Chapter 17: Parallel Databases

- Introduction
- I/O Parallelism
- Interquery Parallelism
- Intraquery Parallelism
- Intraoperation Parallelism
- Interoperation Parallelism
- Design of Parallel Systems

Introduction

- The placement of data on multiple disks and the parallel evaluation of relational operations have been instrumental in the success of parallel databases.

- With microprocessors having become very cheap, parallel machines are becoming quite common and are relatively inexpensive.

- Large-scale parallel database systems are used primarily for storing large volumes of data, and processing decision-support queries on the data.

I/O Parallelism

- Reduce the time required to retrieve relations from disk by partitioning the relations on multiple disks.
- Horizontal partitioning – tuples of a relation are divided among many disks such that each tuple resides on one disk.
- Partitioning techniques:
  - Round-robin: maps the ith tuple to disk $i \bmod n$.
  - Hash partitioning: tuple's disk location is based on applying a hash function to an attribute of the tuple.
  - Range partitioning: groups tuples sharing similar attributes in the same partition.

Comparison of Partitioning Techniques

- Evaluate how well partitioning techniques support the following types of data access:
  - Scanning the entire relation.
  - Locating a tuple associatively – *point* queries.
  - Locating all tuples such that the value of a given attribute lies within a specified range – *range* queries.

Comparison of Partitioning Techniques (Cont.)

- Round-robin.
  - Best suited for sequential scan of entire relation on each query
  - Cannot access tuples associatively
- Hash partitioning.
  - Provides sequential and associative access
  - Counterproductive to clustering related data
- Range partitioning.
  - Provides data clustering in addition to sequential and associate access
  - *Execution skew*: all execution occurs in one partition

Partitioning a Relation across Disks

- If a relation contains only a few tuples which will fit into a single disk block, then assign the relation to a single disk.
- Large relations are preferably partitioned across all the available disks.
- If a relation consists of m disk blocks and there are n disks available in the system, then the relation should be allocated $\mathbf{min}(m, n)$ disks.

Handling of Skew

- Attribute-value skew.
  - Some values appear in the partitioning attributes of many tuples; all the tuples with the same value for the partitioning attribute end up in the same partition.
  - Can occur regardless of whether range-partitioning or hash-partitioning is used.
- Partition skew.
  - May be load imbalance in the partitioning, even without attribute skew.
  - May occur with range-partitioning if partition vector is not carefully chosen.
  - Less likely with hash-partitioning if a good hash-function is chosen.

Handling Skew in Range-Partitioning

- Sort the relation on the join attribute.

- Construct the partition vector by scanning the relation in sorted order as follows.
  - After every $1/n^{th}$ of the relation has been read, the value of the partitioning attribute of the next tuple is added to the partition vector.
  - n denotes the number of partitions to be constructed.

Interquery Parallelism

- Queries/transactions execute in parallel with one another.
- Increases transaction throughput; used primarily to scale up a transaction processing system to support a larger number of transactions per second.
- Easiest form of parallelism to support in a database system, particularly in a shared-memory parallel system.
- Must guarantee *cache coherency* in a shared-disk or shared-nothing architecture.

Intraquery Parallelism

- Execution of a single query in parallel on multiple processors/disks.
- Important for speeding up long-running queries.
- Intraoperation Parallelism – parallelize the execution of each individual operation.
- Interoperation Parallelism – execute the different operations in a query expression in parallel.
- Example discussion of parallelizing algorithms assumes:
  - shared-nothing architecture
  - n processors, $P_0, \ldots, P_{n-1}$, and n disks $D_0, \ldots, D_{n-1}$, where disk D_i is associated with processor P_i.

Parallel Sort

Range-Partitioning Sort

- Assume processors $P_0, \ldots, P_m$, where $m < n$ to sort the relation.
- Redistribute the relation such that all tuples that lie within the i^{th} range are sent to processor P_i, which stores the relation temporarily on disk D_i.
- Each processor sorts its partition of the relation locally, without interaction with the other processors.
- Final merge operation is trivial: range-partitioning ensures that, for $1 \leq i < j \leq m$, the key values in processor P_i are all less than the key values in P_j.

Parallel Sort (Cont.)

Parallel External Sort-Merge

- Assume the relation has already been partitioned among disks $D_0, \ldots, D_{n-1}$.
- Each processor P_i locally sorts the data on disk D_i.
- The sorted runs on each processor are then merged to get the final sorted output.
- Parallelize the merging of sorted runs as follows:
  - The sorted partitions at each processor P_i are range-partitioned across the processors $P_0, \ldots, P_{m-1}$.
  - Each processor P_i performs a merge on the streams as they are received, to get a single sorted run.
  - The sorted runs on processors $P_0, \ldots, P_{m-1}$ are concatenated to get the final result.

Parallel Join

- 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.

- Parallel join algorithms attempt to split the pairs to be tested over several processors. Each processor then computes part of the join locally.

- In a final step, the results from each processor can be collected together to produce the final result.

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 compute the join locally at each processor.

- Assume n processors and relations r and s.
  - r and s each are partitioned into n partitions, denoted $r_0, r_1, \ldots, r_{n-1}$ and $s_0, s_1, \ldots, s_{n-1}$.
  - Partitions r_i and s_i are sent to processor P_i, where their join is computed locally.
  - r and s must be partitioned using the same partitioning function on their join attributes.

Fragment-and-Replicate Join

- For joins were partitioning is not applicable, parallelization can be accomplished by *fragment and replicate* technique.

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

  – One of the relations, say r, is partitioned; any partitioning technique can be used.

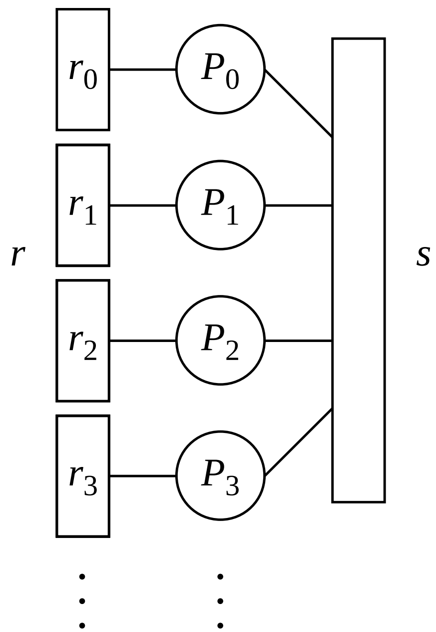
  – The other relation, s, is replicated across all the processors.

  – Processor P_i then locally computes the join of r_i with all of s using any join technique.

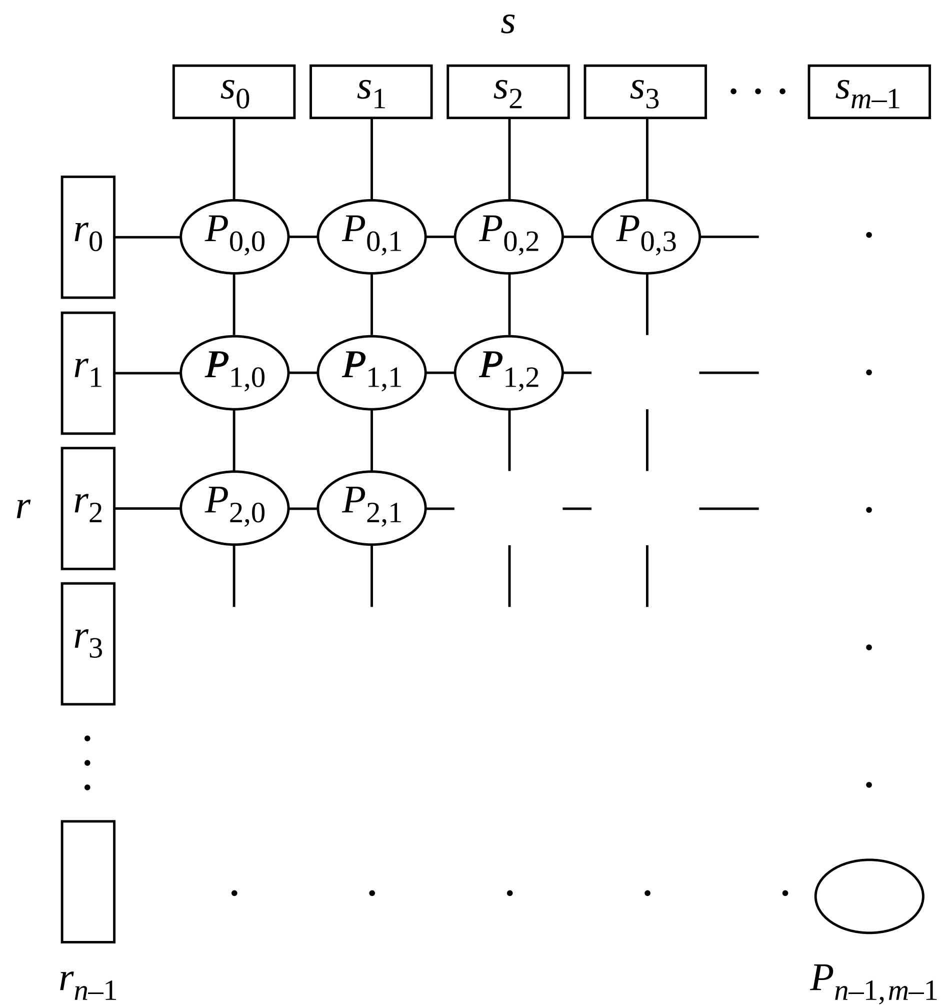
Fragment-and-Replicate Join (Cont.)

- General case: reduces the sizes of the relations at each processor.
  - r is partitioned into n partitions, $r_0, r_1, \ldots, r_{n-1}$; s is partitioned into m partitions, $s_0, s_1, \ldots, s_{m-1}$.
  - There must be at least $m * n$ processors.
  - Label the processors as $P_{0,0}, P_{0,1}, \ldots, P_{0,m-1}, P_{1,0}, \ldots, P_{n-1,m-1}$.
  - $P_{i,j}$ computes the join of r_i with s_j. In order to do so, r_i is replicated to $P_{i,0}, P_{i,1}, \ldots, P_{i,m-1}$, while s_i is replicated to $P_{0,i}, P_{1,i}, \ldots, P_{n-1,i}$.
  - Any join technique can be used at each processor $P_{i,j}$.

Depiction of Fragment-and-Replicate Joins



(a) Asymmetric fragment and replicate



(b) Fragment and replicate

Fragment-and-Replicate Join (Cont.)

- Fragment-and-replicate works with any join condition since every tuple in r can be tested with every tuple in s.
- Usually has a higher cost than partitioning, when both relations are of roughly the same size, since at least one of the relations has to be replicated.
- Say s is small; it may be cheaper to replicate s across all processors, rather than repartition r and s on the join attributes. Here, asymmetric fragment-and-replicate is preferable even though partitioning could be used.

Partitioned Parallel Hash-Join

Assume n processors P_0, $P_1, ..., P_{n-1}$, and two relations r and s, such that r and s are partitioned across multiple disks. Also assume 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 processors.
- Each processor P_i reads the tuples of s that are on its disk D_i, and sends each tuple to the appropriate processor based on hash function h_1.
- Let r_i denote the tuples of relation r that are sent to processor P_i; let s_i denote the tuples of relation s that are sent to processor P_i.
- As tuples of relation s are received at the destination processors, they are partitioned further using another hash function, h_2, which is used to compute the hash-join locally.

Partitioned Parallel Hash-Join (Cont.)

- Once the tuples of s have been distributed, the larger relation r is redistributed across the m processors using hash function h_1.
- As the tuples are received at the destination processors, they are repartitioned using the function h_2 (just as the probe relation is partitioned in the sequential hash-join algorithm).
- Each processor P_i 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.
- Note: Hash-join optimizations can be applied to the parallel case; e.g., the hybrid hash-join algorithm can be used to cache some of the incoming tuples in memory and avoid the cost of writing them and reading them back in.

Parallel Nested-Loop Join

- Assume that
  - relation s is much smaller than relation r and that r is stored by partitioning.
  - there is an index on a join attribute of relation r at each of the partitions of relation r.
- Use asymmetric fragment-and-replicate, with relation s being replicated, and using the existing partitioning of relation r.
- Each processor P_j where a partition of relation s is stored reads the tuples of relation s stored in D_j, and replicates the tuples to every other processor P_i. At the end of this phase, relation s is replicated at all sites that store tuples of relation r.

Parallel Nested-Loop Join (Cont.)

- Each processor P_i performs an indexed nested-loop join of relation s with the ith partition of relation r.
  - This join can actually be overlapped with the distribution of tuples of relation s, to reduce the cost of writing the tuples of relation s to disk and reading them back.
  - However, the replication of relation s must be synchronized with the join so that there is enough space in in-memory buffers at each processor P_i to hold the tuples of relation s that have been received but not yet used in the join.

Other Relational Operations

Selection Example: $\sigma_\theta(r)$

- θ is of the form $a_i = v$ where a_i is an attribute and v a value.
  - If r is partitioned on a_i, the selection is performed at a single processor.
- 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 processor whose partition overlaps with the specified range of values.
- All other cases: the selection is performed in parallel at all the processors.

Other Relational Operations (Cont.)

Duplicate elimination

- Perform by using either of the parallel sort techniques; with the optimization of eliminating duplicates as soon as they are found during sorting.
- Can also partition the tuples (using either range- or hash-partitioning) and performing duplicate elimination locally at each processor.

Projection

- Projection without duplicate elimination can be performed as tuples are read in from disk in parallel.

Other Relational Operations (Cont.)

Grouping/Aggregation

- Partition the relation on the grouping attributes and then compute the aggregate values locally at each processor.
- Reduce cost of transferring tuples during partitioning by partly computing aggregate values before partitioning.
- Consider the **sum** aggregation operation:
  - Perform operation at each processor P_i on those tuples stored on disk D_i; results in tuples with partial sums at each processor.
  - Result of the local aggregation is partitioned on the grouping attributes, and the aggregation performed again at each processor P_i to get the final result.
- Fewer tuples need to be sent to other processors during partitioning.

Cost of Parallel Evaluation of Operations

- The time taken by a parallel operation can be estimated as

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

  - T_{part} is the time for partitioning the relations
  - T_{asm} is the time for assembling the results
  - T_i the time taken for the operation at processor P_i

- Can handle skew in joins with range-partitioning by constructing and storing a frequency table (or *histogram*) of the attribute values for each attribute of each relation.

Interoperation Parallelism

Pipelined Parallelism

- Consider a join of four relations: $r_1 \bowtie r_2 \bowtie r_3 \bowtie r_4$
- Set up a pipeline computes the three joins in parallel.
- Let processor P_1 be assigned the computation of $temp_1 \leftarrow r_1 \bowtie r_2$ and let P_2 be assigned the computation of $r_3 \bowtie temp_1$.
- As P_1 computes tuples in $r_1 \bowtie r_2$, it makes these tuples available to processor P_2.
- Thus, P_2 has available to it some of the tuples in $r_1 \bowtie r_2$ before P_1 has finished its computation. P_2 can use those tuples to begin computation of $temp_1 \bowtie r_3$ even before $r_1 \bowtie r_2$ is fully computed by P_1.
- As P_2 computes tuples in $(r_1 \bowtie r_2) \bowtie r_3$, it makes these tuples available to P_3, which computes the join of these tuples with r_4.

Factors Limiting Utility of Pipeline Parallelism

- Pipeline chains do not attain sufficient length.
- Cannot pipeline operators which do not produce output until all inputs have been accessed (i.e., 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

- Operations in a query expression that do not depend on each other can be executed in parallel.
- Consider the join $r_1 \bowtie r_2 \bowtie r_3 \bowtie r_4$.
- Compute $temp_1 \leftarrow r_1 \bowtie r_2$ in parallel with $temp_2 \leftarrow r_3 \bowtie r_4$.
- When these two computations complete, we compute:

$$temp_1 \bowtie temp_2$$

- To get further parallelism, the tuples in $temp_1$ and $temp_2$ can be pipelined into the computation of $temp_1 \bowtie temp_2$, which is itself carried out using pipelined join.
- Does not provide a high degree of parallelism; less useful in a highly parallel system, although it is useful with a lower degree of parallelism.

Query Optimization

- Query optimization in parallel databases is significantly more complex than query optimization in sequential databases.
- Must take into account partitioning costs and issues such as skew and resource contention.
- In scheduling execution tree in parallel system, must decide:
  - How to parallelize each operation and how many processors to use for it.
  - What operations to pipeline, what operations to execute independently in parallel, and what operations to execute sequentially, one after the other.
  - Constrained by the sequential and pipelined dependencies in the execution tree.

Design of Parallel Systems

- Parallelization of data storage and parallelization of query processing.
- Parallel loading of data from external sources in order to handle large volumes of incoming data.
- Resilience to failure of some processors or disks.
- On-line reorganization of data and schema changes.