UNIT 1 QUERY PROCESSING AND EVALUATION

Structure **Page Nos.**

1.0 INTRODUCTION

The Query Language – SQL is one of the main reasons of success of RDBMS. A user just needs to specify the query in SQL that is close to the English language and does not need to say how such query is to be evaluated. However, a query needs to be evaluated efficiently by the DBMS. But how is a query-evaluated efficiently? This unit attempts to answer this question. The unit covers the basic principles of query evaluation, the cost of query evaluation, the evaluation of join queries, etc. in detail. It also provides information about query evaluation plans and the role of storage in query evaluation and optimisation. This unit thus, introduces you to the complexity of query evaluation in DBMS.

1.1 OBJECTIVES

After going through this unit, you should be able to:

- measure query cost;
- define algorithms for individual relational algebra operations;
- create and modify query expression;
- define evaluation plan choices, and
- define query processing using views.

1.2 QUERY PROCESSING: AN INTRODUCTION

Before defining the measures of query cost, let us begin by defining query processing. Let us take a look at the *Figure 1*.

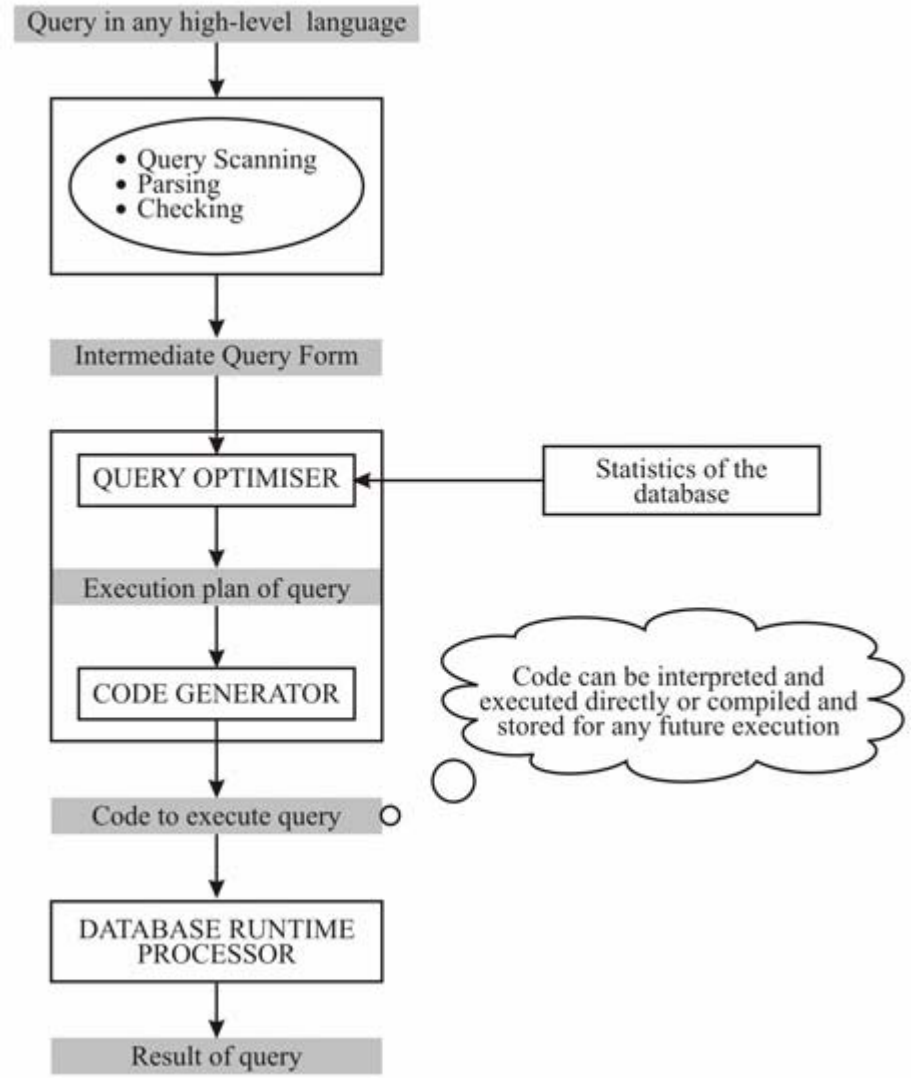


Figure 1: Query processing

In the first step Scanning, Parsing, and Validating is done to translate the query into its internal form. This is then further translated into relational algebra (an intermediate query form). Parser checks syntax and verifies relations. The query then is optimised with a query plan, which then is compiled into a code that can be executed by the database runtime processor.

We can define query evaluation as the query-execution engine taking a query-evaluation plan, executing that plan, and returning the answers to the query. The query processing involves the study of the following concepts:

- how to measure query costs?
- algorithms for evaluating relational algebraic operations.
- how to evaluate a complete expression using algorithms on individual operations?

1.2.1 Optimisation

A relational algebra expression may have many equivalent expressions. For example, $\sigma_{(salary < 5000)}(\pi_{salary}(EMP))$ is equivalent to $\pi_{salary}(\sigma_{salary <5000}(EMP))$.

Each relational algebraic operation can be evaluated using one of the several different algorithms. Correspondingly, a relational-algebraic expression can be evaluated in many ways.

An expression that specifies detailed evaluation strategy is known as evaluation-plan, for example, you can use an index on *salary* to find employees with *salary < 5000* or we can perform complete relation scan and discard employees with salary ≥ 5000. The basis of selection of any of the scheme will be the cost.

Query Optimisation: Amongst all equivalent plans choose the one with the lowest cost. Cost is estimated using statistical information from the database catalogue, for example, number of tuples in each relation, size of tuples, etc.

Thus, in query optimisation we find an evaluation plan with the lowest cost. The cost estimation is made on the basis of heuristic rules.

1.2.2 Measure of Query Cost

Cost is generally measured as total elapsed time for answering the query. There are many factors that contribute to time cost. These are *disk accesses*, CPU time, or even network *communication*.

Typically disk access is the predominant cost as disk transfer is a very slow event and is also relatively easy to estimate. It is measured by taking into account the following activities:

Number of seeks	×	average-seek-cost
Number of blocks read	×	average-block-read-cost
Number of blocks written	×	average-block-written-cost.

Please note that the cost for writing a block is higher than the cost for reading a block. This is due to the fact that the data is read back after being written to ensure that the write was successful. However, for the sake of simplicity we will just use *number of block transfers from disk as the cost measure.* We will also ignore the difference in cost between sequential and random I/O, CPU and communication costs. The I/O cost **depends** on the search criteria i.e., point/range query on an ordering/other fields and the file structures: heap, sorted, hashed. It is also dependent on the use of indices such as primary, clustering, secondary, B+ tree, multilevel, etc. There are other cost factors also, these may include buffering, disk placement, materialisation, overflow / free space management etc.

In the subsequent section, let us try to find the cost of various operations.

1.3 SELECT OPERATION

The selection operation can be performed in a number of ways. Let us discuss the algorithms and the related cost of performing selection operations.

File scan: These are the search algorithms that locate and retrieve records that fulfil a selection condition in a file. The following are the two basic files scan algorithms for selection operation:

1) *Linear search:* This algorithm scans each file block and tests all records to see whether they satisfy the selection condition.

The cost of this algorithm (in terms of block transfer) is defined as:

Cost of searching records satisfying a condition = Number of blocks in a database = N_b.

Cost for searching a key attribute value = Average number of block transfer for locating the value (on an average, half of the file needs to be traversed) so it is = $N_b/2$.

Linear search can be applied regardless of selection condition or ordering of records in the file, or availability of indices.

2) *Binary search:* It is applicable when the selection is an equality comparison on the attribute on which file is ordered. Assume that the blocks of a relation are stored continuously then, cost can be estimated as:

Cost = Cost of locating the first tuple by a binary search on the blocks + sequence of other blocks that continue to satisfy the condition.

$$= \lceil \log_2 (N_b) \rceil + \frac{\text{average number of tuples with the same value}}{\text{Blocking factor (Number of tuples in a block) of the relation}}$$

These two values may be calculated from the statistics of the database.

Index scan: Search algorithms that use an index are restricted because the selection condition must be on the search-key of the index.

3) (a) *Primary index-scan for equality:* This search retrieves a single record that satisfies the corresponding equality condition. The cost here can be calculated as:

Cost = Height traversed in index to locate the block pointer +1(block of the primary key is transferred for access).

(b) *Hash key:* It retrieves a single record in a direct way thus, cost in hash key may also be considered as Block transfer needed for finding hash target +1

4) *Primary index-scan for comparison:* Assuming that the relation is sorted on the attribute(s) that are being compared, (< , > etc.), then we need to locate the first record satisfying the condition after which the records are scanned forward or backward as the condition may be, displaying all the records. Thus cost in this case would be:

Cost = Number of block transfer to locate the value in index + Transferring all the blocks of data satisfying that condition.

Please note we can calculate roughly (from the cost point of view) the number of blocks satisfying the condition as:

Number of values that satisfy the condition × average number of tuples per attribute value/blocking factor of the relation.

5) *Equality on clustering index* to retrieve multiple records: The cost calculations in this case are somewhat similar to that of algorithm (4).

6) (a) *Equality on search-key of secondary index:* Retrieves a single record if the search-key is a candidate key.

Cost = cost of accessing index + 1.

It retrieves multiple records if search-key is not a candidate key.

Cost = *cost of accessing index* + *number of records retrieved* (It can be very expensive).

Each record may be on a different block, thus, requiring one block access for each retrieved record (this is the worst case cost).

(b) *Secondary index, comparison:* For the queries of the type that use comparison on secondary index value ≥ a value, then the index can be used to find first index entry which is greater than that value, scan index sequentially from there till the end and also keep finding the pointers to records.

For the ≤ *type query* just scan leaf pages of index, also keep finding pointers to records, till first entry is found satisfying the condition.

In either case, retrieving records that are pointed to, may require an I/O for each record. Please note linear file scans may be cheaper if many records are to be fetched.

Implementation of Complex Selections

Conjunction: Conjunction is basically a set of AND conditions.

7) *Conjunctive selection using one index:* In such a case, select any algorithm given earlier on one or more conditions. If possible, test other conditions on these tuples after fetching them into memory buffer.

8) *Conjunctive selection using multiple-key index:* Use appropriate composite (multiple-key) index if they are available.

9) *Conjunctive selection by intersection of identifiers* requires indices with record pointers. Use corresponding index for each condition, and take the intersection of all the obtained sets of record pointers. Then fetch records from file if, some conditions do not have appropriate indices, test them after fetching the tuple from the memory buffer.

Disjunction: Specifies a set of OR conditions.

10) *Disjunctive selection by union of identifiers* is applicable if *all* conditions have available indices, otherwise use linear scan. Use corresponding index for each condition, take the union of all the obtained sets of record pointers, and eliminate duplicates, then fetch data from the file.

Negation: Use linear scan on file. However, if very few records are available in the result and an index is applicable on attribute, which is being negated, then find the satisfying records using index and fetch them from the file.

1.4 SORTING

Now we need to take a look at sorting techniques that can be used for calculating costing. There are various methods that can be used in the following ways:

1) Use an existing applicable ordered index (e.g., B+ tree) to read the relation in sorted order.

2) Build an index on the relation, and then use the index to read the relation in sorted order. (Options 1&2 may lead to one block access per tuple).

3) For relations that fit in the memory, techniques like quicksort can be used.

4) For relations that do not fit in the memory, *external sort-merge* is a good choice.

Let us go through the algorithm for External Sort-Merge.

i) **Create Sorted Partitions:**

Let *i* be 0 initially.

Repeat steps (a) to (d) until the end of the relation:

(a) Read *M* blocks of relation into the memory. (Assumption M is the number of available buffers for the algorithm).

(b) Sort these buffered blocks using internal sorting.

(c) Write sorted data to a temporary file – temp (i)

(d) *i = i + 1;*

Let the final value of *i* be denoted by *N;*

Please note that each temporary file is a sorted partition.

ii) **Merging Partitions (N-way merge):**

// We assume (for now) that $N < M$.
// Use *N* blocks of memory to buffer temporary files and 1 block to buffer output.

Read the first block of each temporary file (partition) into its input buffer block;

Repeat steps (a) to (e) until all input buffer blocks are empty;

(a) Select the first record (in sort order) among all input buffer blocks;

(b) Write the record to the output buffer block;

(c) **If** the output buffer block is full then write it to disk and empty it for the next set of data. This step may be performed automatically by the Operating System;

(d) Delete the record from its input buffer block;

(e) **If** the buffer block becomes empty **then** read the next block (if any) of the temporary file into the buffer.

If $N \geq M$, several merge *passes* are required, in each pass, contiguous groups of $M - 1$ partitions are merged and a pass reduces the number of temporary files temp (i) by a factor of $M - 1$. For example, if M=11 and there are 90 temporary files, one pass reduces the number of temporary files to 9, each temporary file begin 10 times the size of the earlier partitions.

Repeated passes are performed till all partitions have been merged into one.

Figure 2 shows an example for external sorting using sort-merge.

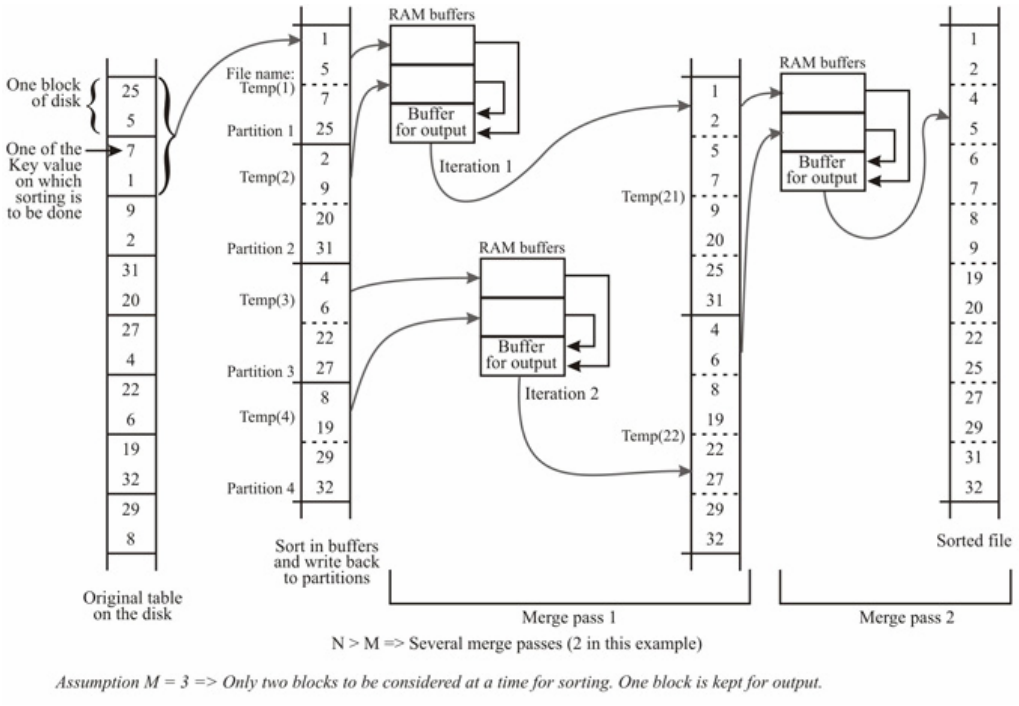


Figure 2: External merge-sort example

Cost Analysis:

Cost Analysis is may be performed, according to the following activities:

- Assume the file has a total of Z blocks.
- Z block input to buffers + Z block output – for temporary file creation.
- Assuming that N ≥ M, then a number of merge passes are required
- Number of merge passes = $\lceil \log_{M-1} (Z/M) \rceil$. Please note that of M buffers 1 is used for output.
- So number of block transfers needed for merge passes $=2\times Z (\lceil \log_{M-1} (Z/M) \rceil)$ as all the blocks will be read and written back of the buffer for each merge pass.
- Thus, the total number of passes for sort-merge algorithm $= 2Z + 2Z (\lceil \log_{M-1} (Z/M) \rceil) = 2Z \times (\lceil \log_{M-1} (Z/M) \rceil + 1)$.

☞ Check Your Progress 1

1) What are the basic steps in query processing?

……

……

……

2) How can the cost of query be measured?

……

……

……

3) What are the various methods adopted in select operation?

……

……

……

4) Define External-Sort-Merge algorithm.

……

……

……

1.5 JOIN OPERATION

There are number of algorithms that can be used to implement joins:

- Nested-loop join
- Block nested-loop join
- Indexed nested-loop join
- Merge-join
- Hash-join
- Complex join.

The choice of join algorithm is based on the cost estimate. Let us use the following information to elaborate the same:

MARKS (enroll no, subject code, marks): 10000 rows, 500 blocks
STUDENT (enroll no, name, dob): 2000 rows, 100 blocks

1.5.1 Nested-Loop Join

Let us show the algorithm for the given relations.

To compute the theta or equi-join

```
for each tuple s in STUDENT
{
        for each tuple m in MARKS
        {
                test s.enrolno = m.enrolno to see if they satisfy the join
                condition if they do, output joined tuple to the result.
        };
};
```

- In the nested loop join there is one *outer* relation and one *inner* relation.
- It does not use or require indices. It can be used with any kind of join condition. However, it is expensive as it examines every pair of tuples in the two relations.
- If there is only enough memory to hold one block of each relation, the number of disk accesses can be calculated as:

  For each tuple of STUDENT, all the MARKS tuples (blocks) that need to be accessed.

However, if both or one of the relations fit entirely in the memory, block transfer will be needed only once, so the total number of transfers in such a case, may be calculated as:

= Number of blocks of STUDENT + Number of blocks of MARKS
= 100 + 500 = 600.

If only the smaller of the two relations fits entirely in the memory then use that as the inner relation and the bound still holds.

Cost for the worst case:

Number of tuples of outer relation × Number of blocks of inner relation + Number of blocks of outer relation.

2000 ∗ 500 + 100 = 1,000,100 with STUDENT as outer relation.

There is one more possible bad case when MARKS is on outer loop and STUDENT in the inner loop. In this case, the number of Block transfer will be:

10000 ∗ 100+500 = 1,000,500 with MARKS as the outer relation.

1.5.2 Block Nested-Loop Join

This is a variant of nested-loop join in which a complete block of outer loop is joined with the block of inner loop.

The algorithm for this may be written has:

```
for each block s of  STUDENT
{
    for each block m of MARKS
    {
        for each tuple si in s
        {
            for each tuple mi in m
            {
                Check if (si and mi) satisfy the join condition
                        if they do output joined tuple to the result
            };
        };
    };
};
```

Worst case of block accesses in this case = Number of Blocks of outer relation (STUDENT) ×Number of blocks of inner relation (MARKS) + Number of blocks of outer relation (STUDENT).

Best case: Blocks of STUDENT + Blocks of MARKS

Number of block transfers assuming worst case:

100*500 + 100 = 50,100 (much less than nested-loop join)

Number of block transfers assuming best case:

400 + 100 = 500 (same as with nested-loop join)

Improvements to Block Nested-Loop Algorithm

The following modifications improve the block Nested method:

Use $M - 2$ disk blocks as the blocking unit for the outer relation, where M = memory size in blocks.

Use one buffer block to buffer the inner relation.

Use one buffer block to buffer the output.

This method minimizes the number of iterations.

1.5.3 Indexed Nested-Loop Join

Index scans can replace file scans if the join is an equi-join or natural join, and an index is available on the inner relation's join attribute.

For each tuple *si* in the outer relation STUDENT, use the index to look up tuples in MARKS that satisfy the join condition with tuple *si.*

In a worst case scenarios, the buffer has space for only one page of STUDENT, and, for each tuple in MARKS, then we should perform an index lookup on *MARKS index.*

Worst case: Block transfer of STUDENT+ number of records in STUDENT * cost of searching through index and retrieving all matching tuples for each tuple of STUDENT.

If a supporting index does not exist than it can be constructed as and when needed.

If indices are available on the join attributes of both STUDENT and MARKS, then use the relation with fewer tuples as the outer relation.

Example of Index Nested-Loop Join Costs

Compute the cost for STUDENT and MARKS join, with STUDENT as the outer relation. Suppose MARKS has a primary B+-tree index on enroll no, which contains 10 entries in each index node. Since MARKS has 10,000 tuples, the height of the tree is 4, and one more access is needed to the actual data. The STUDENT has 2000 tuples. Thus, the cost of indexed nested loops join as:

100 + 2000 * 5 = 10,100 disk accesses

1.5.4 Merge-Join

The merge-join is applicable to equi-joins and natural joins only. It has the following process:

1) Sort both relations on their join attribute (if not already sorted).
2) Merge the sorted relations to join them. The join step is similar to the merge stage of the sort-merge algorithm, the only difference lies in the manner in which duplicate values in join attribute are treated, i.e., every pair with same value on join attribute must be matched.

STUDENT

Enrol no	Name	-----
1001	Ajay	….
1002	Aman	…..
1005	Rakesh	…..
1100	Raman	…..
……	……..	……

MARKS

Enrol no	subject code	Marks
1001	MCS-011	55
1001	MCS-012	75
1002	MCS-013	90
1005	MCS-015	75
…….	……..	……

Figure 3: Sample relations for computing join

The number of block accesses:

Each block needs to be read only once (assuming all tuples for any given value of the join attributes fit in memory). Thus number of block accesses for merge-join is:

Blocks of STUDENT + Blocks of MARKS + the cost of sorting if relations are unsorted.

Hybrid Merge-Join

This is applicable only when the join is an equi-join or a natural join and one relation is sorted and the other has a secondary B+-tree index on the join attribute.

The algorithm is as follows:

> 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 the previous results, to replace addresses by the actual tuples. Sequential scan in such cases is more efficient than the random lookup method.

1.5.5 Hash-Join

This is applicable to both the equi-joins and natural joins. A hash function *h* is used to partition tuples of both relations, where *h* maps joining attribute (enroll no in our example) values to {0, 1, ..., *n-1*}.

The join attribute is hashed to the join-hash partitions. In the example of *Figure 4* we have used mod 100 function to hashing, and n = 100.

Error!

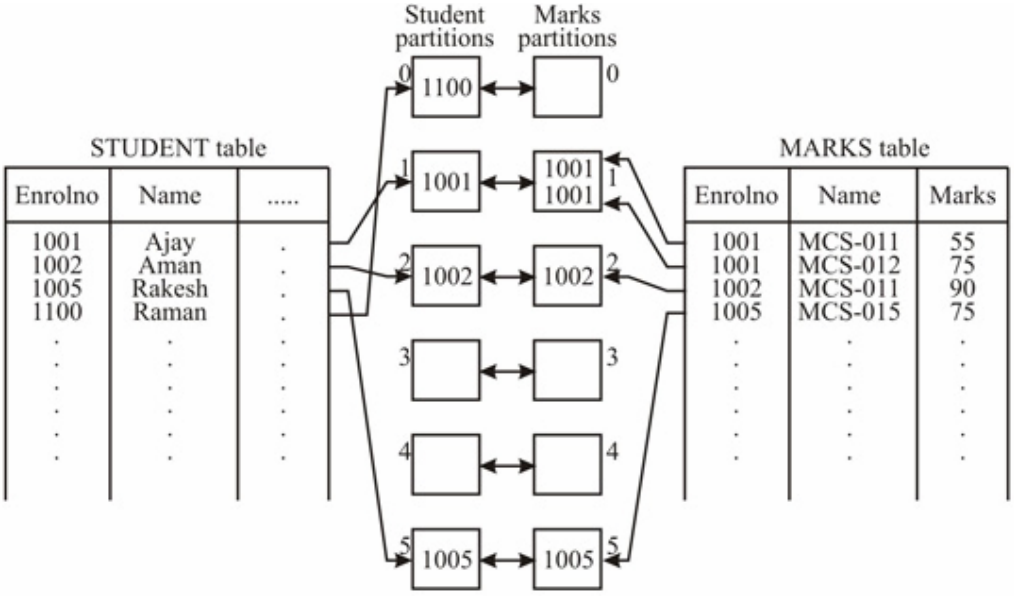


Figure 4: A hash-join example

Once the partition tables of STUDENT and MARKS are made on the enrolment number, then only the corresponding partitions will participate in the join as:

A STUDENT tuple and a MARKS tuple that satisfy the join condition will have the same value for the join attributes. Therefore, they will be hashed to equivalent partition and thus can be joined easily.

Algorithm for Hash-Join

The hash-join of two relations *r* and *s* is computed as follows:

- Partition the relation *r* and *s* using hashing function *h*. (When partitioning a relation, one block of memory is reserved as the output buffer for each partition).

- For each partition *si* of *s,* load the partition into memory and build an in-memory hash index on the join attribute.

- Read the tuples in *ri* from the disk, one block at a time. For each tuple in *ri* locate each matching tuple in *si* using the in-memory hash index and output the concatenation of their attributes.

In this method, the relation *s* is called the *build* relation and *r* is called the *probe* relation. The value *n* (the number of partitions) and the hash function *h* is chosen in such a manner that each *si* should fit in to the memory. Typically *n* is chosen as Number of blocks of s/Number of memory buffers] *f (M) where f is a "fudge factor", typically around 1.2. The probe relation partitions *ri* need not fit in memory.

Average size of a partition *si* will be less than *M* blocks using the formula for *n* as above thereby allowing room for the index. If the build relation *s* is very huge, then the value of *n* as given by the above formula may be greater than *M– 1* i.e., the number of buckets is > the number of buffer pages. In such a case, the relation *s* can be recursively partitioned, instead of partitioning *n* ways, use *M* – 1 partitions for *s and* further partition the *M* – 1 partitions using a different hash function. You should use the same partitioning method on *r.* This method is rarely required, as recursive partitioning is not needed for relations of 1GB or less for a memory size of 2MB, with block size of 4KB.

Cost calculation for Simple Hash-Join

(i) Cost of partitioning *r* and *s*: all the blocks of *r* and *s* are read once and after partitioning written back, so cost 1 = 2 (blocks of *r* + blocks of *s*).

(ii) Cost of performing the hash-join using build and probe will require at least one block transfer for reading the partitions
Cost 2 = (blocks of *r* + blocks of *s*)

(iii) There are a few more blocks in the main memory that may be used for evaluation, they may be read or written back. We ignore this cost as it will be too less in comparison to cost 1 and cost 2.
Thus, the total cost = cost 1 + cost 2
= 3 (blocks of *r* + blocks of *s*)

Cost of Hash-Join requiring recursive partitioning:

(i) The cost of partitioning in this case will increase to number of recursion required, it may be calculated as:

Number of iterations required = ([$\log_{M-1}$ (blocks of s] –1)

Thus, cost 1 will be modified as:

= 2 (blocks of *r* + blocks of *s*) × ([$\log_{M-1}$ (blocks of s)] –1)

The cost for step (ii) and (iii) here will be the same as that given in steps (ii) and (iii) above.

Thus, total cost = 2*(*blocks of *r* + blocks of *s)* ([log_{M-1}(blocks of *s*) – 1]) + (blocks of *r* + blocks of *s)*.

Because *s* is in the inner term in this expression, it is advisable to choose the smaller relation as the build relation. If the entire build input can be kept in the main memory, *n* can be set to 1 and the algorithm need not partition the relations but may still build an in-memory index, in such cases the cost estimate goes down to (Number of blocks *r* + Number of blocks of *s*).

Handling of Overflows

Even if *s* is recursively partitioned *hash-table overflow* can occur, i.e., some partition *si* may not fit in the memory. This may happen if there are many tuples in *s* with the same value for join attributes or a bad hash function.

Partitioning is said to be *skewed* if some partitions have significantly more tuples than the others. This is the overflow condition. The overflow can be handled in a variety of ways:

> *Resolution* (during the build phase): The overflow partition *s* is further partitioned using different hash function. The equivalent partition of *r* must be further partitioned similarily.
>
> *Avoidance* (during build phase): Partition build relations into many partitions, then combines them.

However, such approaches for handling overflows fail with large numbers of duplicates. One option of avoiding such problems is to use block nested-loop join on the overflowed partitions.

Let us explain the hash join and its cost for the natural join STUDENT ⋈ MARKS
Assume a memory size of 25 blocks ⇒ M=25;

SELECT build *s* as STUDENT as it has less number of blocks (100 blocks) and *r* probe as MARKS (500 blocks).

Number of partitions to be created for STUDENT = (block of STUDENT/M)* fudge factor (1.2) = (100/25) × 1.2 = 4.8

Thus, STUDENT relation will be partitioned into 5 partitions of 20 blocks each. MARKS will also have 5 partitions of 100 blocks each. The 25 buffers will be used as –20 blocks for one complete partition of STUDENT plus 4 more blocks for one block of each of the other 4 partitions. One block will be used for input of MARKS partitions.

The total cost = 3(100+500) = 1800 as no recursive partitioning is needed.

Hybrid Hash-Join

This is useful when the size of the memory is relatively large, and the build input is larger than the memory. Hybrid hash join keeps the first partition of the build relation in the memory. The first partition of STUDENT is maintained in the first 20 blocks of the buffer, *and not written to the disk*. The first block of MARKS is used right away for probing, instead of being written and read back. Thus, it has a cost of 3(80 + 400) + 20 +100 = 1560 block transfers for hybrid hash-join, instead of 1800 with plain hash-join.

Hybrid hash-join is most useful if *M* is large, such that we can have bigger partitions.

1.5.6 Complex Joins

A join with a conjunctive condition can be handled, either by using the nested loop or block nested loop join, or alternatively, the result of one of the simpler joins (on a few conditions) can be computed and the final result may be evaluated by finding the tuples that satisfy the remaining conditions in the result.

A join with a disjunctive condition can be calculated either by using the nested loop or block nested loop join, or it may be computed as the union of the records in individual joins.

1.6 OTHER OPERATIONS

There are many other operations that are performed in database systems. Let us introduce these processes in this section.

Duplicate Elimination: Duplicate elimination may be implemented by using hashing or sorting. On sorting, duplicates will be adjacent to each other thus, may be identified and deleted. An optimised method for duplicate elimination can be deletion of duplicates during generation as well as at intermediate merge steps in external sort-merge. Hashing is similar – duplicates will be clubbed together in the same bucket and therefore may be eliminated easily.

Projection: It may be implemented by performing the projection process on each tuple, followed by duplicate elimination.

Aggregate Function Execution: Aggregate functions can be implemented in a manner similar to duplicate elimination. Sorting or hashing can be used to bring tuples in the same group together, and then aggregate functions can be applied to each group. An optimised solution could be to combine tuples in the same group during part time generation and intermediate merges, by computing partial aggregate values. For count, min, max, sum, you may club aggregate values on tuples found so far in the group. When combining partial aggregates for counting, you would need to add up the aggregates. For calculating the average, take the sum of the aggregates and the count/number of aggregates, and then divide the sum with the count at the end.

Set operations ($\cup$, $\cap$ and —) can either use a variant of merge-join after sorting, or a variant of hash-join.

Hashing:

1) Partition both relations using the same hash function, thereby creating *r0, .., rn – 1* and *s0,.., sn – 1*

2) Process each partition *i* as follows: Using a different hashing function, build an in-memory hash index on *ri* after it is brought into the memory.

   $r \cup s$: Add tuples in *si* to the hash index if they are not already in it. At the end of *si* add the tuples in the hash index to the result.

   $r \cap s$: Output tuples in *si* to the result if they are already there in the hash index.
   r – s: For each tuple in *si,* if it is there in the hash index, delete it from the index.

   At end of *si* add remaining tuples in the hash index to the result.

There are many other operations as well. You may wish to refer to them in further readings.

☞ Check Your Progress 2

1) Define the algorithm for Block Nested-Loop Join for the worst-case scenario.

   ……

   ……

   ……

2) Define Hybrid Merge-Join.

   ……

   ……

   ……

3) Give the method for calculating the cost of Hash-Join?

……

……

……

4) Define other operations?

……

……

……

1.7 REPRESENTATION AND EVALUATION OF QUERY EXPRESSIONS

Before we discuss the evaluation of a query expression, let us briefly explain how a SQL query may be represented. Consider the following student and marks relations:

STUDENT (enrolno, name, phone)
MARKS (enrolno, subjectcode, grade)

To find the results of the student(s) whose phone number is '1129250025', the following query may be given.

SELECT enrolno, name, subjectcode, grade
FROM STUDENT *s*, MARKS *m*
WEHRE s.enrolno=m.enrolno AND phone= '1129250025'

The equivalent relational algebraic query for this will be:

$\pi_{\text{enrolno, name, subjectcode}} (\sigma_{\text{phone='1129250025'}} (\text{STUDENT}) \bowtie \text{MARKS})$

This is a very good internal representation however, it may be a good idea to represent the relational algebraic expression to a query tree on which algorithms for query optimisation can be designed easily. In a query tree, nodes are the operators and relations represent the leaf. The query tree for the relational expression above would be:

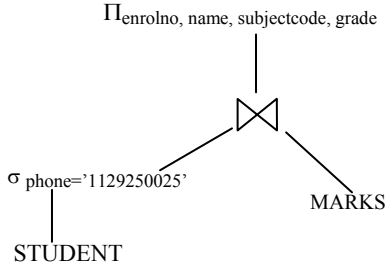


Figure 5: A Sample query tree

In the previous section we have seen the algorithms for individual operations. Now let us look at the methods for evaluating an entire expression. In general we use two methods:

- Materialisation
- Pipelining.

Materialisation: Evaluate a relational algebraic expression from the bottom-up, by explicitly generating and storing the results of each operation in the expression. For example, in *Figure 5* compute and store the result of the selection operation on STUDENT relation, then take the join of this result with MARKS relation and then finally compile the projection operation.

Materialised evaluation is always possible though; the cost of writing/reading results to/from disk can be quite high.

Pipelining

Evaluate operations in a multi-threaded manner, (i.e., passes tuples output from one operation to the next parent operation as input) even as the first operation is being executed. In the previous expression tree, it does not store (materialise) results instead, it passes tuples directly to the join. Similarly, does not store results of join, and passes tuples directly to the projection. Thus, there is no need to store a temporary relation on a disk for each operation. Pipelining may not always be possible or easy for sort, hash-join.

One of the pipelining execution methods may involve a buffer filled by the result tuples of lower level operation while, records may be picked up from the buffer by the higher level operation.

Complex Joins

When an expression involves three relations then we have more than one strategy for the evaluation of the expression. For example, join of relations such as:

STUDENT $\bowtie$ MARKS $\bowtie$ SUBJECTS
may involve the following three strategies:

Strategy 1: Compute STUDENT $\bowtie$ MARKS*; use* result to compute result $\bowtie$ SUBJECTS

Strategy 2: Compute MARKS $\bowtie$ SUBJECTS first, and then join the result with STUDENT

Strategy 3: Perform the pair of joins at the same time. This can be done by building an index of enroll no in STUDENT and on subject code in SUBJECTS.

For each tuple *m* in MARKS, look up the corresponding tuples in STUDENT and the corresponding tuples in SUBJECTS. Each tuple of MARKS will be examined only once.

Strategy 3 combines two operations into one special-purpose operation that may be more efficient than implementing the joins of two relations.

1.8 CREATION OF QUERY EVALUATION PLANS

We have already discussed query representation and its final evaluation in earlier section of this unit, but can something be done during these two stages that optimises the query evaluation? This section deals with this process in detail.

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

1) Generating logically equivalent expressions using **equivalence rules**
2) Annotating resultant expressions to get alternative query plans
3) Choosing the cheapest plan based on **estimated cost.**

The overall process is called **cost based optimisation.**

The cost difference between a good and a bad method of evaluating a query would be enormous. We would therefore, need to estimate the cost of operations and statistical information about relations. For example a number of tuples, a number of distinct values for an attribute etc. Statistics helps in estimating intermediate results to compute the cost of complex expressions.

Let us discuss all the steps in query-evaluation plan development in more details next.

1.8.1 Transformation of Relational Expressions

Two relational algebraic expressions are said to be **equivalent** if on every legal database instance the two expressions generate the same set of tuples (order of tuples is irrelevant).

Let us define certain equivalence rules that may be used to generate equivalent relational expressions.

Equivalence Rules

1) The conjunctive selection operations can be equated to a sequence of individual selections. It can be represented as:

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

2) The selection operations are commutative, that is,

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

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

$$\pi_{attriblist1}(\pi_{attriblist2}(\pi_{attriblist3} \ldots..(E) ..) = \pi_{attriblist1}(E)$$

4) The selection operations can be combined with Cartesian products and theta join operations.

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

and

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

5) The theta-join operations and natural joins are commutative.

$$E1 \bowtie_{\theta} E2 = E2 \bowtie_{\theta} E1$$

6) The Natural join operations are associative. Theta joins are also associative but with the proper distribution of joining condition:

$$(E1 \bowtie E2) \bowtie E3 = E1 \bowtie (E2 \bowtie E3)$$

7) The selection operation distributes over the theta join operation, under conditions when all the attributes in selection predicate involve only the attributes of one of the expressions ($E1$) being joined.

$$\sigma_{\theta 1}(E1 \bowtie_{\theta} E2) = (\sigma_{\theta 1}(E1)) \bowtie_{\theta} E2)$$

8) The projections operation distributes over the theta join operation with only those attributes, which are present in that relation.

$$\pi_{attrlist1 \cup attrlist2}(E1 \bowtie_{\theta} E2) = (\pi_{attrlist1}(E1) \bowtie_{\theta} \pi_{attrlist2}(E2))$$

9) The set operations of union and intersection are commutative. But set difference is not commutative.

$E1 \cup E2 = E2 \cup E$ and similarly for the intersection.

10) Set union and intersection operations are also associative.

$(E1 \cup E2) \cup E3 = E1 \cup (E2 \cup E3)$ and similarly for intersection.

11) The selection operation can be distributed over the union, intersection, and set-differences operations.

$$\sigma_{\theta 1}(E1 - E2) = ((\sigma_{\theta 1}(E1) - (\sigma_{\theta 1}(E2))$$

12) The projection operation can be distributed over the union.

$$\pi_{attriblist1}(E1 \cup E2) = \pi_{attriblist1}(E1) \cup \pi_{attriblist1}(E2)$$

The rules as above are too general and a few heuristics rules may be generated from these rules, which help in modifying the relational expression in a better way. These rules are:

(1) Combining a cascade of selections into a conjunction and testing all the predicates on the tuples at the same time:

$\sigma_{\theta 2}(\sigma_{\theta 1}(E))$ convert to $\sigma_{\theta 2 \wedge \theta 1}(E)$

(2) Combining a cascade of projections into single outer projection:

$\pi_4(\pi_3(\ldots\ldots(E))) = \pi_4(E)$

(3) Commutating the selection and projection or vice-versa sometimes reduces cost

(4) Using associative or commutative rules for Cartesian product or joining to find various alternatives.

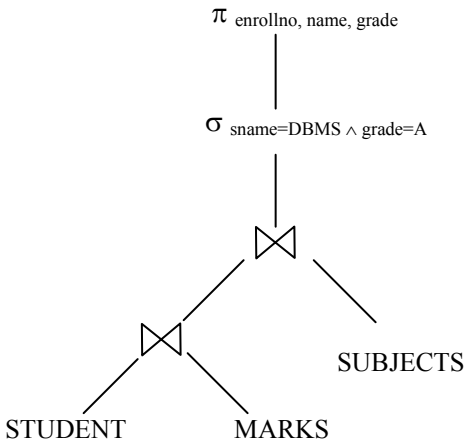
(5) Moving the selection and projection (it may have to be modified) before joins. The selection and projection results in the reduction of the number of tuples and therefore may reduce cost of joining.

(6) Commuting the projection and selection with Cartesian product or union.

Let us explain use of some of these rules with the help of an example. Consider the query for the relations:

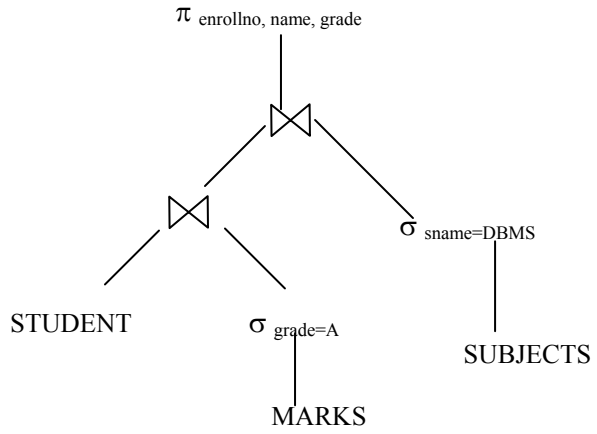
STUDENT (enrollno, name, phone)
MARKS (enrollno, subjectcode, grade)
SUBJECTS (subjectcode, sname)

Consider the query: Find the enrolment number, name, and grade of those students who have secured an A grade in the subject DBMS. One of the possible solutions to this query may be:

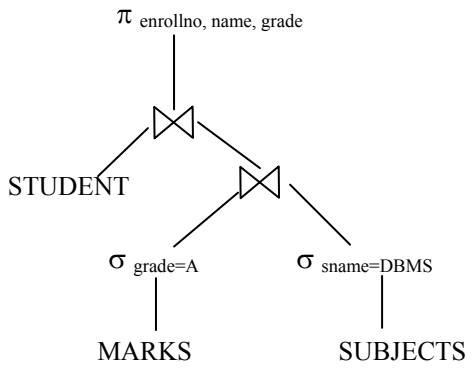


The selection condition may be moved to the join operation. The selection condition given in the *Figure* above is: *sname = DBMS and grade = A*. Both of these

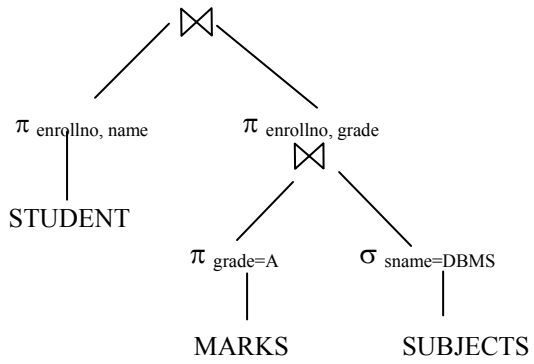
conditions belong to different tables, as *s name* is available only in the SUBJECTS table and *grade* in the MARKS table. Thus, the conditions of selection will be mapped accordingly as shown in the *Figure* below. Thus, the equivalent expression will be:



The expected size of SUBJECTS and MARKS after selection will be small so it may be a good idea to first join MARKS with SUBJECTS. Hence, the associative law of JOIN may be applied.



Finally projection may be moved inside. Thus the resulting query tree may be:



Please note the movement of the projections.

Obtaining Alternative Query Expressions

Query optimisers use equivalence rules to systematically generate expressions equivalent to the given expression. Conceptually, they generate all equivalent expressions by repeatedly executing the equivalence rules until no more expressions are to be found. For each expression found, use all applicable equivalence rules and add newly generated expressions to the set of expressions already found. However, the approach above is very expensive both in time space requirements. The heuristics rules given above may be used to reduce cost and to create a few possible but good equivalent query expression.

1.8.2 Query Evaluation Plans

Let us first define the term Evaluation Plan.

An evaluation plan defines exactly which algorithm is to be used for each operation, and how the execution of the operation is coordinated. For example, *Figure 6* shows the query tree with evaluation plan.

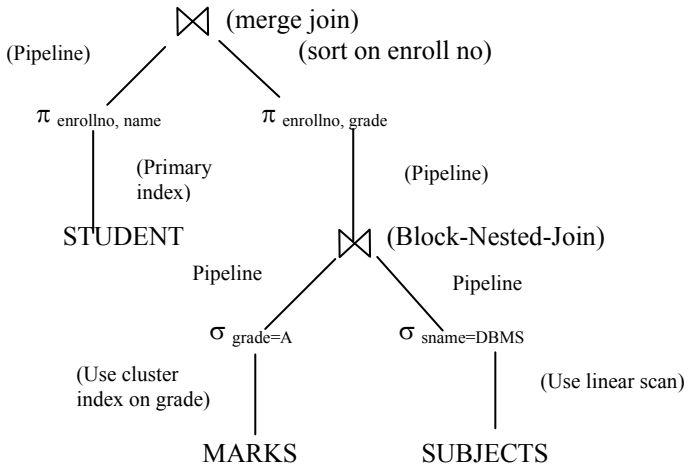


Figure 6: Query evaluation plan

1.8.3 Choice of Evaluation Plans

For choosing an evaluation technique, we must consider the interaction of evaluation techniques. Please note that choosing the cheapest algorithm for each operation independently may not yield best overall algorithm. For example, merge-join may be costlier than hash-join, but may provide a sorted output which reduces the cost for a higher level aggregation. A nested-loop join may provide opportunity for pipelining. Practical query optimisers incorporate elements of the following two broad approaches:

1) Searches all the plans and chooses the best plan in a cost-based fashion.
2) Uses heuristic rules to choose a plan.

1.8.4 Cost Based Optimisation

Cost based optimisation is performed on the basis of the cost of various individual operations that are to be performed as per the query evaluation plan. The cost is calculated as we have explained in the earlier section with respect to the method and operation (JOIN, SELECT, etc.).

1.8.5 Storage and Query Optimisation

Cost calculations are primarily based on disk access, thus, storage has an important role to play in cost. In addition, some of the operations also require intermediate storage thus; cost is further enhanced in such cases. The cost of finding an optimal query plan is usually more than offset by savings at query-execution time, particularly by reducing the number of slow disk accesses.

1.9 VIEWS AND QUERY PROCESSING

A view must be prepared, passing its parameters, which describe the query and the manner in which tuples should be evaluated. This takes the form of a pre-evaluation window, which gives the user of the program the ability to trade off memory usage for faster navigation, or an attempt to balance both of these resources.

The view may maintain the complete set of tuples following evaluation. This requires a lot of memory space, therefore, it may be a good idea to partially pre-evaluate it. This is done by hinting at how the number of that should be present in the evaluated view tuples before and after the current tuple. However, as the current tuple changes, further evaluation changes, thus, such scheme is harder to plan.

1.9.1 Materialised View

A materialised view is a view whose contents are computed and stored.

Materialising the view would be very useful if the result of a view is required frequently as it saves the effort of finding multiple tuples and totalling them up.

Further the task of keeping a materialised view up-to-date with the underlying data is known as materialised view maintenance. Materialised views can be maintained by recomputation on every update. A better option is to use incremental view maintenance, that is, where only the affected part of the view is modified. View maintenance can be done manually by defining triggers on insert, delete, and update of each relation in the view definition. It can also be written manually to update the view whenever database relations are updated or supported directly by the database.

1.9.2 Materialised Views and Query Optimisation

We can perform query optimisation by rewriting queries to use materialised views. For example, assume that a materialised view of the join of two tables *b* and *c* is available as:

$$a = b \ \ NATURAL\ JOIN \ \ c$$

Any query that uses natural join on b and c can use this materialised view '*a*' as:

Consider you are evaluating a query:

$$z = r\ NATURAL\ JOIN \ \ b \ \ NATURAL\ JOIN \ \ c$$

Then this query would be rewritten using the materialised view 'a' as:

$$z = r \ \ NATURAL\ JOIN \ \ a$$

Do we need to perform materialisation? It depends on cost estimates for the two alternatives viz., use of a materialised view by view definition, or simple evaluation.

Query optimiser should be extended to consider all the alternatives of view evaluation and choose the best overall plan. This decision must be made on the basis of the system workload. Indices in such decision-making may be considered as specialised views. Some database systems provide tools to help the database administrator with index and materialised view selection.

☞ Check Your Progress 3

1) Define methods used for evaluation of expressions?

……

……

……

2) How you define cost based optimisation?

……

……

……

3) How you define evaluation plan?

……

……

……

1.10 SUMMARY

In this unit we have discussed query processing and evaluation. A query in a DBMS is a very important operation, as it needs to be efficient. Query processing involves query parsing, representing query in alternative forms, finding the best plan of evaluation of a query and then actually evaluating it. The major query evaluation cost is the disk access time. In this unit, we have discussed the cost of different operations in details. However, an overall query cost will not be a simple addition of all such costs.

1.11 SOLUTIONS/ANSWERS

Check Your Progress 1

1) (a) In the first step scanning, parsing & validating is done
   (b) Translation into relational algebra
   (c) Parser checks syntax, verifies relations
   (d) Query execution engine take a query evaluation plan, executes it and returns answer to the query.

2) Query cost is measured by taking into account following activities:

   * Number of seeks * Number of blocks * Number of block written

   For simplicity we just use number of block transfers from disk as the cost measure. During this activity we generally ignore the difference in cost between sequential and random I/O and CPU/communication cost.

3) Selection operation can be performed in a number of ways such as:

File Scan	**Index Scan**
1. Linear Search	1. (a) Primary index, equality (b) Hash Key
2. Binary Search	2. Primary index, Comparison 3. Equality on clustering index 4. (a) Equality on search key of secondary index (b) Secondary index comparison

4) Algorithm for External Sort-Merge

   1. Create sorted partitions
      (a) Read M blocks of relation into memory
      (b) Write sorted data to partition R_i
   2. Merge the partitions (N-way Merge) until all input buffer blocks are empty.

Check Your Progress 2

1) For each block B_r of r {
   For each block B_s of s {
   For each tuple t_i in Br {
   For each tuple t_i in Bs {
   Test pair(t_i,s_i) to see if they satisfy the join condition

```
                    If they do, add the joined tuple to result
                        };
                    };
                };
            };
```

2) Hybrid Merge-Join is applicable when the join is equi-join or natural join and one relation is sorted.

   Merge the sorted relation with leaf of B+tree.

   (i) Sort the result on address of sorted relations tuples.
   (ii) Scan unsorted relation and merge with previous result.

3) Cost of Hash-join.
   (i) If recursive partitioning not required
   3(Blocks of r +blocks of s)
   (ii) If recursive partitioning required then
   2(blocks of r + blocks of s $(\lceil \log_{m-1}(\text{blocks of s}) \rceil - 1)$ + blocks of r+blocks of s

4) Other operations

   (a) Duplicate elimination
   (b) Projection
   (c) Aggregate functions.

Check Your Progress 3

1) Methods used for evaluation of expressions:

   (a) Materialisation
   (b) Pipelining

2) Cost based optimisation

   (a) Generalising logically equivalent expressions using equivalence rules
   (b) Annotating resultant expressions to get alternative query plans.
   (c) Choosing the cheapest plan based on estimated cost.

3) Evaluation plan defines exactly what algorithms are to be used for each operation and the manner in which the operations are coordinated.