CHAPTER 22

Parallel and Distributed Query Processing

Practice Exercises

22.1 What form of parallelism (interquery, interoperation, or intraoperation) is likely to be the most important for each of the following tasks?

a. Increasing the throughput of a system with many small queries

b. Increasing the throughput of a system with a few large queries when the number of disks and processors is large

Answer:

a. When there are many small queries, interquery parallelism gives good throughput. Parallelizing each of these small queries would increase the initiation overhead, without any significant reduction in response time.

b. With a few large queries, intraquery parallelism is essential to get fast response times. Given that there are large numbers of processors and disks, only intraoperation parallelism can take advantage of the parallel hardware, for queries typically have few operations, but each one needs to process a large number of tuples.

22.2 Describe how partial aggregation can be implemented for the **count** and **avg** aggregate functions to reduce data transfer.

Answer:
FILL

22.3 With pipelined parallelism, it is often a good idea to perform several operations in a pipeline on a single processor, even when many processors are available.

a. Explain why.

b. Would the arguments you advanced in part *a* hold if the machine has a shared-memory architecture? Explain why or why not.

c. Would the arguments in part *a* hold with independent parallelism? (That is, are there cases where, even if the operations are not pipelined and there are many processors available, it is still a good idea to perform several operations on the same processor?)

Answer:

a. The speedup obtained by parallelizing the operations would be offset by the data transfer overhead, as each tuple produced by an operator would have to be transferred to its consumer, which is running on a different processor.

b. In a shared-memory architecture, transferring the tuples is very efficient. So the above argument does not hold to any significant degree.

c. Even if two operations are independent, it may be that they both supply their outputs to a common third operator. In that case, running all three on the same processor may be better than transferring tuples across processors.

22.4 Consider join processing using symmetric fragment and replicate with range partitioning. How can you optimize the evaluation if the join condition is of the form $| r.A - s.B | \leq k$, where k is a small constant? Here, $| x |$ denotes the absolute value of x. A join with such a join condition is called a **band join**.

Answer:

Relation r is partitioned into n partitions, $r_0, r_1, \ldots, r_{n-1}$, and s is also partitioned into n partitions, $s_0, s_1, \ldots, s_{n-1}$. The partitions are replicated and assigned to processors as shown in **??**

Each fragment is replicated on three processors only, unlike in the general case where it is replicated on n processors. The number of processors required is now approximately $3n$, instead of n^2 in the general case. Therefore, given the same number of processors, we can partition the relations into more fragments with this optimization, thus making each local join faster.

22.5 Suppose relation r is stored partitioned and indexed on A, and s is stored partitioned and indexed on B. Consider the query:

$$_{r.C}\gamma_{\text{count}(s.D)}(\ (\sigma_{A>5}(r)) \bowtie_{r.B=s.B} s\ )$$

a. Give a parallel query plan using the exchange operator, for computing the subtree of the query involving only the select and join operators.

b. Now extend the above to compute the aggregate. Make sure to use preaggregation to minimize the data transfer.

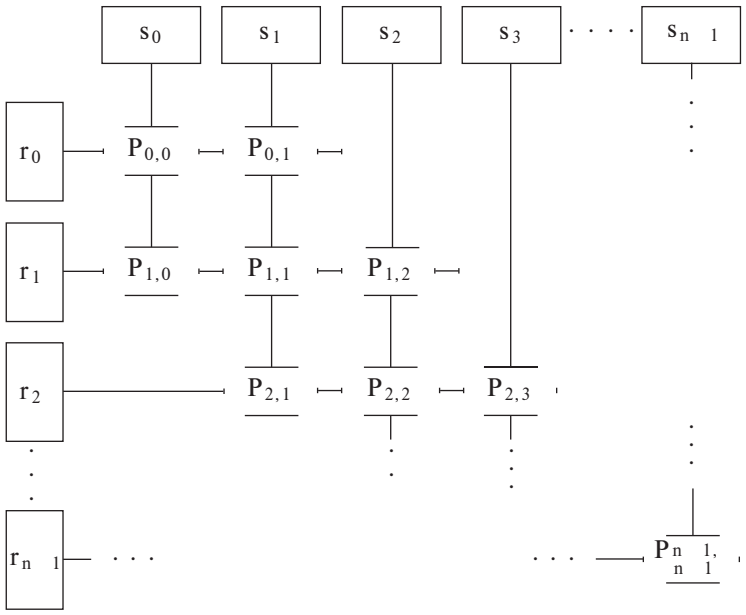

Figure 22.101 The three levels of data abstraction.

c. Skew during aggregation is a serious problem. Explain how pre-aggregation as above can also significantly reduce the effect of skew during aggregation.

Answer:

a. This is a small variant of an example from the chapter.

b. This one is very straightforward, since it is already the example in the chapter

c. Pre-aggregation can greatly reduce the size of the data sent to the final aggregation step. So even if there is skew, the absolute data sizes are smaller, resulting in significant reduction in the impact of the skew.

22.6 Suppose relation r is stored partitioned and indexed on A, and s is stored partitioned and indexed on B. Consider the join $r \bowtie_{r.B=s.B} s$. Suppose s is relatively small, but not small enough to make asymmetric fragment-and-replicate join the best choice, and r is large, with most r tuples not matching any s tuple. A hash-join can be performed but with a semijoin filter used to reduce the data transfer. Explain how semijoin filtering using Bloom filters would work in this parallel join setting.

Answer:

Since s is small, it makes sense to send a Bloom filter on $s.B$ to all partitions of r. Then we use the Bloom filter to find r tuples that may match some s tuple, and repartition the matching r tuples on $r.B$, sending them to the nodes containing s (which is already partitioned on $s.B$). Then the join can be performed at each site storing s tuples. The Bloom filter can significantly reduce the number of r tuples transferred.

Note that repartitioning s does not make sense since it is already partitioned on the join attribute, unlike r.

22.7 Suppose you want to compute $r \overset{\text{left}}{\bowtie}_{r.A=s.A} s$.

a. Suppose s is a small relation, while r is stored partitioned on $r.B$. Give an efficient parallel algorithm for computing the left outer join.

b. Now suppose that r is a small relation, and s is a large relation, stored partitioned on attribute $s.B$. Give an efficient parallel algorithm for computing the above left outer join.

Answer:

a. Replicating s to all nodes, and computing the left outerjoin independently at each node would be a good option in this case.

b. The best technique in this case is to replicate r to all nodes, and compute $r \bowtie s_i$ at each node i. Then, we send back the list of r tuples that had matches at site i back to a single node, which takes the union of the returned r tuples from each node i. Tuples in r that are absent in this union are then padded with nulls and added to the output.

22.8 Suppose you want to compute ${}_{A,B}\gamma_{sum(C)}$ on a relation s which is stored partitioned on $s.B$. Explain how you would do it efficiently, minimizing/avoiding repartitioning, if the number of distinct $s.B$ values is large, and the distribution of number of tuples with each $s.B$ value is relatively uniform.

Answer:

The aggregate can be computed locally at each node, with no repartitioning at all, since partitioning on $s.B$ implies partitioning on $s.A$, $s.B$. To understand why, partitioning on (A, B) requires that tuples with the same value for (A, B) must be in the same partition. Partitioning on just B, ignoring A, also satisfies this requirement.

Of course not partitioning at all also satisfies the requirement, but that defeats the purpose of parallel query processing. As long as the number of distinct $s.B$ values is large enough and the number of tuples with each $s.B$ value are relatively uniform and not highly skewed, using the existing partitioning on $s.B$ will give good performance.

22.9 MapReduce implementations provide fault tolerance, where you can reexecute only failed mappers or reducers. By default, a partitioned parallel join execution would have to be rerun completely in case of even one node failure. It is possible to modify a parallel partitioned join execution to add fault tolerance in a manner similar to MapReduce, so failure of a node does not require full reexecution of the query, but only actions related to that node. Explain what needs to be done at the time of partitioning at the sending node and receiving node to do this.

Answer: This is an application of ideas from MapReduce to join processing. There are two steps: first the data is repartitioned, and then join is performed, corresponding to the map and reduce steps.

A failure during the repartition can be handled by reexecuting the work of the failed node. However, the destination must ensure that tuples are not processed twice. To do so, it can store all received tuples in local disk, and start processing only after all tuples have been received. If the sender fails meanwhile, and a new node takes over, the receivers can discard all tuples received from the failed sender, and receive them again. This part is not too expensive.

Failures during the final join computation can be handled similar to reducer failure, by getting the data again from the partitioners. However, in the MapReduce paradigm tuples to be sent to reducers are stored on disk at the mappers, so they can be resent if required. This can also be done with parallel joins, but there is now a significant extra cost of writing the tuples to disk.

Another option is to find the tuples to be sent to the failed join node by rescanning the input. But now, all partitioners have to reread their entire input, which makes the process very expensive, similar in cost to rerunning the join. As a result this is not viewed as useful.

22.10 If a parallel data-store is used to store two relations r and s and we need to join r and s, it may be useful to maintain the join as a materialized view. What are the benefits and overheads in terms of overall throughput, use of space, and response time to user queries?

Answer:

Performing a join on a cloud data-storage system can be very expensive, if either of the relations to be joined is partitioned on attributes other than the join attributes, since a very large amount of data would need to be transferred to perform the join. However, if $r \bowtie s$ is maintained as a materialized view, it can be updated at a relatively low cost each time each time either r or s is updated, instead of incurring a very large cost when the query is executed. Thus, queries are benefitted at some cost to updates.

With the materialized view, overall throughput will be much better if the join query is executed reasonably often relative to updates, but may be worse if the join is rarely used, but updates are frequent.

The materialized view will certainly require extra space, but given that disk capacities are very high relative to IO (seek) operations and transfer rates, the extra space is likely to not be an major overhead.

The materialized view will obviously be very useful to evaluate join queries, reducing time greatly by reducing data transfer across machines.

22.11 Explain how each of the following join algorithms can be implemented using the MapReduce framework:

a. Broadcast join (also known as asymmetric fragment-and-replicate join).

b. Indexed nested loop join, where the inner relation is stored in a parallel data-store.

c. Partitioned join.

Answer:
FILL