Chapter 18: Parallel Databases

Database System Concepts, 6th Ed.

Chapter 18: Parallel Databases

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

Introduction

- Parallel machines are becoming quite 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
- Databases are growing increasingly large
  - large volumes of transaction data are collected and stored for later analysis.
  - multimedia objects like images are increasingly stored in databases
- Large-scale parallel database systems increasingly used for:
  - storing large volumes of data
  - processing time-consuming decision-support queries
  - providing high throughput for transaction processing

Parallelism in Databases

- Data can be partitioned across multiple disks for parallel I/O.
- Individual relational operations (e.g., sort, join, aggregation) can be executed in parallel
  - data can be partitioned and each processor can work independently on its own partition.
- Queries are expressed in high level language (SQL, translated to relational algebra)
  - makes parallelization easier.
- Different queries can be run in parallel with each other. Concurrency control takes care of conflicts.
- Thus, databases naturally lend themselves to parallelism.

Chapter 18: Parallel Databases

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

I/O Parallelism

- Reduce the time required to retrieve relations from disk by partitioning
- Each relation on multiple disks.
- Horizontal partitioning – tuples of a relation are divided among many disks such that each tuple resides on one disk.
- Partitioning techniques (number of disks = *n*):

Round-robin:

Send the *I* th tuple inserted in the relation to disk *i* mod *n*.

Hash partitioning:

- Choose one or more attributes as the partitioning attributes.
- Choose hash function *h* with range 0…*n* - 1
- Let *i* denote result of hash function *h* applied to the partitioning attribute value of a tuple. Send tuple to disk *i*.

I/O Parallelism (Cont.)

- Partitioning techniques (cont.):
- **Range partitioning**:
  - Choose an attribute as the partitioning attribute.
  - A partitioning vector $[v_o, v_1, ..., v_{n-2}]$ is chosen.
  - Let v be the partitioning attribute value of a tuple. Tuples such that $v_i \leq v_{i+1}$ go to disk $I + 1$. Tuples with $v < v_0$ go to disk 0 and tuples with $v \geq v_{n-2}$ go to disk n-1.

    E.g., with a partitioning vector [5,11], a tuple with partitioning attribute value of 2 will go to disk 0, a tuple with value 8 will go to disk 1, while a tuple with value 20 will go to disk2.

Comparison of Partitioning Techniques (Cont.)

Round robin:

- Advantages
  - Best suited for 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

Comparison of Partitioning Techniques (Cont.)

Hash partitioning:

- Good for scan
  - Assuming hash function is good, and partitioning attributes form a key, tuples will be equally distributed between disks
  - Retrieval work is then well balanced between disks.
- Good for point queries on partitioning attribute
  - Can lookup single disk, leaving others available for answering other queries.
  - Index on partitioning attribute can be local to disk, making lookup and update more efficient
- No clustering, so difficult to answer **range queries**

Comparison of Partitioning Techniques (Cont.)

Range partitioning:

- Provides data clustering by partitioning attribute value.
- Good for sequential access
- Good for point queries on partitioning attribute: only one disk needs to be accessed.
- For range queries on partitioning attribute, one to a few disks may need to be accessed
  - Remaining disks are available for other queries.
  - Good if result tuples are from one to a few blocks.
  - 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
    - Example of execution skew.

Handling of Skew

- The distribution of tuples to disks may be **skewed** — that is, 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 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**.
    - 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.

Chapter 18: Parallel Databases

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

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.

Chapter 18: Parallel Databases

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

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:
  - **Intraoperation Parallelism** – parallelize the execution of each individual operation in the query.
  - **Interoperation Parallelism** – execute the different operations in a query expression in parallel.

the first form 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.

Parallel Processing of Relational Operations

- Our discussion of parallel algorithms assumes:
  - *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.
- If a processor has multiple disks they can simply simulate a single disk D_i.

Parallel Sort

Range-Partitioning Sort

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

Parallel Join

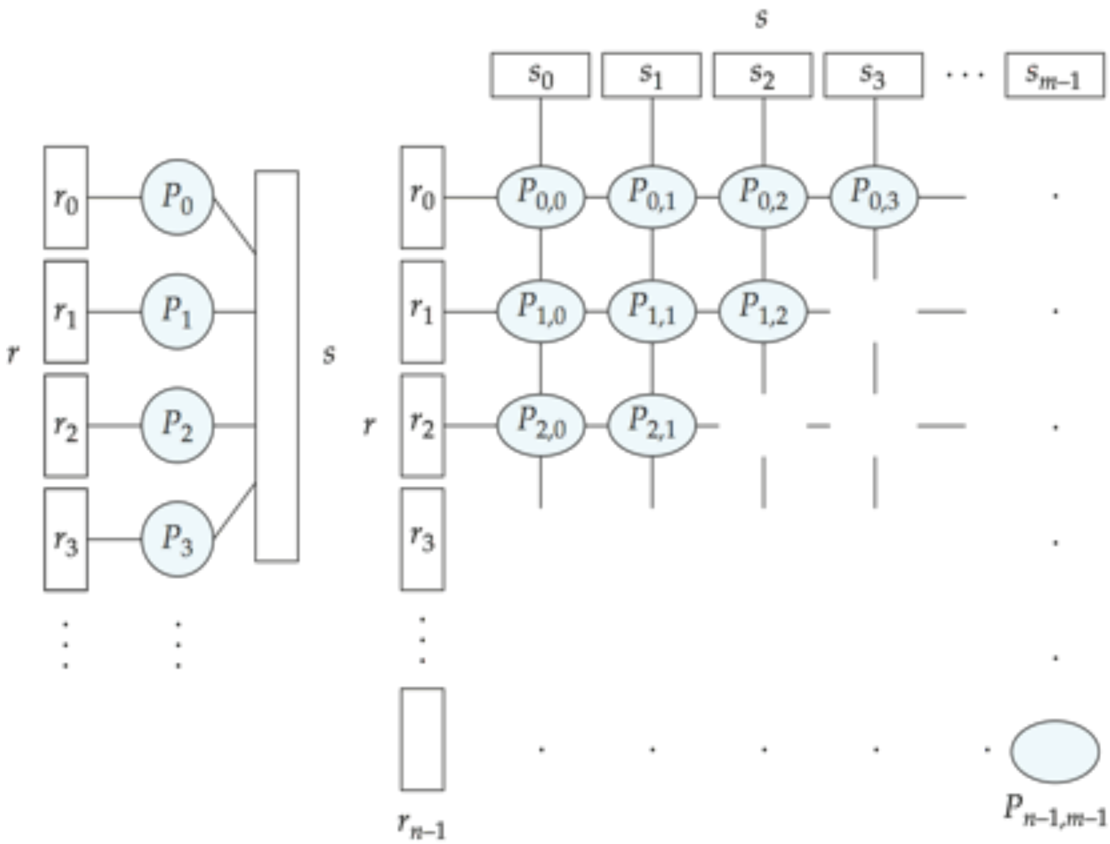
- The join operation requires pairs of tuples to be tested to see if they satisfy the join condition, and if they do, the pair is added to the join output.
- Parallel join algorithms attempt to split the pairs to be tested over several processors. Each processor then computes part of the join locally.
- In a final step, the results from each processor can be collected together to produce the final result.

Partitioned Join

- For 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.
- Let *r* and *s* be the input relations, and 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* on the join attributes *r*.A and *s*.B.
- Partitions r_i and s_i are sent to processor P_i,
- Each processor P_i locally computes $r_i \bowtie_{ri.A=si.B} s_i$. Any of the standard join methods can be used.

Fragment-and-Replicate Join

- Partitioning not possible for some join conditions
  - E.g., non-equijoin conditions, such as r.A > s.B.

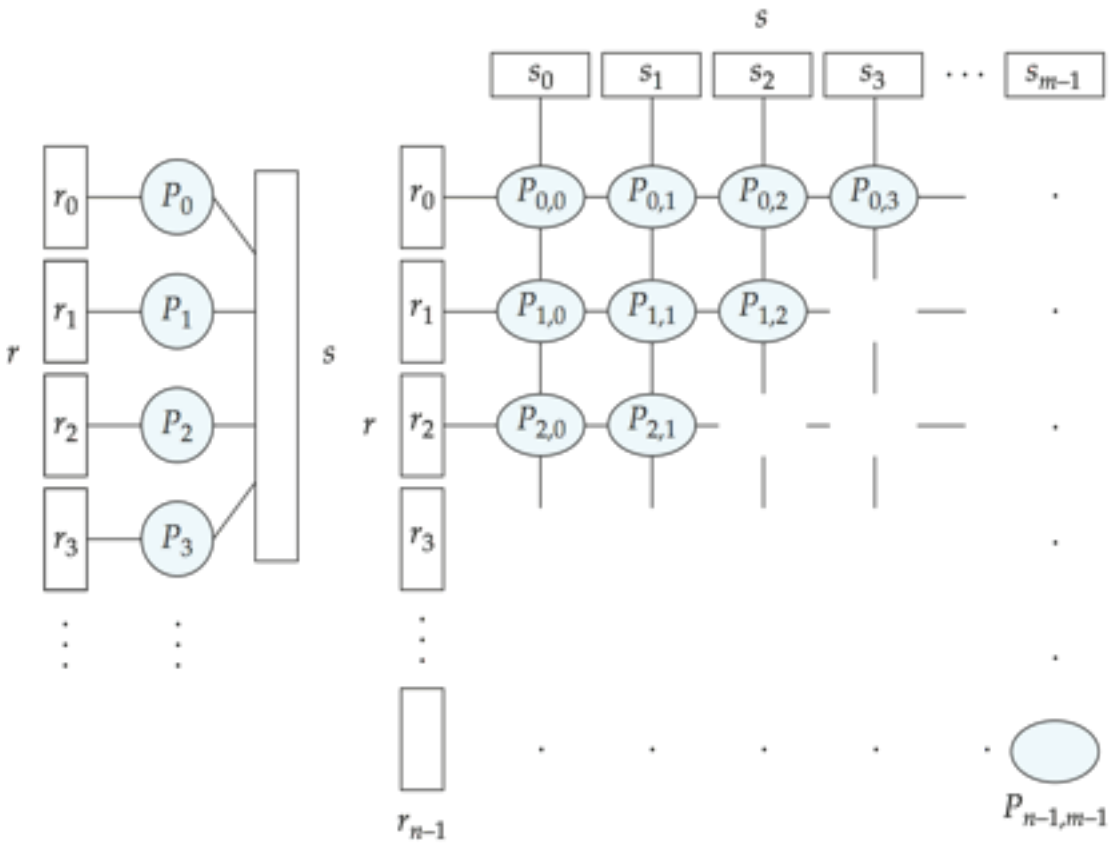


(a) Asymmetric fragment and replicate

(b) Fragment and replicate

Fragment-and-Replicate Join

- Partitioning not possible for some join conditions
  - E.g., non-equijoin conditions, such as r.A > s.B.



(a) Asymmetric fragment and replicate

(b) Fragment and replicate

E.g., assume 16 nodes

1. if R:1GB, S:1GB
(a) R 1 time, S 16 times (17GB)
(b) R 4 times, S 4 times (8GB)

2. if R:1GB, S:1MB
(a) R 1 time, S 16 times (1,016MB)
(b) R 4 times, S 4 times (4,004MB)

Other Relational Operations

Selection $\sigma_\theta(r)$

- If 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.
- If θ is of the form $l <= a_i <= u$ (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.
- In all other cases: the selection is performed in parallel at all the processors.

Grouping/Aggregation

- Partition the relation on the grouping attributes and then compute the aggregate values locally at each processor.
- Can reduce cost of transferring tuples during partitioning by partly computing aggregate values before partitioning.
- Consider the **sum** aggregation operation:
  - Perform aggregation 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.

Interoperator Parallelism

- **Pipelined parallelism**
  - 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
    - Let P1 be assigned the computation of

      $temp1 = r_1 \bowtie r_2$
    - And P2 be assigned the computation of $temp2 = temp1 \bowtie r_3$
    - And P3 be assigned the computation of $temp2 \bowtie r_4$
  - Each of these operations can run in parallel, keeping sending result tuples to the next operation
    - Provided a pipelineable join evaluation algorithm (e.g., indexed nested loops join) is used

Query Optimization

- Query optimization in parallel databases is significantly more complex than query optimization in sequential databases.
- Cost models are more complicated, since we must take into account partitioning costs and issues such as skew and resource contention.
- When **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.
- Determining the amount of resources to allocate for each operation is a problem.
  - E.g., allocating more processors than optimal can result in high communication overhead.
- Long pipelines should be avoided as the final operation may wait a lot for inputs, while holding precious resources

Chapter 18: Parallel Databases

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

Design of Parallel Systems

Some issues in the design of parallel systems:

- Parallel loading of data from external sources is needed in order to handle large volumes of incoming data.
- Resilience to failure of some processors or disks.
  - Probability of some disk or processor failing is higher in a parallel system.
  - Operation (perhaps with degraded performance) should be possible in spite of failure.
  - Redundancy achieved by storing extra copy of every data item at another processor.

Design of Parallel Systems (Cont.)

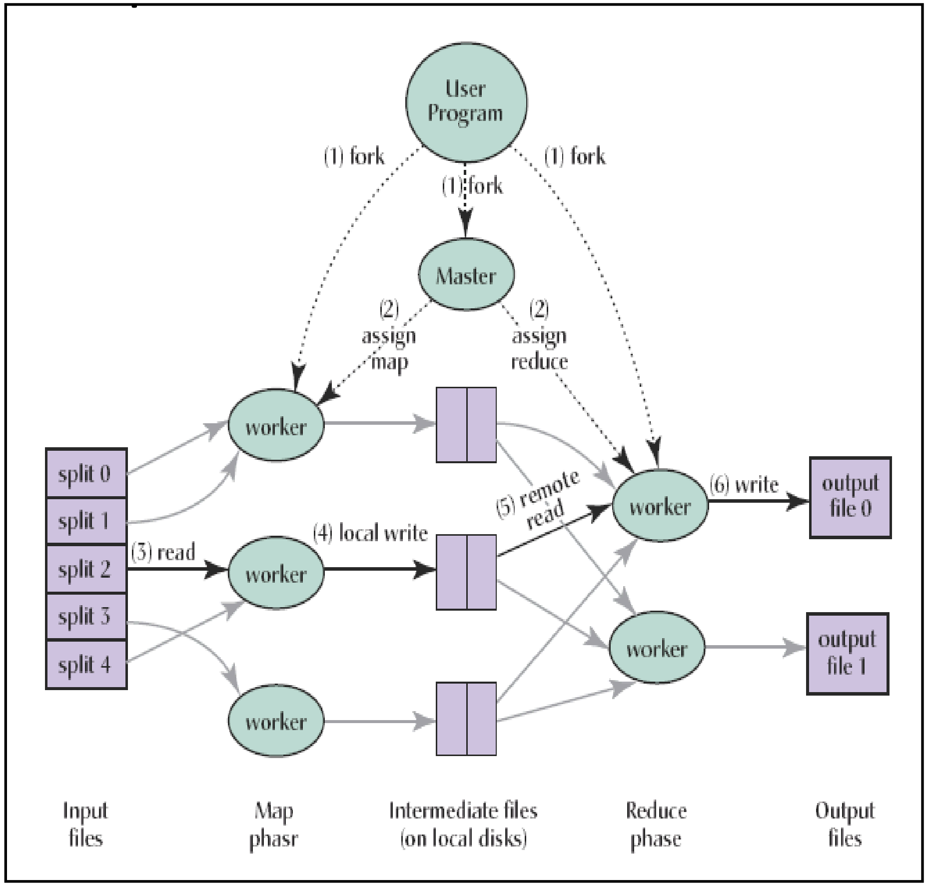
- On-line reorganization of data and schema changes must be supported.
  - For example, index construction on terabyte databases can take hours or days even on a parallel system.
    - Need to allow other processing (insertions/deletions/updates) to be performed on relation even as index is being constructed.
  - Basic idea: index construction tracks changes and “catches up” on changes at the end.
- Also need support for on-line repartitioning and schema changes (executed concurrently with other processing).

Chapter 18: Parallel Databases

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

MapReduce Overview

- Large-scale parallel programming framework
  - express what to compute
  - don't worry about parallelism, fault-tolerance, data distribution, and load balancing
- Applications
  - grep, sort, word count, data mining, etc.
  - @Google, Yahoo!, Amazon, Facebook, IBM, ...
- History
  - 2003: Google implements MapReduce.
  - 2004: MapReduce supports 1000 services at Google.
  - 2006: Hadoop started.
  - 2008: Yahoo! generates search index on Hadoop.



Monday, October 7, 2013

Programming in MapReduce

- WordCount example
  - Map: given a text file, for each word, output the word and count of 1:
    - "to be or not to be" -> (to, 1), (be, 1), (or, 1), (not, 1), (to, 1), (be, 1)
  - Reduce: For each word & the associated set of counts, output the word and the sum:
    - (to, [1, 1]) -> (to, 2)

```
public static void main(String[] args) throws IOException {
  JobConf conf = new JobConf(WordCount.class);
  conf.setJobName("wordcount");

  conf.setOutputKeyClass(Text.class);
  conf.setOutputValueClass(IntWritable.class);

  conf.setMapperClass(WCMap.class);
  conf.setCombinerClass(WCReduce.class);
  conf.setReducerClass(WCReduce.class);

  conf.setInputPath(new Path(args[0]));
  conf.setOutputPath(new Path(args[1]));

  JobClient.runJob(conf);
}
```

```
public class WCMap extends MapReduceBase implements Mapper {

  private static final IntWritable ONE = new IntWritable(1);

  public void map(WritableComparable key, Writable value,
                  OutputCollector output,
                  Reporter reporter) throws IOException {
    StringTokenizer itr = new StringTokenizer(value.toString());
    while (itr.hasMoreTokens()) {
      output.collect(new Text(itr.next()), ONE);
    }
  }
}
```

```
public class WCReduce extends MapReduceBase implements Reducer {

  public void reduce(WritableComparable key, Iterator values,
                     OutputCollector output,
                     Reporter reporter) throws IOException {
    int sum = 0;
    while (values.hasNext()) {
      sum += ((IntWritable) values.next()).get();
    }
    output.collect(key, new IntWritable(sum));
  }
}
```

MapReduce: A major step backwards

- By Michael Stonebraker and David Dewitt

- http://databasecolumn.vertica.com/database-innovation/mapreduce-a-major-step-backwards/