Mini Project : 3

Title - Implement Parallelization of Database Query optimization

Theory -

Query processing is the process through which a Database Management System (DBMS) parses, verifies, and optimizes a given query before creating low-level code that the DB understands.

Query Processing in DBMS, like any other High-Level Language (HLL) where code is first generated and then executed to perform various operations, has two phases: compile-time and runtime.

Query the use of declarative languages and query optimization is one of the main factors contributing to the success of RDBMS technology. Any database allows users to create queries to request specific data, and the database then uses effective methods to locate the requested data.
A database optimization approach based on CMP has been studied by numerous other academics. But the majority of their effort was on optimizing join operations while taking into account the L2-cache and the parallel buffers of the shared main memory.
The following techniques can be used to make a query parallel

- I/O parallelism
- Internal parallelism of queries
- Parallelism among queries
- Within-operation parallelism
- Parallelism in inter-operation

I/O parallelism :

This type of parallelism involves partitioning the relationships among the discs in order to speed up the retrieval of relationships from the disc.
The inputted data is divided within, and each division is processed simultaneously. After processing all of the partitioned data, the results are combined. Another name for it is data partitioning.
Hash partitioning is best suited for point queries that are based on the partitioning attribute and have the benefit of offering an even distribution of data across the discs.
It should be mentioned that partitioning is beneficial for the sequential scans of the full table stored on "n" discs and the speed at which the table may be scanned. For a single disc system, relationship takes around 1/n of the time needed to scan the table. In I/O parallelism, there are four different methods of partitioning:

Hash partitioning :

A hash function is a quick mathematical operation. The partitioning properties are hashed for each row in the original relationship.
Let's say that the data is to be partitioned across 4 drives, numbered disk1, disk2, disk3, and disk4. The row is now stored on disk3 if the function returns.

Range partitioning :
Each disc receives continuous attribute value ranges while using range partitioning. For instance, if we are range partitioning three discs with the numbers 0, 1, and 2, we may assign a relation with a value of less than 5 is written to disk0, numbers from 5 to 40 are sent to disk1, and values above 40 are written to disk2.
It has several benefits, such as putting shuffles on the disc that have attribute values within a specified range.

Round-robin partitioning :
Any order can be used to study the relationships in this method. It sends the ith tuple to the disc number (i% n).
Therefore, new rows of data are received by discs in turn. For applications that want to read the full relation sequentially for each query, this strategy assures an even distribution of tuples across drives.

Schema Partitioning :
Various tables inside a database are put on different discs using a technique called schema partitioning.

Intra-query parallelism :
Using a shared-nothing paralleling architecture technique, intra-query parallelism refers to the processing of a single query in a parallel process on many CPUs.
This employs two different strategies:
First method — In this method, a duplicate task can be executed on a small amount of data by each CPU.
Second method — Using this method, the task can be broken up into various sectors, with each CPU carrying out a separate subtask.
Inter-query parallelism
Each CPU executes numerous transactions when inter-query parallelism is used. Parallel transaction processing is what it is known as. To support inter-query parallelism, DBMS leverages transaction dispatching.
We can also employ a variety of techniques, such as effective lock management. This technique runs each query sequentially, which slows down the running time.
In such circumstances, DBMS must be aware of the locks that various transactions operating on various processes have acquired. When simultaneous transactions don't accept the same data, inter-query parallelism on shared storage architecture works well.
Additionally, the throughput of transactions is boosted, and it is the simplest form of parallelism in DBMS.

Intra-operation parallelism :
In this type of parallelism, we execute each individual operation of a task, such as sorting, joins, projections, and so forth, in parallel. Intra-operation parallelism has a very high parallelism level. Database systems naturally employ this kind of parallelism. Consider the following SQL example:
SELECT * FROM the list of vehicles and sort by model number;

Since a relation might contain a high number of records, the relational operation in the aforementioned query is sorting.
Because this operation can be done on distinct subsets of the relation in several processors, it takes less time to sort the data.

Inter-operation parallelism :
This term refers to the concurrent execution of many operations within a query expression. They come in two varieties:
Pipelined parallelism — In pipeline parallelism, a second operation consumes a row of the first operation's output before the first operation has finished producing the whole set of rows in its output. Additionally, it is feasible to perform these two processes concurrently on several CPUs, allowing one operation to consume tuples concurrently with another operation and thereby reduce them.
It is advantageous for systems with a limited number of CPUs and prevents the storage of interim results on a disc.
Independent parallelism- In this form of parallelism, operations contained within query phrases that are independent of one another may be carried out concurrently. This analogy is extremely helpful when dealing with parallelism of a lower degree.

Execution Of a Parallel Query :
The relational model has been favoured over prior hierarchical and network models because of commercial database technologies. Data independence and high-level query languages are the key advantages that relational database systems (RDBMSs) have over their forerunners (e.g., SQL).
The efficiency of programmers is increased, and routine optimization is encouraged.
Additionally, distributed database management is made easier by the relational model's set-oriented structure. RDBMSs may now offer performance levels comparable to older systems thanks to a decade of development and tuning.
They are therefore widely employed in the processing of commercial data for OLTP (online transaction processing) or decision-support systems. Through the use of many processors working together, parallel processing makes use of multiprocessor computers to run application programmes and boost performance.
It is most commonly used in scientific computing, which it does by the speed of numerical applications' responses.

The development of parallel database systems is an example of how database management and parallel computing can work together. A given SQL statement can be divided up in the parallel database system PQO such that its components can run concurrently on several processors in a multi-processor machine.
Full table scans, sorting, sub-queries, data loading, and other common operations can all be performed in parallel.
As a form of parallel database optimization, Parallel Query enables the division of SELECT or DML operations into many smaller chunks that can be executed by PQ slaves on different CPUs in a single box.
The order of joins and the method for computing each join are fixed in the first part of the Fig, which is sorting and rewriting. The second phase, parallelization, turns the query tree into a parallel plan.

Parallelization divides this stage into two parts: extraction of parallelism and scheduling. Optimizing database queries is an important task in database management systems to improve the performance of database operations. Parallelization of database query optimization can significantly improve query execution time by dividing the workload among multiple processors or nodes.

Here's an overview of how parallelization can be applied to database query optimization:

1. Partitioning: The first step is to partition the data into smaller subsets. The partitioning can be done based on different criteria, such as range partitioning, hash partitioning, or list partitioning. This can be done in parallel by assigning different processors or nodes to handle different parts of the partitioning process.

2. Query optimization: Once the data is partitioned, the next step is to optimize the queries. Query optimization involves finding the most efficient way to execute the query by considering factors such as index usage, join methods, and filtering. This can also be done in parallel by assigning different processors or nodes to handle different parts of the query optimization process.

3. Query execution: After the queries are optimized, the final step is to execute the queries. The execution can be done in parallel by assigning different processors or nodes to handle different parts of the execution process. The results can then be combined to generate the final result set.

To implement parallelization of database query optimization, we can use parallel programming frameworks such as OpenMP or CUDA. These frameworks provide a set of APIs and tools to distribute the workload among multiple processors or nodes and to manage the synchronization and communication between them.

Here's an example of how we can parallelize the query optimization process using OpenMP:

```
//C++
// Partition the data
std::vector<std::vector<int>> partitions;
int num_partitions = omp_get_num_threads();
#pragma omp parallel for
for (int i = 0; i < num_partitions; i++) {
    std::vector<int> partition = partition_data(data, i, num_partitions);
    partitions.push_back(partition);
}

// Optimize the queries in parallel
#pragma omp parallel for
for (int i = 0; i < num_queries; i++) {
    Query query = queries[i];
    int partition_id = get_partition_id(query, partitions);
    std::vector<int> partition = partitions[partition_id];
```

```
    optimize_query(query, partition);
}

// Execute the queries in parallel
#pragma omp parallel for
for (int i = 0; i < num_queries; i++) {
    Query query = queries[i];
    int partition_id = get_partition_id(query, partitions);
    std::vector<int> partition = partitions[partition_id];
    std::vector<int> result = execute_query(query, partition);
    merge_results(result);
}
```

In this example, we first partition the data into smaller subsets using OpenMP parallelism. Then we optimize each query in parallel by assigning different processors or nodes to handle different parts of the optimization process. Finally, we execute the queries in parallel by assigning different processors or nodes to handle different parts of the execution process.

Parallelization of database query optimization can significantly improve the performance of database operations and reduce query execution time. However, it requires careful consideration of the workload distribution, synchronization, and communication between processors or nodes.