CHAPTER - 7

QUERY PROCESSING

7.1 Introduction to Query Processing

Query processing is a translation of high-level queries into low level expressions. It refers to the range of activities that are involved in extracting data from the database which includes translation of queries in high level database languages into expression that can be implemented at the physical level of the file system. Query processing is a stepwise process that can be used at the physical level of the file system, query optimization and actual execution of query to get the result and requires the basic concepts of relational algebra and file structure.

Basic Steps in Query Processing:

1. Parsing and translation
2. Optimization
3. Evaluation

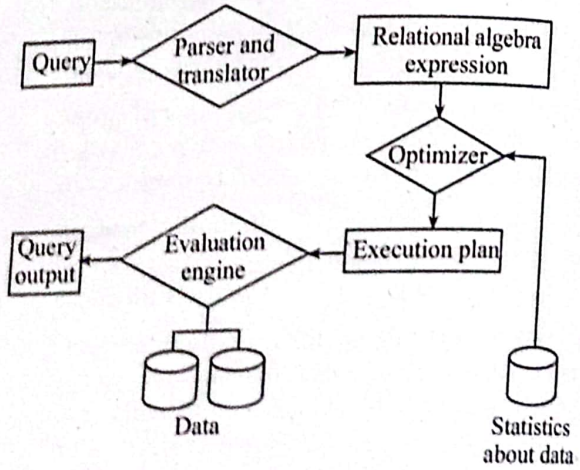


Figure: Steps in query processing

1. **Query Parsing and Translation**

In this phase a query is translated into SQL and into relational algebraic expression. The parser of the query processor module checks the syntax of the query, the user's privileges to execute the query, the table names and attribute names, etc. The correct table names, attribute names and the privilege of the users can be taken from the data dictionary. If the query is valid, then it is converted from high level language SQL to low level instruction in relational algebra which is represented as a query tree data structure. Different possible relational algebra expression may exist for a single query.

Example:

SELECT Ename FROM Employee WHERE Salary>5000;

The query is then translated into Relational Algebra Expression as:

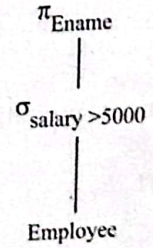
$\sigma_{salary>5000}(\Pi_{Ename}(Employee))$

OR

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

2. **Query Optimization**

Query optimization is the process of selecting the most efficient query evaluation plan among the many strategies usually possible for processing a given query specially if the query is complex. A relational algebra operation annotated with instructions on how to evaluate 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 evaluation.



3. Query Evaluation

The query execution engine takes a query evaluation plan, executes the plan and returns the answers to the query. Specify which access path to follow, which algorithm to use to evaluate operators and how operators interleave.

Output

The final result of the query is shown to the user.

7.2 Equivalence of Expressions

A query can be expressed in several different ways with different cost 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 two expressions generate the same multiset of tuples.

Equivalence Rules

An equivalence rule says that expressions of two forms are equivalent. By applying transformation rules, the optimizer can transform one relational algebra expression into an equivalent expression that is known to be more efficient. Following rules can be applied.

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.

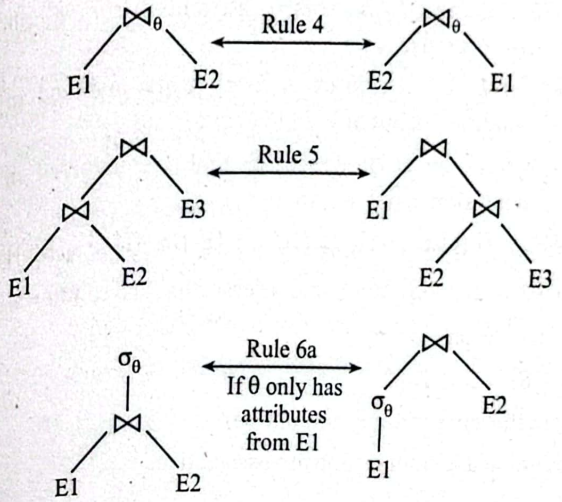
   $\Pi_{t_1}(\Pi_{t_2}(K\ (\Pi_{tn}(E))K)) = \Pi_{t_1}(E)$

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

   a. $\sigma_\theta(E_1 \times E_2) = E_1 \bowtie_\theta E_2$

   b. $\sigma_{\theta 1}(E_1 \bowtie_{\theta 2} E_2) = E_1 \bowtie_{\theta 1 \wedge \theta 2} E_2$

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 join operations are associative:

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

   (b) Theta joins are associative in the following manner:

   $(E_1 \bowtie_{\theta 1} E_2) \bowtie_{\theta 2 \wedge \theta 3} E_3 = E_1 \bowtie_{\theta 2 \wedge \theta 3} (E_2 \bowtie_{\theta 2} E_3)$

   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. When all the attributes in θ_0 involve only the attributes of one of the expressions(E1) being joined.

   $\sigma_{\theta 0}(E_1 \bowtie_\theta E_2) = (\sigma_{\theta 0}(E_1)) \bowtie_\theta E_2$

   b. When θ_1 involves only the attributes of E_1 and θ_2 involves only the 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 projections operation distributes over the theta join operation as follows:

   a. If P involves only attributes from $L_1 \cup L_2$:

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

   b. Consider a join $E_1 \theta E_2$.

   - 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 are not in $L_1 \cup L_2$, and
   - Let L_4 be attributes of E_2 that are involved in join condition θ, but are not in $L_1 \cup L_2$.

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

9. 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).

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 [$\cup$, $\cap$ and $-$]

    $\sigma_\theta(E_1 - E_2) = \sigma_\theta(E_1) - \sigma_\theta(E_2)$ and similarly for $\cup$ and $\cap$ in place of $-$

    Also: $\sigma_\theta(E_1 - E_2) = \sigma_\theta(E_1) - E_2$ and similarly for $\cap$ in place of $-$, but not for $\cup$

12. The projection operation distributes over union

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

Transformation Example:

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

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

Transformation using rule 7a.

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

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

Example with Multiple Transformation

Query: Find the names of all customers with an account at a Brooklyn branch whose account balance is over $1000.

$\Pi_{\text{customer-name}}((\sigma_{\text{branch-city}=\text{"Brooklyn"} \wedge \text{balance} > 1000}(\text{branch} \bowtie (\text{account} \bowtie \text{depositor})))$

Transformation using join associativity (Rule 6a):

$\Pi_{\text{customer-name}}((\sigma_{\text{branch-city}=\text{"Brooklyn"} \wedge \text{balance} > 1000}(\text{branch} \bowtie (\text{account})) \bowtie \text{depositor})$

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

$\sigma_{\text{branch-city}=\text{"Brooklyn"}}(\text{branch}) \bowtie \sigma_{\text{balance}>1000}(\text{account})$

Thus a sequence of transformations can be useful.

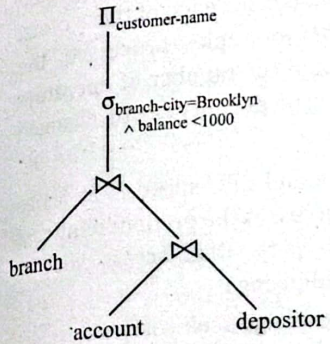


Figure a: Initial Expression Tree

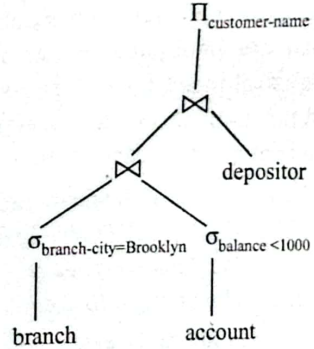


Figure b: Tree After Multiple Transformations

When we compute ($\sigma_{\text{branch-city}=\text{"Brooklyn"}}$(*branch*)$\bowtie$*account*) we obtain a relation whose schema is

(branch-name, branch-city, assets, account-number, balance)

Push projections using equivalence rules 8a and 8b; eliminate unneeded attributes from intermediate results to get:

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

7.3 Query Cost Estimation

The cost of query evaluation can be measured in terms of a number of different resources, including disk access, 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 the computer would account for all these costs, and could be used as a good measure of the cost of the plan.

With data resident in-memory or on SSDs, I/O cost does not dominate the overall cost, and we must include CPU costs when computing the cost of query evaluation. We do not include CPU costs in our model to simplify our presentation, but note that they can be approximated by simple estimators. For example, the cost model used by PostgreSQL (as of 2018) includes (i) a CPU cost per tuple, (ii) a CPU cost for processing each index entry (in addition to the I/O cost), and (iii) a CPU cost per operator or function (such as arithmetic operators, comparison operators, and related functions).The database has default values for each of these costs, which are multiplied by the number of tuples processed, the number of index entries processed, and the number of operators and functions executed, respectively. The defaults can be changed as a configuration parameter.

CPU cost is difficult to calculate and 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 of following :

1. Number of seeks (average-seek-cost)
2. Number of blocks read (average-block-read-cost)
3. Number of blocks written (average-block-write-cost)

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

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

t_T– time to transfer one block

t_S – time for one seek

Cost for b block transfers plus S seeks:

$b * t_T + S * t_S$

We do not include cost to writing output to disk 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.

7.3.1 Selection Operation

In query processing, the file scan is the lowest-level operator to access data. File scans are search algorithms that locate and retrieve records that fulfill a selection condition. In relational systems, a file scan allows an entire relation to be read in those cases where the relation is stored in a single, dedicated file.

a. **Selections Using File Scans and Indices**

Consider a selection operation on a relation whose tuples are stored together in one file. Way of performing a selection is as follows:

A1 (Linear Search)

In a linear search, the system scans each file block and tests all records to see whether they satisfy the selection condition. 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 are not stored contiguously, extra seeks may be required, but we ignore this effect for simplicity.

Cost

One initial seeks plus b_r block transfers, where b_r denotes the number of blocks in the file.

$t_S + b_r * t_T$

A1 (Linear Search, Equality on Key)

Cost of $(b_r/2)$ block transfers plus one seek

$t_S + (b_r/2) * t_T$

Since at most one record satisfies condition, scan can be terminated as soon as the required record is found. In the worst case, b_r blocks transfers are still required.

Linear search can be applied regardless of

i. Selection condition or
ii. Ordering of records in the file, or
iii. Availability of indices

A2(Clustering B+-tree Index, Equality on Key)

For an equality comparison on a key attribute with a clustering index, we can use the index to retrieve a single record that satisfies the corresponding equality condition.

Cost:

$(h_i+1) * (t_T+t_S)$

Where h_i is the height of the tree in i^{th} level

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

A3(Clustering B+-tree Index, Equality on Non-key)

We can retrieve multiple records by using a clustering index when the selection condition specifies an equality comparison on a non-key attribute, A. The only difference from the previous case is that multiple records may need to be fetched. However, the records must be stored consecutively in the file since the file is sorted on the search key.

Cost of the query becomes

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

One seek for each level of the tree, one seek for the first block. Here b is the number of blocks containing records with the specified search key, all of which are read. These blocks are leaf blocks assumed to be stored sequentially (since it is a clustering index) and don't require additional seeks.

A4(Secondary B+-tree Index, Equality on Key)

Selections specifying an equality condition can use a secondary index. This strategy can retrieve a single record if the equality condition is on a key; multiple records may be retrieved if the indexing field is not a key.

In the first case, only one record is retrieved. The time cost in this case is the same as that for a clustering index (case A2).

$(h_i+1) * (t_T+t_S)$

Where h_i is the height of the tree in i^{th} level

In the second case, each record may be resident on a different block, which may result in one I/O operation per retrieved record, with each I/O operation requiring a seek and a block transfer.

The worst-case time cost in this case is

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

where n is the number of records fetched

b. Selections Involving Comparison

We can implement selections of the form $\sigma_{A\leq V}(r)$ or $\sigma_{A\geq V}(r)$ by using a linear file scan or by using indices in the following ways:

A5 (Clustering B+- tree Index, Comparison)

A clustering ordered index (for example, a clustering B+-tree index) can be used when the selection condition is a comparison.

For comparison conditions of the form A > v or A ≥ v, a clustering index on A can be used to direct the retrieval of tuples, as follows: For A ≥ v, we look up the value v in the index to find the first tuple in the file that has a value of A ≥ v. A file scan starting from that tuple up to the end of the file returns all tuples that satisfy the condition. For A > v, the file scan starts with the first tuple such that A > v. The cost estimate for this case is identical to that for case A3.

For comparisons of the form A < v or A ≤ v, an index lookup is not required. For A < v, we use a simple file scan starting

from the beginning of the file, and continuing up to (but not including) the first tuple with attribute A = v. The case of A ≤ v is similar, except that the scan continues up to (but not including) the first tuple with attribute A > v. In either case, the index is not useful.

Cost:

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

Identical to the case of A3, equality on non-key.

A6(Secondary B+-tree Index, Comparison)

We can use a secondary ordered index to guide retrieval for comparison conditions involving <,≤,≥, or >. The lowest-level index blocks are scanned, either from the smallest value up to v (for < and ≤), or from v up to the maximum value (for > and ≥).

The secondary index provides pointers to the records, but to get the actual records we have to fetch the records by using the pointers. This step may require an I/O operation for each record fetched, since consecutive records may be on different disk blocks; as before, each I/O operation requires a disk seek and a block transfer. If the number of retrieved records is large, using the secondary index may be even more expensive than using linear search. Therefore, the secondary index should be used only if very few records are selected.

Cost:

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

Identical to the case of A4, equality on non-key.

Implementation of Complex Selections

So far, we have considered only simple selection conditions of the form A op B, where op is an equality or comparison operation. We now consider more complex selection predicates.

Conjunction

A conjunctive selection is a selection of the form:

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

Disjunction

A disjunctive selection is a selection of the form:

$\sigma_{\theta 1 \vee \theta 2 \vee \cdots \vee \theta n}(r)$

A disjunctive condition is satisfied by the union of all records satisfying the individual, simple conditions θi.

Negation

The result of a selection $\sigma_{\neg\theta}(r)$ is the set of tuples of r for which the condition θ evaluates to false. In the absence of nulls, this set is simply the set of tuples in r that are not in $\sigma_{\theta}(r)$.

We can implement a selection operation involving either a conjunction or a disjunction of simple conditions by using one of the following algorithms

A7 (Conjunctive Selection using One Index)

We first determine whether an access path is available for an attribute in one of the simple conditions. If one is, one of the selection algorithms A2 through A6 can retrieve records satisfying that condition

To reduce the cost we choose a θ_i and one of algorithms A1 through A6 for which the combination results in the least cost for $\sigma_{\theta i}(r)$. The cost of algorithm A7 is given by the cost of the chosen algorithm.

A8 (Conjunctive Selection using Composite Index)

An appropriate composite index (that is, an index on multiple attributes) may be available for some conjunctive selections. If the selection specifies an equality condition on two or more attributes, and a composite index exists on these combined attribute fields, then the index can be searched directly. The type of index determines which of algorithms A2, A3, or A4 will be used.

A9 (Conjunctive Selection by Intersection of Identifiers)

Conjunctive selection operations involve the use of record pointers or record identifiers. This algorithm requires indices with record pointers, on the fields involved in the individual conditions. The algorithm scans each index for pointers to

tuples that satisfy an individual condition. The intersection of all the retrieved pointers is the set of pointers to tuples that satisfy the conjunctive condition. The algorithm then uses the pointers to retrieve the actual records. If indices are not available on all the individual conditions, then the algorithm tests the retrieved records against the remaining conditions.

The cost of algorithm A9 is the sum of the costs of the individual index scans, plus the cost of retrieving the records in the intersection of the retrieved lists of pointers. This cost can be reduced by sorting the list of pointers and retrieving records in the sorted order. Thereby, (1) all pointers to records in a block come together, hence all selected records in the block can be retrieved using a single I/O operation, and (2) blocks are read in sorted order, minimizing disk-arm movement.

A10 (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. The union of all the retrieved pointers yields the set of pointers to all tuples that satisfy the disjunctive condition. We then use the pointers to retrieve the actual records.

However, if even one of the conditions does not have an access path, we have to perform a linear scan of the relation to find tuples that satisfy the condition. Therefore, if there is even one such condition in the disjunct, the most efficient access method is a scan, with the disjunctive condition tested on each tuple during the scan.

7.3.2 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. However, such a process orders the relation only logically, through an index, rather than physically. Hence, the reading of tuples in the sorted order may lead to a disk access (disk seek plus block transfer) for each record, which can be very expensive, since the number of records can be much larger than the number of blocks. For this reason, it may be desirable to order the records physically.

For relations that fit in memory that is records are completely in main memory, techniques like quick sort can be used. For relations that don't fit in memory, external sort-merge is a good choice. External sort- merge, which cumulatively sorts multiple runs of the data based on amount that fits in memory at one time.

External Sort-Merge Algorithm

Sorting of relations that do not fit in memory is called external sorting. The most commonly used technique for external sorting is the external sort-merge algorithm.

Let M denote the number of blocks in the main-memory buffer available for sorting, that is, the number of disk blocks whose contents can be buffered in available main memory.

1. In the first stage, a number of sorted runs are created; each run is sorted, but contains only some of the records of the relation.

```
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 Ri;
    i = i + 1;
until the end of the relation
```

2. In the second stage, the runs are merged. Suppose, for now, that the total number of runs, N, is less than M, so that we can allocate one block to each run and have space left to hold one block of output. The merge stage operates as follows:

```
read one block of each of the N files Ri 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

The output of the merge stage is the sorted relation. The output file is buffered to reduce the number of disk write operations. The preceding merge operation is a generalization of the two-way merge used by the standard in-memory sort–merge algorithm; it merges N runs, so it is called an N-way merge.

If N≥M, several merge passes are required. In each pass, contiguous groups of M - 1 runs are merged. A pass reduces the number of runs by a factor of M -1, and creates runs longer by the same factor.

Example:

If M=11, and there are 90 runs, one pass reduces the number of runs to 9, each 10 times the size of the initial runs. Repeated passes are performed till all runs have been merged into one.

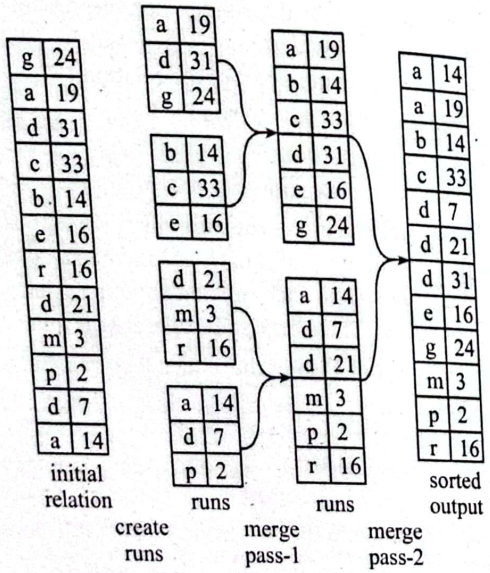


Figure: External sort merge

Total number of block transfers for external sorting:

$$b_r[2[log_{[M/b_b]-1}(b_r/M)]+1)$$

Applying this equation to the example in Figure with b_b set to 1, we get a total of 12 * (4 + 1) = 60 block transfers

The total number of seeks is :

$$2[b_r/M] + [b_r/b_b](2[log_{[M/b_b]-1}(b_r/M)] - 1)$$

Applying this equation to the example in above figure, we get a total of 8 + 12 *(2 * 2 − 1) = 44 disk seeks if we set the number of buffer blocks per run, b_b to 1.

7.3.3 Join Operation

We use the term equi-join to refer to a join of the form $r \bowtie_{A=s.B} s$, where A and B are attributes or sets of attributes of relations r and s, respectively.

We use as a running example the expression:

student ⋈ takes

We assume the following information about the two relations

Number of records of student: $n_{student} = 5,000$.

Number of blocks of student: $b_{student} = 100$.

Number of records of takes: $n_{takes} = 10,000$.

Number of blocks of takes: $b_{takes} = 400$.

Different algorithms to implement joins:

1. Nested-loop
2. Block nested-loop join
3. Indexed nested-loop join
4. Merge-join
5. Hash-join

Choice is based on cost estimate.

1. Nested-Loop Join

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

relation s the inner relation of the join, since the loop for r encloses the loop for s. The algorithm uses the notation $t_r \cdot t_s$, where t_r and t_s are tuples; $t_r \cdot t_s$ denotes the tuple constructed by concatenating the attribute values of tuples t_r and t_s.

```
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
```

The nested-loop join algorithm is expensive, since it examines every pair of tuples in the two relations The number of pairs of tuples to be considered is $n_r * n_s$, where n_r denotes the number of tuples in r , and n_s denotes the number of tuples in s.

For each record in r , we have to perform a complete scan on s. In the worst case, the buffer can hold only one block of each relation, and a total of $n_r * b_s + b_r$ block transfers would be required where b_r and b_s denote the number of blocks containing tuples of r and s, respectively. We need only one seek for each scan on the inner relation s since it is read sequentially, and a total of b_r seeks to read r , leading to a total of $n_r + b_r$ seeks.

In the best case, there is enough space for both relations to fit simultaneously in memory, so each block would have to be read only once; hence, only $b_r + b_s$ block transfers would be required, along with 2 seeks.

Example:

Consider the natural join of student and takes. Assume for now that we have no indices whatsoever on either relation, and that we are not willing to create any index. We can use the nested loops to compute the join; assume that student is the outer relation and takes is the inner relation in the join.

We will have to examine $5000 * 10,000 = 50 * 10^6$ pairs of tuples. In the worst case, the number of block transfers is $5000 * 400 + 100 = 2,000,100$, plus $5000 + 100 = 5100$ seeks.

In the best- case scenario, however, we can read both relations only once, and perform the computation. This computation requires at most 100 + 400 = 500 block transfers, plus 2 seeks. A significant improvement over the worst-case scenario. If we had used takes as the relation for the outer loop and student for the inner loop, the worst-case cost of our final strategy would have been $10,000 * 100 + 400 = 1,000,400$ block transfers, plus 10,400 disk seeks. The number of block transfers is significantly less, and although the number of seeks is higher, the overall cost is reduced, assuming t_S=4 milliseconds and t_T=0.1 milliseconds.

2. **Block Nested-Loop Join**

If the buffer is too small to hold either relation entirely in memory, we can still obtain a major saving in block accesses if we process the relations on a per-block basis, rather than on a per-tuple basis. Block nested-loop is a variant of the nested-loop join where every block of the inner relation is paired with every block of the outer relation. Within each pair of blocks, every tuple in one block is paired with every tuple in the other block, to generate all pairs of tuples. As before, all pairs of tuples that satisfy the join condition are added to the result.

```
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
                test pair (tr , ts) to see if they satisfy
                the join condition
                if they do, add tr · ts to the result;
            end
        end
    end
end
```

In the worst case, each block in the inner relation s is read only once for each block in the outer relation, instead of once for each tuple in the outer relation. Thus, in the worst case, there will be a total of $b_r * b_s + b_r$ block transfers, where b_r and b_s denote the number of blocks containing records of r and s, respectively. Each scan of the inner relation requires one seek, and the scan of the outer relation requires one seek per block, leading to a total of $2 * b_r$ seeks. It is more efficient to use the smaller relation as the outer relation, in case neither of the relations fits in memory.

In the best case, where the inner relation fits in memory, there will be $b_r + b_s$ block transfers and just two seeks (we would choose the smaller relation as the inner relation in this case).

Now return to our example of computing student ⋈ takes, using the block nested loop join algorithm. In the worst case, we have to read each block of takes once for each block of student.

Thus, in the worst case, a total of 100 * 400 + 100 = 40,100 block transfers plus 2 * 100 = 200 seeks are required.

This cost is a significant improvement over the 5000 * 400 + 100 = 2,000,100 block transfers plus 5100 seeks needed in the worst case for the basic nested-loop join.

The best-case cost remains the same–namely, 100 + 400 = 500 block transfers and two seeks.

Indexed Nested-Loop Join

In a nested-loop join, if an index is available on the inner loop's join attribute, index lookups can replace file scans. For each tuple t_r in the outer relation r, the index is used to look up tuples in s that will satisfy the join condition with tuple t_r. This join method is called an indexed nested-loop join; it can be used with existing indices, as well as with temporary indices created for the sole purpose of evaluating the join.

Looking up tuples in s that will satisfy the join conditions with a given tuple t_r is essentially a selection on s. For example, consider student ⋈ takes. Suppose that we have a

student tuple with ID "00128". Then, the relevant tuples in takes are those that satisfy the selection "ID = 00128". The cost of an indexed nested-loop join can be computed as follows: For each tuple in the outer relation r, a lookup is performed on the index for s, and the relevant tuples are retrieved.

In the worst case, there is space in the buffer for only one block of r and one block of the index. Then, b_r I/O operations are needed to read relation r, where b_r denotes the number of blocks containing records of r; each I/O requires a seek and a block transfer, since the disk head may have moved in between each I/O. For each tuple in r, we perform an index lookup on s.

Then, the cost of the join can be computed as:

$b_r(t_T + t_S) + n_r * c$, where nr is the number of records in relation r, and c is the cost of a single selection on s using the join condition. We have seen in how to estimate the cost of a single selection algorithm (possibly using indices); that estimate gives us the value of c. The cost formula indicates that, if indices are available on both relations r and s, it is generally most efficient to use the one with fewer tuples as the outer relation.

For example, consider an indexed nested-loop join of student ⋈ takes, with student as the outer relation. Suppose also that takes has a clustering B+-tree index on the join attribute ID, which contains 20 entries on average in each index node. Since takes has

10,000 tuples, the height of the tree is 4, and one more access is needed to find the actual data. Since $n_{student}$ is 5000, the total cost is 100 + 5000 * 5 = 25,100 disk accesses, each of which requires a seek and a block transfer. In contrast, as we saw before, 40,100 block transfers plus 200 seeks were needed for a block nested-loop join. Although the number of block transfers has been reduced, the seek cost has actually increased, increasing the total cost since a seek is considerably more expensive than a block transfer. However, if we had a selection on the student relation that reduces the

number of rows significantly, indexed nested-loop join could be significantly faster than block nested-loop join.

4. Merge Join

Assuming that b_b buffer blocks are allocated to each relation, the number of disk seeks required would be $\lceil b_r/b_b \rceil + \lceil b_s/b_b \rceil$ disk seeks and block transfer would be $b_r + b_s$. Since seeks are much more expensive than data transfer, it makes sense to allocate multiple buffer blocks to each relation, provided extra memory is available.

For example, with $t_T = 0.1$ milliseconds per 4-kilobyte block, and $t_S = 4$ milliseconds, the buffer size is 400 blocks (or 1.6 megabytes), so the seek time would be 4 milliseconds for every 40 milliseconds of transfer time; in other words, seek time would be just 10 percent of the transfer time.

Suppose the merge-join scheme is applied to our example of student ⋈ takes. The join attribute here is ID. Suppose that the relations are already sorted on the join attribute ID. In this case, the merge join takes a total of 400+100 = 500 block transfers. If we assume that in the worst case only one buffer block is allocated to each input relation (that is, $b_b = 1$), a total of 400 + 100 = 500 seeks would also be required; in reality b_b can be set much higher since we need to buffer blocks for only two relations, and the seek cost would be significantly less.

5. Hash Join

Like the merge-join algorithm, the hash-join algorithm can be used to implement natural joins and equi-joins. In the hash-join algorithm, 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 the same hash value on the join attributes. We assume that:

- h is a hash function mapping JoinAttrs values to {0, 1,⋯, n_h}, where JoinAttrs denotes the common attributes of r and s used in the natural join.

- $r_0, r_1, \cdots, r_{n_h}$ nh denote partitions of r tuples, each initially empty. Each tuple tr ∈ r is put in partition r_i, where i = h(t_r[JoinAttrs]).
- $s_0, s_1, \ldots, s_{n_h}$ denote partitions of s tuples, each initially empty. Each tuple ts ∈ s is put in partition s_i, where i = h(t_s[JoinAttrs]).

Thus, a hash join is estimated to require:

$3(b_r + b_s) + 4n_h$ block transfers.

$2(\lceil b_r/b_b \rceil + \lceil b_s/b_b \rceil) + 2n_h$ seeks.

7.4 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 do not 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.

Aim of query optimization is to transform query into faster, equivalent query. Every method of query optimization depends on database statistics. 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. A relational algebra expression may have many equivalent expressions

Example:

$\sigma_{salary<75000}(\prod_{salary}(instructor))$ is equivalent to $\prod_{salary}(\sigma_{salary<75000}(instructor))$

Each relational algebra operation can be evaluated using one of several different algorithms. Correspondingly, a relational-algebra expression can be evaluated in many ways. Annotated expression specifying detailed evaluation strategy is called an *evaluation-plan.*

We can use an index on *salary* to find instructors with salary < 75000, or can perform complete relation scan and discard instructors with salary ≥ 75000

Types of Optimization

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

i. Cost-Based Query Optimization

A cost-based optimization explores the space of all query evaluation plans that are equivalent to given query an chooses the one with least estimated cost. Cost of physical plans includes processor time and communication time. The most important factor to consider is disk I/Os because it is the most time-consuming action. Some other costs associated are: Operations (joins, unions, intersections).

Steps in cost-based query optimization

1. Generate logically equivalent expressions using equivalence rules
2. Annotate resultant expressions to get alternative query plans
3. Choose the cheapest plan based on estimated cost

Estimation of plan cost is based on:

1. Statistical information about relations.
   Example:
   number of tuples, number of distinct values for an attribute

2. Statistics estimation for intermediate results to compute cost of complex expressions
3. Cost formulae for algorithms, computed using statistics

Consider finding the best join-order for $r_1 \bowtie r_2 \ldots r_n$ where the joins are expressed without any ordering. With n = 3, there are 12 different join orderings:

i. r1 ⋈ (r2 ⋈ r3)
ii. r1 ⋈ (r3 ⋈ r2)
iii. (r2 ⋈ r3) ⋈ r1
iv. (r3 ⋈ r2) ⋈ r1
v. r2 ⋈ (r1 ⋈ r3)
vi. r2 ⋈ (r3 ⋈ r1)
vii. (r1 ⋈ r3) ⋈ r2
viii. (r3 ⋈ r1) ⋈ r2
ix. r3 ⋈ (r1 ⋈ r2)
x. r3 ⋈ (r2 ⋈ r1)
xi. (r1 ⋈ r2) ⋈ r3
xii. (r2 ⋈ r1) ⋈ r3

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. We don't need to generate all the join orders.

For example, suppose we want to find the best join order of the form:

(r1 ⋈ r2 ⋈ r3) ⋈ r4 ⋈ r5

which represents all join orders where r1, r2, and r3 are joined first (in some order), and the result is joined (in some order) with r4 and r5. There are 12 different join orders for computing r1 ⋈ r2 ⋈ r3, and 12 orders for computing the join of this result with r4 and r5. Thus, there appear to be 144 join orders to examine. However, once we have found the best join order for the subset of relations {r1, r2, r3}, we can use that order for further joins with r4 and r5, and we can ignore all costlier join orders of r1 ⋈ r2 ⋈ r3.Thus, instead of 144 choices to examine, we need to examine only 12 + 12 choices.

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 (typical queries have small n, generally < 10)

How cost-based optimization work?

A cost-based optimizer will look at all of the possible scenarios in which a query can be executed. Each scenario will be assigned a 'cost', which indicates how efficiently that query can be run. Then, the cost-based optimizer will pick the scenario that has the least cost and execute the query using that scenario, because that is the most efficient way to run the query.

ii. Heuristic Query Optimization

A drawback of cost-based optimization is the cost of optimization itself. Although the cost of query optimization can be reduced by clever algorithms, the number of different evaluation plans for a query can be very large, and finding the optimal plan from this set requires a lot of computational effort. Hence, optimizers use heuristics to reduce the cost of optimization.

An example of a heuristic rule is the following rule for transforming relational algebra queries:

- Perform selection operations as early as possible.
- Perform projections early.
- Perform most restrictive selection and join operations (i.e. with smallest result size) before other similar operations.

Most practical query optimizers have further heuristics to reduce the cost of optimization.

Example:

Consider the relations as:

Player(Fname Name, Lname, SSN, DraftDate, Address, Salary, CoachSSN,TeamName)

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'.

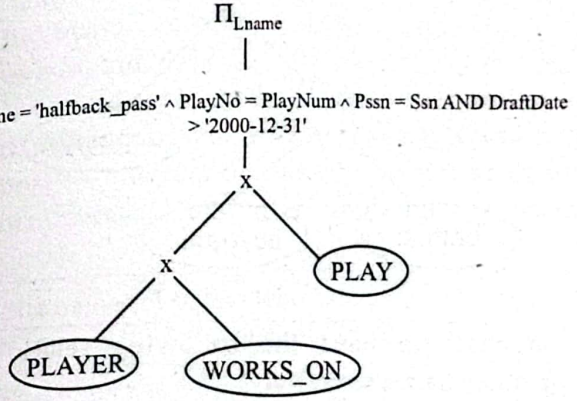
SQL:

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

Equivalent relational algebra is :

Π_{Lname}

$\sigma_{PlayName = 'halfback\_pass' \wedge PlayNo = PlayNum \wedge Pssn = Ssn \wedge DraftDate > '2000\text{-}12\text{-}31'}$ (player×works_on×play)



What's wrong with the tree?

It should work, but it is not optimal Why not? As is, we would 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, the Cartesian product could be huge and would contain lots of unneeded data.

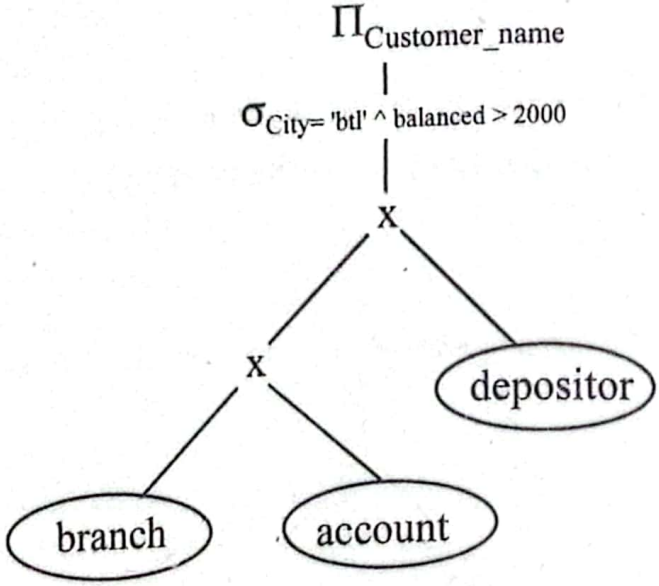
We First, move SELECT operations down the query tree. Second, perform the more restrictive SELECT operations first. Third, replace CARTESIAN PRODUCT and SELECT combinations with JOIN operations. Finally, move PROJECT operations down the query tree.

SOLUTION TO EXAMS' AND OTHER IMPORTANT QUESTIONS

1. Make an operator tree for the following SQL expression
SELECT customer_name
FROM branch,account,depositer
WHERE branch_city= 'btl' AND balance>2000 [2023 Spring]

ANS: Equivalent relational algebra for given SQL expression is:

$\Pi_{customer\_name}$ ($\sigma_{city = 'btl' \wedge balance>2000}$ (brach x account x depositer))



2. Suppose we have the following relation: Employee (person_name, street, city)
Works (person_name, company_name, salary)
Company (company_name, city)
Write the relational algebra expression for the query "Find
the names of all employees who lives in Pokhara". Construct the initial operator tree and final efficient operator tree after applying transformation rules. [2022 Fall]

ANS: Relational algebra expression for given query is:

Π_{person_name} ($\sigma_{city = 'Pokhara'}$ (Employee))

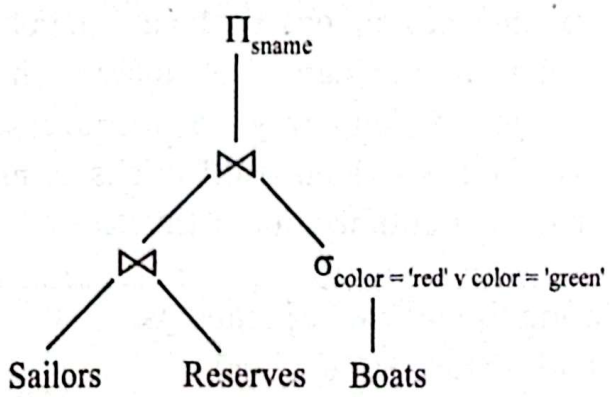
Here's the initial operator tree representation:

When all the attributes in θ_0 involve only the attributes of one of the expressions(E1) being joined then following transformation is possible:

$\sigma_{\theta 0}(E_1 \bowtie_{\theta 1} E_2) = (\sigma_{\theta 0}(E_1)) \bowtie_{\theta 1} E_2$

Hence above query becomes:

Π_{sname} ((Sailors $\bowtie$ Reserves) $\bowtie$($\sigma_{\text{color='red' v color='green'}}$ (Boats))



4. Show how to drive the following equivalences by a sequence of transformations using the equivalence rules.

$\sigma_{\theta 1 \wedge \theta 2 \wedge \theta 3}(E) = \sigma_{\theta 1}(\sigma_{\theta 2}(\sigma_{\theta 3}(E)))$ **[2014 Spring]**

ANS: Applying the associative law of conjunction:

$\sigma_{\theta 1 \wedge \theta 2 \wedge \theta 3}(E) = \sigma_{\theta 1 \wedge (\theta 2 \wedge \theta 3)}(E)$

Applying the selection-pushdown rule

$\sigma_{\theta 1 \wedge (\theta 2 \wedge \theta 3)}(E) = \sigma_{\theta 1}(\sigma_{\theta 2 \wedge \theta 3}(E))$

$\sigma_{\theta 1}(\sigma_{\theta 2 \wedge \theta 3}(E)) = \sigma_{\theta 1}(\sigma_{\theta 2}(\sigma_{\theta 3}(E)))$

◆ ◆ ◆