A parallel DBMS is a DBMS that runs across multiple processors or CPUs and is mainly designed to execute query operations in parallel, wherever possible. The parallel DBMS link a number of smaller machines to achieve the same throughput as expected from a single large machine.

In Parallel Databases, mainly there are three architectural designs for parallel DBMS. They are as follows:

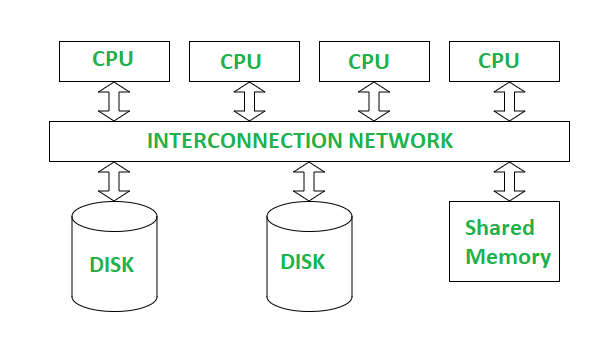
**1)Shared Memory Architecture**

**2)Shared Disk Architecture**

**3)Shared Nothing Architecture**

Let’s discuss them one by one:

**1. Shared Memory Architecture**- In Shared Memory Architecture, there are multiple CPUs that are attached to an interconnection network. They are able to share a single or global main memory and common disk arrays. It is to be noted that, In this architecture, a single copy of a multi-threaded operating system and multithreaded DBMS can support these multiple CPUs. Also, the shared memory is a solid coupled architecture in which multiple CPUs share their memory. It is also known as Symmetric multiprocessing (SMP). This architecture has a very wide range which starts from personal workstations that support a few microprocessors in parallel via RISC.



**Advantages** :

It has high-speed data access for a limited number of processors.

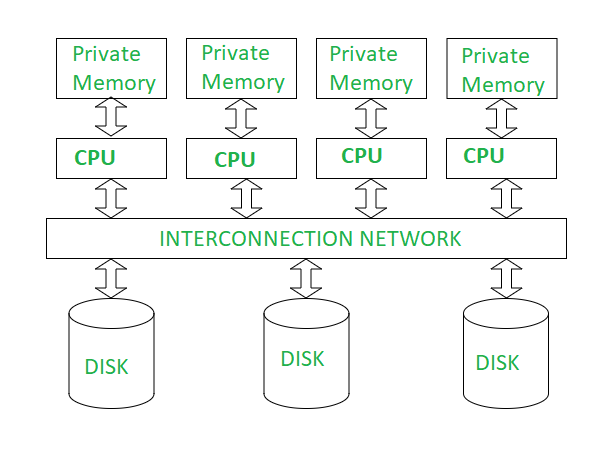
The communication is efficient.

**Disadvantages** :

It cannot use beyond 80 or 100 CPUs in parallel.

The bus or the interconnection network gets block due to the increment of the large number of CPUs.

**2. Shared Disk Architectures :**  
In Shared Disk Architecture, various CPUs are attached to an interconnection network. In this, each CPU has its own memory and all of them have access to the same disk. Also, note that here the memory is not shared among CPUs therefore each node has its own copy of the operating system and DBMS. Shared disk architecture is a loosely coupled architecture optimized for applications that are inherently centralized. They are also known as clusters.



**Advantages** :

The interconnection network is no longer a bottleneck each CPU has its own memory.

Load-balancing is easier in shared disk architecture.

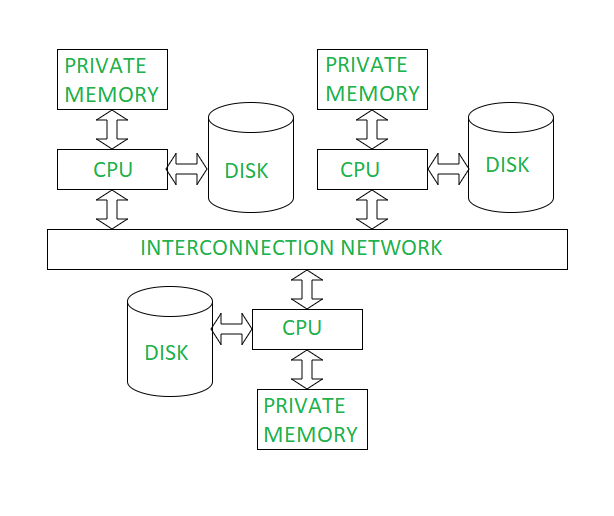
There is better fault tolerance.

**Disadvantages** :

If the number of CPUs increases, the problems of interference and memory contentions also increase.

There’s also exists a scalability problem.

**3. Shared Nothing Architecture :**  
Shared Nothing Architecture is multiple processor architecture in which each processor has its own memory and disk storage. In this, multiple CPUs are attached to an interconnection network through a node. Also, note that no two CPUs can access the same disk area. In this architecture, no sharing of memory or disk resources is done. It is also known as Massively parallel processing (MPP).



**Advantages** :

It has better scalability as no sharing of resources is done

Multiple CPUs can be added

**Disadvantages**:

The cost of communications is higher as it involves sending of data and software interaction at both ends

The cost of non-local disk access is higher than the cost of shared disk architectures.

**Parallel Query Evaluation**

IO Parallelism

**I/O Parallelism** refers to the parallelization of input/output operations in a database system to improve the performance of data retrieval and storage. Since I/O operations, such as reading from and writing to disk, are often the most time-consuming part of query execution, optimizing these through parallelism can significantly enhance overall system performance.

Types

**1)Partitioning**

**Partitioning** in a database context refers to the division of a large table or dataset into smaller, more manageable pieces called partitions. Each partition can be stored and processed separately, allowing for more efficient query execution, load balancing, and management. Partitioning is commonly used to improve performance, enhance scalability, and facilitate parallel processing.

### **Range Partitioning**

* **Concept**: Data is divided into partitions based on specific ranges of values from a column. Each partition holds a subset of the data that falls within a defined range of the column’s values. This is useful when you need to frequently query data within specific value ranges.
* **Implementation**: You define partitions with range boundaries. For example, in a sales database, you might partition data by year, with each partition holding data for a particular year. Queries that filter data by year can quickly access the relevant partition, reducing the amount of data scanned.
* **Example**: Suppose you have a sales table with a sale\_date column. You could set up partitions for each quarter: Q1 (January to March), Q2 (April to June), Q3 (July to September), and Q4 (October to December). Queries that need data from Q1 will only access the corresponding partition.

### **Hash Partitioning**

* **Concept**: Data is distributed across partitions using a hash function applied to one or more columns. The hash function generates a hash value, which determines the partition where the row will be stored. This helps ensure an even distribution of rows across partitions, which can balance the load and improve query performance.
* **Implementation**: A hash function is applied to a column value (or a combination of columns), and the result is used to determine the partition. This method is particularly useful for balancing data distribution and managing high-transaction workloads.
* **Example**: In a customer table, you might hash the customer\_id to assign rows to partitions. For instance, if you have four partitions, the hash function might ensure that the customer IDs are evenly distributed among these four partitions. This helps in balancing the data and workload across partitions.

### **Round-Robin Partitioning**

* **Concept**: Data rows are distributed evenly across partitions in a cyclic or sequential order, regardless of the row's values. This method does not consider the data content and simply allocates rows to partitions in a predictable, repetitive pattern.
* **Implementation**: Rows are assigned to partitions in the order they are inserted. If there are three partitions, the first row goes to Partition 1, the second row to Partition 2, the third to Partition 3, and the fourth row cycles back to Partition 1, and so on. This method ensures that each partition gets a roughly equal number of rows.
* **Example**: If you have a log table with many rows being inserted continuously, you can use round-robin partitioning to distribute these rows evenly across partitions. This helps in maintaining even storage and balancing the workload without considering specific data values.

**2)IntraQuery Parallelism**

Intra-query parallelism refers to the execution of a single query in a parallel process on different CPUs using a shared-nothing paralleling architecture technique.

**Types**

### **1. Intraoperation Parallelism**

**Intraoperation Parallelism** involves parallelizing different parts of a single operation within a query. It focuses on breaking down one type of operation into smaller tasks that can be executed concurrently.

### **SUBTYPES OF INTRAOPERATION ||el**

### **-- Parallel Sort**

**Parallel Sort** involves dividing the sorting operation into smaller tasks that can be executed concurrently. This approach speeds up the sorting process by leveraging multiple processors or nodes to handle different parts of the data simultaneously.

#### **How It Works:**

1. **Data Partitioning**: The data to be sorted is divided into smaller partitions. Each partition is a subset of the data that can be sorted independently.
2. **Concurrent Sorting**: Each partition is sorted concurrently by different processors or nodes. This is done using parallel sorting algorithms, such as merge sort or quicksort.
3. **Merging**: Once all partitions are sorted, the sorted partitions are merged together to form the final sorted dataset. This merge step is also parallelized to improve performance.

#### **Example:**

For a large dataset of customer records, the data is partitioned into chunks (e.g., by customer ID ranges). Each chunk is sorted in parallel. After sorting, the sorted chunks are merged to produce the final sorted list of customer records.

### **-- Parallel Join**

**Parallel Join** involves executing the join operation across multiple processors or nodes to speed up the process. The goal is to handle large datasets efficiently by breaking the join operation into smaller, concurrent tasks.

#### **How It Works:**

**Data Partitioning**: Both tables involved in the join are partitioned based on the join key. This ensures that each processor or node handles a manageable subset of the data.

**Concurrent Joins**: Each partition of one table is joined with the corresponding partition of the other table concurrently. This can be done using various join algorithms like hash join or merge join.

**Combining Results**: The results from the concurrent join operations are combined to form the final result set.

#### **Example:**

Consider two large tables, Orders and Customers, joined on CustomerID. Both tables are partitioned by CustomerID. Each partition of the Orders table is joined with the corresponding partition of the Customers table in parallel, and the results are aggregated.

### **-- Partitioned Join**

**Partitioned Join** is a specific type of parallel join where the tables are partitioned based on the join key to ensure that matching rows from both tables are located in the same partition.

#### **How It Works:**

**Partitioning Tables**: Both tables are partitioned using the same partitioning scheme based on the join key. For instance, if joining on CustomerID, both tables are partitioned by CustomerID ranges.

**Local Joins**: Each partition of one table is joined with the corresponding partition of the other table. This local join is performed concurrently by different processors or nodes.

**Result Combination**: The results of the local joins are combined to form the final joined dataset.

#### **Example:**

For a join on OrderID, if both Orders and Products tables are partitioned by OrderID ranges, each partition of the Orders table is joined with the corresponding partition of the Products table. This ensures that each processor handles a specific subset of the data.

### **-- Fragment and Replicate Join**

**Fragment and Replicate Join** is a join technique where one table is fragmented into multiple subsets, and each fragment is replicated across multiple nodes. The goal is to speed up the join operation by distributing the data across nodes.

#### **How It Works:**

**Fragmentation**: The join table (typically the smaller table) is fragmented into smaller pieces. Each fragment is a subset of the data that can be replicated.

**Replication**: Each fragment is replicated across multiple nodes to ensure that all nodes have access to the entire fragment.

**Concurrent Joins**: Each node performs the join operation using the replicated fragments. This ensures that all nodes can participate in the join operation concurrently.

**Combining Results**: The results from the concurrent join operations are aggregated to form the final result set.

#### **Example:**

For a join operation between a large Orders table and a smaller Products table, the Products table is fragmented into smaller pieces. Each fragment is replicated across all nodes. Each node then performs the join operation with its local copy of the Products fragments, ensuring that the join is performed efficiently.

**2)Interoperation Query ||el**

Certainly! Here’s a revised explanation of pipelined and independent parallelism in the context of DBMS, without the benefits:

### **--Pipelined Parallelism**

**Pipelined parallelism** involves breaking a task into a series of stages, where each stage performs a specific operation. This technique is used to improve the efficiency of query execution by overlapping the processing of different parts of the query.

**How it works:**

**Task Decomposition**: A complex database operation (like a query) is decomposed into several stages. For example, a query might involve stages such as selection, projection, and join.

**Stage Execution**: Each stage processes its input and passes the result to the next stage. This allows multiple stages to be active simultaneously.

**Overlap**: While one tuple (row) is being processed in the selection stage, another tuple can be processed in the projection stage, and so on.

**Example:** Consider a query that involves a join operation. In a pipelined parallelism setup:

* **Stage 1**: Scan the first relation.
* **Stage 2**: Join the result of Stage 1 with the second relation.
* **Stage 3**: Apply additional operations like filtering or aggregation.

While Stage 1 processes tuples from the first relation, Stage 2 can start working on tuples that have already been processed, and Stage 3 can begin processing tuples that have been joined.

### **--Independent Parallelism**

**Independent parallelism** focuses on breaking down a task into independent subtasks that can be executed concurrently without depending on each other. In a DBMS context, this often applies to the parallel execution of multiple queries or operations that do not interfere with one another.

**How it works:**

**Task Decomposition**: A large query or a set of queries is divided into smaller, independent tasks. For example, if multiple queries are running, they might be executed in parallel if they do not share resources or data.

**Parallel Execution**: These independent tasks are executed simultaneously, making full use of available processing resources.

**Example:** Consider a scenario where a DBMS needs to handle multiple queries:

* **Query 1**: Retrieve data from Table A.
* **Query 2**: Retrieve data from Table B.
* **Query 3**: Perform a complex aggregation on Table C.

If the queries are independent, they can be executed in parallel, utilizing different CPUs or cores. Since the queries do not interfere with each other, they can be processed simultaneously without waiting for one another to complete.

**3)Interquery ||el**

In Inter-query parallelism, there is an execution of multiple transactions by each CPU. It is called parallel transaction processing. DBMS uses transaction dispatching to carry inter query parallelism. We can also use some different methods, like efficient lock management. In this method, each query is run sequentially, which leads to slowing down the running of long queries. In such cases, DBMS must understand the locks held by different transactions running on different processes. Inter query parallelism on shared disk architecture performs best when transactions that execute in parallel do not accept the same data. Also, it is the easiest form of parallelism in DBMS, and there is an increased transaction throughput.

**Disadvantages of Parallel Databases**

* **Increased Complexity**: Designing and managing a parallel database system is more complex due to the need for data partitioning, task coordination, and query optimization.
* **Data Partitioning Issues**: Deciding how to effectively partition data across nodes can be challenging and may introduce overhead or inefficiencies if not done properly.
* **Synchronization Overhead**: Ensuring consistency and managing communication between different nodes can add overhead and affect performance.
* **Scalability Limits**: While parallel databases can scale well, there are practical limits to how much you can scale, and beyond a certain point, the benefits may diminish.
* **Cost**: Implementing and maintaining a parallel database system can be expensive due to the need for multiple processors, servers, and additional software or infrastructure.

### **Key Aspects of Parallel Query Optimization**

**1.Query Decomposition**

* + **Break Down**: Divide a query into smaller, manageable sub-queries or tasks that can be executed in parallel.
  + **Task Scheduling**: Determine the order and timing of executing these sub-queries to maximize efficiency.

**2.Data Partitioning**

* + **Horizontal Partitioning**: Split data into subsets (partitions) based on rows, allowing different partitions to be processed simultaneously.
  + **Vertical Partitioning**: Divide data columns into separate partitions, enabling parallel processing of different attributes.

**3.Load Balancing**

* + **Distribute Load**: Ensure that tasks are evenly distributed across available processors or nodes to avoid bottlenecks and overloading.
  + **Dynamic Adjustment**: Adjust task distribution dynamically based on current system load and performance metrics.

**4.Synchronization and Coordination**

* + **Manage Dependencies**: Handle interdependencies between different parts of the query to ensure correct results.
  + **Aggregate Results**: Collect and combine results from different parallel tasks to produce the final output.

**5.Resource Management**

* + **Monitor Usage**: Track and manage the usage of system resources like CPU, memory, and I/O to prevent contention and ensure efficient execution.
  + **Optimize Access**: Optimize data access patterns to reduce contention and improve performance.

**6.Query Plan Generation**

* + **Cost Estimation**: Estimate the cost of various parallel execution plans to select the most efficient one.
  + **Plan Selection**: Choose the optimal plan based on factors such as data distribution, query complexity, and system resources.

### **Example**

For a query involving a large table join:

* **Decompose**: Break the join operation into separate scans and join tasks.
* **Partition**: Distribute the table rows across different nodes for parallel processing.
* **Execute**: Process the partitions concurrently, performing joins in parallel.
* **Combine**: Aggregate the results from different nodes to generate the final output.