Parallel Sort - Range Partitioning Sort

Assumptions:

Assume n processors, P_0, P_1, …, P_{n-1} and n disks D_0, D_1, …, D_{n-1}.

Disk D_i is associated with Processor P_i.

Relation R is partitioned into R_0, R_1, …, R_{n-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) R_i that resides on n disks on an attribute A in parallel.

Steps:

Step 1: Partition the relations R_i on the sorting attribute A at every processor using a range vector **v**. Send the partitioned records which fall in the i^{th} range to Processor P_i where they are temporarily stored in D_i.

Step 2: Sort each partition locally at each processor P_i. And, send the sorted results for merging with all the other sorted results which is trivial process.

Point to note:

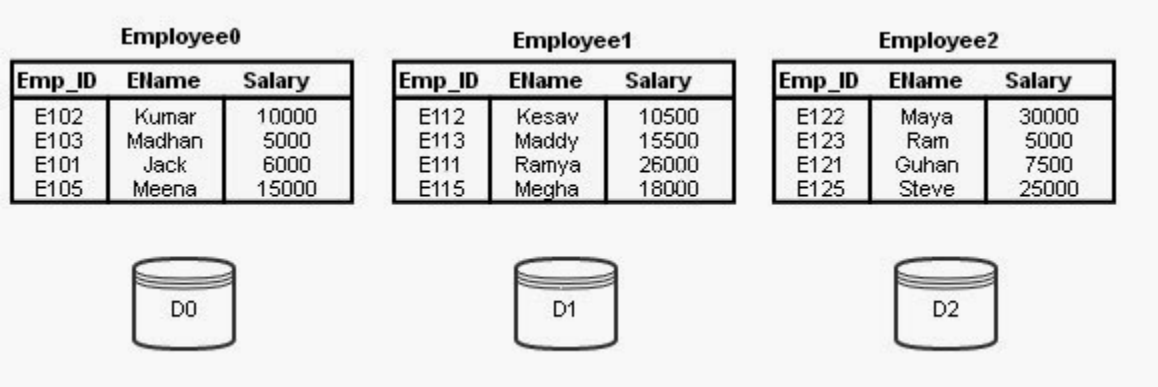
Range partition must be done using a good range-partitioning vector. Otherwise, skew might be the problem.

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 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.



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:

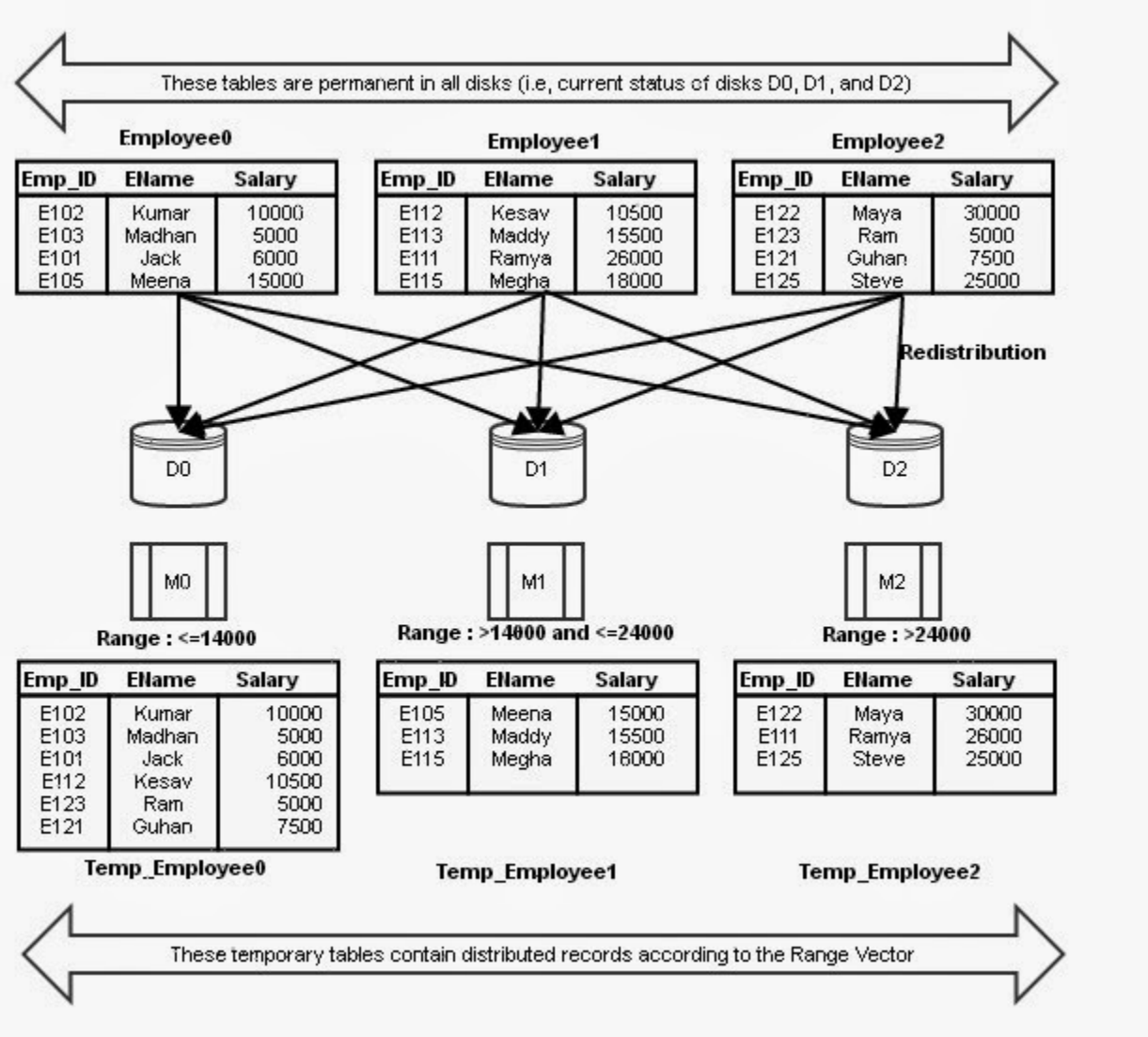
At first we have to identify a range vector v on the Salary attribute. The range vector is of the form **v[v0, v1, …, vn-2]**. For our example, let us assume the following range vector;

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 D_0, D_1, and D_2. (They can also be stored in Main memories (M_0, M_1, M_2) if they fit into RAM).



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.

Parallel External Sort-Merge

Assumptions:

Assume n processors, P_0, P_1, …, P_{n-1} and n disks D_0, D_1, …, D_{n-1}.

Disk D_i is associated with Processor P_i.

Relation R is partitioned into R_0, R_1, …, R_{n-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) R_i on an attribute A in parallel where R resides on n disks.

Steps:

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

Step 2: Identify a range partition vector v and range partition every R_i into processors, P_0, P_1, …, P_{n-1} using vector v.

Step 3: Each processor P_i 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:

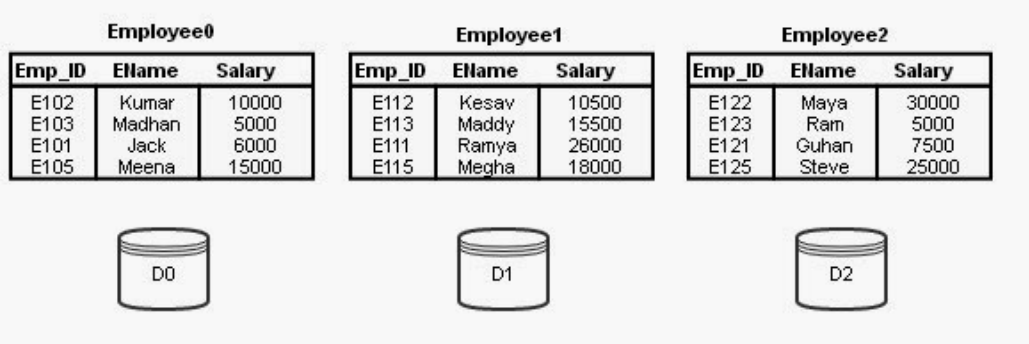
Range partition must be done using a good range-partitioning vector. Otherwise, skew might be the problem.

Example:

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

Employee (Emp_ID, EName, Salary)

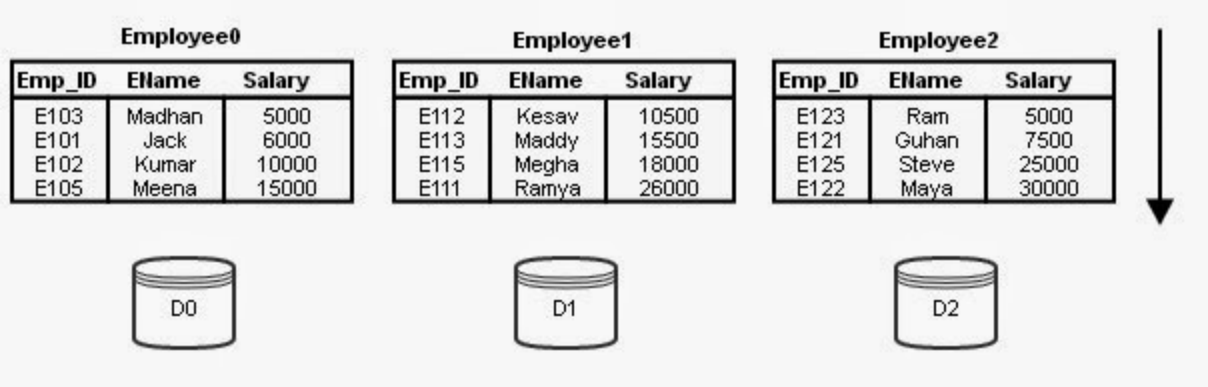
Assume that relation ***Employee is permanently partitioned using Round-robin technique*** into 3 disks D_0, D_1, and D_2 which are associated with processors P_0, P_1, and P_2. At processors P_0, P_1, and P_2, the relations are named Employee0, Employee1 and Employee2 respectively. This initial state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



Assume that the following sorting query is initiated. 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 $Employee_i$ contains salary values of range minimum to maximum. The partitions sorted in ascending order is shown below, in Figure 2.



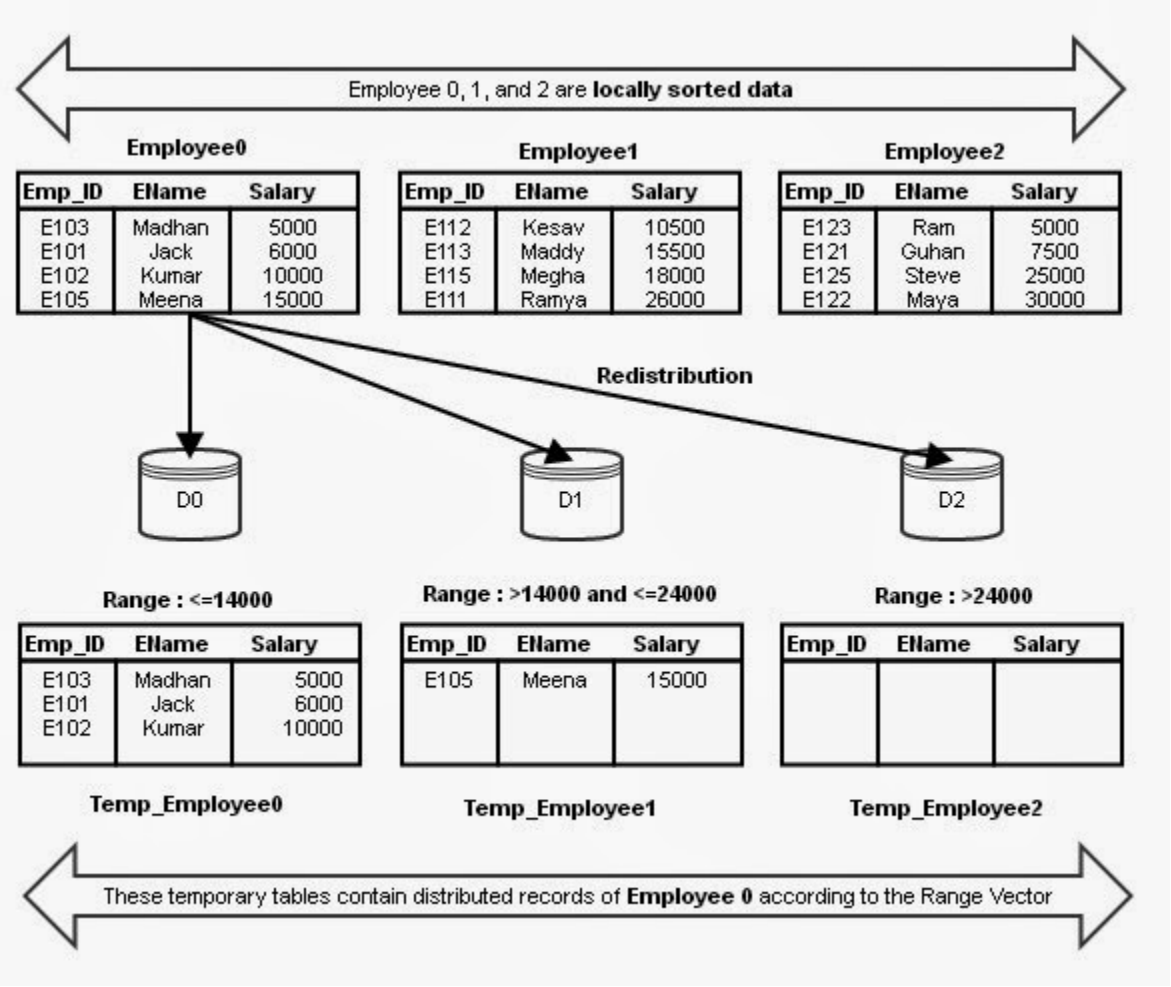
Step 2:

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]

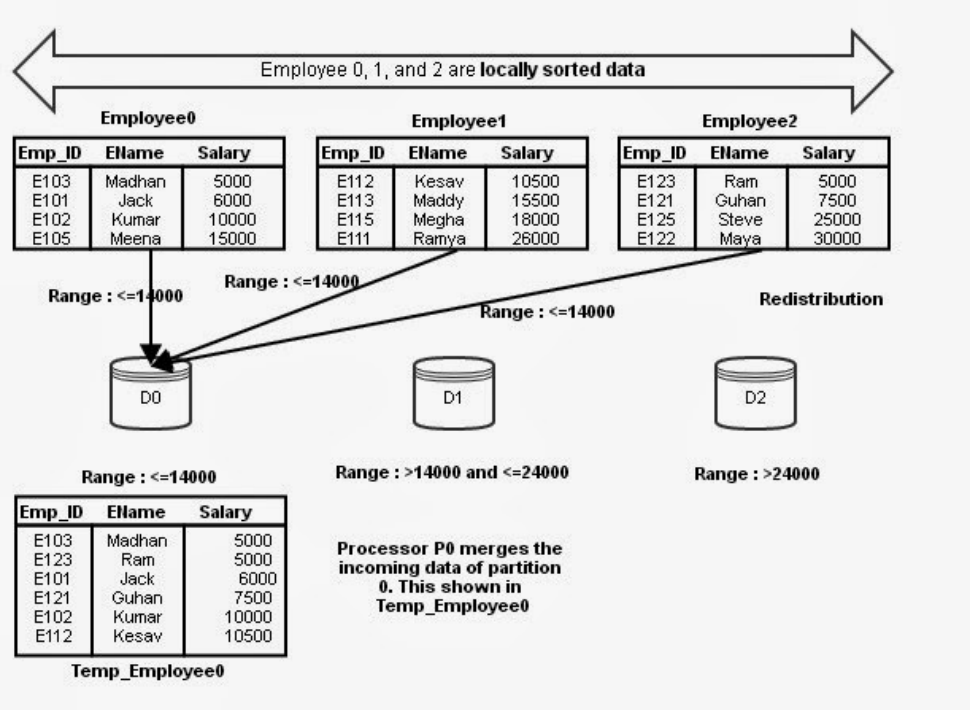
This range vector represents 3 ranges, range 0 (14000 and less), range 1 (14001 to 24000) and range 2 (24001 and more).

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.

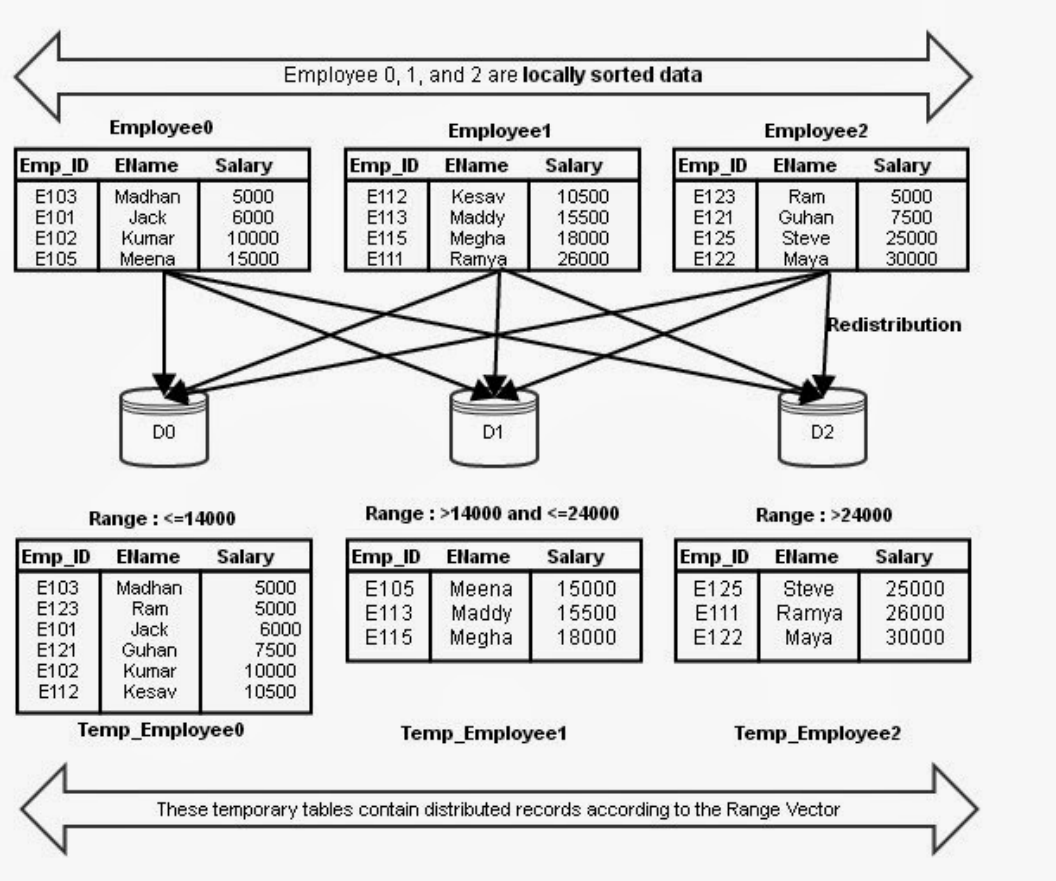


Step 3: Actually, the above said distribution is executed at all processors in parallel such that processors P0, P1, and P2 are sending the first partition of Employee 0, 1, and 2 to disk 0. Upon

receiving the records from various partitions, the receiving processor P0 merges the sorted data. This is shown in Figure 4.



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.



Step 4:

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

Parallel Join – Partitioned Join

The relational tables that are to be joined gets partitioned on the joining attributes of both tables using same partitioning function to perform Join operation in parallel.

How does Partitioned Join work?

Assume that relational tables r and s need to be joined using attributes r.A and s.B. The system partitions both r and s using same partitioning technique into n partitions. In this process A and B attributes (joining attributes) to be used as partitioning attributes as well for r and s respectively. r is partitioned into $\mathbf{r_0}$, $\mathbf{r_1}$, $\mathbf{r_2}$, …, $\mathbf{r_{n-1}}$ and s is partitioned into $\mathbf{s_0}$, $\mathbf{s_1}$, $\mathbf{s_2}$, …, $\mathbf{s_{n-1}}$. Then, the system sends partitions $\mathbf{r_i}$ and $\mathbf{s_i}$ into processor $\mathbf{P_i}$, where the join is performed locally.

What type of joins can we perform in parallel using Partitioned Join?

Only joins such as Equi-Joins and Natural Joins can be performed using Partitioned join technique.

Why Equi-Jion and Natural Jion?

Equi-Join or Natural Join is done between two tables using an equality condition such as r.A = s.B. The tuples which are satisfying this condition, i.e, same value for both A and B, are joined together. Others are discarded. Hence, if we partition the relations r and s on applying certain partitioning technique on both A and B, then the tuples having same value for A and B will end up in the same partition. Let us analyze this using simple example;

RegNo	*SName*	*Gen*	*Phone*
1	Sundar	M	9898786756
3	Karthik	M	8798987867
4	John	M	7898886756
2	Ram	M	9897786776

Table 1 - STUDENT

RegNo	*Courses*
4	Database
2	Database
3	Data Structures
1	Multimedia

Table 2 – COURSES_REGD

Let us assume the following;

The RegNo attributes of tables STUDENT and COURSES_REGD are used for joining.

Observe the order of tuples in both tables. They are not in particular order. They are stored in random order on RegNo.

Partition the tables on RegNo attribute using Hash Partition. We have 2 disks and we need to partition the relational tables into two partitions (possibly equal). Hence, n is 2.

The hash function is, h(RegNo) = (RegNo mod n) = (RegNo mod 2). And, if we apply the hash function we shall get the tables STUDENT and COURSES_REGD partitioned into $Disk_0$ and $Disk_1$ as stated below.

Partition 0

RegNo	SName	Gen	Phone
4	John	M	7898886756
2	Ram	M	9897786776

STUDENT_0

RegNo	Courses
4	Database
2	Database

COURSES_REGD_0

Partition 1

RegNo	SName	Gen	Phone
1	Sundar	M	9898786756
3	Karthik	M	8798987867

STUDENT_1

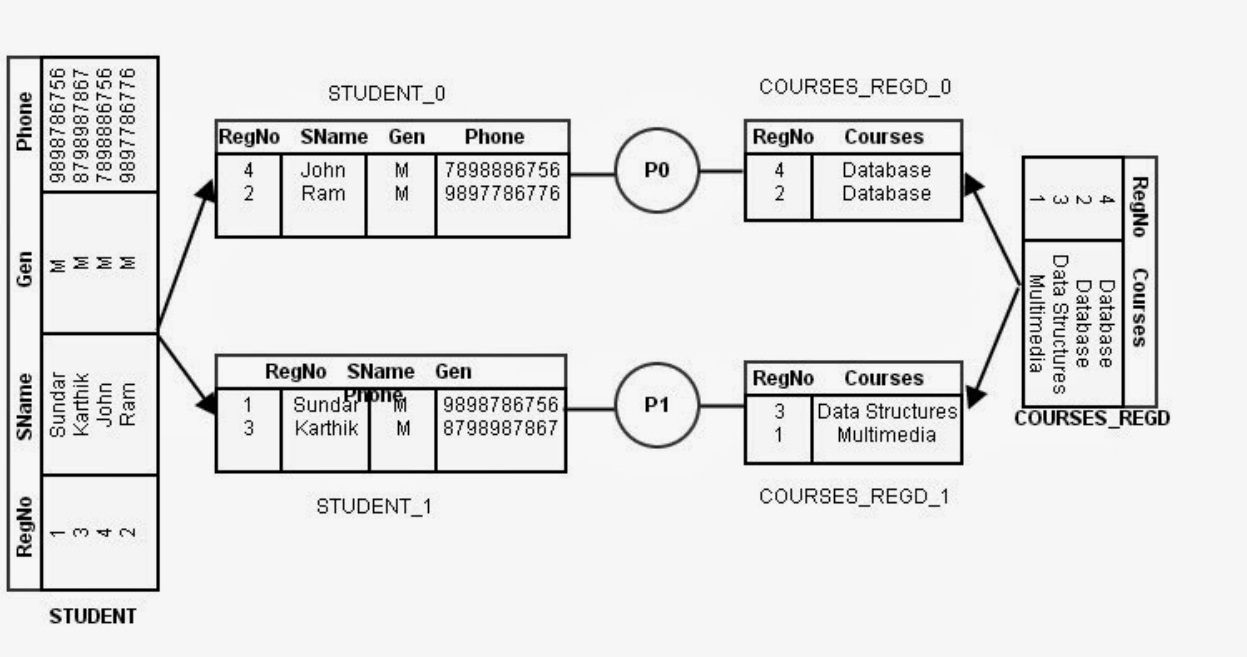
RegNo	Courses
3	Data Structures
1	Multimedia

COURSES_REGD_1

From the above table, it is very clear that the same RegNo values of both tables STUDENT and COURSES_REGD are sent to same partitions. Now, join can be performed locally at every processor in parallel.

One more interesting fact about this join is, only 4 (2 Student records X 2 Courses_regd records) comparisons need to be done in every partition for our example. Hence, we need total of 8 comparisons in partitioned join against 16 (4 X 4) in conventional join.

The above discussed process is shown in Figure 1.



Points to note:

1. There are only two ways of partitioning the relations,
   Range partitioning on the join attributes or
   Hash partitioning on the join attributes.
2. Only equi-joins and natural joins can be performed in parallel using Partitioned Join technique.
3. Non-equi-joins cannot be performed with this method.
4. After successful partitioning, the records at every processor can be joined locally using any of the joining techniques hash join, merge join, or nested loop join.

5. If Range partitioning technique is used to partition the relations into n processors, **Skew** may present a special problem. That is, for some partitions, we may get fewer records (tuples) for one relation for a given range and many records for other relation for the same range.

6. With Hash partitioning, if there are many tuples with same value in one relation then the difference both relations is possible in one partition. Otherwise, skew has minimal effect.

7. The number of comparisons between relations are well reduced in partitioned join parallel technique.

Fragment and Replicate Join

Introduction

All of us know about different types of joins in terms of the nature of the join conditions and the operators used for join. They are Equi-join and Non-Equi Join. Equi-join is performed through checking an equality condition between different joining attributes of different tables. Non-equi join is performed through checking an inequality condition between joining attributes.

Equi-join is of the form,

SELECT columns FROM list_of_tables WHERE table1.column = table2.column;

whereas, Non-equi join is of the form,

SELECT columns FROM list_of_tables WHERE table1.column < table2.column;

(Or, any other operators >, <>, <=, >= etc. in the place of < in the above example)

We have discussed Partitioned Join in the previous post, where we partitioned the relational tables that are to be joined, into equal partitions and we performed join on individual partitions locally at every processor. Partitioning the relations on the joining attribute and join them will work only for joins that involve equality conditions.

Clearly, joining the tables by partitioning will work only for Equi-joins or natural joins. For inequality joins, partitioning will not work. Consider a join condition as given below;

$$r \bowtie_{r.a>s.b} s$$

In this non-equal join condition, the tuples (records) of r must be joined with records of s for all the records where the value of attribute r.a is greater than s.b. In other words, ***all records of r join with some records of s and vice versa***. That is, one of the relations' all records must be joined with some of the records of other relation. For clear example, see Non-equi join post.

What does fragment and replicate mean?

Fragment means partitioning a table either horizontally or vertically (Horizontal and Vertical fragmentation). Replicate means duplicating the relation, i.e, generating similar copies of a table. This join is performed by fragmenting and replicating the tables to be joined.

Asymmetric Fragment and Replicate Join

(How does Asymmetric Fragment and Replicate Join work?)

It is a variant of Fragment and Replicate join. It works as follows;

1. The system fragments table r into n fragments such that r_0, r_1, r_2, .., r_{n-1}, where r is one of the tables that is to be joined and n represents the number of processors. Any partitioning technique, round-robin, hash or range partitioning could be used to partition the relation.

2. The system replicates the other table, say s into n processors. That is, the system generates n copies of s and sends to n processors.

3. Now we have, r_0 and s in processor P_0, r_1 and s in processor P_1, r_2 and s in processor P_2, .., r_{n-1} and s in processor P_{n-1}. The processor Pi is performing the join locally on r_i and s.

Figure 1 given below shows the process of Asymmetric Fragment-and-Replicate join (it may not be the appropriate example, but it clearly shows the process);

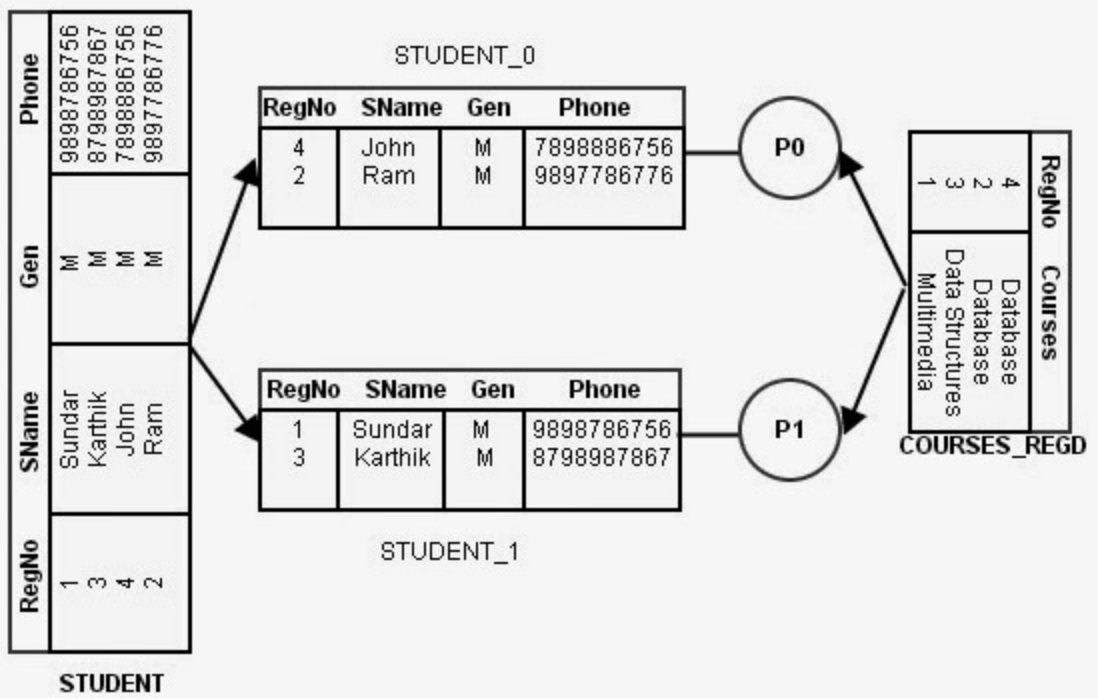


Figure 1 - process of Asymetric Fragment and Replicate Parallel Join

Points to Note:

1. Non-equal join can be performed in parallel.

2. If one of the relations to be joined is already partitioned into n processors, this technique is best suited, because we need to replicate the other relation.

3. Unlike in Partitioned Join, any partitioning techniques can be used.

4. If one of the relations to be joined is very small, the technique performs better.

Fragment and Replicate Join
(How does Fragment and Replicate Join work?)

It is the general case of Asymmetric Fragment-and-Replicate join technique. Asymmetric technique is best suited if one of the relations to be joined is small, and if it can fit into memory. If the relations that are to be joined are large, and the joins is non-equal then we need to use Fragment-and-Replicate Join. It works as follows;

1. The system fragments table r into m fragments such that r_0, r_1, r_2, .., r_{m-1}, and s into n fragments such that s_0, s_1, s_2, .., s_{n-1} . Any partitioning technique, round-robin, hash or range partitioning could be used to partition the relations.

2. The values for m and n are chosen based on the availability of processor. That is, we need at least m*n processors to perform join.

3. Now we have to distribute all the partitions of r and s into available processors. And, ***remember that we need to compare every tuple of one relation with every tuple of other relation. That is the records of r_0 partition should be compared with all partitions of s, and the records of partition s_0 should be compared with all partitions of r.*** This must be done with all the partitions of r and s as mentioned above. Hence, the data distribution is done as follows;

a. As we need m*n processors, let us assume that we have processors $P_{0,0}$, $P_{0,1}$, …, $P_{0,n-1}$, $P_{1,0}$, $P_{1,1}$, …, $P_{m-1,n-1}$. Thus, processor $P_{i,j}$ performs the join of r_i with s_j.

b. To ensure the comparison of every partition of r with every other partition of s, we replicate r_i with the processors, $P_{i,0}$, $P_{i,1}$, $P_{i,2}$, …, $P_{i,n-1}$, where 0, 1, 2, …, n-1 are partitions of s. This replication ensures the comparison of every r_i with complete s.

c. To ensure the comparison of every partition of s with every other partition of r, we replicate s_i with the processors, $P_{0,i}$, $P_{1,i}$, $P_{2,i}$, …, $P_{m-1,i}$, where 0, 1, 2, …, m-1 are partitions of r. This replication ensures the comparison of every s_i with complete r.

4. $P_{i,j}$ computes the join locally to produce the join result.

Figure 2 given below shows the process of general case Fragment-and-Replicate join (it may not be the appropriate example, but it clearly shows the process);

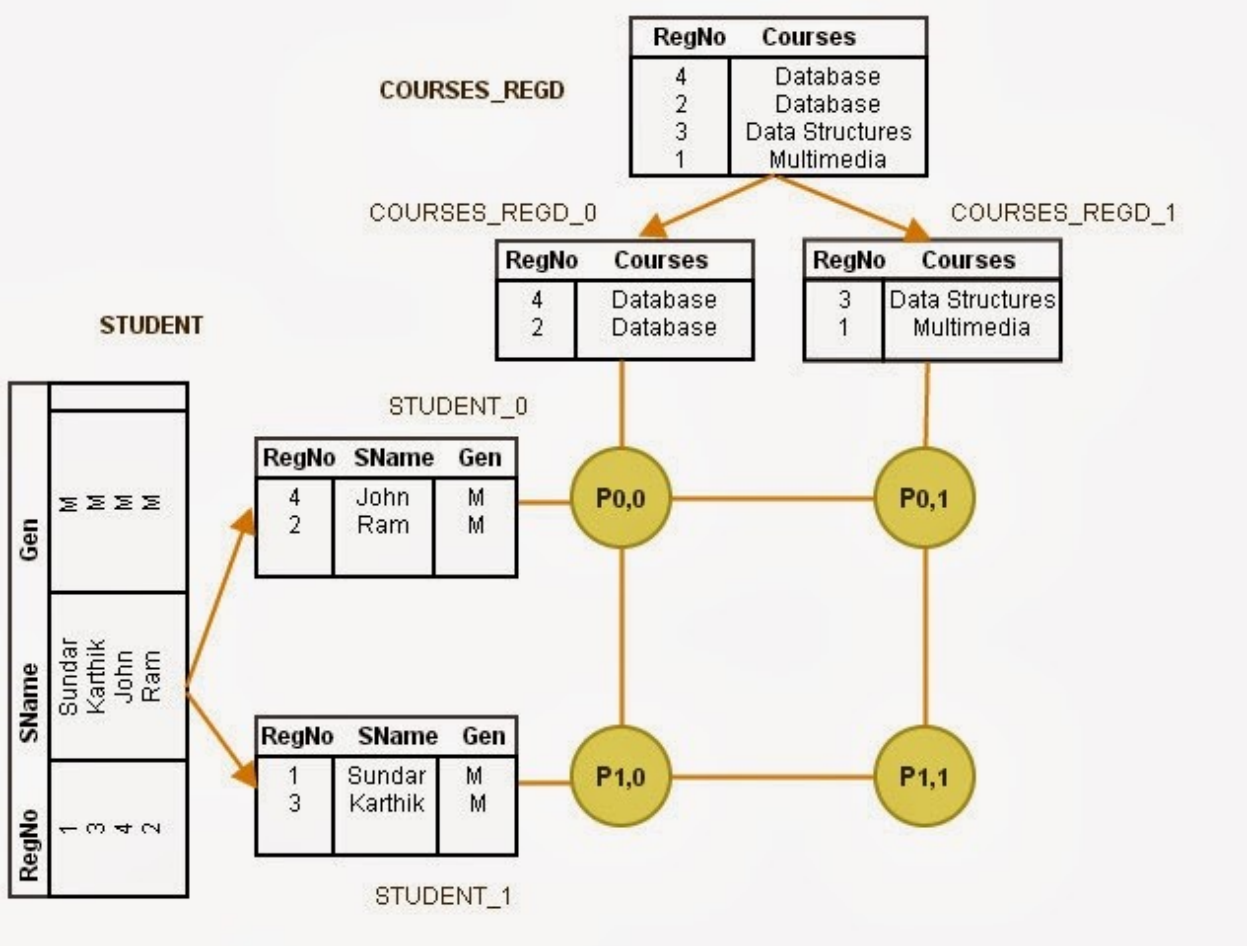


Figure 2 - process of general case Fragment-and-Replicate Join

Points to Note:

1. Asymmetric Fragment-and-replicate join is the special case of general case Fragment-and-replicate join, where n or m is 1, i.e, if one of the relation does not have partitions.

2. When compared to asymmetric technique, Fragment-and-replicate join reduces the size of the tables at every processor.

3. Any partitioning techniques can be used and any joining technique can be used as well.

4. Fragment-and-replicate technique suits both Equi-join and Non-equi join.

5. It involves higher cost in partitioning.

Partitioned Parallel Hash Join in Parallel Database / Parallel Hash Join Technique

The Sequential Hash Join technique discussed in the post Hash Join technique in DBMS can be parallelized.

The Technique

Please go through the post Hash Join technique in DBMS for better understanding of Parallel Hash-Join.

Assume that we have,

- n processors, P0, P1, …, Pn-1
- two tables, r and s (r and s are already partitioned into disks of n processors),
- s is the smaller table

Parallel Hash-Join Algorithm:

Step 1: Choose a hash function *h1*. *h1* should be able to take the ***join attribute value*** of each tuple of relations r and s, and maps them to one of the n processors.

Step 2: Smaller relations s is taken as the Build relation.

Step 3: Each processor Pi reads the tuples of smaller relation s in it's disk Di, and sends them to different processors based on the hash function h1.

Step 4: On receiving the records of s_i at every destination processor Pi, the Pi further partitions the tuples using another hash function *h2*. This hash function is used for joining the tuples locally at every processor. This step is independent for any processor.

Step 5: On completion of the distribution of all the records of smaller relation s, it is now the turn for the larger relation r. Step 3 is done for all the records of relation r.

Step 6: On receiving the records of r_i at every destination processor Pi, the Pi further partitions the tuples using another hash function *h2*. This phase is called the Probe Phase.

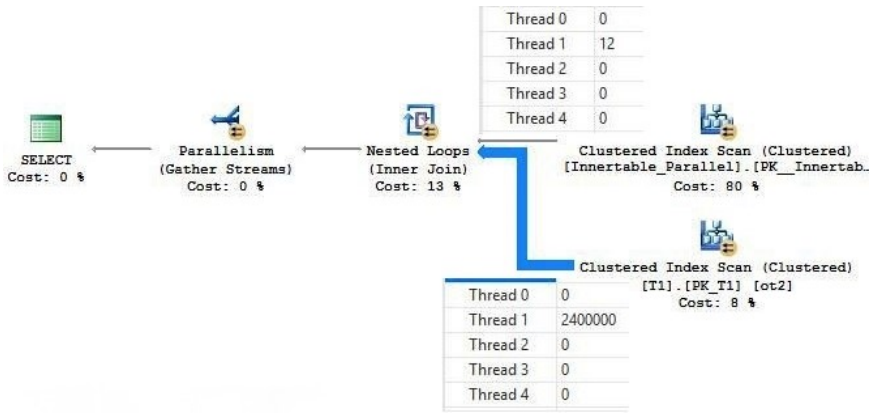
Step 7: At last, all the records which can be joined will be in same processors' same partitions. Each processor Pi executes the build and probe phases of the hash-join algorithm on the local partitions r_i and s_i of r and s to produce a partition of the final result of the hash-join.

Points to remember:

Hash Join at each processor is independent of the other.

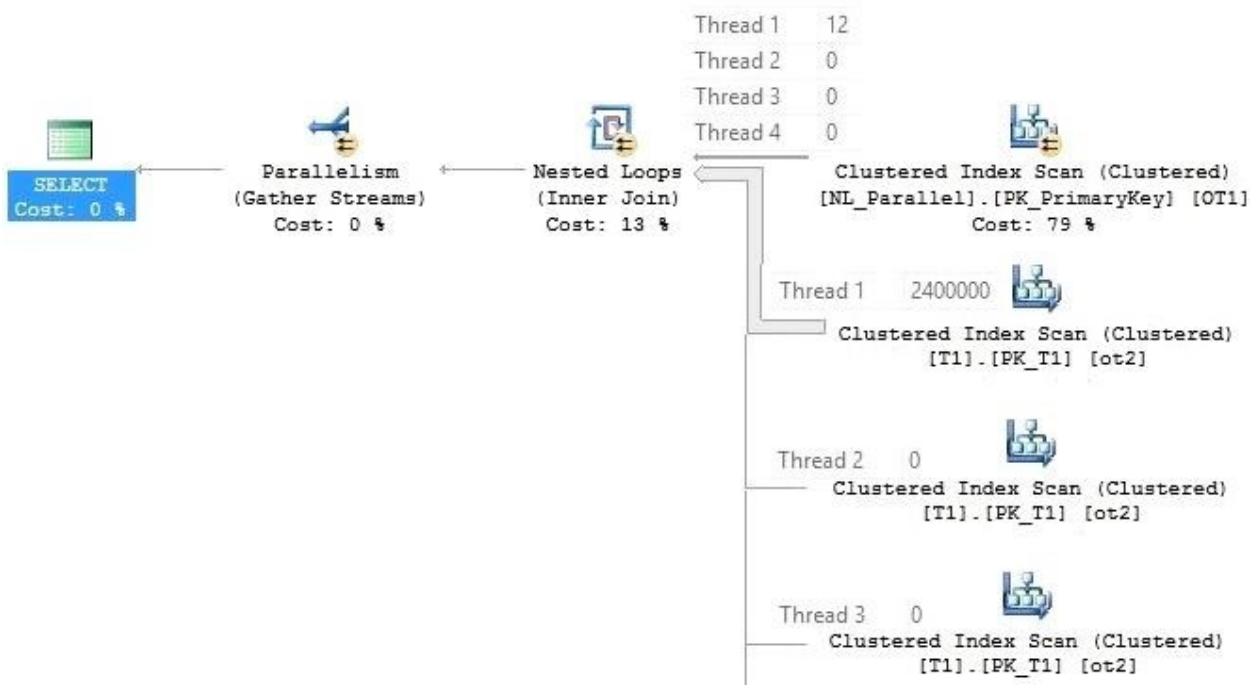
It works for equi-join and natural join conditions

Parallel Nested-Loop Join

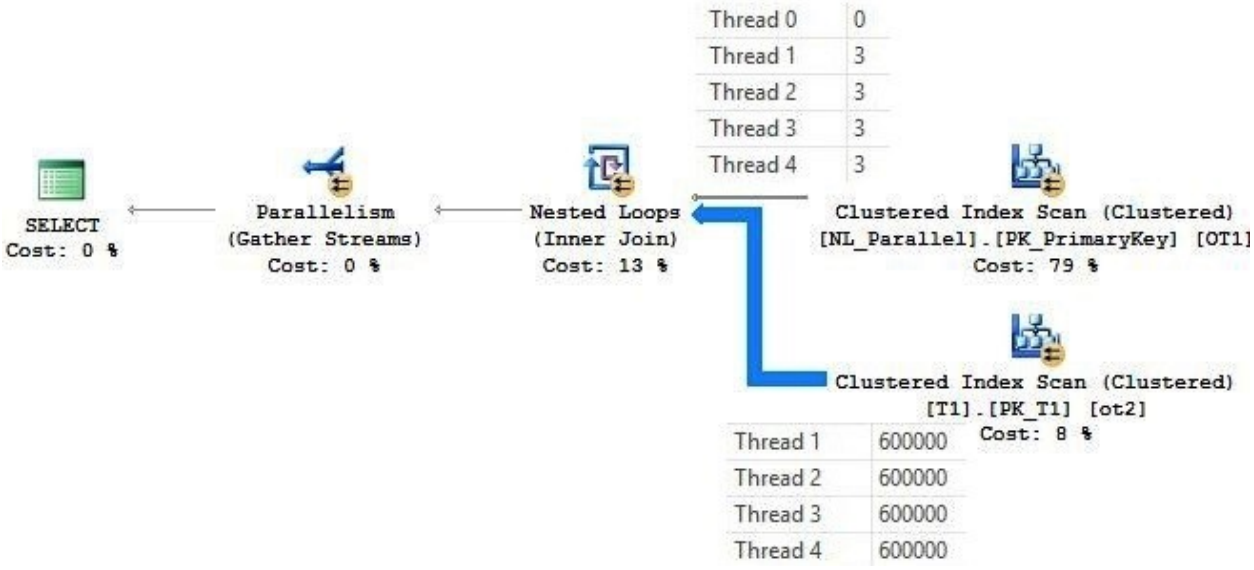


Have a look at the above query plan's outer table. There exists a parallel scan, thus parallel page supplier (parallel page supplier is not a part of the relational engine so it cannot be seen in the execution plan) distributes a bunch of rows to all the threads. Then the filter is applied on all the active threads. All the rows that satisfy the filter predicate are in the thread 1 (Depends on the timings and data structure of the table, if you keep running the query you might see the rows in different threads).

The inner side of the parallel Nested Loop Join executes as per the outer side of the thread's execution. To have a better understanding we have elaborated the inner side of an execution plan.



As we can see in the outer table, only thread 1 has all the filtered rows, other threads do not have any rows at all. Thus, on the inner side of the loop join only thread 1 will take part in this activity while the rest of the threads won't able to take part in this.



Now look at the above execution plan on the NL_Parallel table. All four threads have equal amount of rows and, because of that, the inner side of execution processed is parallel. This execution plan is slightly better than the above execution plan, as it uses the Parallelism evenly in the inner table.

Other Relational Operations:

- **Selection**
- **Duplicate elimination**
- **Projection**
- **Aggregation**

Cost of Parallel Evaluation of Operations:

We achieve parallelism by partitioning the I/O among multiple disks, and partitioning the CPU work among multiple processors. If such a split is achieved without any overhead, and if there is no skew in the splitting of work, a parallel operation using n processors will take 1/n times as long as the same operation on a single processor. We already know how to estimate the cost of an operation such as a join or a selection. The time cost of parallel processing would then be 1/n of the time cost of sequential processing of the operation.

We must also account for the following costs:
• Start-up costs for initiating the operation at multiple processors.
• Skew in the distribution ofwork among the processors,with some processors getting a larger number of tuples than others.
• Contention for resources—such as memory, disk, and the communication network—resulting in delays.
• Cost of assembling the final result by transmitting partial results from each processor.
The time taken by a parallel operation can be estimated as:

$$T_{\text{part}} + T_{\text{asm}} + \max(T_0, T_1, \ldots, T_{n-1})$$

where *T*part is the time for partitioning the relations, *T*asm is the time for assembling the results, and *Ti* is the time taken for the operation at processor *Pi* . Assuming that the tuples are distributed without any skew, the number of tuples sent to each processor can be estimated as 1*/n* of the total number of tuples. The preceding estimate will be an optimistic estimate, since skew is common. Even though breaking down a single query into a number of parallel steps reduces the size of the average step, it is the time for processing the single slowest step that determines the time taken for processing the query as a whole. Thus, any skew in the distribution of the work across processors greatly affects performance. The problem of skew in partitioning is closely related to the problem of partition overflow in sequential hash joins.

Interoperation 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]

Types of Interoperation Parallelism: 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 $\bowtie$ r2 $\bowtie$ r3 $\bowtie$ r4, the portion r1 $\bowtie$ r2 can be done in one processor, and r3 $\bowtie$ 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.

Query Optimization:

Query optimizers for parallel query evaluation are more complicated than query optimizers for sequential query evaluation. First, the cost models are more complicated, since partitioning costs have to be accounted for, and issues such as skew and resource contention must be taken into account.

To evaluate an operator tree in a parallel system, we must make the following decisions:

- How to parallelize each operation, and how many processors to use for it.
- What operations to pipeline across different processors, what operations to execute independently in parallel, and what operations to execute sequentially, one after the other.

These decisions constitute the task of scheduling the execution tree. Determining the resources of each kind—such as processors, disks, and memory— that should be allocated to each operation in the tree is another aspect of the optimization problem.

Disadvantage: One concern is that long pipelines do not lend themselves to good resource utilization. Unless the operations are coarse grained, the final operation of the pipeline may wait for a long time to get inputs, while holding precious resources, such as memory. Hence, long pipelines should be avoided.

Heuristic approaches to reduce the number of parallel execution plans that we have to consider. We describe two popular heuristics here.

The first heuristic is to consider only evaluation plans that parallelize every operation across all processors, and that do not use any pipelining. This approach is used in the Teradata systems. Finding the best such execution plan is like doing query optimization in a sequential system.

The second heuristic is to choose the most efficient sequential evaluation plan, and then to parallelize the operations in that evaluation plan. The Volcano parallel database popularized a model of parallelization called the exchange-operator model. This model uses existing implementations of operations, operating on local copies of data, coupled with an exchange operation that moves data around between different processors.

Design of Parallel Systems:

Parallel loading of data from external sources is an important requirement, if we are to handle large volumes of incoming data. A large parallel database system must also address these availability issues:

- Resilience to failure of some processors or disks.
- Online reorganization of data and schema changes.

With a large number of processors and disks, the probability that at least one processor or disk will malfunction is significantly greater than in a singleprocessor system with one disk. A poorly designed parallel system will stop functioning if any component (processor or disk) fails. Assuming that the probability of failure of a single processor or disk is small, the probability of failure of the system goes up linearly with the number of processors and disks.

Data are replicated across at least two processors. If a processor fails, the data that it stored can still be accessed from the other processors. The system keeps track of failed processors and distributes the work among functioning processors. Requests for data stored at the failed site are automatically routed to the backup sites that store a replica of the data. If all the data of a processor Aare replicated at a single processor B, B will have to handle all the requests to A as well as those to itself, and that will result in B becoming a bottleneck. Therefore, the replicas of the data of a processor are partitioned across multiple other processors.

In recent years, a number of companies have developed newparallel database products, including Netezza, DATAllegro (which was acquired by Microsoft), Greenplum, and Aster Data. Each of these products runs on systems containing tens to thousands of nodes, with each node running an instance of an underlying database; Each product manages the partitioning of data, as well as parallel processing of queries, across the database instances. Netezza, Greenplum and Aster Data use PostgreSQL as the underlying database; DATAllegro originally used Ingres as the underlying database system, but moved to SQL Server subsequent to its acquisition by Microsoft.

Parallelism on Multicore Processors:

Parallelism has become commonplace on most computers today, even some of the smallest, due to current trends in computer architecture.

Parallelism versus Raw Speed:

This increase results from an exponential growth in the number of transistors that could be fit within a unit area of a silicon chip, and is known popularly as Moore's law, named after Intel co-founder Gordon Moore. Technically, Moore's law is not a law, but rather an observation and a prediction regarding technology trends. Until recently, the increase in the number of transistors and the decrease in their size led to ever-faster processors.

Fast processors are power inefficient. This is problematic in terms of energy consumption and cost, battery life for mobile computers, and heat dissipation. As a result, modern processors typically are not one single processor but rather consist of several processors on one chip. To maintain a distinction between on-chip multiprocessors and traditional processors, the term core is used for an on-chip processor. Thus we say that a machine has a multicore processor.

Cache Memory and Multithreading

Processors are able to process data faster than it can be accessed from main memory, main memory can become a bottleneck that limits overall performance. For this reason, computer designers include one or more levels of cache memory in a computer system. Cache memory is more costly than main memory on a per-byte basis, but offers a faster access time. The computer hardware maintains control over the transfer of data among the various levels of cache and between cache and main memory. Despite this lack of direct control, the database system's performance can be affected by how cache is utilized. Because main memory is so much slower than processors, a significant amount of potential processing speed may be lost while a core waits for data from main memory. These waits are referred to as **cache misses.**

A thread is an execution stream that shares memory with other threads running on the same core. If the thread currently executing on a core suffers a cache miss (or other type of wait), the core proceeds to execute another thread, thereby not wasting computing speed while waiting. Each new generation of processors supports more cores and more threads. **The Sun UltraSPARC T2 processor** has **8 cores**, each of which supports **8 threads**, for a total of 64 threads on one processor chip.

Adapting Database System Design for Modern Architectures

As we allow a higher degree of concurrency to take advantage of the parallelism of modern processors, we increase the amount of data that needs to be in cache. This can result in more cache misses, perhaps so many that even a multithreaded core has to wait for data from memory. Concurrent transactions need some sort of concurrency control to ensure the ACID properties, when concurrent transactions access data in common, some sort of restrictions must be imposed on that concurrent access. Those restrictions, whether based on locks, timestamps, or validation, result in waiting or the loss of work due to transaction aborts. Finally, there are components of a database system shared by all transactions. In a system using locking, the lock table is shared by all transactions and access to it can become a bottleneck. Similar problems exist for other forms of concurrency control. Similarly, the buffer manager, the log manager, and the recovery manager serve all transactions and are potential bottlenecks.

****