Advanced DB

CHAPTER 21
PARALLEL DATABASES

Chapter 21: Parallel Databases

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

Introduction

- Parallel machines are becoming quite common and affordable
  - Prices of microprocessors, memory and disks have dropped sharply
- Databases are growing increasingly large
  - large volumes of transaction data are collected and stored for later analysis (eg. data warehouses)
  - multimedia objects like images are increasingly stored in databases
  - demand for high throughput transaction systems are common
- Databases naturally lend themselves to parallelism
  - Data can be partitioned across multiple disks for parallel I/O.
  - Individual relational operations (e.g., sort, join, aggregation) can be executed in parallel
  - Queries are expressed in high level language
  - Different queries can be run in parallel with each other
    - Concurrency control takes care of conflicts

Exploiting Parallelism

- I/O Parallelism
- Interquery Parallelism
    - e.g., assign one query to each node
- Intraquery Parallelism
    - One query is processed by multiple nodes
  - **Inter-operation Parallelism**
    - Breakup query into subqueries (operations)
    - Assign each operation to a node
  - **Intra-operation Parallelism**
    - Utilize multiple nodes to execute one operation
    - Parallel join, parallel sort, etc.

I/O Parallelism

- Reduce the time required to retrieve relations from disk by partitioning the relations on multiple disks.
- Horizontal partitioning used mainly
- Partitioning techniques
  - Round-robin
  - Hash partitioning
  - Range partitioning
- Evaluate how well each technique supports different types of data access
  - ***Table scan***: scanning the entire relation
  - ***Point queries***: locating a tuple associatively: e.g., $r.A = 25$.
  - ***Range queries***: locating all tuples attribute value lies within a specified range: e.g., $10 \le r.A < 25$.

Round-robin

- Method
  - Send the ith tuple inserted in the relation to disk (i mod n)
- Advantages
  - Best suited for sequential scan of entire relation on each query
  - All disks have almost an equal number of tuples; retrieval work is thus well balanced between disks.
- Range queries are difficult to process
  - No clustering - tuples are scattered across all disks

Hash Partitioning

- Method
  1. Choose one or more attributes as the partitioning attributes (say, A)
  2. Choose hash function h with range $0 \cdots n - 1$
  3. Send tuple to disk $h(t[A])$
- Good for sequential access
  - If partitioning attribute is a key and hash function is good
    => tuples will be equally distributed between disks
  - Retrieval work is then well balanced between disks.
- Good for point queries on partitioning attribute
  - Can locate correct disk
  - Index on partitioning attribute can be local to disk
    => making lookup and update more efficient
- No clustering, so difficult to answer range queries

Range Partitioning

- Method
  1. Choose an attribute as the partitioning attribute (say, A)
  2. A partitioning vector $[v_o, v_1, ..., v_{n-2}]$ is chosen, for $v_i \leq v_{i+1}$
  3. If $t[A]$ is between v_i and v_{i+1} , t goes to disk $i + 1$

     If $t[A] < v_0$, t goes to disk 0

     if $t[A] \geq v_{n-2}$, t goes to disk n-1.
- Good for sequential access
- Good for point queries on partitioning attribute: only one disk needs to be accessed.
- Provides data clustering by partitioning attribute value
  - For range queries on partitioning attribute, only a few disks may need to be accessed
  - If many blocks are to be fetched, they are still fetched from one to a few disks, and potential parallelism in disk access is wasted

Data Skew

- Distribution of tuples to disks may be *skewed*
  - some disks have many tuples, while others may have fewer tuples
- Types of skew:
  - Attribute-value skew
    - Some values appear in the partitioning attributes of many tuples
    - all tuples with same value for the partitioning attribute end up in the same partition
    - can occur with range-partitioning and hash-partitioning.
  - Partition skew
    - With range-partitioning, badly chosen partition 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.

Interquery Parallelism

- Queries/transactions execute in parallel with one another
- Increases transaction throughput
  - used primarily to scale up a transaction processing system
- Easiest form of parallelism
  - particularly in a shared-memory parallel database
  - similar to concurrent processing in a sequential database system
- More complicated on shared-disk or shared-nothing architectures
  - Locking and logging must be coordinated by passing messages between processors
  - Data in a local buffer may have been updated at another processor
  - ***Cache-coherency*** has to be maintained — reads and writes of data in buffer must find latest version of data.

Intraquery Parallelism

- Execution of a single query in parallel on multiple processors/disks
  - important for speeding up long-running queries.
- Two complementary forms of intraquery parallelism :
  - **Interoperation Parallelism** – execute the different operations in a query expression in parallel
  - **Intraoperation Parallelism** – parallelize the execution of each individual operation in the query
    - scales better with increasing parallelism because
      the number of tuples processed by each operation is typically more than the number of operations in a query

Intraoperation parallelism

- Parallel processing of relational operators
- We assume:
  - *read-only* queries
  - shared-nothing architecture
  - *n* processors, P_0, ..., P_{n-1}, and *n* disks D_0, ..., D_{n-1}, where disk D_i is associated with processor P_i
- Shared-nothing architectures can be efficiently simulated on shared-memory and shared-disk systems.
  - Algorithms for shared-nothing systems can be run on shared-memory and shared-disk systems
  - some optimizations may be possible

Range-Partitioning Sort

- Choose m ($\leq n - 1$) processors $P_0, ..., P_m$ for sorting
- Create range-partition vector with m entries, on the sorting attribute(s)
- Redistribute the relation using range partitioning
  - all tuples in the ith range are sent to processor P_i
  - P_i stores the tuples temporarily on disk D_i.
  - This step requires I/O and communication overhead.
- Each P_i sorts its partition of the relation locally
  - Each processor executes same operation (sort) in parallel with other processors, without any interaction with the others
- Final merge operation
  - is trivial (concatenation)
  - range-partitioning ensures that, for $1 \leq j \leq m$, key values in P_{j-1} are all less than the key values in P_j.

Parallel External Sort Merge

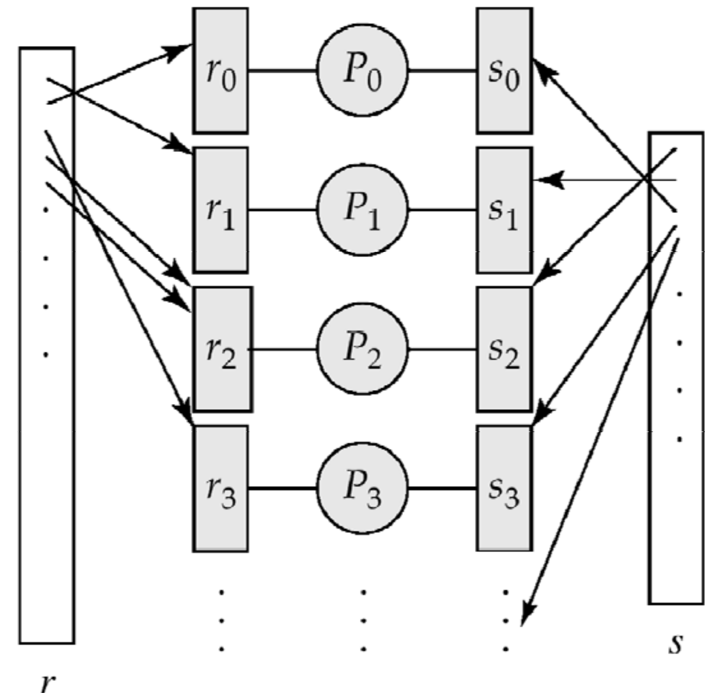
- Assume the relation is partitioned among disks D_0, ..., D_{n-1}
  - not necessarily on sort key
- Each processor P_i locally sorts the data on disk D_i.
- The sorted runs on each processor are then merged to get the final sorted output.
- Parallelize merge operation:
  - The sorted runs at each processor are range-partitioned across the processors P_0, ..., P_{m-1}.
  - Each processor performs a merge on the streams as they are received
  - The sorted runs are concatenated to get the final result

Parallel Join

- The join operation
  - pairs of tuples need to be tested to see if they satisfy the join condition
  - if they do, the pair is added to the join output
- Parallel join algorithms
  - 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 equi-joins
  - *partition* the two input relations across the processors
  - and compute the join locally at each processor
- Suppose we want to compute $r \bowtie_{r.A=s.B} s$
- r and s each are partitioned into n partitions
  - denoted $r_0, r_1, ..., r_{n-1}$ and $s_0, s_1, ..., s_{n-1}$.
  - Can use either *range partitioning* or *hash partitioning*.
  - must be partitioned on join attributes $r.A$ and $s.B$
  - using the same range-part. vector or hash func.
- Partitions r_i and s_i are sent to processor P_i
- Each P_i locally computes $r_i \bowtie_{ri.A=\text{si}.B} s_i$
  - Any of the standard join methods can be used.



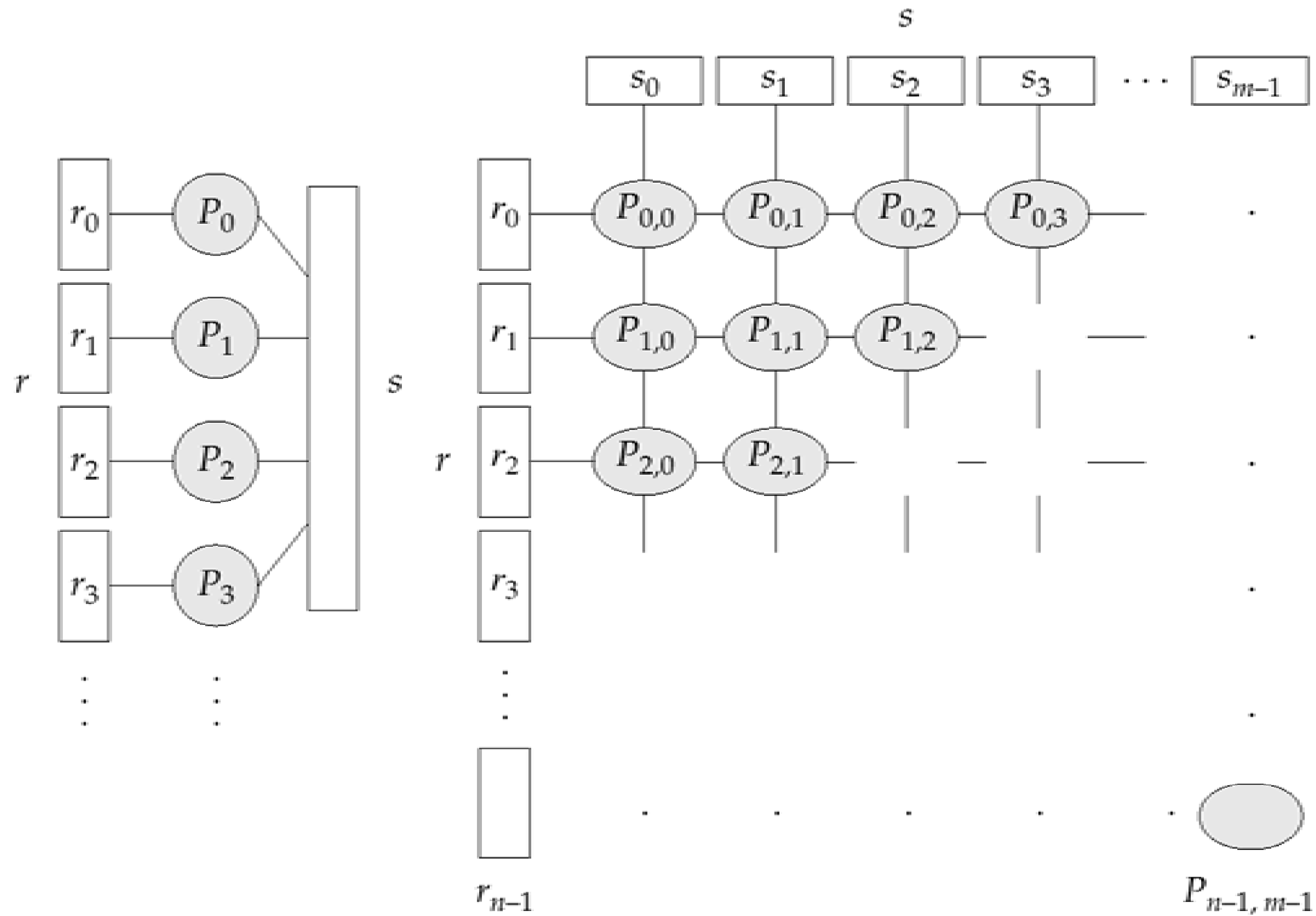
© Silberschatz, Korth and Sudarshan

Copyright © 2006 - 2008 by S.-g. Lee

Fragment-and-Replicate Join

- Partitioned Join is not possible for non-equijoin conditions
  - e.g., $r.A > s.B$
- Fragment and Replicate
  - Partition r into n partitions, s into m partitions
  - Sent to $m*n$ processors
- Special case – *asymmetric fragment-and-replicate*
  - m=1 (or n=1).
  - Processor P_i locally computes the join of r_i with all of s using any join technique

Fragment-and-Replicate Join (cont.)



© Silberschatz, Korth and Sudarshan

Copyright © 2006 - 2008 by S.-g. Lee

Fragment-and-Replicate Join (cont.)

- Partition *r* and *s*: any partitioning technique may be used
  - r is partitioned into n partitions: $r_0, r_1, ..., r_{n-1}$
  - s is partitioned into m partitions: $s_0, s_1, ..., s_{m-1}$
- There must be at least $m * n$ processors
  - Label the processors as $P_{0,0}, P_{0,1}, ..., P_{0,m-1}, P_{1,0}, ..., P_{n-1,m-1}$
- $P_{i,j}$ computes $r_i \bowtie s_j$
  - r_i is replicated to $P_{i,0}, P_{i,1}, ..., P_{i,m-1}$
  - s_i is replicated to $P_{0,i}, P_{1,i}, ..., P_{n-1,i}$
- Any join technique can be used at each processor $P_{i,j}$.

Fragment-and-Replicate Join (cont.)

- Works with any join condition (general or asymmetric)
  - 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 over partitioned join.
  - e.g., say *s* is small and *r* is large, and already partitioned.

    It may be cheaper to replicate *s* across all processors,

    rather than repartition *r* and *s* on the join attributes.

Other Relational Operations

- Selection: $\sigma_{\theta}(r)$
  - If θ is $a_i = v$, 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.
  - If θ is of the form $l <= a_i <= u$ (i.e., θ is a range selection)
    - if r has been range-partitioned on a_i
    - then selection is performed at each processor whose partition overlaps with the specified range of values.
  - In all other cases: the selection is performed in parallel at all the processors.
- Duplicate elimination
  - Perform by using either of the parallel sort techniques
    - eliminate duplicates as soon as they are found during sorting.
  - Can also partition the tuples (using either range- or hash- partitioning) and perform duplicate elimination locally at each processor.

Cost of Parallel Evaluation of Operations

- Ideally, without skew or overhead, expected speed-up will be n (runtime: $1/n$)
- Actual parallel operation time can be estimated as

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

  - T_{part} is the time for partitioning the relations
  - T_{asm} is the time for assembling the results
  - T_i is the time taken for the operation at processor P_i
    - skew and time wasted in contentions can be additional overhead

Interoperation Parallelism

- Consider a join of four relations $r_1 \bowtie r_2 \bowtie r_3 \bowtie r_4$
- Pipelined parallelism
  - Assign P_1 to compute $temp_1 = r_1 \bowtie r_2$
    P_2 to compute $temp_2 = temp1 \bowtie r_3$
    P_3 to compute $temp_2 \bowtie r_4$
  - Each of these operations can be executed in parallel
    - Send result tuples to the next operation as they are being generated
  - Requires a pipeline-able join evaluation algorithm (e.g. indexed nested loops join)
- Independent parallelism
  - Assign P_1 to compute $temp_1 = r_1 \bowtie r_2$
    P_2 to compute $temp_2 = r_3 \bowtie r_4$
    P_3 to compute $temp_1 \bowtie temp_2$
  - P_1 and P_2 can work independently in parallel
  - P_3 has to wait for input from P_1 and P_2
  - Can pipeline output of P_1 and P_2 to P_3, combining independent parallelism and pipelined parallelism

Interoperation Parallelism (Cont.)

- Pipeline parallelism is useful since it avoids writing intermediate results to disk
- Useful with small number of processors
- But does not scale up well with more processors
  - only a limited number of operations in a query
  - pipeline chains do not attain sufficient length
- Cannot pipeline operators which do not produce output until all inputs have been accessed (e.g. aggregate and sort)
- Little speedup in cases of skew where one operator's execution cost is much higher than the others (typical)

Query Optimization

- Query optimization in parallel databases is significantly more complex than in sequential databases.
- Cost models are more complicated
  - partitioning costs
  - skew
  - resource contention
- Optimizer must decide:
  - How to parallelize each operation
  - How many processors to use
    - allocating more processors than optimal can result in high communication overhead
  - Which operations to pipeline (long pipelines should be avoided), to execute independently in parallel, or to execute sequentially one after the other
- The number of "equivalent" parallel evaluation plans is much larger
  - heuristics are needed

END OF CHAPTER 21