CHAPTER 22

Parallel and Distributed Query Processing

Solutions for the Practice Exercises of Chapter 22

Practice Exercises

22.1

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

Answer:
FILL

22.3

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.

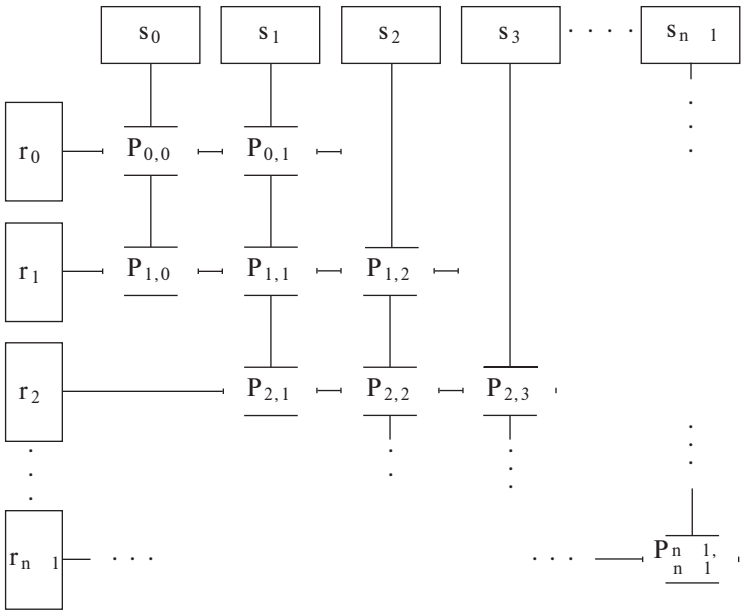

Figure 22.101 The three levels of data abstraction.

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

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 Figure 22.101

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

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

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

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

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

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

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

Answer:
FILL