Inter-operational and Intra-operational parallelism

Intra-operation Parallelism

It is about parallelizing a single relational operation given in a query.

SELECT * FROM Email ORDER BY Start_Date;

In the above query, the relational operation is Sorting. Since a table may have large number of records in it, the operation can be performed on different subsets of the table in multiple processors, which reduces the time required to sort.

Consider another query,

SELECT * FROM Student, CourseRegd WHERE Student.Regno = CourseRegd.Regno;

In this query, the relational operation is Join. The query joins two tables Student, and CourseRegd on common attribute Regno. Parallelism is required here if the size of tables is very large. Usually, order of tuples does not matter in DBMS. Hence, the tables arranged in random order needs every record of one table should be matched with every record of other table to complete the join process. For example, if Student has 10000 records and CourseRegd has 60000 records, then join requires 10000 X 60000 comparisons. If we exploit parallelism in here, we could achieve better performance.

There are many such relational operations which can be executed in parallel using many processors on subsets of the table/tables mentioned in the query. The following list includes the relational operations and various techniques used to implement those operations in parallel.

- **Parallel Sort**
    - **Range-Partitioning Sort**
    - **Parallel External Sort-Merge**
- **Parallel Join**
    - **Partitioned Join**
    - **Fragment and Replicate Join**
    - **Partitioned Parallel Hash Join**
    - **Parallel Nested-Loop Join**
- **Selection**
- **DuplicateElimination**
- **Projection**
- **Aggregation**

Parallel Sort

1)Range Partitioning Sort

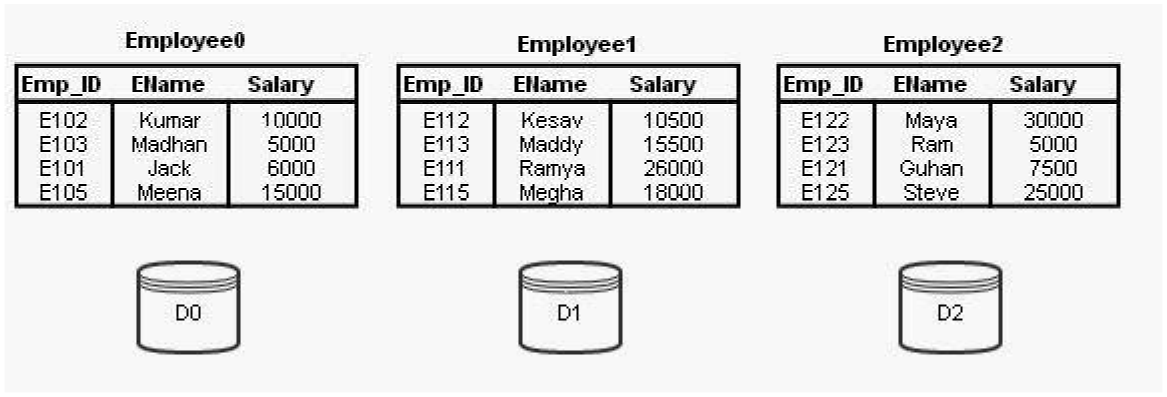
Assumptions:
Assume n processors, P0, P1, …, Pn-1 and n disks D0, D1, …, Dn-1.
Disk Di is associated with Processor Pi.
Relation R is partitioned into R0, R1, …, Rn-1 using Round-robin technique or Hash Partitioning technique or Range Partitioning technique (if range partitioned on some other attribute other than sorting attribute)
Objective:
Our objective is to sort a relation (table) Ri that resides on n disks on an attribute A in parallel.

Steps:
Step 1: Partition the relations Ri on the sorting attribute A at every processor using a range vector v. Send the partitioned records which fall in the ith range to Processor Pi where they are temporarily stored in Di.
Step 2: Sort each partition locally at each processor Pi. And, send the sorted results for merging with all the other sorted results which is trivial process.

Example:
Let us explain the above said process with simple example.Consider the following relation schema Employee;

Employee (Emp_ID, EName, Salary)

Assume that relation ***Employee is permanently partitioned using Round-robin technique*** into 3 disks D_0, D_1, and D_2 which are associated with processors P_0, P_1, and P_2. At processors P_0, P_1, and P_2, the relations are named Employee0, Employee1 and Employee2 respectively. This initial state is given in Figure 1.



Employee0

Emp_ID	EName	Salary
E102	Kumar	10000
E103	Madhan	5000
E101	Jack	6000
E105	Meena	15000

Employee1

Emp_ID	EName	Salary
E112	Kesav	10500
E113	Maddy	15500
E111	Ramya	26000
E115	Megha	18000

Employee2

Emp_ID	EName	Salary
E122	Maya	30000
E123	Ram	5000
E121	Guhan	7500
E125	Steve	25000

Figure 1

Assume that the following sorting query is initiated.

SELECT * FROM Employee ORDER BY Salary;

As already said, the table Employee is not partitioned on the sorting attribute Salary. Then, the Range-Partitioning technique works as follows;

Step 1:

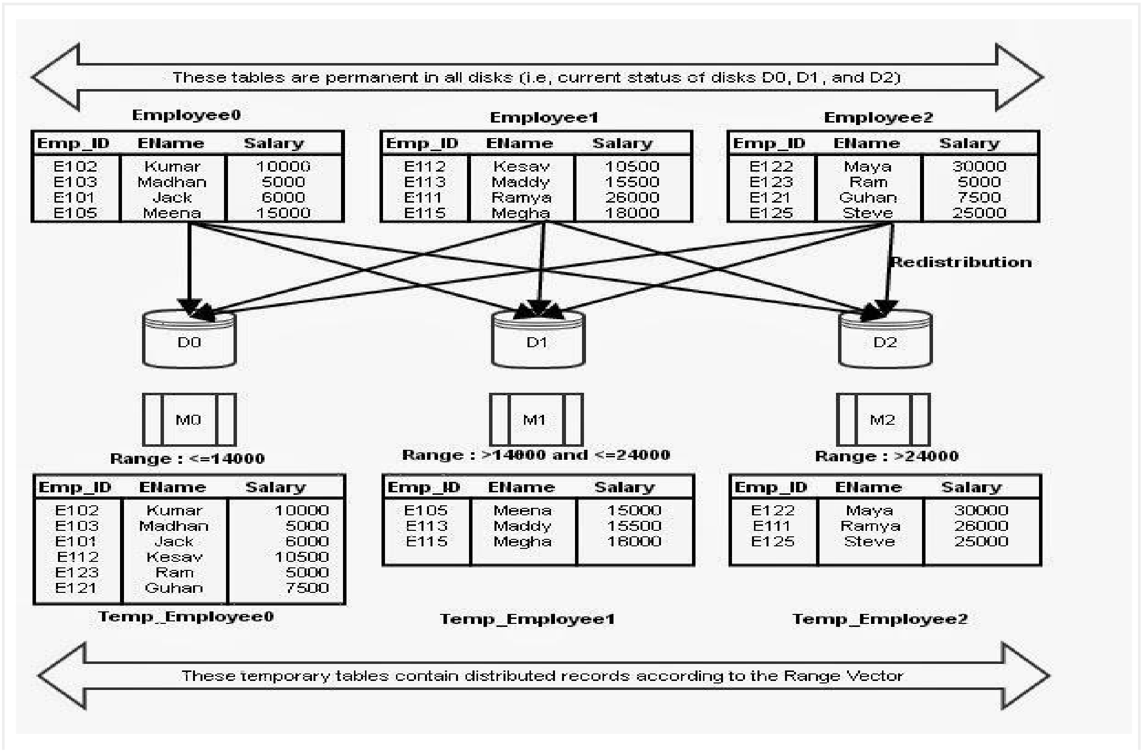
At first we have to identify a range vector v on the Salary attribute. The range vector is of the form **v[v0, v1, …, vn-2]**. For our example, let us assume the following range vector;

v[14000, 24000]

This range vector represents 3 ranges, range 0 (14000 and less), range 1 (14001 to 24000) and range 2 (24001 and more).

Redistribute the relations Employee0, Employee1 and Employee2 using these range vectors into 3 disks temporarily. After this distribution disk 0 will have range 0 records (i.e, records with salary value less than or equal to 14000), disk 1 will have range 1 records (i.e, records with salary value greater than 14000 and less than or equal to 24000), and disk 2 will have range 2 records (i.e, records with salary value greater than 24000).

This redistribution according to range vector v is represented in Figure 2 as links to all the disks from all the relations. Temp_Employee0, Temp_Employee1, and Temp_Employee2, are the relations after successful redistribution. These tables are stored temporarily in disks D0, D1, and D2. (They can also be stored in Main memories (M0, M1, M2) if they fit into RAM).


These tables are permanent in all disks (i.e, current status of disks D0, D1, and D2)
Redistribution
D0
D1
D2
M0
M1
M2
Range : <=14000
Range : >14000 and <=24000
Range : >24000
These temporary tables contain distributed records according to the Range Vector

Employee0

Emp_ID	EName	Salary
E102	Kumar	10000
E103	Madhan	5000
E101	Jack	6000
E105	Meena	15000

Employee1

Emp_ID	EName	Salary
E112	Kesav	10500
E113	Maddy	15500
E111	Ramya	26000
E115	Megha	18000

Employee2

Emp_ID	EName	Salary
E122	Maya	30000
E123	Ram	5000
E121	Guhan	7500
E125	Steve	25000

Emp_ID	EName	Salary
E102	Kumar	10000
E103	Madhan	5000
E101	Jack	6000
E112	Kesav	10500
E123	Ram	5000
E121	Guhan	7500

Temp_Employee0

Emp_ID	EName	Salary
E105	Meena	15000
E113	Maddy	15500
E115	Megha	18000

Temp_Employee1

Emp_ID	EName	Salary
E122	Maya	30000
E111	Ramya	26000
E125	Steve	25000

Temp_Employee2

Figure 2

Step 2:

Now, we got temporary relations at all the disks after redistribution.

At this point, all the processors sort the data assigned to them in ascending order of Salary individually. The process of performing the same operation in parallel on different sets of data is called ***Data Parallelism***.

Final Result:

After the processors completed the sorting, we can simply collect the data from different processors and merge them. This merge process is straightforward as we have data already sorted for every range. Hence, collecting sorted records from partition 0, partition 1 and partition 2 and merging them will give us final sorted output.

2) **Parallel External Sort-Merge Technique**

Parallel External Sort-Merge

Assumptions:

Assume n processors, P0, P1, …, Pn-1 and n disks D0, D1, …, Dn-1.

Disk Di is associated with Processor Pi.

Relation R is partitioned into R0, R1, …, Rn-1 using Round-robin technique or Hash Partitioning technique or Range Partitioning technique (partitioned on any attribute)

Objective:

Our objective is to sort a relation (table) Ri on an attribute A in parallel where R resides on n disks.

Steps:

Step 1: Sort the relation partition Ri which is stored on disk Di on the sorting attribute of the query.

Step 2: Identify a range partition vector v and range partition every Ri into processors, P0, P1, …, Pn-1 using vector v.

Step 3: Each processor Pi performs a merge on the incoming range partitioned data from every other processors (The data are actually transferred in order. That is, all processors send first partition into P0, then all processors sends second partition into P1, and so on).

Step 4: Finally, concatenate all the sorted data from different processors to get the final result.

Point to note:

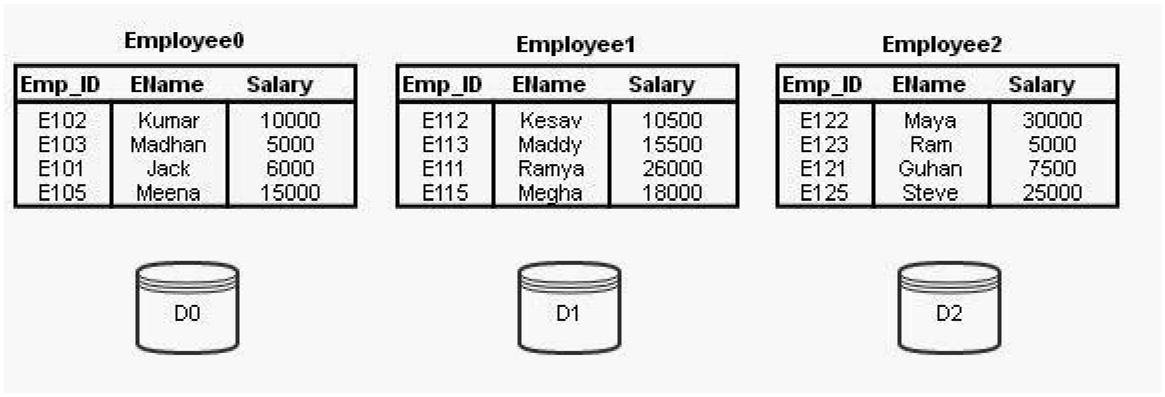
Range partition must be done using a good range-partitioning vector. Otherwise, skew might be the problem.

Example:

Let us explain the above said process with simple example. Consider the following relation schema Employee;

Employee (Emp_ID, EName, Salary)

Assume that relation Employee is permanently partitioned using Round-robin technique into 3 disks D0, D1, and D2 which are associated with processors P0, P1, and P2. At processors P0, P1, and P2, the relations are named Employee0, Employee1 and Employee2 respectively. This initial state is given in Figure 1.



Employee0

Emp_ID	EName	Salary
E102	Kumar	10000
E103	Madhan	5000
E101	Jack	6000
E105	Meena	15000

Employee1

Emp_ID	EName	Salary
E112	Kesav	10500
E113	Maddy	15500
E111	Ramya	26000
E115	Megha	18000

Employee2

Emp_ID	EName	Salary
E122	Maya	30000
E123	Ram	5000
E121	Guhan	7500
E125	Steve	25000

Figure 1 - Partitioned Employee

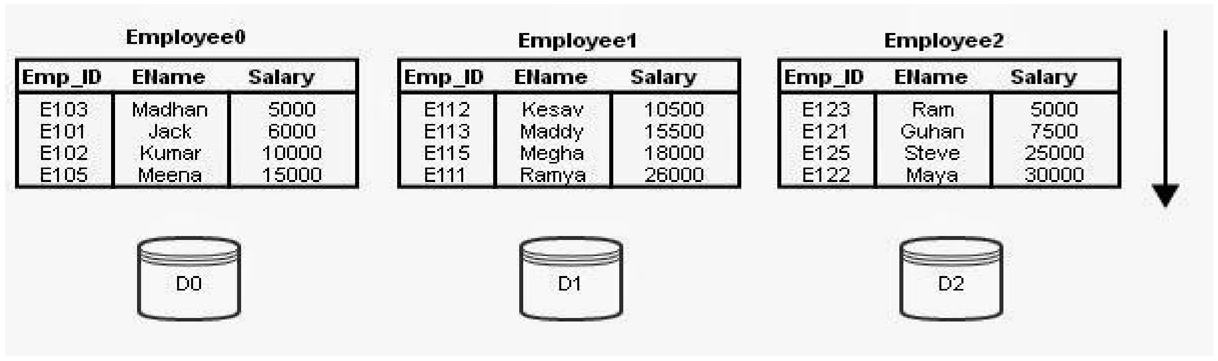
Assume that the following sorting query is initiated.

SELECT * FROM Employee ORDER BY Salary;

As already said, the table Employee is not partitioned on the sorting attribute Salary. Then, the Parallel External Sort-Merge technique works as follows;

Step 1:

Sort the data stored in every partition (every disk) using the ordering attribute Salary. (Sorting of data in every partition is done temporarily). At this stage every Employeei contains salary values of range minimum to maximum. The partitions sorted in ascending order is shown below, in Figure 2.



Employee0

Emp_ID	EName	Salary
E103	Madhan	5000
E101	Jack	6000
E102	Kumar	10000
E105	Meena	15000

Employee1

Emp_ID	EName	Salary
E112	Kesav	10500
E113	Maddy	15500
E115	Megha	18000
E111	Ramya	26000

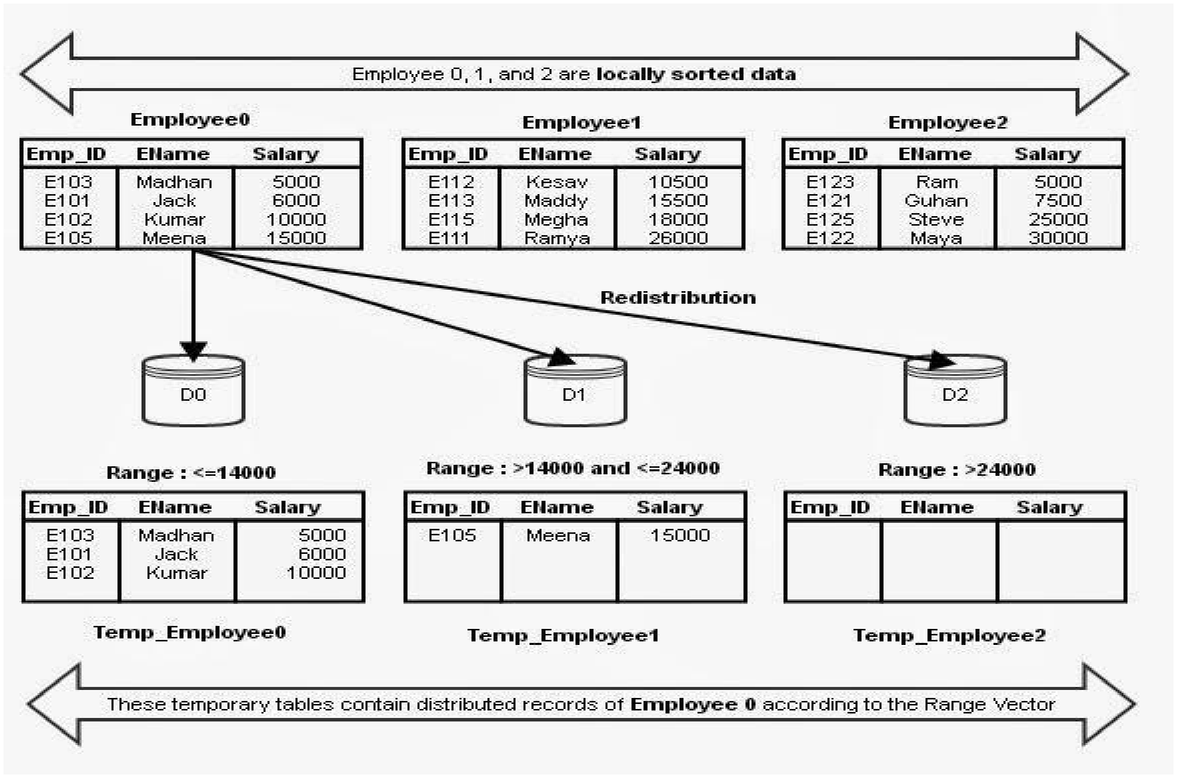
Employee2

Emp_ID	EName	Salary
E123	Ram	5000
E121	Guhan	7500
E125	Steve	25000
E122	Maya	30000

Figure 2 - Partitioned Employee in ascending order on Salary attribute

Step 2:

We have to identify a range vector v on the Salary attribute. The range vector is of the form v[v0, v1, …, vn-2]. For our example, let us assume the following range vector;
v[14000, 24000]
This range vector represents 3 ranges, range 0 (14000 and less), range 1 (14001 to 24000) and range 2 (24001 and more).
Redistribute every partition (Employee0, Employee1 and Employee2) using these range vectors into 3 disks temporarily. What would be the status of Temp_Employee 0, 1, and 2 after distributing Employee 0 is given in Figure 3.


Employee 0, 1, and 2 are locally sorted data
Employee0
Employee1
Employee2
Redistribution
D0
D1
D2
Range : <=14000
Range : >14000 and <=24000
Range : >24000
Temp_Employee0
Temp_Employee1
Temp_Employee2
These temporary tables contain distributed records of Employee 0 according to the Range Vector

Employee0

Emp_ID	EName	Salary
E103	Madhan	5000
E101	Jack	6000
E102	Kumar	10000
E105	Meena	15000

Employee1

Emp_ID	EName	Salary
E112	Kesav	10500
E113	Maddy	15500
E115	Megha	18000
E111	Ramya	26000

Employee2

Emp_ID	EName	Salary
E123	Ram	5000
E121	Guhan	7500
E125	Steve	25000
E122	Maya	30000

Temp_Employee0 (Range : <=14000)

Emp_ID	EName	Salary
E103	Madhan	5000
E101	Jack	6000
E102	Kumar	10000

Temp_Employee1 (Range : >14000 and <=24000)

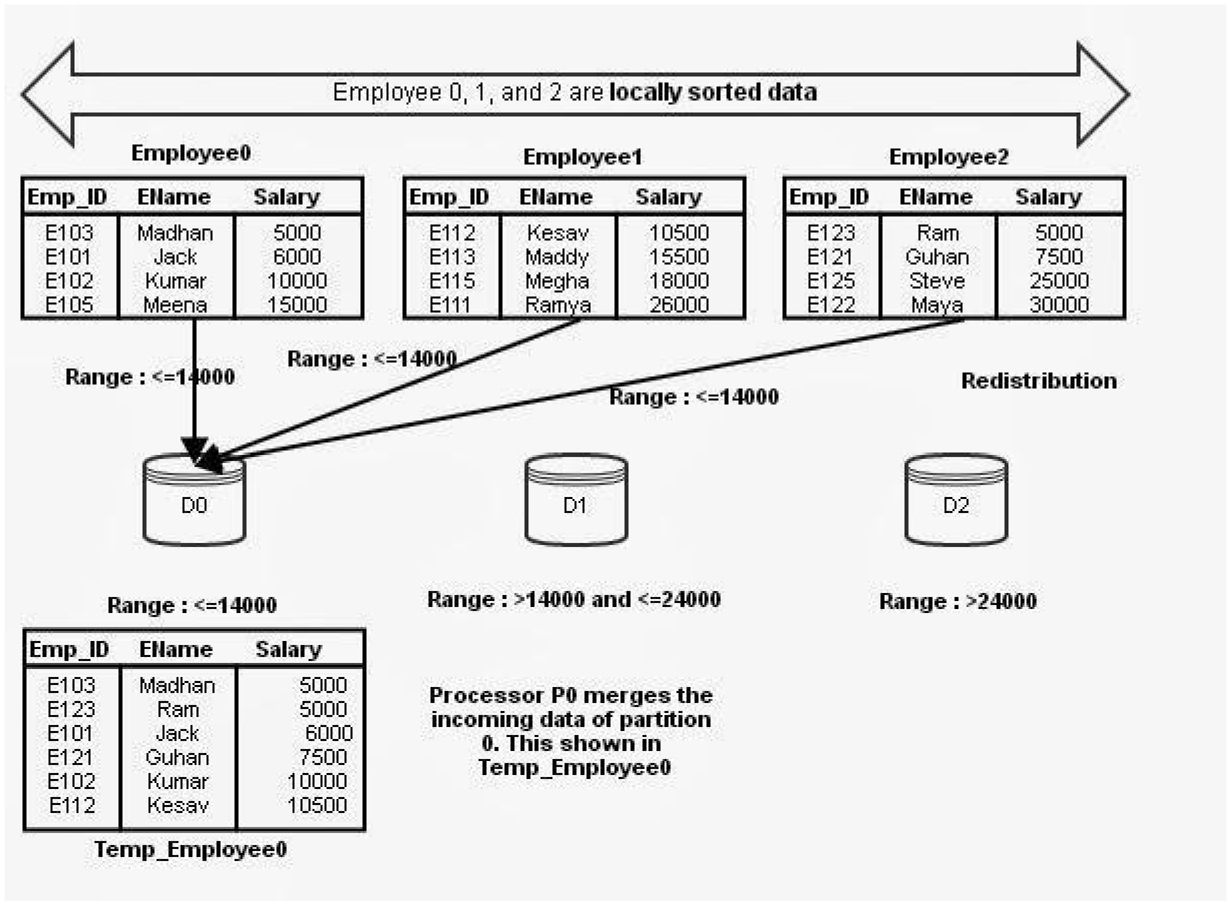
Emp_ID	EName	Salary
E105	Meena	15000

Temp_Employee2 (Range : >24000)

Emp_ID	EName	Salary

Figure 3 - Distribution of Employee 0 according to the range vector

Step 3:
Actually, the above said distribution is executed at all processors in parallel such that processors P0, P1, and P2 are sending the first partition of Employee 0, 1, and 2 to disk 0. Upon receiving the records from various partitions, the receiving processor P0 merges the sorted data. This is shown in Figure 4.



Employee 0, 1, and 2 are **locally sorted data**
Range : <=14000
Range : <=14000
Range : <=14000
Redistribution
D0
D1
D2
Range : <=14000
Range : >14000 and <=24000
Range : >24000
Processor P0 merges the incoming data of partition 0. This shown in Temp_Employee0

Employee0

Emp_ID	EName	Salary
E103	Madhan	5000
E101	Jack	6000
E102	Kumar	10000
E105	Meena	15000

Employee1

Emp_ID	EName	Salary
E112	Kesav	10500
E113	Maddy	15500
E115	Megha	18000
E111	Ramya	26000

Employee2

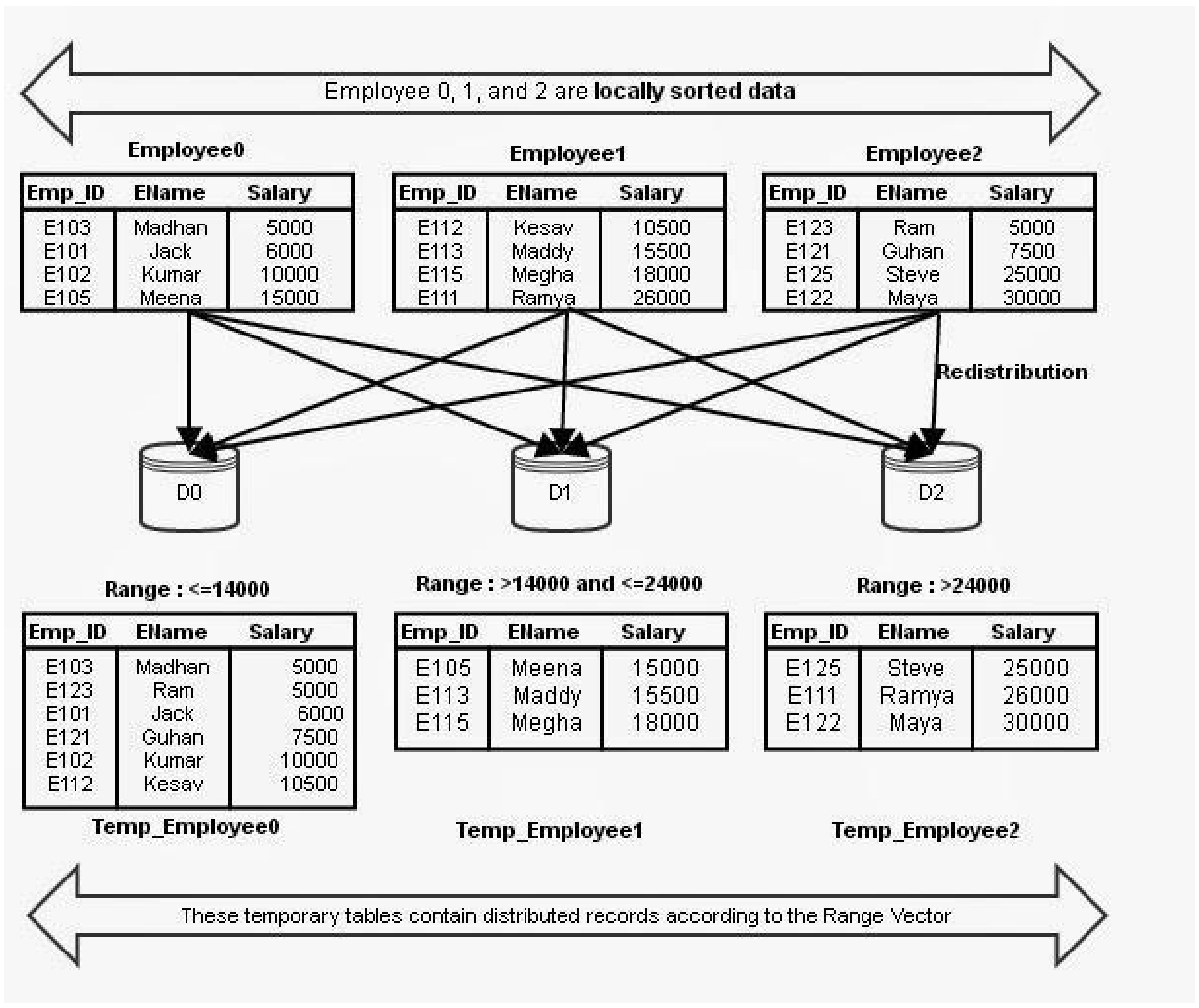
Emp_ID	EName	Salary
E123	Ram	5000
E121	Guhan	7500
E125	Steve	25000
E122	Maya	30000

Emp_ID	EName	Salary
E103	Madhan	5000
E123	Ram	5000
E101	Jack	6000
E121	Guhan	7500
E102	Kumar	10000
E112	Kesav	10500

Temp_Employee0

Figure 4 - Merging data of range <=14000 at D0 from all the disks

The above said process is done at all processors for different partitions. The final version of Temp_Employee 0, 1, and 2 are shown in Figure 5.


Employee 0, 1, and 2 are locally sorted data
Employee0
Employee1
Employee2
Redistribution
D0
D1
D2
Range : <=14000
Range : >14000 and <=24000
Range : >24000
Temp_Employee0
Temp_Employee1
Temp_Employee2
These temporary tables contain distributed records according to the Range Vector

Employee0

Emp_ID	EName	Salary
E103	Madhan	5000
E101	Jack	6000
E102	Kumar	10000
E105	Meena	15000

Employee1

Emp_ID	EName	Salary
E112	Kesav	10500
E113	Maddy	15500
E115	Megha	18000
E111	Ramya	26000

Employee2

Emp_ID	EName	Salary
E123	Ram	5000
E121	Guhan	7500
E125	Steve	25000
E122	Maya	30000

Temp_Employee0 (Range : <=14000)

Emp_ID	EName	Salary
E103	Madhan	5000
E123	Ram	5000
E101	Jack	6000
E121	Guhan	7500
E102	Kumar	10000
E112	Kesav	10500

Temp_Employee1 (Range : >14000 and <=24000)

Emp_ID	EName	Salary
E105	Meena	15000
E113	Maddy	15500
E115	Megha	18000

Temp_Employee2 (Range : >24000)

Emp_ID	EName	Salary
E125	Steve	25000
E111	Ramya	26000
E122	Maya	30000

Figure 5 - Status of Temp_Employee 0, 1, and 2 after distributing all ranges from all disks

Step 4:

The final concatenation of sorted data from all the disks is trivial.

The join operation requires that the system test pairs of tuples to see whether they satisfy the join condition; if they do, the system adds the pair to the join output. Parallel join algorithms attempt to split the pairs to be tested over several processors. Each processor then computes part of the join locally.

Partitioned Join

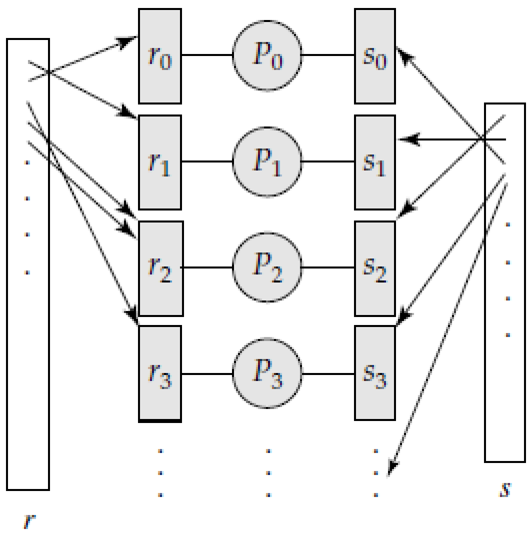
For certain kinds of joins, such as equi-joins and natural joins, it is possible to partition the two input relations across the processors, and to compute the join locally at each processor. Suppose that we are using n processors, and that the relations to be joined are r and s. Partitioned join then works this way: The system partitions the relations r and s each into n partitions, denoted r0, r1, . . . , rn−1 and s0, s1, . . . , sn−1. The system sends partitions ri and si to processor Pi, where their join is computed locally.

The partitioned join technique works correctly only if the join is an equi-join and if we partition r and s by the same partitioning function on their join attributes. The idea of partitioning is exactly the same as that behind the partitioning step of hash–join. In a partitioned join, however, there are two different ways of partitioning r and s:

• Range partitioning on the join attributes

• Hash partitioning on the join attributes

In either case, the same partitioning function must be used for both relations. For range partitioning, the same partition vector must be used for both relations. For hash partitioning, the same hash function must be used on both relations. Figure 20.2 depicts the partitioning in a partitioned parallel join.



Once the relations are partitioned, we can use any join technique locally at each processor Pi to compute the join of ri and si. For example, hash–join, merge join, or nested-loop join could be used. Thus, we can use partitioning to parallelize any join technique.

If one or both of the relations r and s are already partitioned on the join attributes (by either hash partitioning or range partitioning), the work needed for partitioning is reduced greatly. If the relations are not partitioned, or are partitioned on attributes other than the join attributes, then the tuples need to be repartitioned. Each processor Pi reads in the tuples on disk Di, computes for each tuple t the partition j to which t belongs, and sends tuple t to processor Pj . Processor Pj stores the tuples on disk Dj .

Inter-operation parallelism

It is about executing different operations of a query in parallel. A single query may involve multiple operations at once. We may exploit parallelism to achieve better performance of such queries. Consider the example query given below;

SELECT AVG(Salary) FROM Employee GROUP BY Dept_Id;

It involves two operations. First one is an Aggregation and the second is grouping. For executing this query,

We need to group all the employee records based on the attribute Dept_Id first.

Then, for every group we can apply the AVG aggregate function to get the final result.

We can use Interoperation parallelism concept to parallelize these two operations.

[Note: Intra-operation is about executing single operation of a query using multiple processors in parallel]

The following are the variants using which we would achieve Interoperation Parallelism;

1. Pipelined Parallelism

2. Independent Parallelism

1. Pipelined Parallelism

In Pipelined Parallelism, the idea is to consume the result produced by one operation by the next operation in the pipeline. For example, consider the following operation;

$$r1 \bowtie r2 \bowtie r3 \bowtie r4$$

The above expression shows a natural join operation. This actually joins four tables. This operation can be pipelined as follows;

Perform temp1 $\leftarrow$ r1 $\bowtie$ r2 at processor P1 and send the result temp1 to processor P2 to perform temp2 $\leftarrow$ temp1 $\bowtie$ r3 and send the result temp2 to processor P3 to perform result $\leftarrow$ temp2 $\bowtie$ r4.

The advantage is, we do not need to store the intermediate results, and instead the result produced at one processor can be consumed directly by the other. Hence, we would start receiving tuples well before P1 completes the join assigned to it.

Disadvantages:

1. Pipelined parallelism is not the good choice, if degree of parallelism is high.

2. Useful with small number of processors.

3. Not all operations can be pipelined. For example, consider the query given in the first section. Here, you need to group at least one department employees. Then only the output can be given for aggregate operation at the next processor.

4. Cannot expect full speedup.

2. Independent Parallelism:

Operations that are not depending on each other can be executed in parallel at different processors. This is called as Independent Parallelism.

For example, in the expression r1 ⋈ r2 ⋈ r3 ⋈ r4, the portion r1 ⋈ r2 can be done in one processor, and r3 ⋈ r4 can be performed in the other processor. Both results can be pipelined into the third processor to get the final result.

Disadvantages:

Does not work well in case of high degree of parallelism.

References

http://www.faadooengineers.com/

https://www.exploredatabase.com/

Textbook: Database System Concepts by Abraham Silberschatz, H. Korth, Sudarshan