000$ $B_D = 13$

$H_{SSN} = 4.$ $H_{MGRSSN} = 2$

$H_{DNO} = 2.$ $H_{DNO} = 1$

$JS = \dfrac{10000.}{125 \times 10000}$ 1/125 $BF_{ED} = 4$ records/block

J1

EMP is outer loop

① $C_{J1} = B_E + (\underline{B_E} \times B_D) + (Js * N_E * N_D) / BF_{ED}$

$= 2000 + 2000 \times 13 + \boxed{\frac{1}{125} \times \frac{10000 * 125}{4}} = \underline{30,500}$ 2500

② DEP is outer loop

~~CJ1~~

$= 13 + (13 \times 2000) + \frac{1}{125} \times \frac{10000 \times 125}{4} = \underline{28,513}$ $\boxed{\cancel{6000}}$

J2

③ EMP is outer loop

$CJ2 = B_E + (N_E * (\overset{1}{H_{D.DNO}} + 1)) + 2500 =$ ↙ primary index

$= 2000 + 10000 \times 2 + 2500 = \underline{24500}$

(4) DEP in outer loop (

~~13 + (125~~

$B_D + (N_D * (H_{\substack{DNO\\EMP}} + \overline{\text{Selectivity of DNO in E}})) + 2500$ unclustered index

$= 13 + (125 * (2 + \frac{10000}{125})) + 2500$ (80)

$= \boxed{12763}$

w/ Memory, M blocks - 10 blocks

(2a) Cost J1 for DEP in outer loop

$= B_D + \lceil \frac{B_D}{M-2} \rceil * B_E + 2500$

$= 13 + 2 * 2000 + 2500 = \underline{6513}$

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

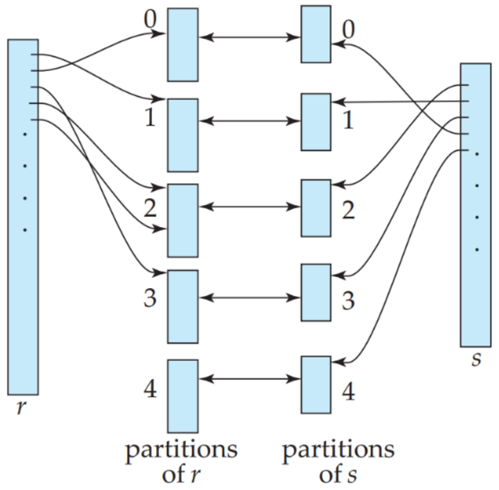
Cost Estimate (with sorting):
$(2 \times B) + \big(2 \times B \times (log_{dM}N)\big) + B_R + B_S \;+\; (\,js \times |R| \times |S|/BF_{RS})$

(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.

This is most promising join techniques. However possible to be used only for equi and natural joins. A conservative estimate of Hash Join is 3(Br + Bs)

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.

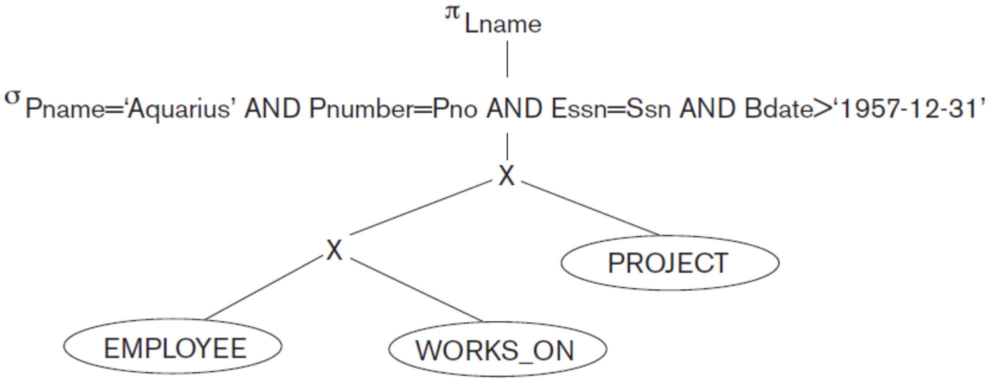
Let us say SQL Query for

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

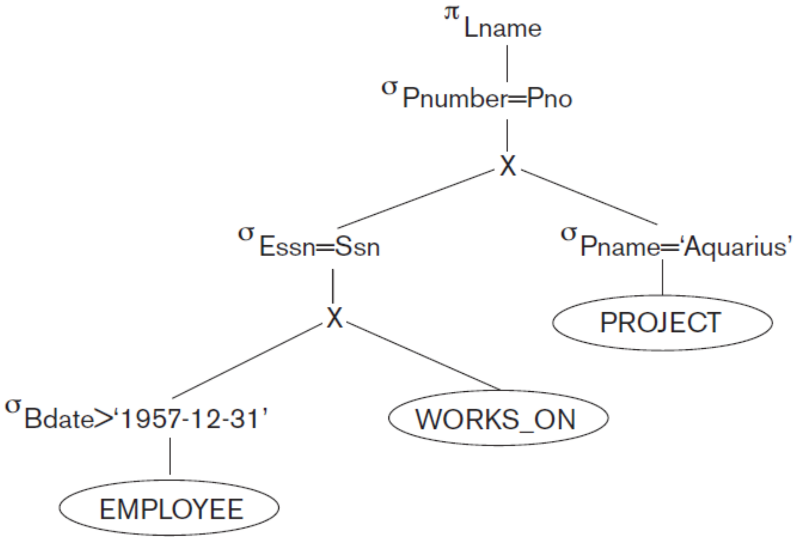
Say expressed SQL as

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

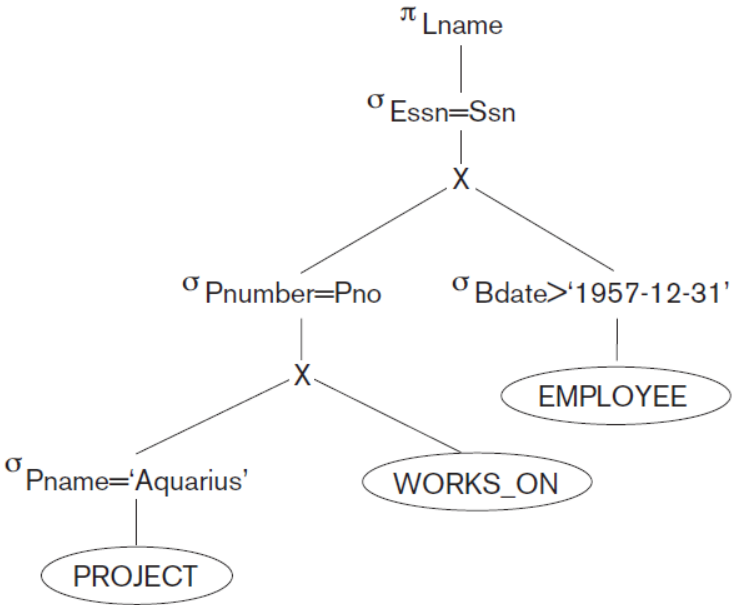
(a) Initial query tree for SQL query above: without any optimization, just translation


π Lname
σ Pname='Aquarius' AND Pnumber=Pno AND Essn=Ssn AND Bdate>'1957-12-31'
X
X
PROJECT
EMPLOYEE
WORKS_ON

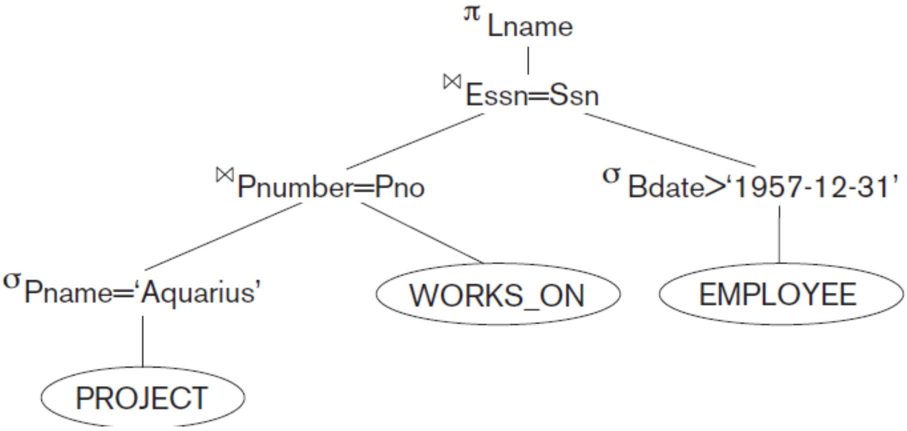
(b) Below is tree after moving SELECT operations down the query tree



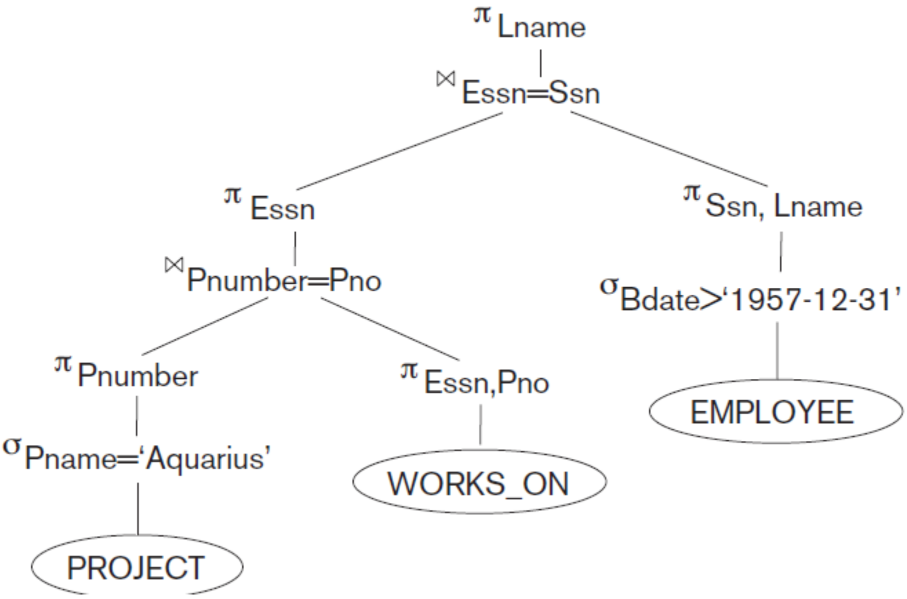
(c) Below is tree after applying the more restrictive SELECT operation first.



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



(e) May reduce tuple size of intermediate results to produce smaller records and reducing blocks to be processed. 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 $\bowtie$ (and ×)

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

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

6. Commuting σ with $\bowtie$ (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 $\bowtie$ (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 each SELECT operation as far as down the query tree as is permitted by the attributes involved in the select condition.
2. Move more restrictive selects down so that intermediate results are smaller; most restrictive condition is the one that has less *selectivity*
3. If a condition involves attributes from multiple relation, then it is likely to be a join condition. Replace Cartesian products with JOIN
4. 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

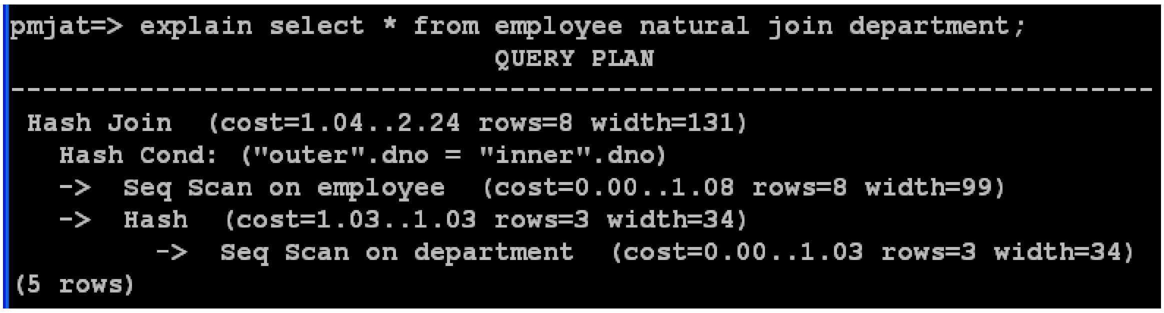
The main intuition of query optimization is order the operations such that they minimize the size intermediate results – preform selection and projection as early as possible.

EXPLAIN command of SQL

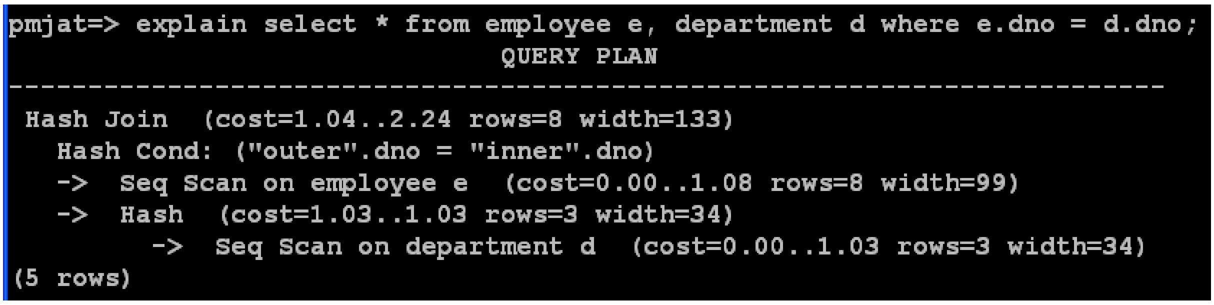
- You can get a fair idea how query planner work by observing various execution plans by using EXPLAIN command in Postgres

```
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

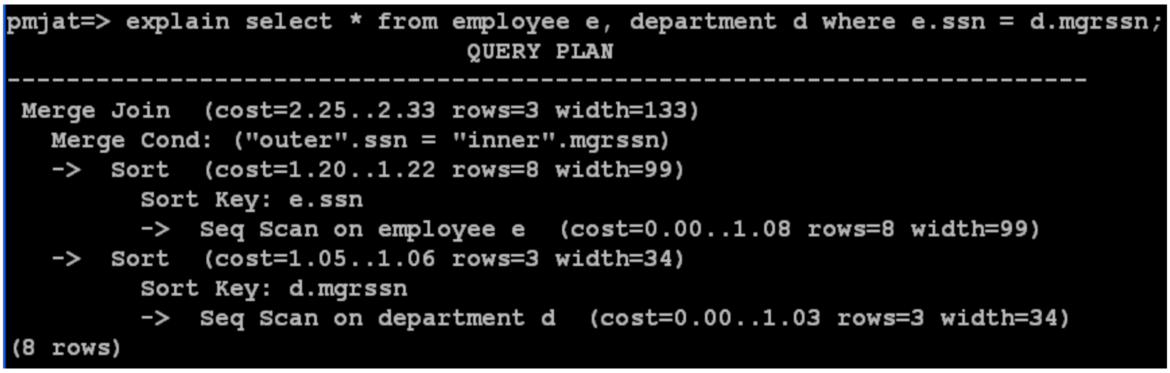


```
pmjat=> explain select * from employee natural join department;
                              QUERY PLAN
-----------------------------------------------------------------------
 Hash Join  (cost=1.04..2.24 rows=8 width=131)
   Hash Cond: ("outer".dno = "inner".dno)
   ->  Seq Scan on employee  (cost=0.00..1.08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

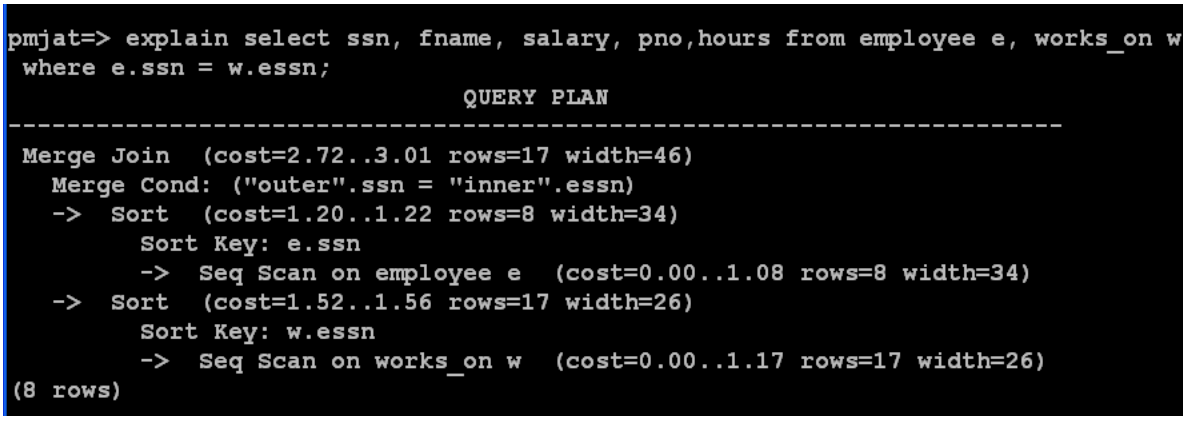


```
pmjat=> explain select * from employee e, department d where e.dno = d.dno;
                              QUERY PLAN
------------------------------------------------------------------------
 Hash Join  (cost=1.04..2.24 rows=8 width=133)
   Hash Cond: ("outer".dno = "inner".dno)
   ->  Seq Scan on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

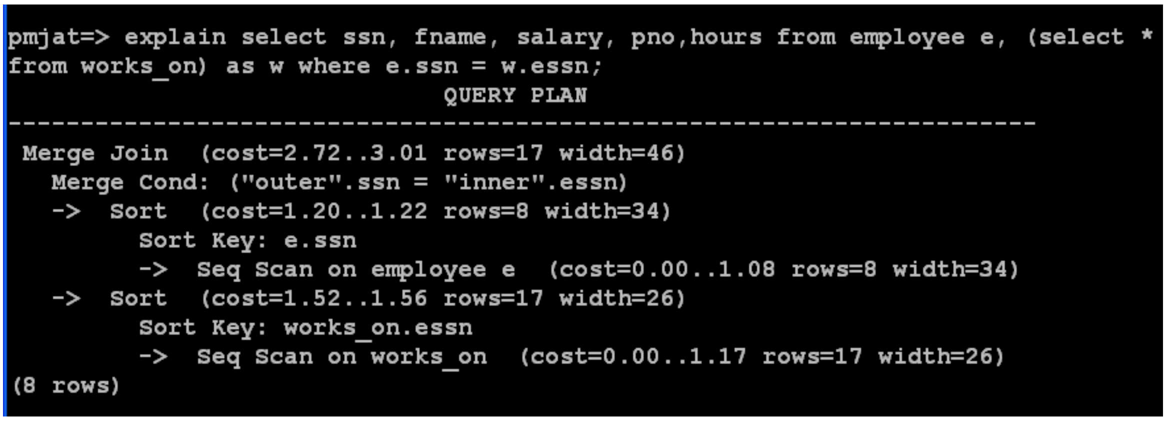
A blog https://use-the-index-luke.com/sql/explain-plan/postgresql/operations provides a basic introduction to postgresql "explain" operations.



```
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)
```