**Parallelism** in databases: speedup and scale-up.

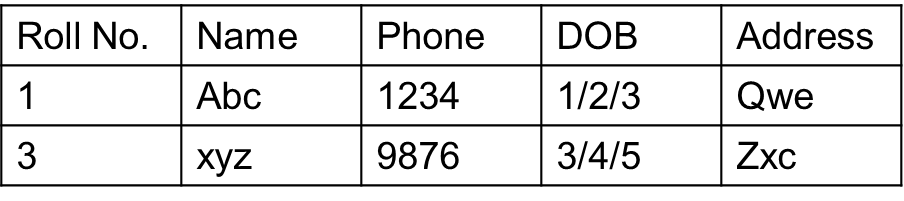
- It enables faster query execution using multiple resources (processors and disks).

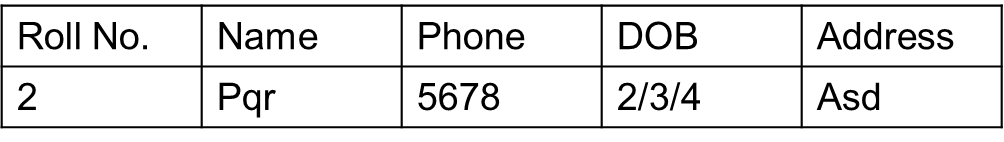
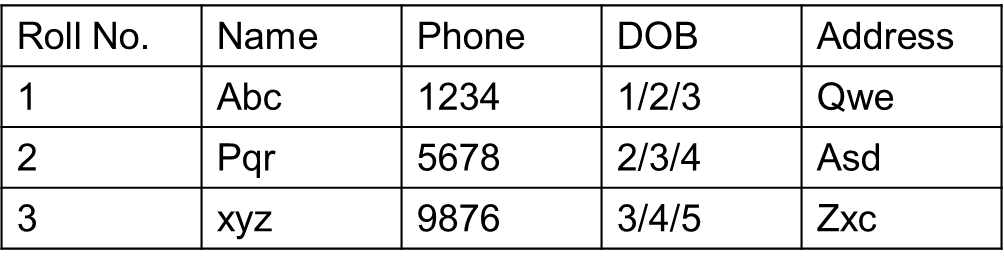
- Queries decomposed into parts that can work simultaneously, facilitating parallel execution.

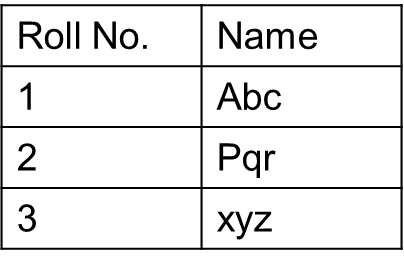
- Data partitioning across multiple disks enables parallel I/O operations.

- Relational operations like sorting, joining, and aggregation can be executed concurrently.

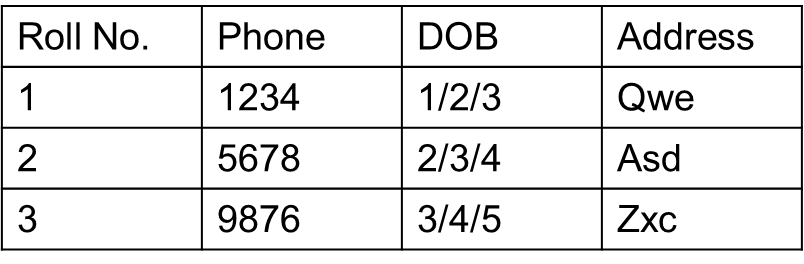
- High-level query languages like SQL make parallelization more accessible.

- Concurrency control mechanisms ensure different queries can run in parallel w/o conflicts.





Vertical Partitioning Horizontal Partitioning



**Vertical Partitioning:**

*Pros:*

* *Improved Query Performance:* Only specific columns’ queries benefit, as they access smaller tables with fewer columns, reducing I/O.
* *Maintenance and Indexing:* Easier to manage and index specific columns. Indexing becomes more efficient as indexes are created on smaller tables.
* *Data Integrity:* Provide more flexibility in managing different types of data.
* *Use Cases:* Suitable for tables with a wide range of columns where not all columns are frequently accessed together.

*Cons:*

* *Joins:* May increase query complexity and processing time, to access columns of other partitions using “Join”.
* *Data Distribution:* Some columns might logically belong together based on their reln or characteristics, but they end up being separated into different partitions. Complex.
* *Scalability:* If the data distribution pattern changes frequently, it may result in uneven distribution of data (performance issues and inefficiencies in query processing).

**Horizontal Partitioning:**

*Pros:*

* *Improved Query Performance:* The database can access only relevant partition, reducing I/O & improving query speed.
* *Data Distribution:* Rows with related data are placed in the same table, which aligns well with grouping of data based on criteria like ranges.
* *Indexing:* Each partition can have its own index, improving query performance and reducing maintenance overhead by allowing for independent index management.
* *Data Integrity:* Partitioning maintains data integrity by isolating subsets of data into separate partitions, reducing the risk of conflicts and inconsistencies across the entire dataset.

*Cons:*

* *Joins:* Joins not needed, but managing and querying data can still introduce complexity, especially when data from multiple partitions need to be combined.
* *Maintenance Overhead:* Managing multiple partitions can introduce extra complexity, especially as the number of partitions increases.
* *Use Cases:* It may not be suitable for tables with a wide range of columns or complex data relationships.

**Hash Partitioning:**

Uses a hash function on partitioning attributes to distribute rows across disks. When we apply hash function to data (rows?), it calculates a unique numeric value (output of hash, numbers like 1, 2, 3) based on data, which helps decide on which disk the data should be stored (if a number is 1, then disk 1, for 5 its disk 5, etc.)

*Advantages:*

* Good for sequential access.
* Provides even distribution of tuples/records between disks.
* Good at quickly finding specific data based on the attributes used to divide the data.

*Disadvantages:*

* No clustering (related entries close together), making range queries challenging.

**Range Partitioning:**

Assigns continuous attribute value ranges to different disks. (For example, values less than 5 are assigned to disk 0, values between 5 and 40 go to disk 1, and values greater than 40 are placed on disk 2).

*Advantages:*

* Organizes data based on the value of a specific attribute.
* It works well for going through data sequentially and quickly finding specific information using the partitioning attributes.

*Disadvantages:*

* May limit data amount we can access from multiple disks, affecting speed of queries.
* Risk of uneven distribution of workload among disks, causing execution skew.

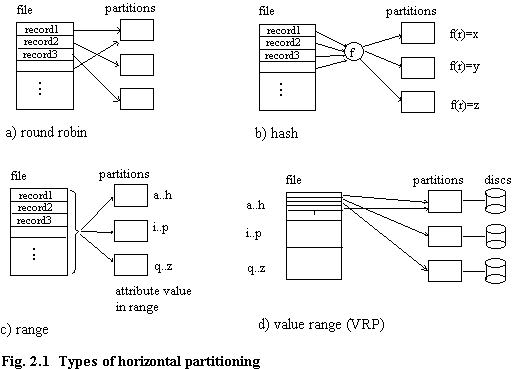
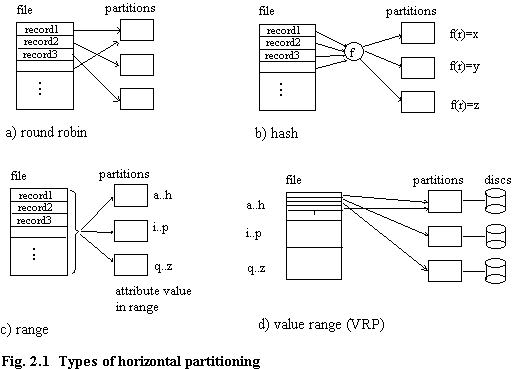
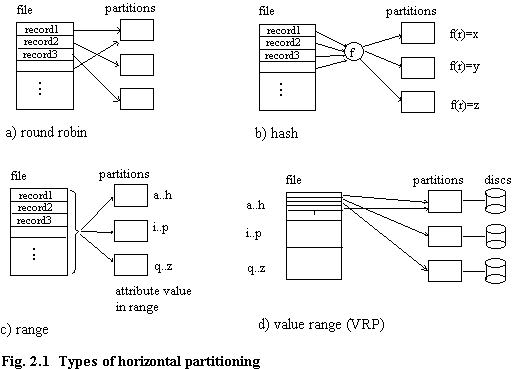
**Round-Robin Partitioning:**

Sends tuples to disks in a cyclic order, evenly distributing them. (Modulo %)

*Advantages:*

* Ideal for scanning through the entire dataset in sequence for every query.
* Evenly distributes data tuples across disks, ensuring balanced workload distribution.

*Disadvantages:*

* Range queries can be challenging to execute efficiently due to the scattered distribution of tuples across disks.
* No clustering, tuples being scattered across all disks = some queries less efficient.
* *Round Robin:* Best for sequential scans.
* *Hash:* Efficient for point queries on partitioning attributes.
* *Range:* Suitable for range queries, less efficient for large fetches.
* *Locating Tuple Associatively:* Hard in RoundRobin, balanced workload in Hash.
* *Locating Tuples within Specified Range:* Range queries challenging in Round Robin, suitable in Range.

**Skew(Asymmetry) in Tuple Distribution:**

- *Types of Skew:*

1. *Attribute-Value Skew:*

* Occurs when specific values in the partitioning attribute are highly frequent. As a result, tuples with the same value for the partitioning attribute tend to concentrate in a single partition.
* Examples: Date attribute partition.
* Skewed distribution affects query performance as accessing data related to the skewed attribute may increase load on a particular partition.
* Relevant to both range-partitioning and hash-partitioning techniques.

1. *Partition Skew:*

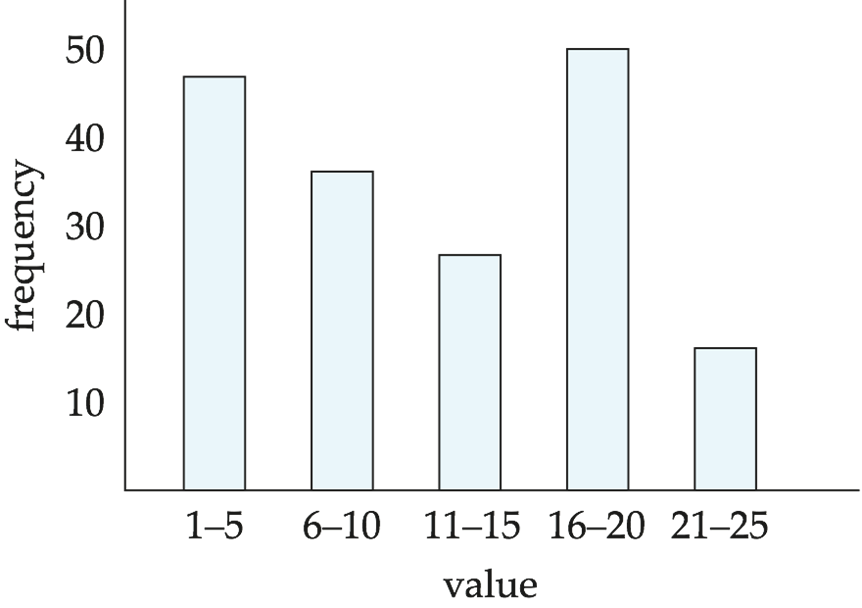
* Associated with range-partitioning, where an inadequately chosen partition can result in uneven distribution of tuples.
* Example: If the partition vector in range-partitioning is not well-marked, some partitions may have more tuples than others.
* Uneven distribution affects the workload balance = performance issues.
* Less likely to occur with hash-partitioning, especially if a well-designed hash function is used.

**Balancing / Even Distribution Techniques for Range-Partitioning:**

1. *Sorting and Constructing Partition Vector:*

* Sort the relation based on the partitioning attribute, helps to organize data.
* After Sorting, build the partition vector by scanning the sorted relation. For example, if there are 10 partitions, the partition vector would be created by adding the attribute value after every 1/10th of the relation is read.
* If the partitioning attribute has duplicate values and the sorting process doesn't account for them properly, the partition vector may end up with duplicate entries, leading to uneven distribution of data among partitions.

1. *Histogram-Based Technique:*

* Construct a balanced partitioning vector using a histogram.
* It shows the distribution of data values within each range of the attribute.
* Assumes a uniform distribution within each range, ensures that each partition receives fair share of data.
* Histogram can be built by scanning the relation or sampling tuples.

**Handling Skew in Range Partitioning with Virtual Node Partitioning:**

* Instead of directly partitioning the data among the physical processors, a larger number of virtual partitions, or virtual nodes, are created. These virtual nodes are more than the actual physical processors.
* Virtual processors are then assigned to these virtual partitions:-
  + *Round-robin fashion:* Each virtual partition is sequentially assigned virtual processors, distributing the workload evenly.
  + *Based on estimated processing cost:* The estimated processing cost of each virtual partition is considered, and virtual processors are allocated accordingly to balance the workload.
* By distributing the workload evenly across virtual nodes, the system aims to handle skew effectively and prevent any single partition from becoming overloaded.

*Limitations:*

*-Changes in Data Distribution:*

* Effective in distributing workload and handling skew, may face challenges if the distribution of data changes over time.
* Recomputing the partitioning scheme, and adjusting the virtual nodes based on the evolving data distribution, can be an expensive operation.

-*Dynamic Repartitioning:*

* Dynamic repartitioning involves efficiently splitting a virtual node into two when it has too many tuples or experiences a high load, similar to the concept of splitting a B+-tree node when it becomes overfull.
* This adaptability allows the system to respond dynamically to shifts in data distribution, optimizing the workload distribution among virtual nodes.

**Sort Operation?:-**

*1. Range-Partitioning Sort:-*

* The relation is initially redistributed across multiple processors (sets P0 through Pm where m ≤ n - 1) using range partitioning based on specified sorting attributes. Each processor Pi receives tuples falling within a specific range and sorts them locally.
* Tuples falling within the ith range are sent to processor Pi.
* Each processor Pi stores the received tuples temporarily on its disk Di, which involves input/output (I/O) operations and communication overhead.
* Each processor Pi sorts its partition of the relation locally.
* Each processor independently executes the sort operation in parallel with others, demonstrating data parallelism.
* Range-partitioning guarantees that key values in Pi are less than those in Pj for 1 ≤ j ≤ m.
* Consequently, the final merge operation becomes straightforward as the data is already partially sorted within each partition.

*2. Parallel External Sort-Merge:*

* In this scenario, where the relation is already partitioned among disks D0 through Dn-1, and each processor Pi has sorted the data locally on disk Di, the next step involves merging the sorted runs in parallel.
* This merging process is performed in parallel across all processors, allowing for efficient utilization of resources.
* Sorted partitions at each processor Pi are range-partitioned across processors P0, ..., Pm-1.
* Pi merges incoming streams to generate a single sorted run. This concatenation operation combines the sorted runs from each processor to form a complete and globally sorted dataset.

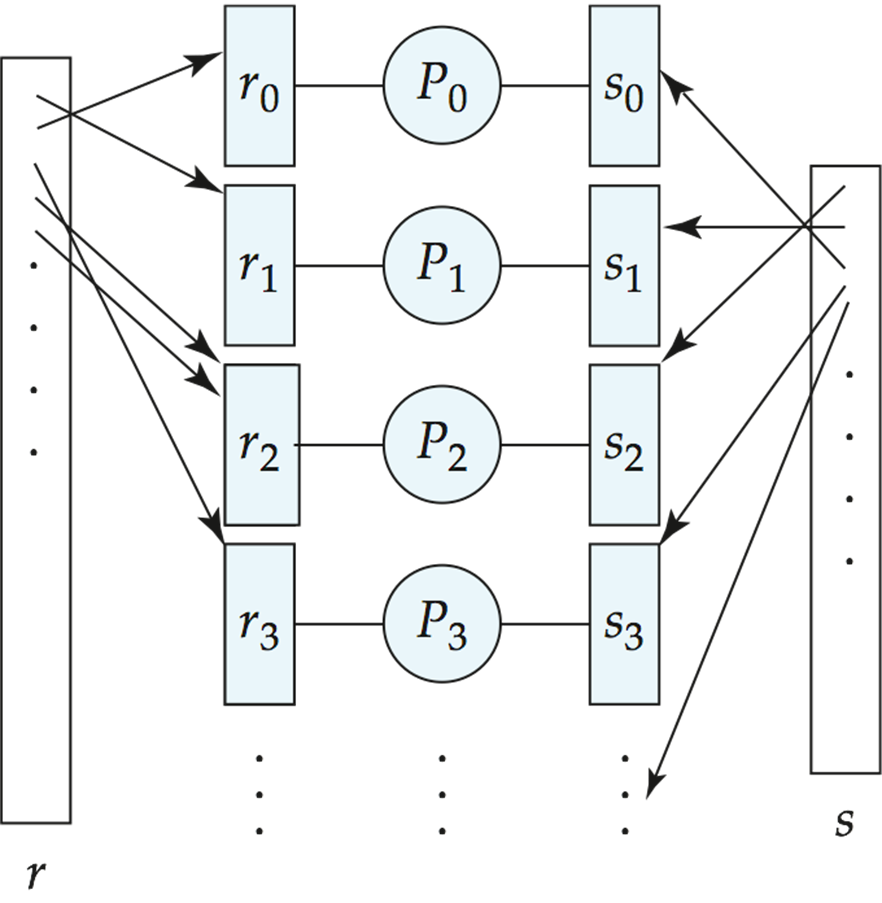
**Join Operation?:-**

***1. Parallel Join:***

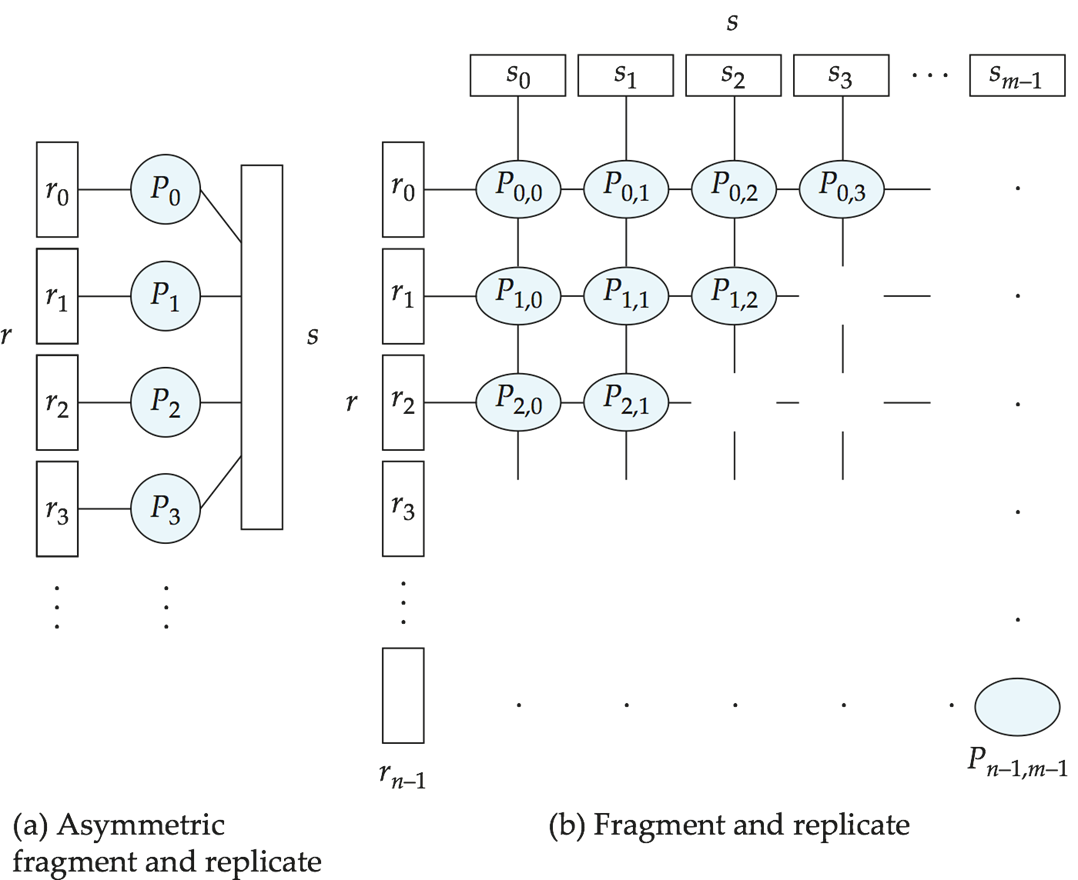
* The join operation involves testing pairs of tuples against a specified condition. Pairs that satisfy the join condition are added to the join output.
* Parallel join algorithms aim to distribute the workload of testing tuple pairs across multiple processors. Each processor is assigned a portion of the tuple pairs, and computations are performed locally on each processor.
* On each processor, the local computation involves testing the assigned tuple pairs for the join condition. The goal is to independently compute parts of the join operation on each processor, using parallel processing capabilities.
* After local computations are complete, the locally computed join results from each processor are combined to produce the overall output of the parallel join.

***2. Partitioned Join:***

*Equi-Joins and Natural Joins:*

* When we're dealing with join operations between two relations, like r and s, in a parallel computing environment, we can divide each of these into partitions.
* Assume r & s are partitioned into n partitions: r0, r1, ..., rn-1 & s0, s1, ..., sn-1.
* Range or hash partitioning can be used.
* We need to ensure that the data in these partitions is organized in a way that makes sense for the join operation, by partitioning the data based on the attributes we want to join on, such as r.A and s.B.
* Partitions ri and si are sent to processor Pi.
* Each processor Pi computes ri.A ⋈si.B locally using a standard join method.

*Fragment and Replicate:*

* Some join conditions, like non-equijoin conditions (e.g., r.A > s.B), may not be suitable for partitioning, so we use this method.
* Involves breaking down one relation (let's say r) into partitions using any partitioning technique. The other relation (let's say s) is replicated across all processors.
* Each processor computes the join of its partition of r with all tuples of s.
* Both relations (r and s) are partitioned into multiple partitions using any partitioning technique. Each processor computes the join of its partition of r with its corresponding partition of s.
* Processors are labeled as P0,0, P0,1, ..., P0,m-1, P1,0, ..., Pn-1,m-1.
* Each processor Pi,j computes the join of its partition r with its corresponding partition s. Any join technique can be used at each processor to compute the join.
* Fragment and replicate usually incur higher costs compared to partitioning.
* Asymmetric fragment-and-replicate may be preferable in cases where one relation is small and already partitioned, while the other is large.

***3. Partitioned Parallel Hash Join:***

* Assuming relation s is smaller than r, making s the build relation.
* Each tuple in s is mapped to one of the n processors using hash function h1 based on the join attribute value.
* The processor responsible for a particular tuple of s reads it from its disk and sends it to the correct processor based on h1. At the receiving processors, the tuples of s are further divided using another hash function h2 for local processing.
* Tuples of r are redistributed across m processors using h1 & repartitioned using h2.
* Each processor executes the build and probe phases of the hash-join algorithm. This means comparing tuples of r with corresponding tuples of s to find matches. The result is a partition of the final output of the hash-join operation.
* Hash-join optimization techniques can enhance performance and efficiency in the parallel hash-join process.

***3. Parallel Nested Loop Join:***

Relation s is significantly smaller than relation r. Relation r is partitioned, and within each partition of the relation r, there exists an index created on a specific attribute that is used for joining r with another relation, such as s.

*Fragment-and-Replicate:* Use asymmetric fragment-and-replicate (s smaller, and replicated and r remains partitioned or divided across the processors, with each processor handling a portion of it). This ensures that every processor has its own copy of s, making it easier to perform joins with the partitioned r without moving s around too much.

*Nested-Loop Join:* Each processor, containing a partition of relation r, performs a nested-loop join with its corresponding partition of relation s. With an index on the join attribute of r, the join operation becomes faster as it can quickly locate relevant tuples.

**Grouping/Aggregation:**

* Divide the dataset into smaller subsets based on the grouping attributes specified in the query. Each processor is assigned a portion of the data that shares common values for the grouping attributes. Partitioning ensures that related data is grouped together, which is essential for aggregation operations.
* Each processor independently calculates aggregate (sum, count, average, max, min, etc.) values for the data subset it holds. By performing computations locally, processors avoid the need to exchange large volumes of data over the network, which can significantly improve performance.
* By pre-computing aggregates, processors may already possess partial results, which minimizes the need for extensive data exchange during the partitioning process. This optimization helps improve its efficiency.

**Cost of Parallel Evaluation:**

1. Without Skew and Overhead:

Expected speed-up: 1/n.

1. Considering Skew and Overheads:

* Tpart (partitioning time)
* Tasm (result assembling time)
* Ti (operation time at processor Pi)

**Pipelined Parallelism:**

* Join operation involving four relations: r1, r2, r3, r4.nEstablish a pipeline to compute the three joins simultaneously.

- P1 computes temp1 = r1 ⋈ r2.

- P2 computes temp2 = temp1 ⋈ r3.

- P3 computes final result temp2 ⋈ r4.

* Each operation executes in parallel, passing result tuples to the next operation while computing further results.

*Factors Limiting Utility:*

* *Scalability:* Useful with a small number of processors but doesn't scale well with more processors. Pipeline chains don't achieve sufficient length.
* *Incompatibility:* Operators that don't produce output until all inputs are accessed (e.g., aggregate and sort) cannot be pipelined.
* *Skew:* Little speedup in cases where one operator's execution cost is much higher than others due to skew.

**Independent Parallelism:**

Divide the join of four relations into independent tasks.

- P1 computes temp1 = r1 ⋈ r2.

- P2 computes temp2 = r3 ⋈ r4.

P1 and P2 work independently in parallel. P3 waits for input from both P1 and P2 before processing. P3 computes the final result by joining temp1 and temp2. Output of P1 and P2 can be pipelined to P3 to combine independent and pipelined parallelism.

Utility and Limitations: Useful with a lower degree of parallelism. Less effective in highly parallel systems.

**Query Optimization in Parallel Databases:**

* *Complexity:* Optimizing queries in parallel databases is more complex due to the need to account for parallel execution and distribution of data.
* *Cost Models:* Cost models in parallel databases consider factors like partitioning costs, data skew, and resource contention (competition for system resources) to estimate the most efficient query execution plan.
* *Heuristics:* Different heuristics are used to generate parallel plans, including strategies like pipelining and inter-operation pipelining. These heuristics help in choosing the most efficient plan for parallel execution.
* *Exchange Operator:* The introduction of an exchange operator helps in partitioning and distributing tuples across processors, allowing each operation to work independently on local data.
* *Pipelined Parallelism:* Pipelined parallelism is explored as an option to optimize query execution by streamlining the flow of data between operations.
* *Physical Organization:* Choosing an appropriate physical organization, such as the partitioning technique, is crucial for speeding up query execution in parallel databases.

**Design of Parallel Systems:**

* Parallel loading of data from external sources is necessary for handling large volumes of incoming data.
* Resilience required for processor or disk failures. Redundancy is achieved by storing extra copies of data items at other processors.
* Support for on-line reorganization of data and schema changes.
* On-line repartitioning and schema changes supported concurrently with other processing.