Parallel and Distributed Query Processing

- **Parallel Systems and Distributed Systems**
- I/O Parallelism
- Parallel Query Processing
- Distributed Query Processing

Parallel Systems

- Parallel machines have become common and affordable
  - Prices of microprocessors, memory and disks have dropped sharply
  - Recent desktop computers feature multiple processors and this trend is projected to accelerate

- Data storage needs are growing increasingly large
  - user data at web-scale
    - 100’s of millions of users, petabytes of data
  - large volumes of data are collected and stored for analysis.
  - multimedia objects like images/videos

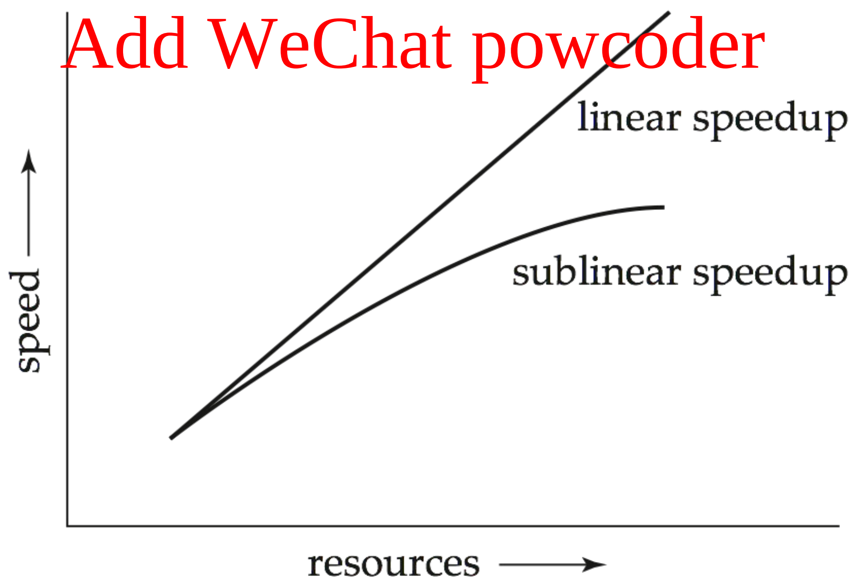
- Large-scale parallel database systems increasingly used for:
  - storing large volumes of data
  - processing time-consuming decision-support queries
  - providing high throughput for query processing

Parallel Systems (cont.)

- Parallel database systems consist of multiple processors and multiple disks connected by a fast interconnection network.
  - A **coarse-grain parallel** machine consists of a small number of powerful processors
  - A **massively parallel** or **fine-grain parallel** machine utilizes thousands of smaller processors.
    - Typically hosted in a **data center**
- Two main performance measures:
  - **throughput**: number of tasks that can be completed in a given time interval
    - A system can improve throughput by processing many tasks in parallel
  - **response time**: amount of time it takes to complete a single task from the time it is submitted
    - A system can improve response time by performing subtasks of each task in parallel

Speed-Up

- **Speedup**
  - Parallelism is used to provide ***speedup***, where tasks are executed ***faster*** because more resources are provided.
  - A fixed-sized problem executing on a small system is given to a system which is *N*-times larger.
    - $speedup = \dfrac{small\ system\ elapsed\ time}{large\ system\ elapsed\ time}$
  - Speedup is **linear** if equation equals *N*.

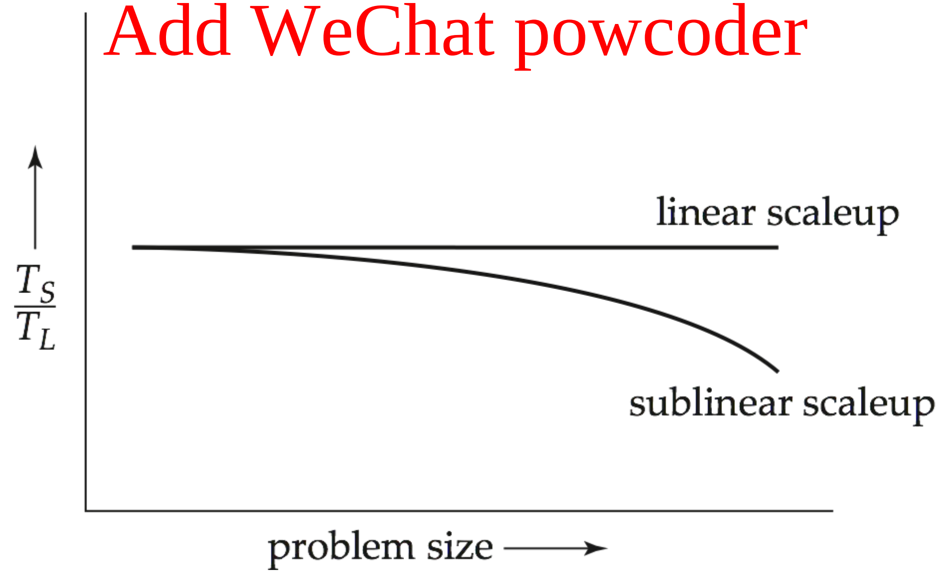


Scale-Up

- **Scaleup**:
  - Parallelism is used to provide ***scaleup***, where ***larger*** tasks are processed in the ***same*** amount of time by providing more resources.
  - Increase the size of both the problem and the system: *N*-times larger system used to perform *N*-times larger job.

$$\textit{scaleup} = \frac{\textit{small system small problem elapsed time}}{\textit{big system big problem elapsed time}}$$

  - Scale up is **linear** if equation equals 1.



Factors Limiting Speedup and Scaleup

Speedup and scaleup are often sublinear due to:

- **Startup/sequential costs**
  - Cost of starting up multiple processes and sequential computation before/after parallel computation may dominate computation time, if the degree of parallelism is high.

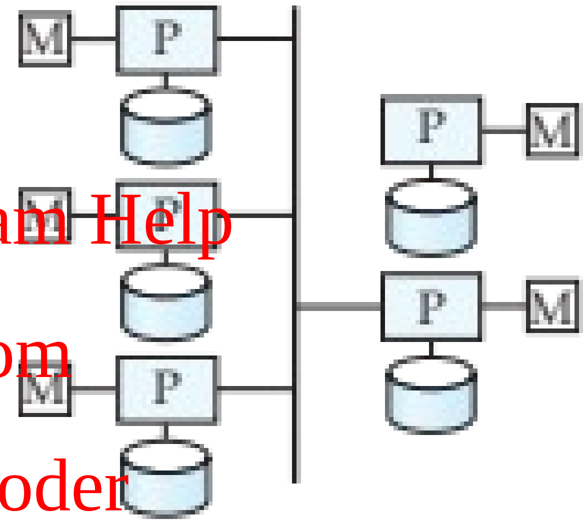
- **Interference**

  - Processes accessing shared resources (e.g., system bus, disks, or locks) compete with each other, thus spending time waiting on other processes, rather than performing useful work.

- **Skew**
  - It is often difficult to divide a task into exactly equal-sized parts, the way that the sizes are distributed is therefore ***skewed***.
  - The service time for the single **slowest** step will determine the service time for the task as a whole.
    - Example: A task of size 100 is divided into 10 parts and one task happens to be of size 20, the speedup is only 5, not 10.

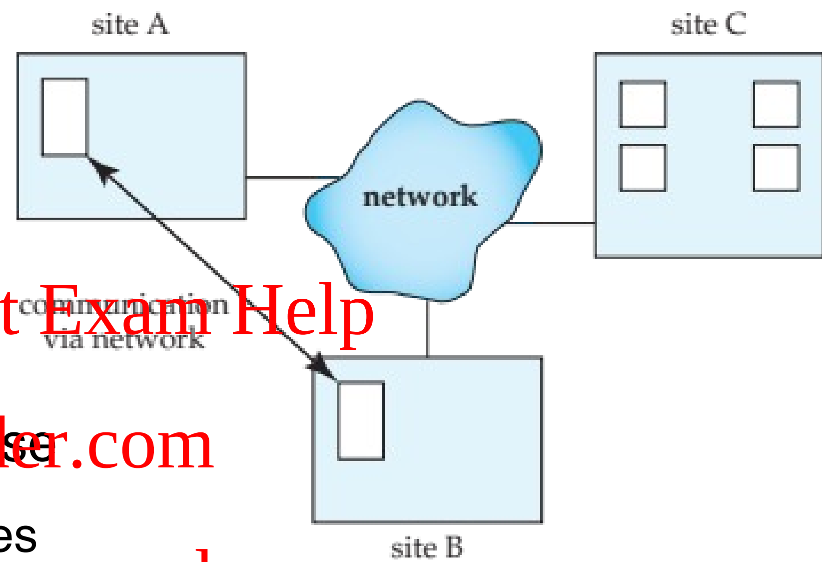
Parallel Database Architecture: Shared Nothing

- Node consists of a processor, memory, and one or more disks
- All communication via interconnection network
- Can be scaled up to thousands of processors without interference.
- Main drawback: cost of communication and non-local disk access; sending data involves software interaction at both ends.



Distributed Systems

- A distributed system consists of loosely coupled sites (or nodes) that share no physical component



- Data spread over multiple sites.
- Homogeneous distributed database
  - Same software/schema on all sites
  - Goal: provide a view of a single database, hiding details of distribution
- Heterogeneous distributed database
  - Different software/schema on different sites
  - Goal: integrate existing databases to provide useful functionality

Parallel and Distributed Query Processing

- Parallel Systems and Distributed Systems
- **I/O Parallelism**
- Parallel Query Processing
- Distributed Query Processing

I/O Parallelism

- Reduce the time required to retrieve relations from disk by partitioning the relations on multiple disks, on *multiple* ***nodes***.
- ***Horizontal partitioning*** – tuples of a relation are divided among many nodes such that some subset of tuples resides on each node.

- Partitioning techniques (number of nodes = n):

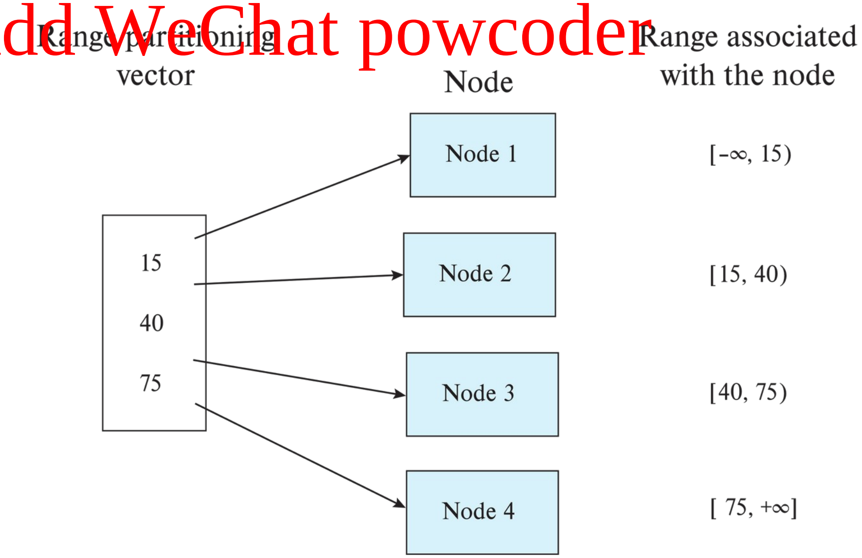
  Round-robin:
  - Send the i^{th} tuple inserted in the relation to node i mod n.

  Hash partitioning:
  - Choose one or more attributes as the partitioning attributes.
  - Choose hash function h with range $0 \ldots n - 1$
  - Let i denote result of hash function h applied to the partitioning attribute value of a tuple. Send tuple to node i.

I/O Parallelism (Cont.)

Range partitioning:

- Choose an attribute as the partitioning attribute.
- Choose a partitioning vector [v_1, v_2, ..., v_{n-1}].
- Let x be the partitioning attribute value of a tuple.
  - Tuples with $x < v_1$ go to N_1
  - Tuples with $x \geq v_{n-1}$ go to N_n
  - Tuples such that $v_i \leq x < v_{i+1}$ go to N_{i+1}
- Example:



Types of Skew

- **Data-distribution skew:** some nodes have many tuples, while others may have fewer tuples.
  - **Attribute-value skew**
    - Some partitioning-attribute values appear in many tuples; all the tuples with the same value for the partitioning attribute end up in the same partition.
    - Can occur with range-partitioning and hash-partitioning.
  - **Partition skew**
    - Imbalance, even without attribute-value skew
    - Badly chosen range-partitioning vector may assign too many tuples to some partitions and too few to others.
    - Less likely with hash-partitioning if a good hash-function is chosen.

Handling of Skew (Cont.)

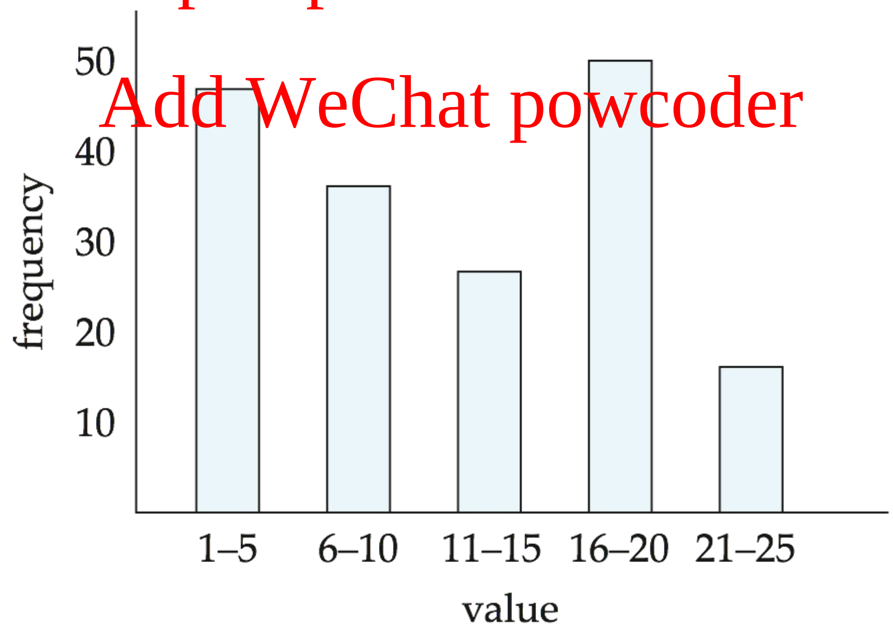
- A small skew can result in a significant decrease in performance.
  - Skew becomes an increasing problem with a higher degree of parallelism.
  - Example: Consider a relation of 1000 tuples.
    - i) If it is divided into 10 equal parts,
      - $\text{speedup} = \dfrac{\text{time taken to scan the relation in a single disk system}}{\text{time taken to scan the relation in a multiple - disk system}} = 10$
    - ii) If it is divided into 10 unequal parts and even one partition has 200 tuples,
      - speedup = 5
    - iii) If it is divided to 100 equal parts,
      - speedup = 100
    - iv) If it is divided into 100 unequal parts and even one partition has 40 tuples,
      - speedup = 25
  - The loss of speedup due to skew increases with parallelism.

Handling Skew in Range-Partitioning

- Data-distribution skew can be avoided with range-partitioning by creating **balanced range-partitioning vectors**
  - Sort the relation on the partitioning attribute.
  - Construct the partitioning vector by scanning the relation in sorted order as follows.
    - After every $1/n$ of the relation has been read, the value of the partitioning attribute of the next tuple is added to the partitioning vector.
    - n denotes the number of partitions to be constructed.
  - Imbalances can result if duplicates are present in partitioning attributes.
  - Extra I/O overhead in doing the initial sort.

Handling Skew using Histograms

- Balanced partitioning vector can be constructed from histogram, which can be stored in the system catalog.
  - In a histogram, the values for the attribute are divided into a number of ranges, and with each range the histogram associates the number of tuples whose attribute value lies in that range.
  - Reduced I/O overhead.



Handling Skew Using Virtual Node Partitioning

- Key idea: pretend there are several times (10x to 20x) as many **virtual nodes** as real nodes
  - Virtual nodes are mapped to real nodes
  - Tuples partitioned across virtual nodes using range-partitioning vector
    - Hash partitioning is also possible.

- Mapping of virtual nodes to real nodes
  - **Round-robin**: virtual node *i* mapped to real node (*i*-1 mod *n*)+1

  - **Mapping table**: mapping table *virtual_to_real_map*[] tracks which virtual node is on which real node

    - Allows skew to be handled by moving virtual nodes from more loaded nodes to less loaded nodes
    - Both data distribution skew and execution skew can be handled
- Basic idea:
  - If any normal partition would have been skewed, it is very likely the skew is spread over a number of virtual partitions.
  - Skewed virtual partitions get spread across a number of nodes, so work gets distributed evenly!

Parallel and Distributed Query Processing

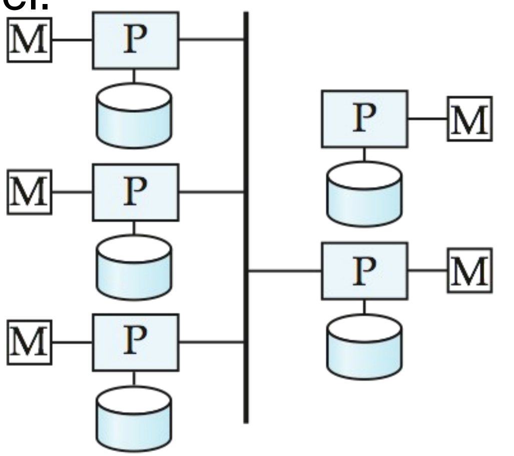
- Parallel Systems and Distributed Systems
- I/O Parallelism
- **Parallel Query Processing**
- Distributed Query Processing

Parallel Query Processing

- **Interquery parallelism**
  - Different queries can be run in parallel with each other.
  - Increases throughput; used primarily to ***scale up*** a database system to support a larger number of queries per second.
  - The response times of individual queries are no faster.

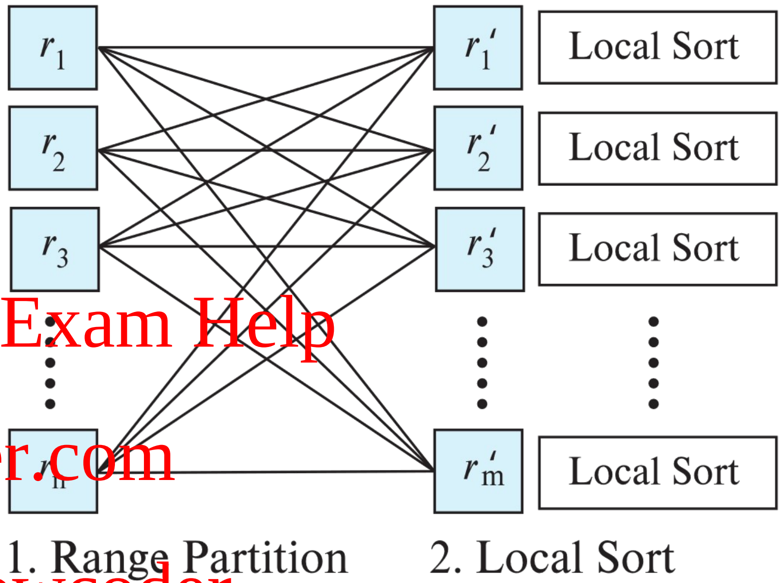
Intraquery Parallelism

- Execution of a single query in parallel on multiple nodes; important for ***speeding up*** long-running queries.
- Two complementary forms of intraquery parallelism:
  - **Intraoperation Parallelism**
    - Parallelize execution of each individual operation in a query, e.g., sort, select, project, join.
    - Data can be partitioned and each node can work independently on its own partition.
    - High degree of parallelism
  - **Interoperation Parallelism**
    - Execute different operations in a query in parallel.
    - Limited degree of parallelism
- Our discussion of parallel algorithms assumes:
  - *read-only* queries
  - shared-nothing architecture: n nodes, N_1, ..., N_n, each assumed to have disks and processors



Range-Partitioning Sort

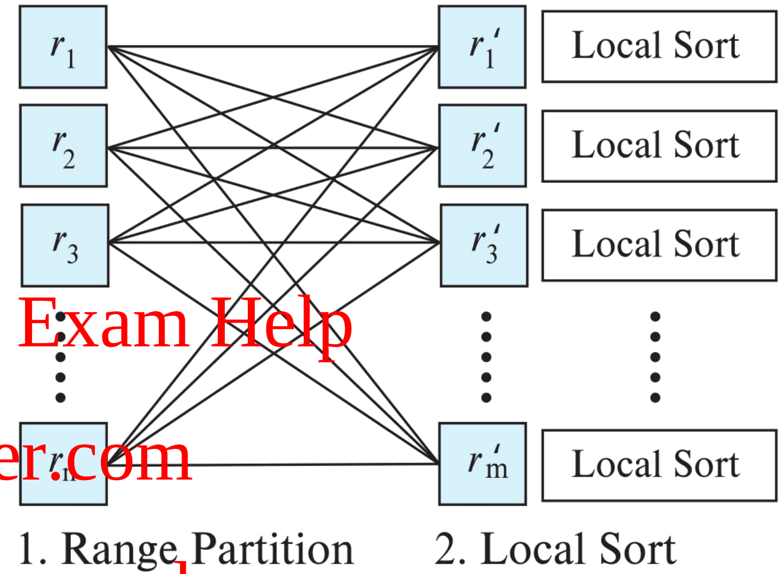
- Suppose a relation is partitioned among nodes $N_1, ..., N_n$.
- Choose nodes $N_1, ..., N_m$ to do sorting.
- Create range-partitioning vector with m-1 entries on the ***sorting attributes***
- Redistribute the relation using range partitioning
  - All tuples that lie in the i^{th} range are sent to node N_i
  - N_i stores the tuples it received temporarily on its local disk.
  - This step requires I/O and communication overhead.



(a) Range Partitioning Sort

Range-Partitioning Sort (cont.)

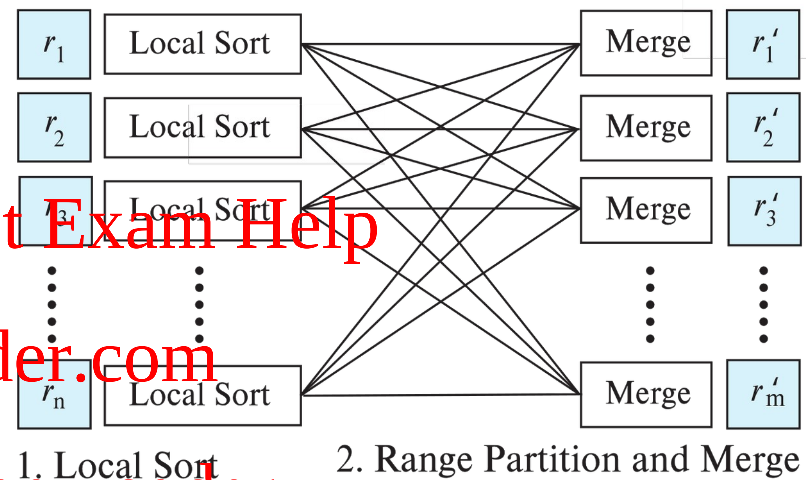
- Each node N_i sorts its partition of the relation locally.
  - **Data parallelism**: each node executes same operation (sort) in parallel with other nodes, without any interaction with the others.
- Concatenate the results to get the fully sorted relation.
  - range-partitioning ensures that, if $i < j$, all key values in node N_i are all less than all key values in N_j.



(a) Range Partitioning Sort

Parallel External Sort-Merge

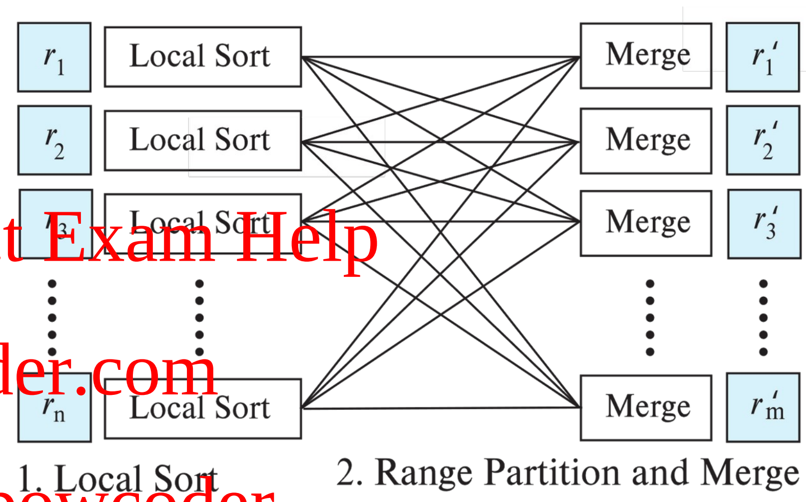
- Suppose a relation is partitioned among nodes N_1, ..., N_n.
- Each node N_i locally sorts the data.
- The sorted runs on each node are then merged in parallel to get the final sorted output.



(b) Parallel External Sort-Merge

Parallel External Sort-Merge (Cont.)

- Parallelize the merging of sorted runs as follows:
  - The sorted partitions at each node N_i are ***range-partitioned*** across the nodes N_1, ..., N_m (all by the ***same*** partitioning vector) and the tuples are sent in sorted order, so each node receives the tuples as sorted streams.
  - Each node N_i performs a merge on the sorted streams as they are received, to get a single sorted run.
  - The sorted runs on nodes N_1, ..., N_m are concatenated to get the final result.



(b) Parallel External Sort-Merge

Parallel Join

- *Reminder*: The join operation requires pairs of tuples to be tested to see if they satisfy the join condition, and if they do, the pair is added to the join output.
- Basic idea:
  - Divide the tuples of the input relations over several nodes.
  - Each node then computes part of the join locally.
  - The results from each node can be collected together to produce the final result.

Partitioned Parallel Join

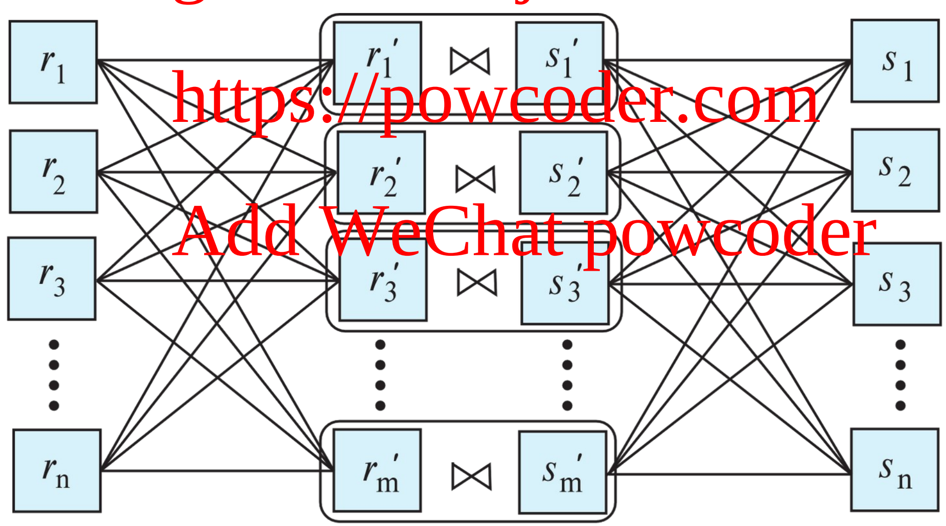
- For equi-joins and natural joins, it is possible to *partition* the two input relations across the nodes, and compute the join locally at each node.
- Let *r* and *s* be the input relations, and we want to compute $r \bowtie_{r.A=s.B} s$.

- *r* and *s* each is partitioned into *m* partitions, denoted $r_1', r_2', \ldots, r_m'$ and $s_1', s_2', \ldots, s_m'$.

- Can use either *range partitioning* or *hash partitioning*.
- *r* and *s* must be partitioned on their ***join attributes*** *r*.A and *s*.B, using the ***same*** range-partitioning vector or hash function.
- Partitions r_i' and s_i' are sent to node N_i.

Partitioned Parallel Join (cont.)

- Join can be computed at each node using any of
  - Hash join, leading to **partitioned parallel hash join**
  - Merge join, leading to **partitioned parallel merge join**
  - Nested loops join, leading to **partitioned parallel nested-loops join** or **partitioned parallel index nested-loops join**



Step 1: Partition r Step 2: Partition s

Step 3: Each node N_i computes $r_i' \bowtie s_i'$

Partitioned Parallel Hash-Join

Parallelizing partitioned hash join:

- Assume relations *r* and *s* are partitioned and *s* is smaller than *r* and therefore *s* is chosen as the ***build relation***.
- A hash function h_1 takes the ***join attribute*** value of each tuple in *s* and maps this tuple to one of the *n* nodes.
  - Each node N_i reads the tuples of *s* that are on its local disk, and sends each tuple to the appropriate node based on hash function h_1.
  - Let s_i denote the tuples of relation *s* that are sent to node N_i.
- As tuples of relation *s* are received at the destination nodes, they are partitioned further using another hash function, h_2, which is used to compute the hash-join locally.

Partitioned Parallel Hash-Join (Cont.)

- Next, the larger relation r is redistributed across the n nodes using the hash function h_1
  - Let r_i denote the tuples of relation r that are sent to node N_i.

- As the r tuples are received at the destination nodes, they are repartitioned using the function h_2

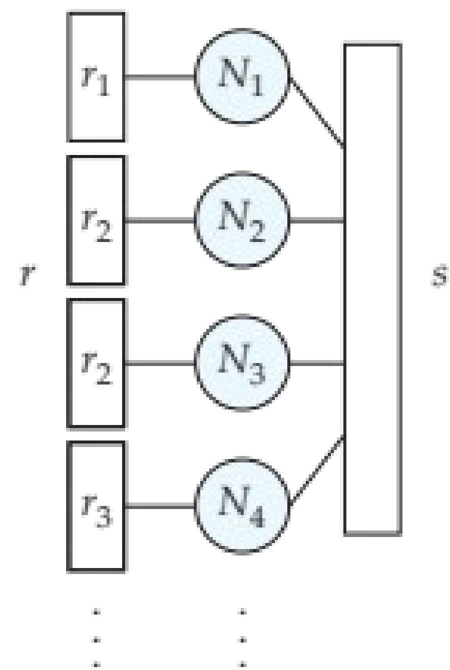
- Each node N_i executes the ***build and probe phase*** of the hash-join algorithm on the local partitions r_i and s_i of r and s to produce a partition of the final result of the hash-join.

Fragment-and-Replicate Join

- Partitioning is not possible for some join conditions
  - E.g., non-equijoin conditions, such as r.A > s.B.
- In these cases, parallelization can be accomplished by **fragment and replicate** technique

- Special case – **asymmetric fragment-and-replicate**:

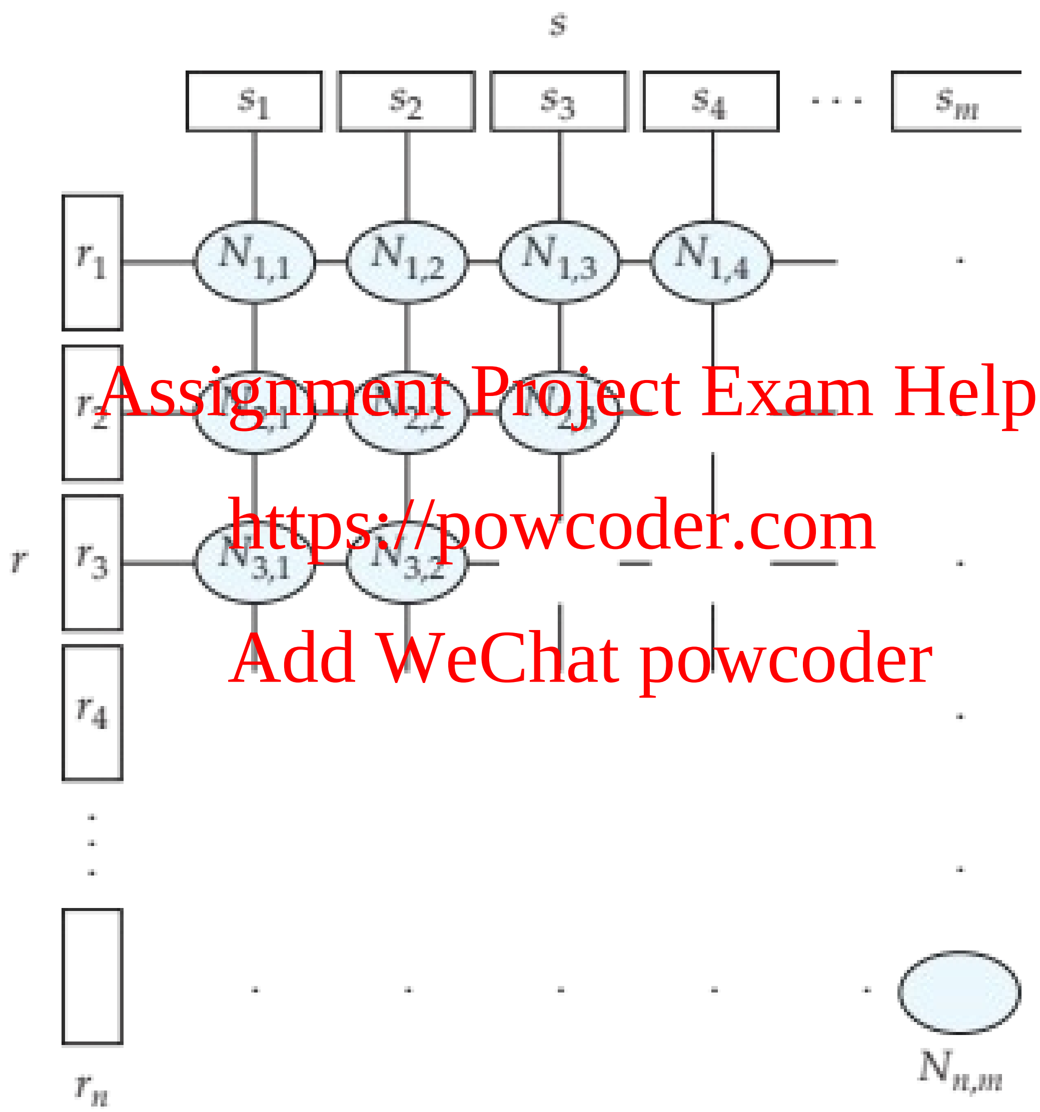
  - One of the relations, say *r*, is partitioned using any partitioning technique.
  - The other relation, *s*, is replicated across all the nodes.
  - Node N_i then locally computes the join of r_i with all of *s* using any join technique.
  - Also referred to as **broadcast join**



Fragment-and-Replicate Join (Cont.)

- General case
  - Reduces the sizes of the relations at each node.
  - There must be at least $m * n$ nodes.
  - Label the nodes as $N_{1,1}$, $N_{1,2}$, ..., $N_{1,m}$, $N_{2,1}$, ..., $N_{n,m}$.
  - r is partitioned into n partitions, r_1, r_2, ..., r_n
  - s is partitioned into m partitions, s_1, s_2, ..., s_m
  - Any partitioning technique may be used.
  - r_i is replicated to $N_{i,1}$, $N_{i,2}$, ..., $N_{i,m}$ (a row)
  - s_i is replicated to $N_{1,i}$, $N_{2,i}$, ..., $N_{n,i}$ (a column)
  - $N_{i,j}$ computes the join of r_i with s_j.
  - Any join technique can be used at each node $N_{i,j}$.

Fragment-and-Replicate Join (cont.)



Fragment-and-Replicate Join (Cont.)

- Both versions of fragment-and-replicate work with any join condition, since ***every*** tuple in *r* can be tested with ***every*** tuple in *s*.
- Usually has a higher cost than partitioning, since one of the relations (for asymmetric fragment-and-replicate) or both relations (for general fragment-and-replicate) have to be replicated.
- Sometimes asymmetric fragment-and-replicate is preferable even though partitioning could be used.
  - E.g., if *s* is small and *r* is large, and already partitioned. It may be cheaper to replicate *s* across all nodes, rather than repartition *r* and *s* on the join attributes.

Selection

- Selection $\sigma_\theta(r)$
  - If θ is of the form $a_i = v$, where a_i is an attribute and v is a value.

    - If r is partitioned on a_i, the selection is performed at a single node.

  - If θ is of the form $l \leq a_i \leq u$ (i.e., θ is a range selection) and the relation has been range-partitioned on a_i
    - Selection is performed at each node whose partition overlaps with the specified range of values.
  - In all other cases: the selection is performed in parallel at all the nodes.

Duplicate Elimination and Projection

- Duplicate elimination
  - Perform by using either of the parallel sort techniques
    - eliminate duplicates as soon as they are found during sorting.
  - Or, partition the tuples (using either range- or hash- partitioning) and perform duplicate elimination locally at each node.

- Projection
  - Projection without duplicate elimination can be performed as tuples are read in from disk in parallel.
  - If duplicate elimination is required, any of the above duplicate elimination techniques can be used.

Grouping/Aggregation

- A straight-forward way:
  - partition the relation on the ***grouping attributes***
  - compute the aggregate values locally at each node.
- **Optimization:** Can reduce cost of transferring tuples during partitioning by **partial aggregation** before partitioning
- Consider the **sum** aggregation operation:
  - Perform aggregation operation at each node N_i on those tuples stored on its local disk
    - results in tuples with partial sums at each node
    - there is one tuple at N_i for each value of the grouping attribute
  - Result of the local aggregation is partitioned on the ***grouping attribute***, and the aggregation performed again on tuples with the partial sums at each node N_i to get the final result.
- Fewer tuples need to be sent to other nodes during partitioning.

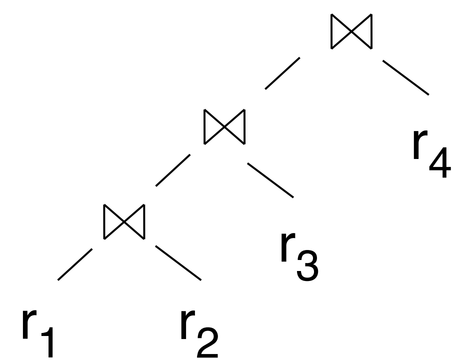
Interoperator Parallelism

- *Reminder*: In pipelining, the output tuples of one operation, *A*, are consumed by a second operation, *B*, even before *A* has produced the entire set of tuples in its output.
- **Pipelined parallelism**
  - Run *A* and *B* simultaneously on different nodes, so that *B* consumes tuples in parallel with *A* producing them.

  - Consider a join of four relations

    - $r_1 \bowtie r_2 \bowtie r_3 \bowtie r_4$

  - Set up a pipeline that computes the three joins in parallel
  - Each of these operations can execute in parallel, sending result tuples it computes to the next operation even as it is computing further results, provided a pipelineable join evaluation algorithm (e.g., indexed nested loops join) is used



Factors Limiting Utility of Pipelined Parallelism

- Pipelined parallelism is useful since it avoids writing intermediate results to disk
- Useful with small number of nodes, but does not scale up well with more nodes.
  - Does not provide a high degree of parallelism since pipeline chains are not very long
  - Cannot pipeline operators which do not produce output until all inputs have been accessed (e.g., aggregate and sort)
  - Little speedup is obtained for the frequent cases of skew in which one operator's execution cost is much higher than the others.

Independent Parallelism

- **Independent parallelism**
  - Operations in a query expression that do not depend on one another can be executed in parallel.
  - Consider a join of four relations

    $r_1 \bowtie r_2 \bowtie r_3 \bowtie r_4$

    - Let N_1 be assigned the computation of $temp_1 = r_1 \bowtie r_2$

    - And N_2 be assigned the computation of $temp_2 = r_3 \bowtie r_4$
    - And N_3 be assigned the computation of $temp_1 \bowtie temp_2$

    - N_1 and N_2 can work **independently in parallel**
    - N_3 has to wait for input from N_1 and N_2
      - Can pipeline output of N_1 and N_2 to N_3, combining independent parallelism and pipelined parallelism
  - Does not provide a high degree of parallelism
    - useful with a lower degree of parallelism.
    - less useful in a highly parallel system.

Query Optimization for Parallel Execution

- *Reminder*: A query optimizer takes a query and finds the cheapest execution plan.
- Query optimization in parallel databases is significantly more complex than query optimization in sequential databases.
  - Different options for partitioning inputs and intermediate results
  - Cost models are more complicated, since we must take into account partitioning costs and issues such as skew and resource contention.

Parallel Query Plan Space

A parallel query plan must specify

- How to parallelize each operation, including which algorithm to use, and how to partition inputs and intermediate results
- How the plan is to be **scheduled**
  - How many nodes to use for each operation
  - What operations to pipeline within same node or different nodes
  - What operations to execute independently in parallel, and
  - What operations to execute sequentially, one after the other.
- E.g., In query ${}_{r.A}\gamma_{sum(s.C)}(r \bowtie_{r.A=s.A \wedge r.B=s.B} s)$
  - Partitioning *r* and *s* on (*A,B*) for join will require repartitioning for aggregation
  - But partitioning *r* and *s* on (*A*) for join will allow aggregation with no further repartitioning
- Query optimizer has to choose best plan taking above issues into account

Choosing Query Plans

- The number of parallel evaluation plans from which to choose from is much larger than the number of sequential evaluation plans.
- Two alternatives often used for choosing parallel plans:
  - First choose most efficient sequential plan and then choose how best to parallelize the operations in that plan
    - Heuristic, since best sequential plan may not lead to best parallel plan
  - Parallelize every operation across all nodes

Parallel and Distributed Query Processing

- Parallel Systems and Distributed Systems
- I/O Parallelism
- Parallel Query Processing
- **Distributed Query Processing**

Distributed Query Processing

- Many database applications require data from multiple databases.
- For centralized systems, the primary criterion for measuring the cost of a particular strategy is the number of disk accesses.

- In a distributed system, other issues must be taken into account:

  - The cost of a data transmission over the network.

  - The potential gain in performance from having several sites process parts of the query in parallel.

Join Locations and Join Ordering

- Consider the following relational algebra expression

  $r_1 \bowtie r_2 \bowtie r_3$

- r_1 is stored at site S_1

- r_2 at S_2

- r_3 at S_3

- For a query issued at site S_l, the system needs to produce the result at site S_l

Possible Query Processing Strategies

- Strategy 1
  - Ship copies of all three relations to site S_I and choose a strategy for processing the entire locally at site S_I.
- Strategy 2
  - Ship a copy of the r_1 relation to site S_2 and compute $temp_1 = r_1 \bowtie r_2$ *at* S_2.
  - Ship $temp_1$ from S_2 to S_3, and compute $temp_2 = temp_1 \bowtie r_3$ at S_3
  - Ship the result $temp_2$ to S_I.
- Devise similar strategies, exchanging the roles S_1, S_2, S_3
- Must consider following factors:
  - amount of data being shipped
  - cost of transmitting a data block between sites
  - relative processing speed at each site

Semijoin Strategy

- Let r_1 be a relation with schema R_1 stored at site S_1

  Let r_2 be a relation with schema R_2 stored at site S_2
- Evaluate the expression $r_1 \bowtie r_2$ and obtain the result at S_1.

- Strategy 1:
  - Ship r_2 to S_1. However, if there are many tuples of r_2 that do not join with any tuple of r_1, then this entails shipping useless tuples.

- Strategy 2:
  1. Compute $temp_1 \leftarrow \prod_{R1 \cap R2} (r_1)$ at S_1.
  2. Ship $temp_1$ from S_1 to S_2.
  3. Compute $temp_2 \leftarrow r_2 \bowtie temp_1$ at S_2.
  4. Ship $temp_2$ from S_2 to S_1.
  5. Compute $r_1 \bowtie temp_2$ at S_1. This is the same as $r_1 \bowtie r_2$.

Semijoin Strategy (Cont.)

- Strategy 2
  - $temp_2$ contains all of the tuples that qualify the join condition but other attributes of r_1 are missing
  - Cost savings: ship only $temp_2$, rather than all of r_2 to S_1

    versus
  - Overhead: ship $temp_1$ to S_2.
  - Particularly advantageous when relatively few tuples of r_2 contribute to the join
    - $temp_2$ may have significantly fewer tuples than r_2.
    - overhead will be dominated by savings

Semijoin Strategy (Cont.)

- The **semijoin** of r_1 with r_2, is denoted by:

$$r_1 \ltimes r_2$$

- it is defined by:

$$\prod_{R1} (r_1 \bowtie r_2)$$

- Thus, $r_1 \ltimes r_2$ selects those tuples of r_1 that contributed to $r_1 \bowtie r_2$.

- In step 3 above, $temp_2 = r_2 \ltimes r_1$.
- For joins of several relations, the above strategy can be extended to a series of semijoin steps.

Distributed Query Optimization

- Extensions to existing query optimization techniques
  - Record the location of data
  - Annotate operators with the site where they are executed

    - Operators typically operate only on local data

    - Remote data is typically fetched locally before operator is executed

  - Consider semijoin operations to reduce data transfer costs
    - Heuristic: restrict semijoins only on database tables, not on intermediate join results