Chapter - 5

Date.

Query Processing and Optimization.

5.1 Query Cost Estimation
5.2 Query Operations
5.3 Evaluation of Expressions
5.4 Query optimization
5.5 Query Decomposition.
5.6 Performance Tuning.

Query Processing.

↳ Query processing is a translation of high-level queries into low level expression.

↳ It refers to the range of activities that are involved in extracting data from the database.

↳ It includes translation of queries in high-level database languages into expressions that can be implemented at the physical level of the file system.

↳ It is a step wise process that can be used ~~ato~~ at the physical level of the

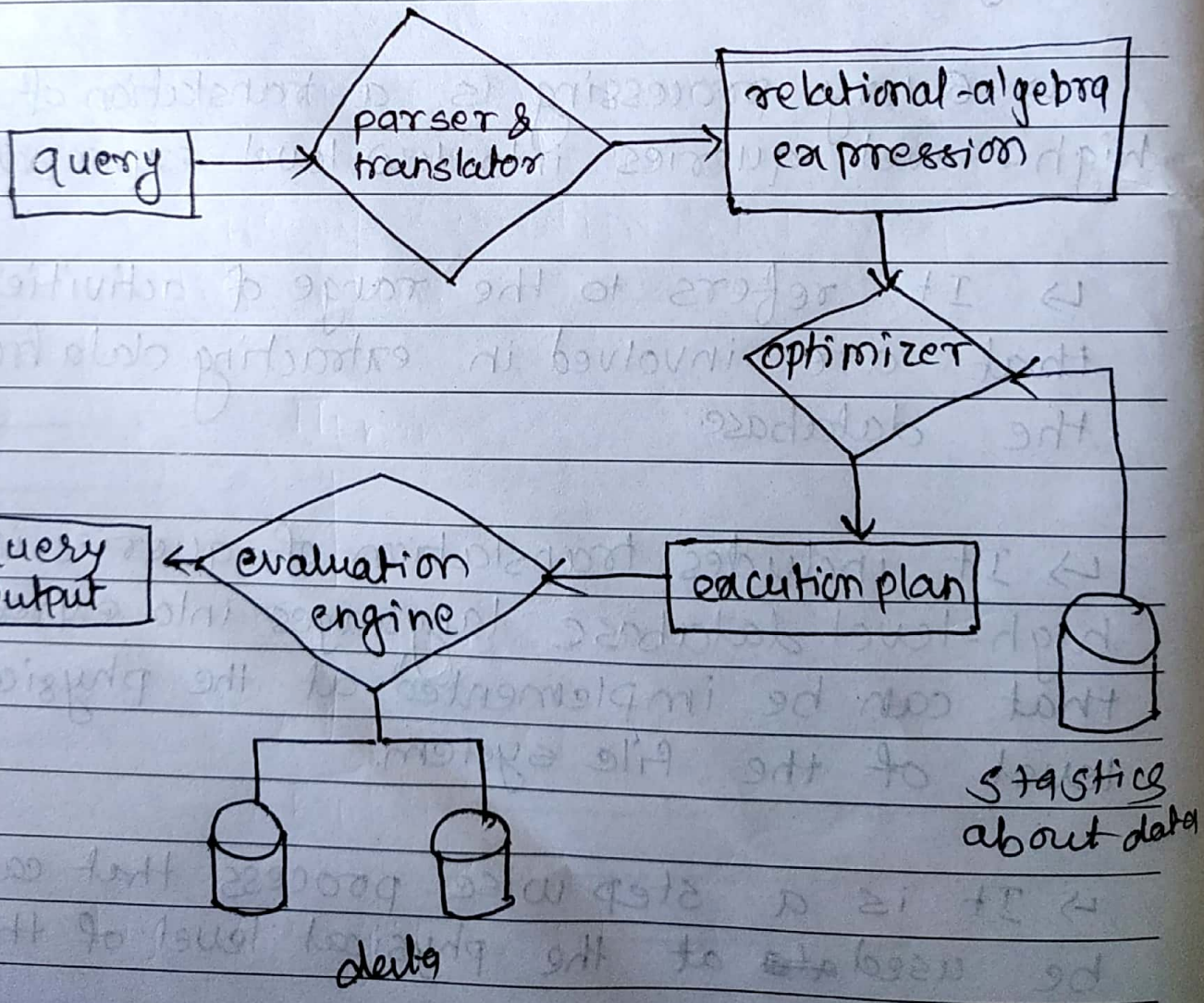
Date.

file system, query optimization and actual execution of query to get the result.

↳ It requires the basic concepts of relational algebra and file structure.

Basic steps in Query Processing.

i. Parsing and translation.
ii. Optimization.
iii. Evaluation.



Date.

Query parsing and translation (query compiler)

↳ A query is translated into SQL and into a relational algebraic expression.

↳ Parser checks the syntax and verifies the relations and the attributes which are used in the query.

↳ Different possible relational algebra expression for a single query.

Example:

```
SELECT Ename FROM Employee
WHERE salary > 5000;
```

Translated into Relational Algebra Expression

$\sigma_{salary>5000}(\Pi_{Ename}(Employee))$

OR

$\Pi_{Ename}(\sigma_{salary>5000}(Employee))$

Date.

Query optimization (query optimizer)

↳ Query optimization is the process of selecting the most efficient query-evaluation plan from among the many strategegies usually possible for processing a given query, especially if the query is complex.

↳ A relational-algebra operation annotated with instructions on how ~~much~~ to evalute it is called an evaluation primitive.

↳ A sequence of primitive operations that can be used to evaluate a query is a query execution plan or query evalution plan.

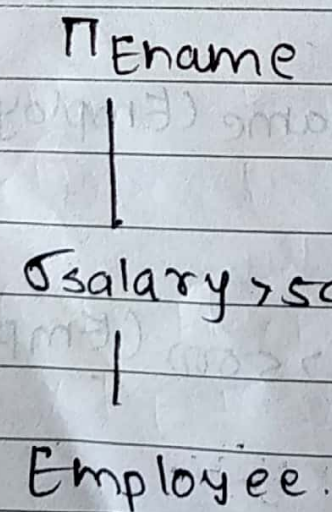


Fig: Query Execution plan.

Date.

Query evaluation (command processor)

↳ The query-execution engine takes a query-evaluation plan, executes that plan and returns the answers to the query.

↳ specify which access path to follow.
↳ Specify which algorithm to use to evaluate operator.
↳ Specify how operators interleave.

Query cost Estimation.

↳ The cost of query evalution can be measured in terms of a number of different resources, including disk accesses, CPU time to execute a query, and, in distributed or parallel database system, the cost of communication, The response time for a query-evaluation plan, assuming no other activity is going on on the computer, would account for all these costs, and could be used as a good measure of the cost of the plan.

↳ CPU cost is difficult to calculate.
↳ CPU speed is at faster rate than disk speed.
↳ Typically disk access is the predominant cost, and is also relatively easy to estimate.

↳ Disk access cost is measured by taking into account following:

i. Numbes of seeks [no. of attemps to find]
ii. Numbes of blocks read.
iii. Number of blocks write.

Cost to write is greater than cost to read a block.

For simplicity we just use the number of block transfers from disk and numbs of ~~bloc~~ seeks as the cost measures

t_T - time to transfer one block
t_S - time for one seek.

Cost of b block transfer plus S seek
$b * t_T + S * t_S$

We ighore CPU costs for simplicity
Real systems do take CPU cost

Date.

into account.

We donot include cost to writing output in our cost formulae.

↳ several algorithms can reduce disk IO by using extra buffer space.

↳ Amount of real memory available to buffer depends on other concurrent queries and os processes, known only during execution.

↳ We often use worst case estimates, assuming only the minimum amount of memory needed for the operation is available.

↳ Required data may be buffer resident already, avoiding disk I/O but hard to take into account for cost estimation.

Date.

Algorithms for RA operator.

Selection operator:

File scan

Read one file at a time sequentially

Index scan

Read file randomly.

File scan:

search algorithms that locate & retrieve records that fulfill a selection condition.

A1 (Linear search)

↳ Here each record is read from the beginning of the file till search record is reached

↳ It checks each record for filter condition one after the other.

↳ An initial seek is required to access the first block of the file.

↳ In case blocks of the file arenot stored contiguously, extra seeks maybe required, but we ignore this effect for simplicity

cost of br block transfers plus 1 seek.

cost estimate = $t_S + br * t_T$

br denotes the number of blocks in the file.

↳ If selection is on a key attribute, can stop on finding record.

Average case:

cost of (br/2) block-transfers plus 1 seek.

$t_S + (br/2) * t_T$

for worst case br blocks are required.

Index scan:

↳ Search algorithms that use an index are referred to as index scans.

↳ we use the selection predicate to guide us in the choice of index to use in processing

Date.

the query.

A2 (Primary index, equality on key)

↳ In this method search key value will be the primary key and index will be created on it.

↳ These types of indexes are usually stored in B+ tree structure with height 'h'.

↳ worst case of the query is given as:

$$(hi+1) * (tT + ts)$$

hi = height of the index.

↳ (hi+1) is given because h seek to traverse the B+ tree of height h and we need one seek to traverse the data file where the record is stored.

A3 (Primary index, equality on nonkey)

↳ We can retrive multiple records by using a primary index when the selection condition specifies an equality comparison on a nonkey attribute.

↳ cost of query becomes:

$$h_i * (t_T + t_S) + b * t_T$$

↳ seek and traversal time for B+ tree of height h to get index of the record. The we have to seek b times the data files and traverse b times all the fetched records from the data files

A4 (Secondary index, equality on nonkey)

↳ Selections specifying an equality condition can use a secondary index.

↳ This strategy can retrieve a single record if the equality condition is on key; multiple records may be retrieved if the indexing field is not a key

Date.

A3 (Primary index, equality on nonkey)

↳ We can retrive multiple records by using a primary index when the selection condition specifies an equality comparison on a nonkey attribute.

↳ cost of query becomes:

$$hi * (t_T + t_S) + b * t_T$$

↳ seek and traversal time for B+ tree of height h to get index of the record. The we have to seek b times the data files and traverse b times all the fetched records from the data files

A4 (Secondary index, equality on nonkey)

↳ Selections specifying an equality condition can use a secondary index.

↳ This strategy can retrieve a single record if the equality condition is on key; multiple records may be retrieved if the indexing field is not a key

Date.

cost when search-key is a candidate key
(hi+1) * (+T+tS)

cost when search-key is not a candidate key.
(hi+n) * (+T+tS)

where n is numbes of records fetched.

↳ can be very expensive.

(selections Involving comparisons)

can implement selections of the form $\sigma_{A \leq V}(r)$ or $\sigma_{A \geq V}(r)$ by using

i. a linear file scan
ii. By using indices

A5 (Primary index, comparison.)

↳ In this method comparison operators like <, <=, and >= are used to retrieve more than one record from the file.

Date.

↳ This comparision operator is used on the primary key, on which index is created

cost:

$$hi*(tT+ts)+b*tT$$

A6 (Secondary index, comparison)

cost: $(hi+h)*(tT+ts)$

Implementation of complex selections

Conjunction:

A conjunctive selection is a selection of the form

$$\sigma_{\theta 1 \wedge \theta 2 \wedge \ldots \wedge \theta n}(r)$$

A7 (Conjuctive selection using one index)

↳ We first determine whether an access path is available for an attribute in one of the simple conditions

Date.

↳ if one is, one of the selection algorithms A2 through A6 can retrieve records satisfying that condition.

↳ The cost of algorithm A7 is given by the cost of chosen algorithm.

A8 (conjuctive selection using composite index)

↳ A appropriate composite index (i.e index on multiple attributes) may be available for some conjuctive selections

↳ The type of index determines which of algorithms A2, A3 or A4 will be used.

A9 (conjuctive selection by intersection of identifiers)

↳ Conjuctive selection operation involves the use of record pointers, ~~on the~~ & record identifiers

↳ This algorithm requires indices with

Date.

record pointers on the fields involved in individual conditions.

↳ The algorithm scans index for pointers to tuples that satisfy individual condition.

↳ The intersection of all retrieved pointers is the set of pointers to tuples that satisfy the conjuctive condition.

↳ The algorithm then uses the pointers to retrieve actual records

Disjunction:

A disjunctive selection is a selection of the form

$$\sigma \theta_1 \wedge \theta_2 \wedge \cdots \theta_n(r)$$

Algo (Disjunctive selection by union of identifiers)

If access paths are available on all the conditions of a disjunctive selection, each index is scanned for pointers to tuples that satisfy the individual condition.

Date.

↳ The union of all the retrieved pointers yields the set of pointers to all tuples that satisfy disjunctive condition.

↳ We then use the pointers to retrieve the actual records

Negation:

↳ The result of a selection $\sigma_{\neg\theta}(r)$ is the set of tuples of r for which that condition θ evaluates to false.

↳ In the absence of nulls, this set is simply the set of tuples r that are not in $\sigma_{\theta}(r)$

Sorting

↳ We can sort a relation by building an index on the sort key, and then using that index to read the relation in sorted order.

↳ Such process orders that relation only logically, through an index, rather

than physically

↳ For relations that fit in memory that is records are completely in main memory techniques like quicksort can be used For relations that don't fit in memory, external sort-merge is a good choice.

↳ External sort-merge, which ~~co~~ cumulatively sorts multiple runs of the data based on amount that fits in memory at one time.

Join operation:

Different algorithms to implement joins.

i. Nested-loop join. ✓
ii. Block nested-loop join.
iii. Indexed nested-loop join
iv. Merge-join ✓
v. Hash-join ✓

Choice is based on cost estimate.

Date.

Nested loop join.

↳ Simple algorithm to compute the theta join $r \bowtie_\theta s$, of two relations r and s. This algorithm is called nested-loop join algorithm, since it basically consists of a pair of nested for loops.

```
for each tuple tr in r do begin
    for each tuple ts in s do begin
        test pair (tr, ts) to see if they
        satisfy the join condition θ
        if they do, add tr.ts to the result.
    end
end
```

↳ r is called the outer relation & s the inner relation of the join.

↳ Requires no indices and can be used with any kind of join condition.

↳ Expensive since it examines every pair of tuples in two relations.

↳ In the worst case, if there is enough

memory only to hold one block of each relation, the estimated cost is.

$n_r * b_s + b_r$, block transfers plus $n_r + b_r$ seeks

Where n_r denotes the number of tuples in r

b_r and b_s denotes number of blocks containing tuples r and s respectively.

If the smaller relation fits entirely in memory, use that as the inner relation.

cost: $b_r + b_s$ block transfers and 2 seeks.

Example:

Number of records of

$n_{student} = 5,000$ $n_{takes} = 10000$

Number of blocks of:

Date.

$b_{student} = 100$ $b_{takes} = 400$

worst case: (with student as outer relation)

$5000 * 400 + 100 = 2,000,100$ block transfers

$5000 + 100 = 5100$ seeks.

with takes as outer relation

$1000 * 100 + 400 = 1,000,400$ block transfers

10,400 seeks.

If student fits entirely in memory, the cost estimate will be 500 block transfers.

Block Nested-loop join

Varient of nested loop join in which every block of inner relation is paired with every block of outer relation.

```
for each block Br of r do begin
    for each block Bs of s do begin
        for each tuple tr in Br do begin
            for each tuple ts in Bs do begin
```

check if (tr, ts) satisfy the join condition
if they do, add tr.ts to the result.
end
end
end
end
end

worst case estimate
br * bs + br block transfers plus 2 * br seeks.

Best case br + bs block transfers plus 2 seeks.

Merge-Join:

↳ sort both relations on their join attributes (if not already sorted on the join attributes)

↳ The Merge Join operator is one of four operators that join data from two input streams into a single combined output stream.

Date.

↳ It has two inputs, called the left and right input.

↳ Merge join can't be used without adding extra sort operators. These extra sort increase the total plan cost.

↳ The Merge join operator supports all ten logical join operations: inner join; left, right and full outer join; left and right semi and anti semi join; as well a concatenation & union.

↳ Algorithm requires at least one equality based join predicate.

↳ can be used only for equi-joins and natural joins.

Cost of merge join is:

$b_r + b_s$ block transfers + $\lceil b_r/b_b \rceil + \lceil b_s/b_b \rceil$ seeks.

+ The cost of sorting if relations are unsorted.

Date.

hybrid merge-join

↳ If one relation is sorted, and the other has a secondary B+-tree index on the join attribute.

↳ Merge the sorted relation with the leaf entries of the B+-tree.

↳ sort the result on the addresses of the unsorted relation's tuples.

↳ scan the unsorted relation in physical address order and merge with previous result, to replace address by the actual tuples.

↳ Sequential scan more efficient than random lookup.

Hash join:

↳ A hash function h is used to partition tuples of both relations. The basic idea is to partition the tuples of each of the relations into sets that have same hash value on the join attributes.

Date.

we assume that:

↳ h is a hash function mapping joinAttrs values to $\{0, 1, \ldots, n_h\}$, where JoinAttrs denotes

↳ $r_0, r_1, \ldots, r_n$ denote partitions of r tuples; each initially empty. Each tuple $t_r \in r$ is put in partition r_i where $i = h(t_r[\text{joinAttrs}])$.

↳ $r_0, r_1, \ldots r_n$ denotes partitions s_i, where $i = h(t_s[\text{joinAttrs}])$.

↳ suppose that an r tuple and ~~an~~ s tuple satisfy the join condition; then, they will have the same value for the join attributes. If that value is hashed to some value i, the r tuple has to be in r_i and s tuples in s_i.

↳ r tuples in r_i need only to be compared with s tuples in s_i Need not be compared with s tuples in any other partition.

Date.

↳ If the value of n_h is greater than or equal to the number of page frames of memory, the partition cannot be done in one pass. Since these will not be enough buffer pages. Instead partitioning has to be done in repeated passes.

↳ If recursive partitioning is not required cost of hash join is

$3(b_r + b_s) + 4 * n_h$ block transfers +
$2(\lceil b_r/b_b \rceil + \lceil b_s/b_b \rceil)$ seeks.

↳ if recursive partitioning required.

Total cost estimate

$2(b_r + b_s)\lceil \log_{\lceil M/b_b \rceil - 1}(b/M) \rceil + b_r + b_s$ block transfers + $2(\lceil b_r/b_b \rceil + \lceil b_s/b_b \rceil)$ $\log_{\lceil M b_b \rceil - 1}(b_s/M)$ seeks.

↳ If the entire build input can be kept in main memory no partitioning is required. Cost estimate goes down to

$b_r + b_s$

Date.

Example:

instructor ⋈ teaches

Assume that memory size is 20 blocks

$b_{instructor} = 100$ and $b_{teaches} = 400$

instructor is to be used as build input. Partition it into five partitions, each of size 20 blocks. This partitioning can be done in one pass.

Similarly, partition teaches into five partitions each of size 80. This is also done in one pass.

Therefore total cost, ignoring cost of writing partially filled blocks

$3(100 + 400) = 1500$ block transfers +
$2(\lceil 100/3 \rceil + \lceil 400/3 \rceil) = 336$ seeks

Evaluation of Expressions:

↳ In reality expressions cannot exist as a single operation. instead a typical query combines them.

For example: select customes_name from account natural join customer whese balance < 3000

This example consists of a selection, a natural join and a projection.

So Alternatives for evaluating an entire expression tree:

i. Materialization
ii. Pipelining.

Materialization:

↳ In materialization an expression is evaluated one opesation at a time in an appropriate order.

↳ The result of each evaluation is

Date.

materialized in a temporary relation for subsequent use.

↳ Evaluation starts with one operation at a time, at the lowest-level. Use intermediate results materialized into temporary relations to evaluate next-level operations.

Consider the expressions

i. $\Pi_{customer\_name}(\sigma_{balance<2500}(account) \bowtie customer)$

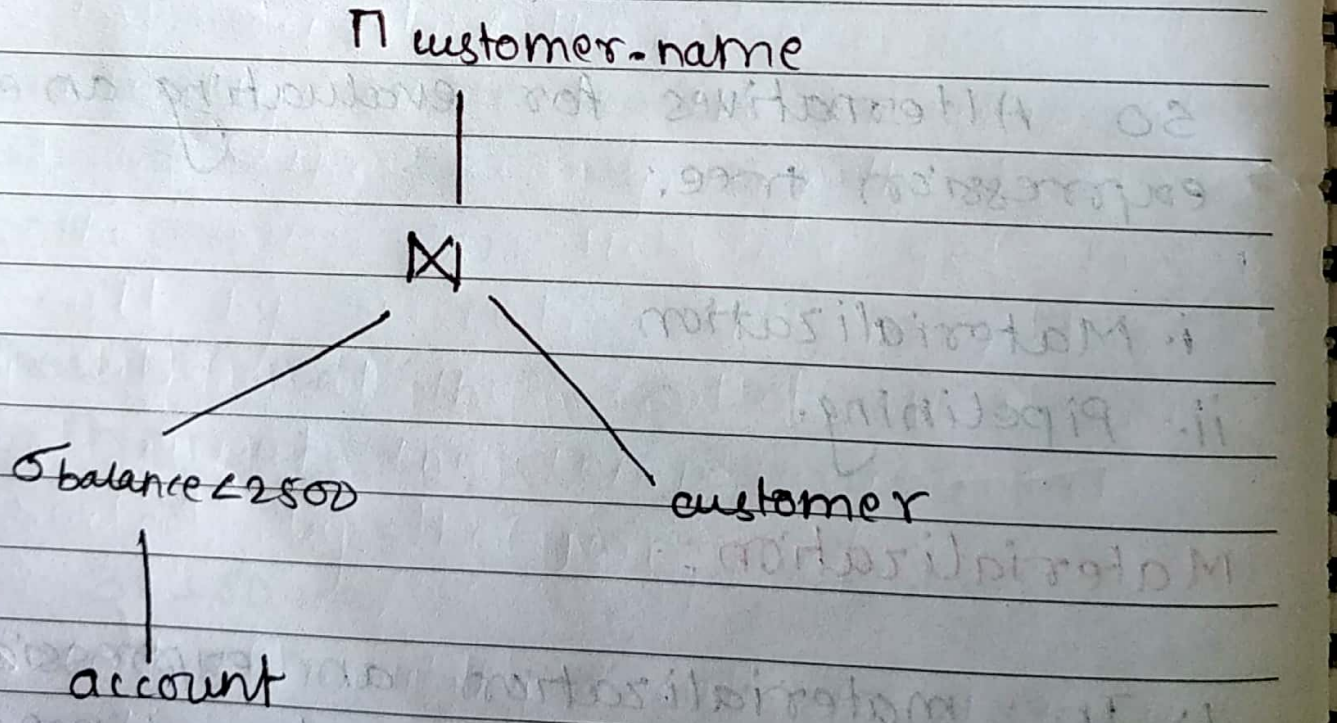
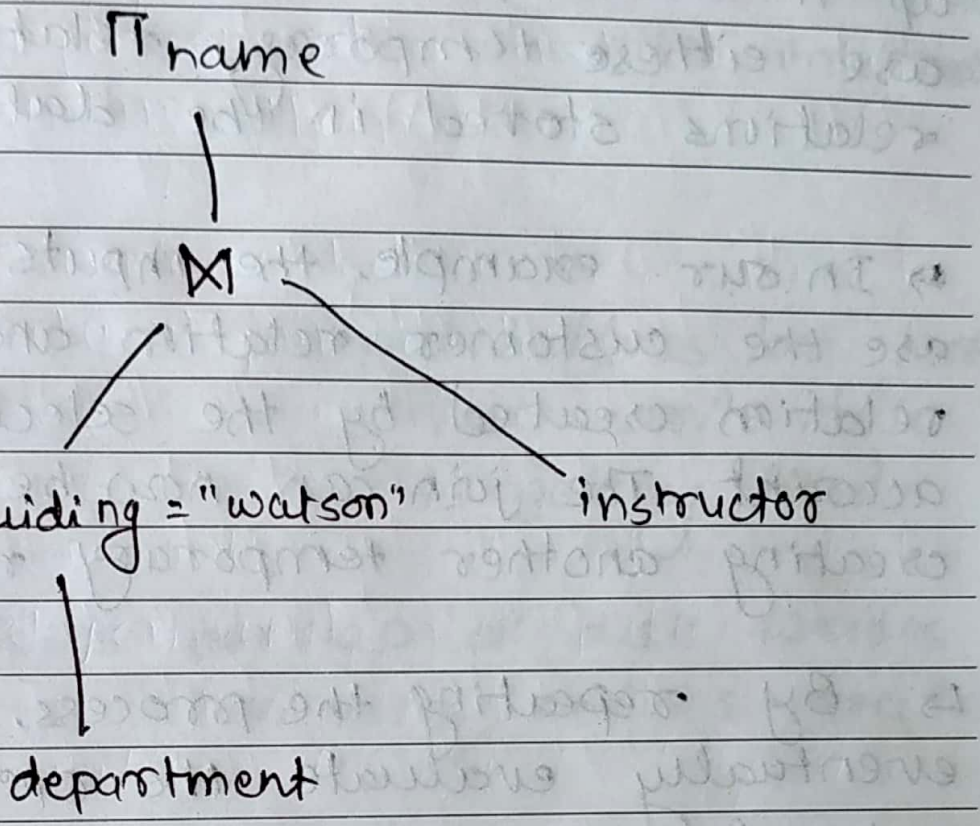


Fig: Pictorial representation of an expression.

ii. $\sigma_{building = "watson"}(department) \bowtie instructor$



↳ Materialized evaluation is always applicable.

In the example 1, we start from the lowest-level operations in the expressions (at the bottom of the tree). Here only one such operation exists; the selection operation on account. The inputs to the lowest-level operations are relations in the database. We execute these operations by using suitable algorithm, we store the results in temporary relations.

~~We~~

Date.

we can use these temporary relations to execute the operations at the next level up in the tree, where the inputs ~~are~~ now are either temporary relations or relations stored in the database.

↳ In our example, the inputs to the join are the customer relation and temporary relation created by the selection on account. The join can now be evaluated creating another temporary relation.

↳ By repeating the process, we will eventually evaluate the operation at the root of the tree giving the final result of the expression. Here we get final result by executing the projection operation at the root of the tree, using as input the temporary relation created by the join.

↳ Cost of writing results to disk and reading them back can be quite high.

↳ Our cost formulas for operations

ignore cost of writing results to disk, so:

Overall cost = sum of costs of individual operations + cost of writing intermediate results to disk.

Double buffering:

↳ use two output buffers for each operation, when one is full write it to disk while the other is getting filled.

↳ Allows overlap of disk writes with computation and reduces execution time.

Pipelining:

↳ We can improve query-evaluation efficiency by reducing the number of temporary files that are produced.

↳ We achieve this reduction by combining several relational operations into a pipeline of operations, in which the result of one operation are passed along to the next operation in the

Date.

pipeline. Evaluation as just described is called pipelined evaluation.

↳ For example: Consider the expression $(\Pi_{a1, a2}(r \bowtie s))$

If materialization were applied, evaluation would involve creating temporary relation to hold the result of the join, and then back in the result to perform the projection. These operations can be combined: when the join operation generates a tuple of its result, it passes that tuple immediately to the project operation for processing. By combining the join and the projection, we avoid creating the intermediate result and instead create the final result directly.

↳ Pipelining is much cheaper than materialization: no need to store a temporary relation to disk.

↳ Pipeling may not always be possible e.g., sort, hash-join.

Date.

For pipeling to be effective, use evaluation algorithms that generate output tuples even as tuples are received for inputs to the operation.

↳ Pipelines can be executed in either of two ways:

i. Demand driven.
ii. Producer driven.

Demand driven:

In demand driven or lazy evaluation.

↳ the system makes repeated requests for tuples from the operation at the top of the pipeline.

↳ Each time that an operation receives a request for tuples, it computes the next tuple (or tuples) to be returned, and then returns that tuple.

↳ In between calls, operation has to maintain "state" so it knows what to return next.

Date.

Producer-driven or eager pipeling:

In producer driven or eager pipeling

↳ operations donot wait for requests to produce tuples, but instead genesate the tuples eagesly.

↳ Each operation at the bottom of a pipeline continually generates output tuples, and puts them in its output buffer until the buffer is full.

↳ An operation at any other level of a pipeline genesates output tuples when it gets input tuples from lowes down in the pipeline until its output buffer is full.

↳ Once the operation uses a tuple from a pipelinined input, it removes the tuple from its input buffer.

↳ The operation repeats this process until all the output tuples have been generated.

Alternative name: pull and push models of pipelining.

Transformation of Relational Expressions.

↳ A query can be expressed in several different ways, with different costs of evaluation

↳ Rather than taking the relational expression as given, we consider alternative, equivalent expressions.

↳ Two-relational-algebra expressions are said to be equivalent if, on every legal database instance, the two expressions generate the same set of tuples.

↳ In SQL, inputs and outputs are multisets of tuples, two expressions in the multiset version of the relational algebra are said to be equivalent if on every legal database instance the two expressions generate the same multiset of tuples.

↳ An equivalence rule says that expressions of two forms are equivalent; can replace

Date.

expression of first form by second, or vice versa.

↳ Generation of query-evaluation plans for an expression involves several steps:

i. Generating logically equivalent expressions
ii. Use equivalence rules to transform an expression into an equivalent one.
iii. Annontating resultant expressions to get alternative query plans.
iv. Chosing the cheapest plan based on estimated cost.

↳ The overall process is called cost based optimization.

↳ Equivalence Rules:

An equivalence rule says that expressions of two forms are equivalent.

1. Conjunctive selection operations can be deconstructed into a sequence of individual selections.

$$\sigma_{\theta_1 \wedge \theta_2}(E) = \sigma_{\theta_1}(\sigma_{\theta_2}(E))$$

2. Selection operations are commutative.

$$\sigma_{\theta_1}(\sigma_{\theta_2}(E)) = \sigma_{\theta_2}(\sigma_{\theta_1}(E))$$

3. Only the last in a sequence of projection operations is needed, the others can be omitted.

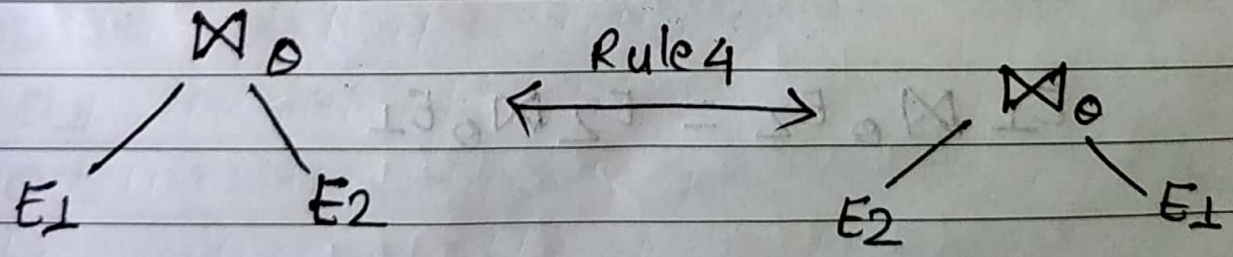
$$\Pi_{t_1}(\Pi_{t_2}(\ldots(\Pi_{t_n}(E))\ldots)) = \Pi_{t_1}(E)$$

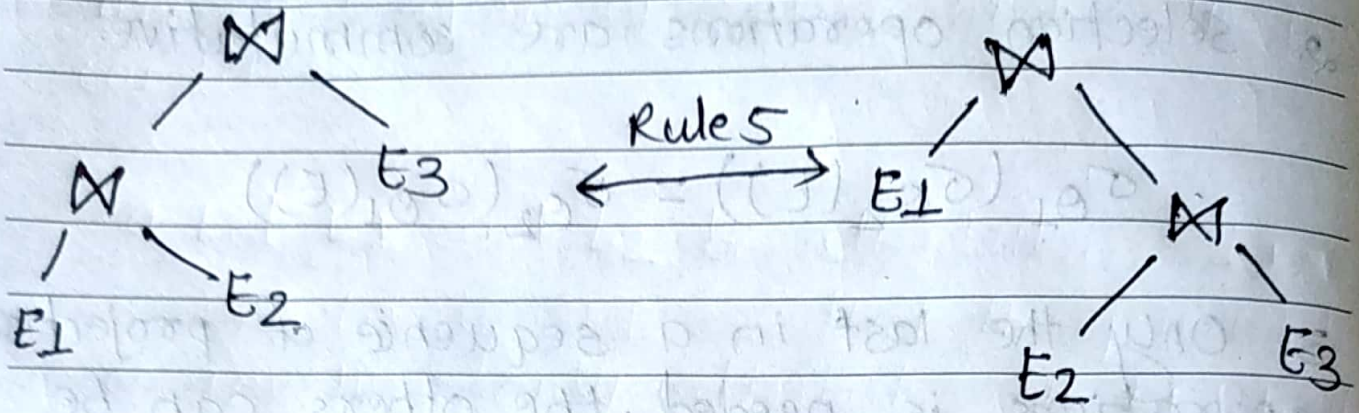
4. selections can be combined with cartesian products and theta joins.

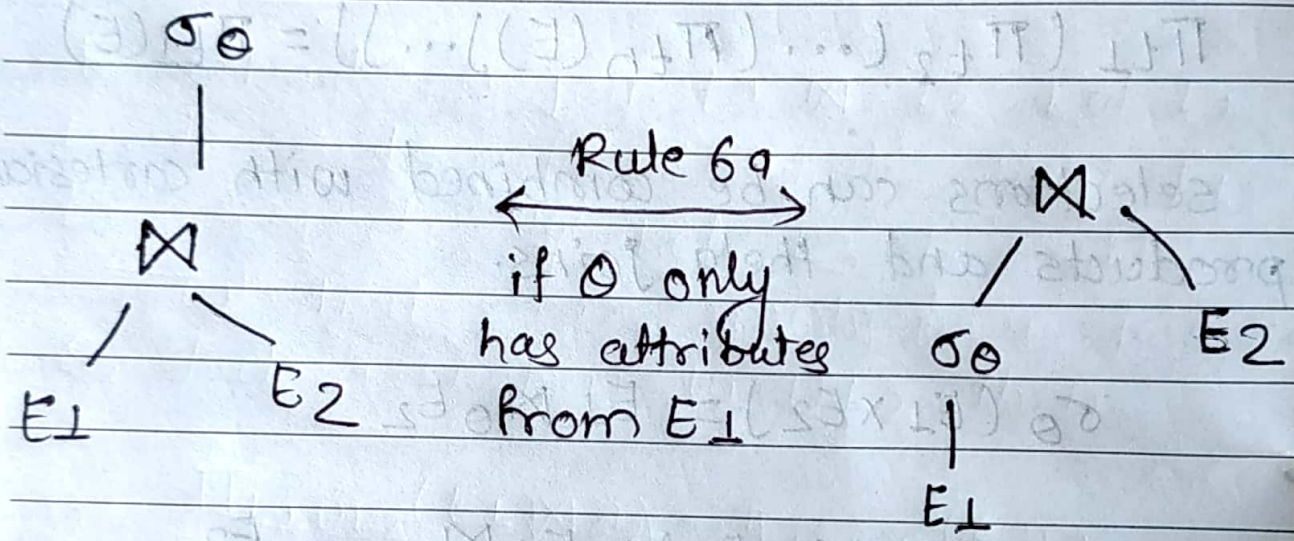
$$\sigma_{\theta}(E_1 \times E_2) = E_1 \bowtie_{\theta} E_2$$

$$\sigma_{\theta_1}(E_1 \bowtie_{\theta_2} E_2) = E_1 \bowtie_{\theta_1 \wedge \theta_2} E_2$$

Pictorial Depiction of Equivalence Rules:







5. Theta-join operations (and natural joins) are commutative.

$$E_1 \bowtie_\theta E_2 = E_2 \bowtie_\theta E_1$$

6 a. Natural joins ~~are~~ operations are associative

$$(E_1 \bowtie E_2) \bowtie E_3 = E_1 \bowtie (E_2 \bowtie E_3)$$

Date.

b. Theta joins are associative in the following manner:

$$(E_1 \bowtie_{\theta_1} E_2) \bowtie_{\theta_2 \wedge \theta_3} E_3 = E_1 \bowtie_{\theta_1 \wedge \theta_3} (E_2 \bowtie_{\theta_2} E_3)$$

corrected

where θ_2 involves attributes from only E_2 and E_3.

7. The selection operation distributes over the theta-join operation under the following two conditions:

a. It distributes when all the attributes in selection condition θ_0 involve only the attributes of one of the expressions (say E_1) being joined.

$$\sigma_{\theta_0}(E_1 \bowtie_{\theta} E_2) = (\sigma_{\theta_0}(E_1)) \bowtie_{\theta} E_2$$

b. It distributes when selection condition θ_1 involves only the attributes of E_1 & θ_2 involves only attributes of E_2

$$\sigma_{\theta_1 \wedge \theta_2}(E_1 \bowtie_{\theta} E_2) = (\sigma_{\theta_1}(E_1)) \bowtie_{\theta} (\sigma_{\theta_2}(E_2))$$

8. The projection operation distributes over

Date.

the theta-join operation under the following conditions.

a. Let L_1 and L_2 be attributes of E_1 & E_2 respectively. suppose that the join condition θ involves only attributes in $L_1 \cup L_2$. Then

$$\Pi_{L_1 \cup L_2}(E_1 \bowtie_\theta E_2) = (\Pi_{L_1}(E_1)) \bowtie_\theta (\Pi_{L_2}(E_2))$$

b. Consider a join $E_1 \bowtie_\theta E_2$. Let L_1 and L_2 be sets of attributes from E_1 and E_2 respectively. Let L_3 be attributes of E_1 that are involved in join condition θ, but arenot in $L_1 \cup L_2$. Then

$$\Pi_{L_1 \cup L_2}(E_1 \bowtie_\theta E_2) = \Pi_{L_1 \cup L_2}((\Pi_{L_1 \cup L_3}(E_1)) \bowtie_\theta (\Pi_{L_2 \cup L_4}(E_2)))$$

g. The set operations union and intersection are commutative.

$$E_1 \cup E_2 = E_2 \cup E_1$$

$$E_1 \cap E_2 = E_2 \cap E_1$$

set difference is not commutative.

Date.

10. Set Union and intersection are associative

$$(E_1 \cup E_2) \cup E_3 = E_1 \cup (E_2 \cup E_3)$$

$$(E_1 \cap E_2) \cap E_3 = E_1 \cap (E_2 \cap E_3)$$

11. The selection operation distributes over the union, intersection, and set-difference operations.

$$\sigma_p(E_1 - E_2) = \sigma_p(E_1) - \sigma_p(E_2)$$

Similarly, the preceding equivalence, with $-$ replaced with either $\cup$ or $\cap$ also holds. Further:

$$\sigma_p(E_1 - E_2) = \sigma_p(E_1) - E_2$$

The preceding equivalence with $-$ replaced by $\cap$ also holds, but does not hold if $-$ is replaced by $\cup$.

12. The projection operation distributes over the union operation.

$$\Pi_L(E_1 \cup E_2) = (\Pi_L(E_1)) \cup (\Pi_L(E_2))$$

Date.

Transformation Example:

Query: Find the names of all customers who have an account at some branch located in kathmandu.

$\Pi_{customer\text{-}name}(\sigma_{branch\text{-}city = \text{"kathmandu"}}(branch \bowtie account \bowtie depositor))$

Transformation using rule 7a.

$\Pi_{customer\text{-}name}((\sigma_{branch\text{-}city = \text{"kathmandu"}}(branch)) \bowtie account \bowtie depositer)$

performing the selection as early as possible reduces the size of the relation to be joined.

Query: Find the names of all customers with an account at kathmandy branch whose account balance is over 10 000

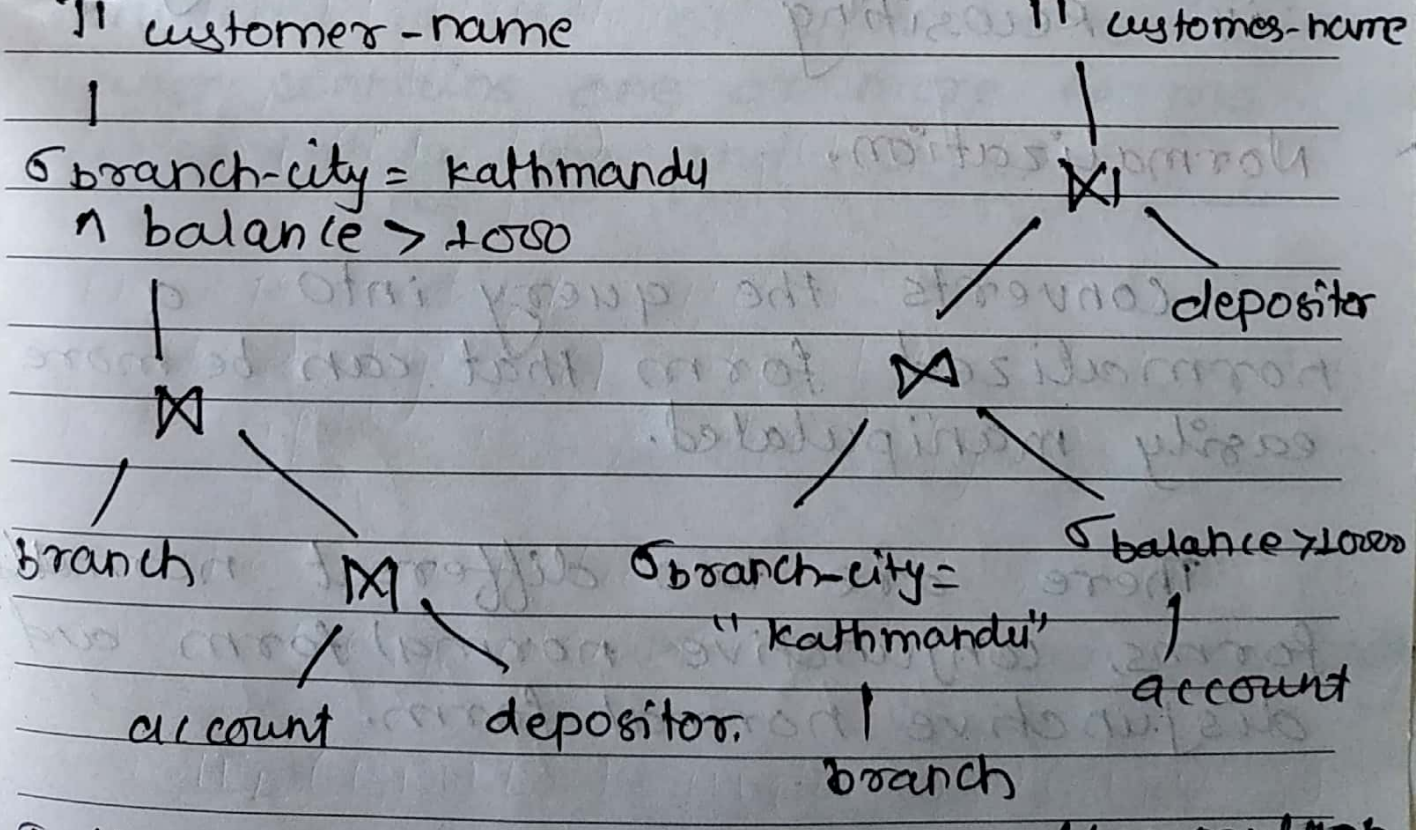
$\Pi_{customes\text{-}name}(\sigma_{branch\text{-}city = \text{"kathmandu"} \wedge balance > 10000}(branch \bowtie account \bowtie depositor))$

Transformation using join associatively (Rule 6a)

$\Pi_{customer\_name}((\sigma_{branch\_city = "kathmandu" \wedge balance > 1000}(branch \bowtie account)) \bowtie depositor)$

second form provides an opportunity to apply the "perform selections early" rule, resulting in the subexpression.

$\sigma_{branch\_city = "kathmandu"}(branch) \bowtie \sigma_{balance > 1000}(account)$



Initial Expression Tree

Tree after multiple Transformations

Date.

Query Decomposition.

The Query decomposition is the first phase of query processing whose aims are to transfer a high-level query into a relational algebra query & to check whether that query is syntactically and semantically correct.

Stages of Query Decomposition.

i. Normalization
ii. Analysis
iii. Elimination of Redundancy
iv. Rewriting

Normalization.

Converts the query into a normalized form that can be more easily manipulated.

There are two different normal forms, conjunctive normal form and disjunctive normal form.

Date.

Conjunctive normal form.

A sequence of conjuncts that are connected with the and operator. Each conjunct contains one or more terms connected by the or operator.

for example:

(position = 'Manager' V salary > 20000) ∧ branch NO = 'B003'.

Disjunctive normal form.

A sequnce of disjuncts that are connected by the or operator. Each disjunct contains one or more terms connected by the and operator.

for example:

(position = 'Manager' ∧ branchNO = 'B003') V (salary ≥ 20000 ∧ branchNO = 'B003')

Analysis:

The query is lexically and syntactically analyzed using the techniques of programming language compilers

Date.

↳ verifies that the relations and attributes specified in the query are defined in the system catalog.

↳ verifies that any operations applied to database objects are appropriate for the object type.

↳ On completion of the analysis, the high-level query has been transformed into some internal representation (query tree) that is more suitable for processing.

Root
↑
intermediate operation
↑
Leaves.

Simplification.

TO detect redundant qualifications eliminate common subexpressions, and

transform the query to a semantically equivalent but more easily and efficiently computed form.

↳ Access restrictions, view definitions, & integrity constraints are considered at this stage.

↳ Expression replacement already used in views.

Query restructuring / rewriting.

↳ The final stage of query decomposition.

↳ The query is restructured to provide a more efficient implementation.

↳ Operator Tree is used.

Leaf nodes are operand relations

Non leaf are intesmediate tables produced as a result of some relational operators.

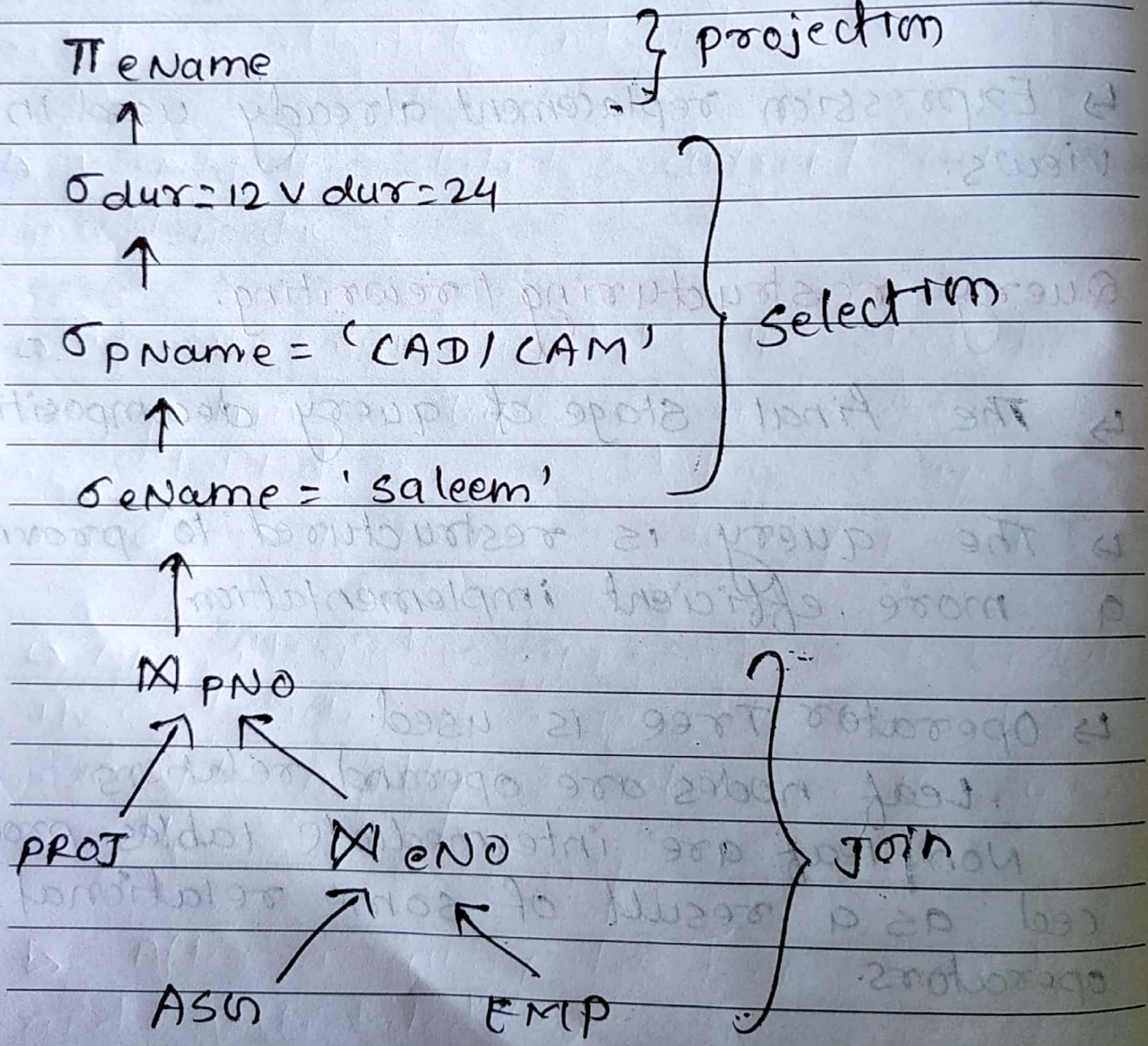
Example:

Select eName FROM EMP, ASG, PROJ

Where

Date.

EMP.eN = ASG.eNO
AND ASG.pNO = PROJ.pNO
AND eName ≠ 'saleem'
AND pName = 'CAD/CAM'
AND (dur = 36 or dur = 24)



Date.

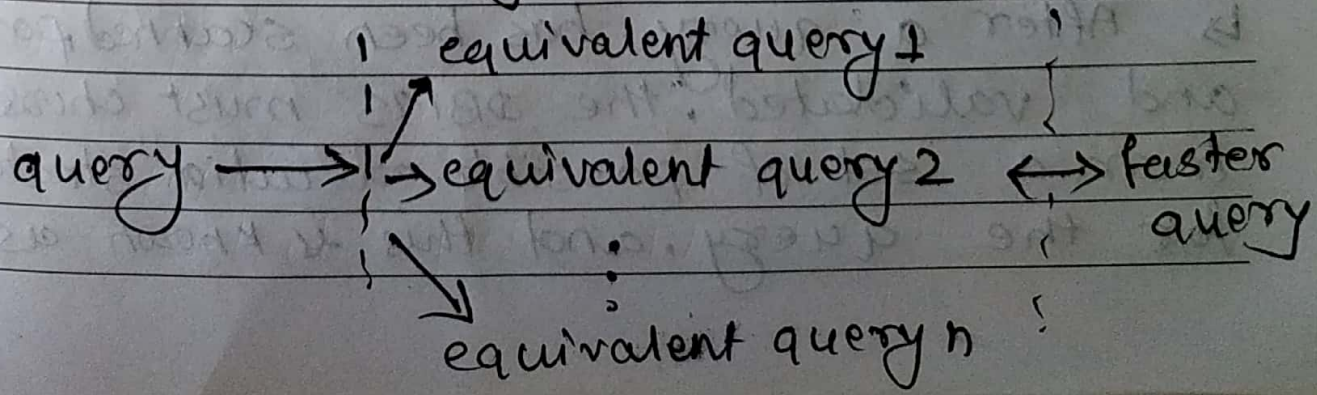
Query optimization.

↳ Query optimization is the process of selecting the most efficient query-evaluation plan from among many strategies usually possible for processing a given query, especially query is complex.

↳ We donot expect users to write their queries so that they can be processed efficiently. Rather, we expect the system to construct a query evaluation plan that minimizes the cost of query evaluation. This is where query optimization comes into play.

↳ An important aspect of query processing is query optimization.

↳ The aim of query optimization is to choose the one that minimizes resource usage.



Date.

Aim: transfer query into faster, equivalent query.

↳ Every method of query optimization depend on database statics.

↳ The statistics cover information about relations, attribute, and indexes.

↳ Keeping the statistics current can be problematic.

↳ If the DBMS updates the statistics every time a tuple is inserted, updated, or deleted, this would have a significant impact on performance during peak period.

↳ An alternative approach is to update the statistics on a periodic basis, for example nightly or whenever the system is idle.

↳ After a query has been scanned, parsed and validated, the DBMS must choose from a set of possible execution strategies for the query, and this is known as

query optimization.

Types:

i. Heuristic (Logical) query optimization.
ii. cost-based (Physical) query optimization.

Cost Based optimization.

↳ A cost based optimization explores the space of all query-evaluation plans that are equivalent to the given query, and chooses the one with the least estimated cost.

↳ Cost of physical plans include processor time and communication time. The most important factor to consider is disk IOs because it is the most time consuming action.

↳ some other costs associated are:
operations (joins, unions, intersections).

Steps in cost-based query optimization.

i. Generate logically equivalent expressions using equivalence rules.

Date.

ii. Annonate resultant expressions to get alternative query plans.

iii. Choose the cheapest plan based on estimated cost.

Estimation of plan cost based on:

↳ statical information about relations. Examples: number of tuples, number of distinct value for an attribute.

↳ statics estimation for intermediate results to compute cost of complex expressions

↳ cost formula for algorithms, computed using statistics.

Consider finding the best join order for $r_1 \bowtie r_2 \ldots r_n$.

There are $(2(n-1)!/(n-1)!)$ different join orders for above expression. With $n=7$, the number is 665280, with $n=10$, the number is greater than 176 billion!

↳ No need to generate all the join orders. Using dynamic programming, the least-cost join order for any subset of $\{r_1, r_2, \ldots r_n\}$ is computed only once and stored for future use.

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

Dynamic Programming in Optimization

To find best join tree for a set of n relations.

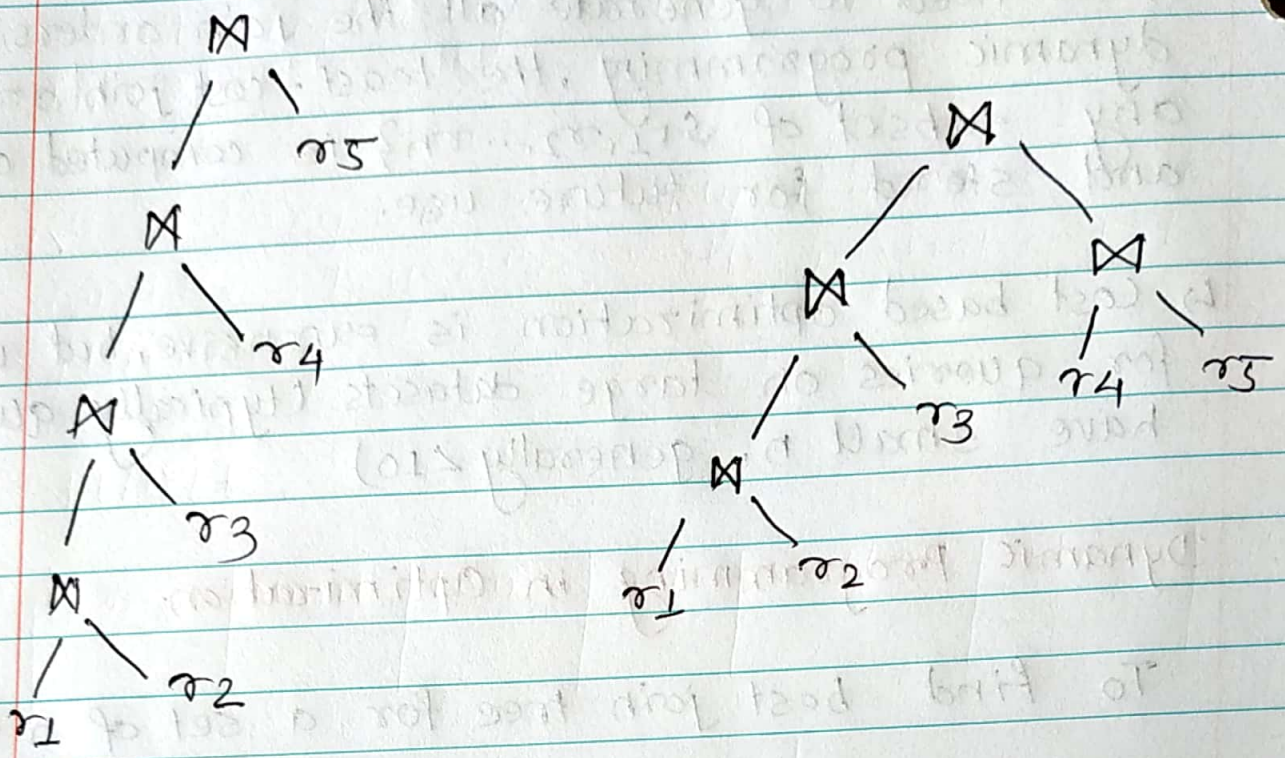
↳ To find best plan for a set S of n relations, consider all possible plans of the form: $S_1 \bowtie (S - S_1)$ where S_1 is any non-empty subset of S.

↳ Recursively compute costs for joining subsets of S to find the cost of each plan. Choose the cheapest of the $2^n - 1$ alternatives.

↳ When plan for any subset is computed, store it and reuse it when it is required again, instead of recomputing it.

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.



Left-deep join tree. Non-left-deep join tree.

Cost of optimization.

With dynamic programming time complexity with bushy tree is $O(3^n)$

with $n=10$, this number is 59000 instead of 176 billion.

Space complexity is $O(2^n)$

To find best left-deep join tree for a set of n relations.

↳ consider n alternatives with one relation as right hand side input and other relations

as left hand side input.

↳ Using (recursively computed and stored) least-cost join order for each alternative on left-hand side, choose the cheapest of the n alternatives.

If only left-deep trees are considered, time complexity of finding best join order is $O(n2^n)$

space complexity remains at $O(2^n)$

Heuristic optimization.

↳ Heuristic optimization transforms the query-tree by using a set of rules (Heristics) that typically (but not in all cases) improve execution performance

i. Perform selection early (reduces the number of tuples)
ii. Perform projection early (reduces the number of attributes)
iii. Perform most restrictive selection & join operations (i.e. with smallest result size) before ~~the~~ other similar operations.

↳ Generate initial query tree from SQL statement & a single query tree is involved.

↳ Transform query tree into more efficient query tree, via series of tree modifications, each of

which hopefully reduces the execution time.

↳ some system use only heuristics; others combine heuristics with partial cost based optimization.

PLAYER

Fname	Minit	Lname	SSN	Draft Date	Address	Salary

coach_ssn	Team Name

UNIT

Uname	Unumber	coach_ssn

PLAY

PlayName	PlayNum	PlayType	Unum

WORKS_ON

PSSN	PlayNO	Hours

Query:

Find the last names of players who were drafted after 2000 who have worked on a play called 'halfback-pass'.

SQL:

```
SELECT Lname
FROM PLAYER, WORKS_ON, PLAY
WHERE PlayName = 'halfback_pass' AND
      PlayNO = PLayNum AND Pssn = Ssn AND
          DraftDate > '2000-12-31'
```

PROJECT (Lname)

|

SELECT (PlayName = 'halfback-pass' AND playNO = PlayNum AND Pssn = Ssn AND DraftDate > '2000-12-31'

|

X

X

Play

PLAYER

WORKS-ON

↳ It should work, but it is not optimal

↳ We have to find the cartesian product of the PLAYER, WORKS_ON, and PLAY files. Depending on the size and number of records in each table, cartesian product could be huge and would contain lots of unneeded data.

↳ First, move SELECT operation down the query tree.

↳ Second, perform the more restrictive SELECT operations first.

↳ Third replace CARTESIAN PRODUCT and SELECT combination with join operations.

↳ Finally, move PROJECT operations down the query tree.

↳ This is called heuristic optimization.

Performance Tuning:

Tuning the performance of a system involves adjusting parameters and design choices to improve its performance for a specific application.

↳ Various aspects of a database-system design- ranging from high level aspects such as the schema and transaction design, to database parameters such as buffer sizes, down to hardware issues such as number of disks - affect the performance of an application. Each of these aspects can be adjusted so that performance is improved.

i. Location of Bottleneck
ii. Tunable parameters
iii. Tuning of Hardware
iv. Tuning of the schema
v. Tuning of indices
vi. Using Materialized views
vii. Tuning of Transactions
viii. Performance simulation.

Location of Bottleneck.

↳ As a result of the numerous queues in the database, bottlenecks in database system typically show up in the form of long queues for particular service or equivalently in high utilizations for a particular service.

↳ The utilization of a resource should be kept low enough that queue length is short.

Tunable parameters.

↳ Database administrators can tune database system at three levels.

↳ The lowest level is at the hardware level

↳ options for tuning at this level includes adding disks or using a RAID system if disk I/O is bottleneck, adding more memory if the disk buffer size is a bottleneck, or moving to a faster processor if CPU use is bottleneck.

↳ The second level consists of the database system parameters, such as buffersize and checkpointing intervals.

↳ The exact set of of database-system parameters that can be tuned depends on the specific database system.

↳ Most database-system manuals provide information on what database-system parameters can be adjusted, and how you should choose value for the parameters.

↳ The third level is the highest level. It includes the schema and transactions.

↳ The administrater can tune the design

of the schema, the indices that are created, and the transactions that are executed, to improve performance. Tuning at this level is comparatively system independent.