Unit 5 Query Execution

Structure:

5.1 Introduction

In the previous unit, you studied query optimization and its various components such as query execution algorithm, heuristics in query optimization, semantic query optimization, multi-query optimization and applications. You also learned execution strategies for SQL sub queries and query processing for SQL updates. Now, you will study about query execution in this unit.

The query processor is the group of components of a DBMS that turns user queries and data-modification commands into a sequence of database operations and executes those operations. Since SQL lets us express queries at a very high level, the query processor must supply a lot of detail regarding how the query is to be executed.

In this unit, we will study query execution, that is, the algorithms that manipulate the data of the database. We shall cover the principal methods for execution of the operations of relational algebra. We shall introduce you to the basic building blocks of physical query plans. You will be also introduced to the more complex algorithms that implement operators of relational algebra efficiently; these algorithms also form a necessary part of physical query plans. You will also study about "iterators". Iterator is an object that enables a programmer to traverse a container.

Objectives:

After studying this unit, you should be able to:

- explain the physical-query-plan operators
- discuss the one-pass algorithm for database operations
- identify and demonstrate nested-Loop joins
- explain two-pass algorithms based on sorting and hashing
- discuss Index-based algorithms
- discuss buffer management
- demonstrate parallel algorithms for relational operations
- explain heuristics in query optimisation
- identify basic algorithms for executing query operations

5.2 Introduction to Physical-Query-Plan Operators

Physical query strategy is made from operators. Each of these operators implements one step of the plan. The physical operators are often specific implementations for one of the operators of the relational algebra, although we also require physical operators for other tasks that do not involve an operator of the relational algebra.

For instance, we often need to "scan" a table. In other words, we need to bring into the main memory, each tuple of some relation that is an operand of a relational-algebra expression.

5.2.1 Scanning tables

One of the most fundamental things that we can do in a physical query plan is to read the entire list of contents of a relation **R**. This step is especially necessary when we take the union or join of **R** with another relation. A variation of this operator includes an easy predicate. Here we read only those tuples of the relation **R** that suit the predicate. There are primarily two fundamental approaches for locating the tuples of a relation **R**. They are given below:

1. In several cases, there is an index on any attribute of **R**. We may be able to u**s**e this index to get all the tuples of **R**. For instance**,** a sparse index on **R** can be used to lead **u**s to all the blocks holding **R**, even if we don't kn**o**w which blocks these are. This operation is known as ***index-scan.***
2. In certain cases, the relation **R** is stored in an area of secondary memory with its tuples set in blocks. The blocks which contain the tuples of **R** are known to the system. It is possible to get the blocks one by one. This operation is known as ***table-scan.***

We shall resume the study of index-scan in section 5.7, where we will discuss the implementation of the operator. But an important observation for now is that we can use the index not only to get all the tuples of the relation it indexes, but also to get only those tuples that have a specific value (or a specific range of values) in the attribute or attributes that make the search key for the index.

5.2.2 Sorting while scanning tables

There can be several reasons behind sorting a relation as we read its tuples. One reason could be that the query could include an '**ORDER BY'** clause, (we will be using capital letters in some words to emphasise and highlight them.) requiring that a relation be sorted. Another reason is that different algorithms for relational-algebra operations need one or both the arguments to become sorted relations.

The physical-query-plan operator which is sort-scan gets a relation **R,** as well as a measurement of the attributes in which sort is to be made. It produces **R** in that sorted order. There are certain ways in which sort-scan can be implemented. They are given below:

1. In case **R** is too large to fit in main memory, then the multi-way merging approach is a good choice. However, rather than storing the final sorted **R** back on the disk, we can produce one block of the sorted **R** at a time, since its tuples are needed.
2. If we have to produce a relation **R** sorted by attribute **a**, and there is a B-tree index present on **a,** or **R** is stored as an indexed-sequential file ordered by a, then a scan of the index allows us the production of **R** in the desired order.

3. If the relation **R** that we want to retrieve in sorted order is sufficiently small to fit in the main memory, then we can retrieve its tuples by using a table scan or index scan. Then we can use one of the many possible main-memory sorting algorithms which are also efficient.

Significance of Iterators

When Iterators are composed within query plans, they support proficient execution. They contrast with a *materialization* strategy, where the result of each operator is produced entirely and not in parts. They are stored either on the disk or the main memory.

When iterators are used, many operations become active. Tuples pass among operators as needed. This reduces the need for storage.

Self Assessment Questions

1. The system knows the blocks containing the tuples of R, and it is not possible to get the blocks one by one. (True/ False)
2. We can use the index not only to get all the tuples of the relation it indexes, but also _________ .
3. It is open function that initiates the process of getting tuples, but it does not get a tuple. (True/ False)

Activity 1

In a group of four, analyze and explain how the index can be used not only to get all the tuples of the relation indexed by it as well as those tuples that possess a specific value in the attribute or attributes that make up the search key for the index.

5.3 One-Pass Algorithms for Database Operations

We are about to begin our study of one of the most important topics in query optimization: how to execute the individual steps - for instance, a join or selection - of a logical query plan?

The selection of an algorithm for every single operator is an essential element of the process of transformation of a logical query plan into a physical query plan. The proposed algorithms for operators largely fall into three classes:

1. Index-based technique which is explained in Section 5.7
2. Sorting-based technique which is covered in Section 5.5.

3. Hash-based technique which is described in Section 5.6 and Section 5.9, among other places.

Additionally, it is possible to divide algorithms for operators into the following degrees of cost and difficulty:

(a) There are some methods which work without a set limit on the size of the data. These methods use three or more passes to do their jobs, and are natural, recursive generalizations of the two-pass algorithms;

(b) Some methods involve reading the data only once from disk. These are the one-pass algorithms. They work when at least one of the arguments of the operation fits in main memory. But with selection and projection operations there are exceptions.;

There are certain methods that can be used for data that is too large for the available main memory but not for the largest possible data sets. An example of such an algorithm is the two-phase, multiway merge sort. These *two-pass* algorithms are described by reading data first time from the disk, processing it somehow, writing all or majority of it to the disk, and then again reading it for further processing through the second pass.

In this section, we shall concentrate on the one-pass methods. However, both in this section and subsequently, we shall classify operators into three broad groups:

1. ***Full-relation, binary operations:*** All other operations are in this class: set and bag versions of union, intersection, difference, joins, and products. Except for bag union, each of these operations requires at least one argument to be limited to size M, if we are to use a one-pass algorithm.
2. ***Tuple-at-a-time, unary operations:*** These operations require neither an entire relation, nor a large part of it, in memory at once. This enables us to read one block at a particular time, use the main memory buffer, and produce the output.
3. ***Fill-relation, unary operations:*** These one-argument operations need to consider all or most of the tuples in memory at once. So one-pass algorithms are limited to relations that are approximately of size *hl* (the number of main-memory buffers available) or less. The operations of this

class that we consider here are Y (the grouping operator) and S (the duplicate elimination operator).

One-pass algorithms for tuple-at-a-time operations

The tuple-at-a-time operations **(σ)** ***(R)*** and ***π(R)*** have observable algorithms, regardless of whether relation fits in main memory. You can understand the blocks of ***R*** individually into an input buffer; execute the operation on every tuple. Besides, you can move the selected tuples or the projected tuples to the output buffer. Refer to Figure 5.1 for the performance of selection or projection on relation R.

Since the output buffer may be an input buffer of some other operator or may be sending data to a user or application, we do not count the output buffer as needed space. Thus, we basically want ***M – 1*** for the input buffer irrespective of **B**, where M is available memory buffer/block, B is the block taken by each argument

The disk I/O necessity for this method depends on just how the argument relation R is given. If R is primarily on disk, then the cost is whatsoever it takes to execute a table-scan or index-scan of R.

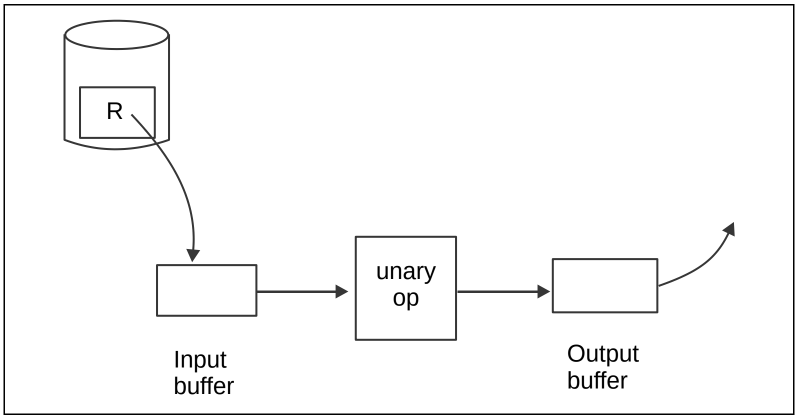


Figure 5.1: Performance of a Selection or Projection on Relation R

Self Assessment Questions

4. The selection of an algorithm for each operator is one of the most fundamental elements of the process of transformation of a logical query plan into a physical query plan. (True/ False)
5. Tuple-at-a-time, unary operations require neither _____________ nor _________.

Activity 2
If the output of the operation can be stored on full cylinders, we waste almost no time writing. Analyze what you have understood from this statement. Explain with an example.

5.4 Nested-Loop Joins

Before proceeding to the more complex algorithms in the next sections, we shall turn our attention to a family of algorithms for the join operator called "nested loop" joins.

These algorithms are, in a sense, "one-and-a-half" passes, since in each difference one of the two arguments includes its tuples read only once. Contrary to this, the other argument will be read repeatedly.

5.4.1 Tuple-based nested-loop join

The effortless variation of nested-loop join has loops that range over single tuples of the relations concerned. In this algorithm, which we call, tuple-based nested-loop join, we calculate the join R(X, Y) ⋈ S(Y, Z) as below:

```
    For each tuple s in S DO
    For each tuple r in R DO
    If r and s join to make a tuple t
Then
Output is t
```

S is called the outer relation and R the inner relation of the join. One buffer is for outer relation and one buffer for inner relation. Then the I/O cost of this algorithm is T(R)T(S) disk. It is expensive since it examines every pair of tuples in the two relations. However, there are many situations where this algorithm can be modified to have much lower cost. A next development looks much more cautiously at the way tuples of R and S are split between blocks, and utilizes as much of the memory as it can to decrease the number of disk I/O's as we go through the inner loop.

5.4.2 Iterator for a tuple-based nested-loop join

One benefit of a nested-loop join is that it fits well into an iterator structure. It prevents us from storing intermediate relations on disk in some situations. The iterator for $R \bowtie S$ is easy to build from the iterators for R and S, which support functions R. Open ( ). The code for the three iterator functions for nested-loop join is in Figure 5.2. It makes the assumption that neither relation R nor *S* is empty.

```
Open () {
    R. Open () ;
    S. Open () ;
    s : = S. GetNext () ;
}
GetNext () {
    REPEAT {
        r : = R. GetNext () ;
        IF (r = NotFound) { /* R is exhausted for the current s */
                R. Close () ;
                s : = S. GetNext () ;
                IF (s = NotFound) RETURN NotFound ;
                    /* both R and S are exhausted */
                R. Open () ;
                r : = R. GetNext () ;
            }
    }
    UNTIL (r and s join) ;
    RETURN the join of r and s ;
}
Close () {
    R. Close () ;
    S. Close () ;
}
```

Figure 5.2: Iterator Functions for Tuple-based Nested-loop Join of R and S

The functions used in Figure 5.2 as defined below:

Open():

R.Open() will initialize a main-memory structure to represent a set of tuples of R.

GetNext():

REPEAT R.GetNext() will continue until tuple r is not returned.

Close():

R.Close() will release the memory.

Self Assessment Questions

6. _________ joins can be used for relations of any size. One relation does not need to necessarily fit in the main memory.
7. Nested-loop does not allow us to avoid storing intermediate relations on disk in some situations. (True/ False)

5.5 Two-Pass Algorithms based on Sorting

We shall now begin the study of multi-pass algorithms for performing relational algebra operations on relations that are larger than what the one-pass algorithms of Section 5.3 can handle. We focus on two-pass algorithms, where data from the operand relations is examine into main memory, processed in somehow written out to disk again and then reread from disk to complete the operation.

We can naturally extend this idea to any number of passes, where the data is read many times into main memory. However we concentrate on two-pass algorithms because:

1. Two passes are usually enough, even for very large relations
2. Generalizing to more than two passes is not hard

In this section, we consider sorting as a tool for implementing relational operations. The fundamental idea is below. If we have a big relation **R**, where ***B(R)*** is greater than ***M***, the number of memory buffers available, then we can continually:

1. Read ***h1*** blocks of R into the main memory.
2. Sort the **Y** blocks in the main memory, using a main-memory sorting algorithm which is also efficient. Such an algorithm will take an amount

of processor time that is just slightly more than linear in the number of tuples in main memory. So we expect that the time to sort will definitely not exceed the disk I/O (input/output) time for step (1).

3. Finally, write the sorted list into M blocks of disk. We shall refer to the contents of these blocks as one of the sorted sublists of *R.*

   All the algorithms we shall discuss then use a second pass to "merge" the sorted sublists in some way to execute the desired operator.

Self Assessment Questions

8. In _________ algorithms, data is read into main memory from the operand relations.
9. In the second pass, all the sorted sublists are _________.

5.6 Two-Pass Algorithms Based on Hashing

There is a family of hash-based algorithms that attack the same problems as in Section 5.5. The essential idea behind all these algorithms is as follows. If the data is extremely big to store in main-memory buffers, hash all the tuples of the argument or arguments using an appropriate hash key. For all the general operations, there is a way to select the hash key so all the tuples that require to be measured together when we perform the operation have the same hash value.

We then perform the operation by working on one bucket at a time (or on a pair of buckets with the same hash value in the case of a binary operation). In fact we have decreased the size of the operand(s) by a factor equivalent to the number of buckets.

If there are M buffers available, then we can pick M as the number of buckets. This helps in gaining a factor of M in the size of the relations that we can handle easily. Notice that the sort-based algorithms of Section 5.5 also gain a factor of M by pre-processing, although the sorting and hashing approaches achieve their similar gains by rather different means.

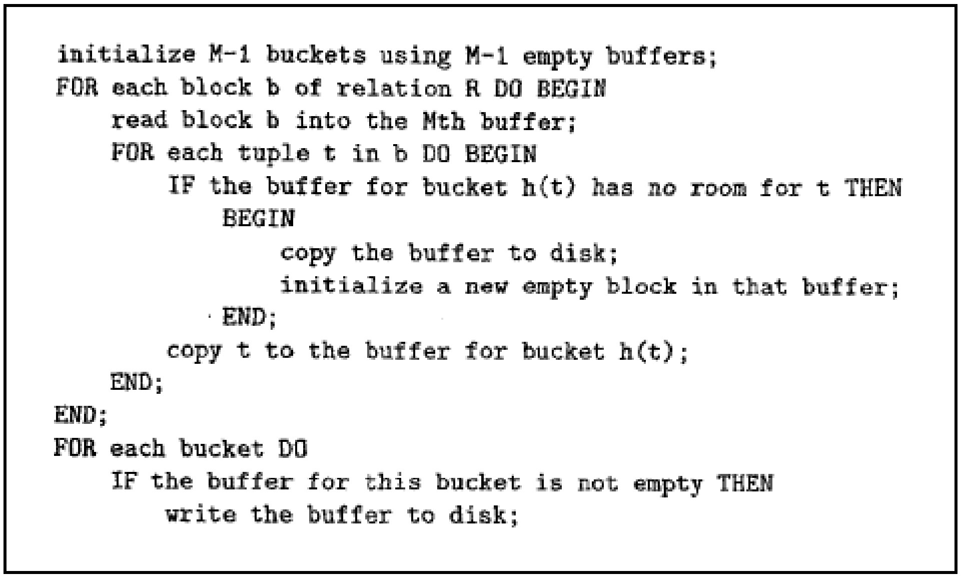
Partitioning Relations by Hashing: To begin, let us review the way we would take a relation R by using M buffers and partitioning R into M - 1 buckets of roughly equal size.

We shall assume that h is the hash function, and that h takes complete tuples of R as its argument (i.e., all attributes of R are part of the hash key).

We connect one buffer with every bucket. The last buffer holds blocks of R, individually.

Every tuple *t* in the block is hashed to bucket *h (t)* and copied to the suitable buffer. But if that buffer is full, we write it out to disk, and initialize one more block for the similar bucket. Finally, we write out the final block of each bucket if it is not empty.

The algorithm is given in more detail in Figure 5.3. Note that it assumes that tuples, while they may be variable-length, are never too large to fit in an empty buffer.



```
initialize M-1 buckets using M-1 empty buffers;
FOR each block b of relation R DO BEGIN
    read block b into the Mth buffer;
    FOR each tuple t in b DO BEGIN
        IF the buffer for bucket h(t) has no room for t THEN
            BEGIN
                copy the buffer to disk;
                initialize a new empty block in that buffer;
            END;
        copy t to the buffer for bucket h(t);
    END;
END;
FOR each bucket DO
    IF the buffer for this bucket is not empty THEN
        write the buffer to disk;
```

Figure 5.3: Partitioning a Relation R into M - 1 Buckets

Self Assessment Questions

10. If there are M buffers available and we can pick M as the number of buckets, we can gain a factor of M in the size of the relations that we can handle. (True/ False)
11. The essential idea behind all hash-based algorithms is _________.

5.7 Index-Based Algorithms

The existence of an index on one or more attributes of a relation makes available some algorithms that would not be feasible without the index. Index-based algorithms are especially useful for the selection operator.

However, algorithms for join and other binary operators also use indexes to very good advantage.

In this section, we shall introduce these algorithms. We also continue with the discussion of the index-scan operator for accessing a stored table with an index that we began in Section 5.2.1. To appreciate many of the issues, we first need to study "clustering" indexes.

Clustering and non-clustering indexes

A relation is said to be 'clustered' if its tuples are packed into the least blocks that can possibly hold those tuples. All the analysis we have done so far assume that relations are clustered.

We may also speak of clustering index, which are indexes on an attribute or else attributes such that all the tuples through a fixed value for the search key of this index appear on approximately as few blocks as can hold them.

It is noteworthy that a relation that is not clustered cannot have a clustering index, but a clustered relation can also have non-clustering indexes. (See Figure 5.4)

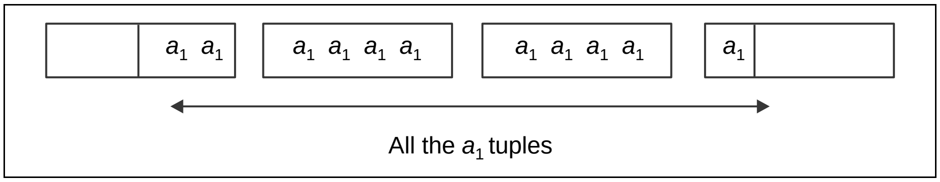


Figure 5.4: A Clustering Index having all the Tuples with a Fixed Value Packed into (close to) the Minimum Possible Number of Blocks

Self Assessment Questions

12. The existence of an index on one or more attributes of a relation makes available some algorithms that _________.
13. Index-based algorithms are extremely useful for the selection operator. (True/ False)

5.8 Buffer Management

We have assumed that operators on relations have some number M of main-memory buffers that they can utilize to store required data. Actually, these buffers are not often allocated in advance to the operator, as well as the value of M might differ, depending on system conditions.

The essential task of creating main-memory buffers accessible to processes, for example, queries that act on the database is given to the buffer manager.

It is the responsibility of the buffer manager to let processes get the memory that they need for reduction of the delayed and unsatisfiable requests. The role of the buffer manager is illustrated in Figure 5.5.

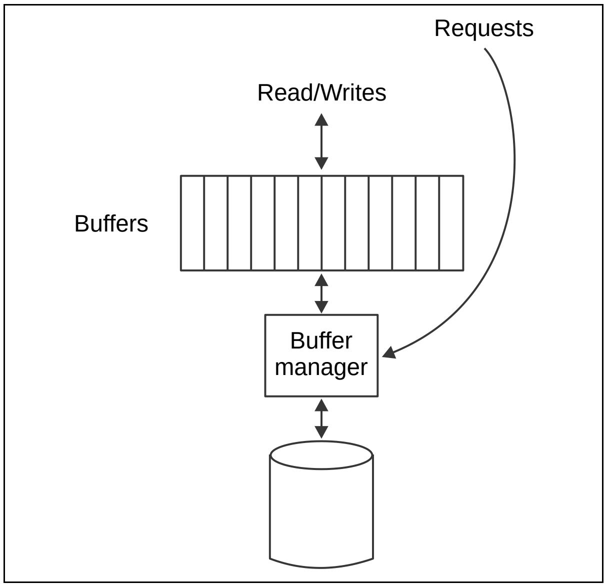


Figure 5.5: Role of Buffer Manager

Buffer management architecture

Buffer management architectures are broadly divided into two main categories:

1. In most of the relational database management system, the buffer manager controls main memory directly
2. The buffer manager allocates buffers in virtual memory. It permits the operating system to decide which buffers are actually in main memory at any time and which are in the “swap space" on disk that the operating system manages. Many main memory DBMSs and “object-oriented" DBMSs operate this way.

Whichever approach a DBMS uses, the same problem arises: how to fit the number of buffers into the available main memory. The buffer manager should try to limit the number of buffers in usage so that they can fit in it.

When the buffer manager controls main memory directly, and requests exceed available space, it has to select a buffer to empty, by returning its contents to disk. If the buffered block has not been changed, then it may simply be erased from main memory, but if the block has changed it must be written back to its place on the disk.

When the buffer manager allocates space in virtual memory, it has the option to allocate more buffers than can fit in main memory. However, if all these buffers are really in use, then there will be "thrashing," a common operating-system problem, where many blocks are moved in and out of the disk's swap space. In this situation, the system spends the majority time in swapping blocks, whereas very little useful work gets done.

Normally, when the Database Management System is initialized then several buffers parameter are set. We would expect that this number is set so that the buffers occupy the available main memory, regardless of whether the buffers are allocated in main or virtual memory.

Self Assessment Questions

14. The buffers are rarely allocated in advance to the _________ , and the value of M may vary depending on system conditions.
15. If the buffered block has not been changed, then it may simply be erased from _________.

5.9 Parallel Algorithms for Relational Operations

Database operations, frequently being time-consuming and involving a lot of data, can generally profit from parallel processing. In this section, we shall review the principal architectures for parallel machines. We then concentrate on the "shared-nothing" architecture, which appears to be the most cost effective for database operations, although it may not be superior for other parallel applications.

There are simple modifications of the standard algorithms for most relational operations that will exploit parallelism almost perfectly. That is, the time to complete an operation on a ***p*** processor machine is about l/p of the time it takes to complete the operation on a uni-processor.

Models of parallelism

You can say that collection of processors is the heart of parallel machines. Often the number of processors ***p*** is large, in the hundreds or thousands.

We shall assume that each processor has its own local cache, which we do not show explicitly in our diagrams.

In most organizations, each processor also has local memory, which we do show. Of great importance to database processing is the fact that along with these processors are many disks, perhaps one or more per processor or in some architecture a large collection of disks accessible to all processors directly.

Additionally, parallel computers all have some communications facility for passing information among processors. In our diagrams, we show the communications as if there were a shared bus for all the elements of the machine.

However, in practice a bus cannot interconnect as many processors or other elements as are found in the largest machines. So the interconnection system is, in much architecture, a powerful switch, perhaps augmented by busses that connect subsets of the processors in local *clusters.*

The three most important classes of parallel machines are:

1. ***Shared Memory:*** In this architecture, as illustrated in Figure 5.6, each processor has access to all the memory of all the processors. That is, there is a single physical address space for the entire machine, rather than one address space for each processor.

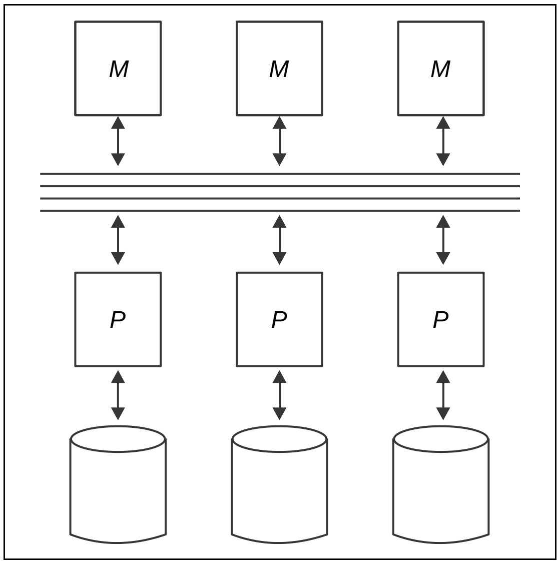


Figure 5.6: A Shared-Memory Machine

The diagram given in Figure 5.6 is, in fact too extreme, signifying that processors have no private memory at all. Rather, each processor has some local main memory, which it typically uses whenever it can.

However, it has direct access to the memory of other processors when it needs to. Large machines of this class are of the *NUMA* (non uniform memory access) kind, meaning that it takes rather more time for a processor to access data in a memory that "belongs" to some other processor than it does to access its "own" memory, or the memory of processors in its local cluster.

However, the difference in memory-access times is not great in current architectures. Rather, all memory accesses, no matter where the data is, take much more time than a cache access. So the critical issue is whether or not the data a processor needs is in its own cache.

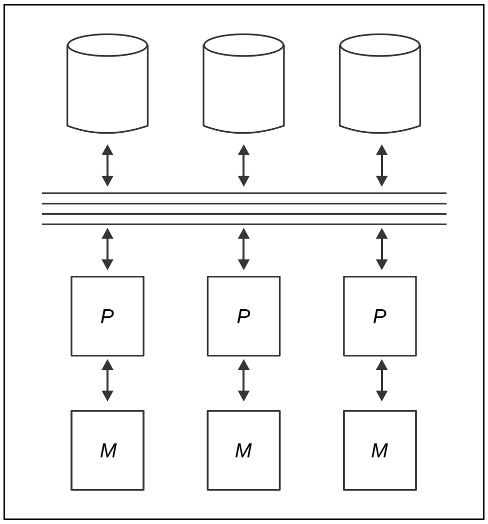


Figure 5.7: A Shared-Disk Machine

2. ***Shared Disk:*** In this architecture, as shown in Figure 5.7; every processor has its own memory, which is not accessible directly from other processors. However, the disks are accessible from any of the processors through the communication network.

   Disk controllers manage the potentially competing requests from different processors. The number of disks stored and processors need not be identical.

3. ***Shared Nothing:*** Here, all processors have their own memory and their own disk or disks, as in Figure 5.8. All communication is via communication network, from processor to processor. For example, if one processor p wants to read tuples from the disk of another Processor Q, then processor P sends a message to Q asking for the data. Then, Q obtains the tuples from its disk and ships them over the network in another message, which is received by P.

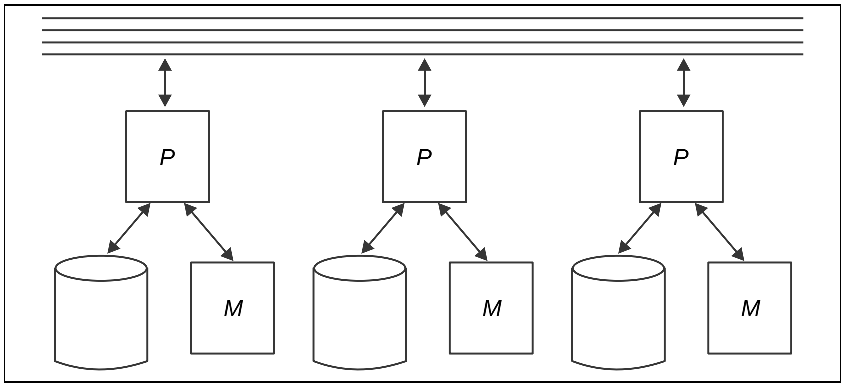


Figure 5.8: A Shared-Nothing Machine

The shared-nothing architecture is the most commonly used architecture for high-performance database systems.

Shared nothing machinery is comparatively economical to make, however when we design algorithms for these machines we should be aware that it is costly to send data from one processor to another.

Normally, data must be sent between processors in a message, which has considerable overhead associated with it. Both processors must execute a program that supports the message transfer, and there may be contention or delays associated with the communication network as well.

Usually, the value of a message can be broken down into a large fixed overhead and a small amount of time per byte transmitted. Thus, there is an important advantage to designing a parallel algorithm so that communication among processors includes large amounts of data sent at once.

For instance, we might buffer several blocks of data at processor P all bound for processor Q. If Q does not need the data immediately, it may be much more efficient to wait until we have a long message at P and then send it to Q.

Self Assessment Questions

16. The disks are accessible from any of the processors through the _________ network.
17. The number of disks stored and processors need not be identical. (True/ false)

5.10 Using Heuristics in Query Optimisation

One of the chief heuristic rules is that before applying the **JOIN** or other binary operations, one must apply **SELECT** and **PROJECT** operations.

- A query tree is a tree data structure. Its purpose is to communicate to a relational algebra expression. It symbolises the relational algebra operations as internal nodes, and signifies the input relations of the query as leaf nodes of the tree.
- An implementation of the query tree includes execution of an internal node operation when its operands are accessible and then swapping that internal node with the relation which results from the execution of the operation.
- The execution ends at the execution of the root node. The output is the result relation for the query.

Heuristic Optimization of Query Trees

- Different relational algebra expressions can be equivalent. In other words, they can correspond to the same query.
- A standard initial query tree is made by the query parser.
- This first query tree is then transformed by the heuristic query optimiser into a final query tree that is efficient to execute.

SELECT NAME FROM

EMPLOYEE, WORKS_ON, PROJECT WHERE PNAME='Aquarius' AND ESSN=SSN AND BDATE > '1-DEC-56'AND PNUMBER=PNO;

For Execution of this query, we do not require creating a huge file containing the CARTESIAN PRODUCT of the whole EMPLOYEE, PROJECT, and WORKS_ON files. This query basically requires a single record from the PROJECT relation and only the employee records for those whose date of birth is after '1-DEC-56'.

The basic outline of a Heuristic Algebraic Optimization Algorithm is given below:

1. Break up any SELECT operations by means of conjunctive operations into a cascade of SELECT operations.
2. Move each and every SELECT operation as far down the query tree as is allowed by the attributes which are involved in the select condition.
3. Rearrange the leaf nodes of the tree by the following criteria:
   - Position the leaf node relation with the most limiting SELECT operations so that they are executed first in the query representation.
   - Ensure that the ordering of leaf nodes does not cause CARTESIAN PRODUCT operation.
4. Through a subsequent SELECT operation, combine a CARTESIAN PRODUCT operation in the tree into a JOIN operation, but the condition should signify a join condition.
5. Break down and move lists of projection attributes down the tree as much as possible by creating new PROJECT operations as desired.
6. Recognize those sub-trees that stand for those groups of operations that can be implemented by only one algorithm.

Self Assessment Questions

18. The _________ terminates at the execution of the root node. This makes the result relation for the query.
19. The first query tree is transformed by the heuristic query optimizer into a final query tree that is efficient to execute. (True/ False)

5.11 Basic Algorithm for Executing Query Operations

External Sorting is the essential algorithm for implementation of query operation. Sorting is one of the main algorithms used in query processing (an example is ORDER BY- clause which requires a sorting).External sorting is appropriate for huge files of records stored on disk that do not fit completely in main memory.

A sort-merge approach is utilized by the usual external sorting algorithm. The algorithm consists of two phases. They are given below:

1. Sorting Phase
2. Merging Phase

Implementing the SELECT Operation

Many search algorithms are feasible for selecting records from a file. The following search techniques are available:

- Linear search (brute force)
- Binary search
- Using a primary index (or hash function)
- Using a primary index to retrieve multiple records
- Using a clustering index to retrieve multiple records
- Implementing the JOIN Operation
- Using a secondary (B^+-tree) index on an equality comparison

The JOIN operation is one of the lengthiest operations in query processing. Most common methods for performing a join are:

1. Nested-loop join (brute force)
2. Sort-merge join
3. Single-loop join (using an access structure to retrieve the matching records)
4. Hash-join

Implementing PROJECT as well as Set Operations

- Implementation of a PROJECT operation is easy if attribute lists includes a key of relation R.
- If attribute list does not contain a key of R, duplicate tuples must be eliminated.
- Set operations (u, n,-,x) are at times expensive to implement. The Cartesian product operation is especially quite expensive.
- Since union, intersection, set difference apply only to union–compatible relations, their implementation can be done by using certain variations of the sort-merge technique.
- Hashing can moreover be utilised to execute UNION, INTERSECTION, and SET DIFFERENCE.

Implementing Aggregate Operations

- The aggregate operations (MAX, MIN, SUM, AVERAGE, COUNT), when applied to an entire table, can be computed by a table scan or else by using an appropriate index. For example:

  ```
  SELECT MAX (SALARY) FROM EMPLOYEE;
  ```

If an ascending index on salary exists for the EMPLOYEE relation, it can be utilized (or else we can scan the entire table).

- when a GROUP The index can also be used for computing the SUM, AVERAGE, and COUNT aggregates. However, the index must be dense, i.e. there must bean index entry for each and every record in the main file.
- The aggregate operator must be applied independently to each group of tuples BY clause is used in a query.

Self Assessment Questions

20. The index can be used for _________.
21. _________ can be used to implement INTERSECTION, UNION and SET DIFFERENCE.

5.12 Summary

Let us recapitulate the important points discussed in this unit:

- The principal methods for execution of the operations of relational algebra are; scanning, hashing, sorting, and indexing are the major approaches.
- One reason is that various algorithms for relational-algebra operations require either one or both of their arguments to be known as sorted relations.
- Another reason is that the query could include an ORDER BY clause, requiring that a relation should be sorted.
- Iterators support proficient execution when they are composed within the query plans.
- We have assumed that operators on relations have available some number M of main-memory buffers that can be used to store the needed data.
- The basic algorithm for execution of query operation is External Sorting.
- External sorting is suitable for huge files of records stored on disk that do not fit entirely in main memory.
- The number of buffers is a parameter set when the DBMS is initialised.

5.13 Glossary

- ***Iterators:*** An object that enables a programmer to traverse a container.

- ***NUMA:*** Non Uniform Memory Access. It is a computer memory design used in multiprocessing, where the memory access time depends on the memory location relative to a processor.
- ***Scanning tables:*** A variation of this operator involves a simple predicate, where we read only those tuples of the relation R that satisfy the predicate.
- ***Shared disk:*** The disks are accessible from any of the processors through the communication network.
- ***Table-scan:*** Those blocks which contain the tuples of R are known to the system.
- ***Two-pass algorithms:*** Data from the operand relations is read into main memory and processed in some way written out to disk again. This data is then read again from the disk to complete the operation.

5.14 Terminal Questions

1. Explain the physical query plan operators.
2. Discuss the One-Pass algorithm for database.
3. What is buffer management?
4. Describe the most important classes of parallel machines.

5.15 Answers

Self Assessment Questions

1. False
2. Secondary
3. Open function
4. True
5. False
6. Nested-loop
7. False
8. merged
9. Two-pass
10. True
11. Last
12. non-clustering
13. False
14. Operator

15. Main memory
16. Communication
17. True
18. Execution
19. True
20. JOIN
21. Hashing

Terminal Questions

1. Physical query plans are built from operators. Every single operator implements one step of the plan. Refer Section 5.2 for more details.
2. One-pass algorithms read the data only once from disk. Refer Section 5.3 for more details.
3. Buffer manager allows processes to get the memory they need, while the unsatisfiable and delayed requests are minimized. Refer Section 5.8 for more details.
4. Shared memory, shared disk, etc. are the important phases of parallel machines. Refer Section 5.9 for more details.

References:

- Raghu Ramakrishnan, Johannes Gehrke, *Database Management Systems,* (3rdEd.), McGraw-Hill, Higher Education
- Peter Rob, Carlos Coronel, *Database Systems: Design, Implementation, and Management*, (7thEd.), Thomson Learning
- Silberschatz, Korth, Sudarshan, *Database System Concepts, (*4th Ed.), McGraw-Hill
- Elmasari Navathe, *Fundamentals of Database Systems, (*3rdEd.), Pearson Education Asia

E-references:

- www.wisegeek.com
- www.dbms.edu.in
- www.neilconway.org/docs/dbms_notes.pdf
- www.unixspace.com/context/