CHAPTER 18

Parallel Databases

This chapter is suitable for an advanced course, but can also be used for independent study projects by students of a first course. The chapter covers several aspects of the design of parallel database systems — partitioning of data, parallelization of individual relational operations, and parallelization of relational expressions. The chapter also briefly covers some systems issues, such as cache coherency and failure resiliency.

The most important applications of parallel databases today are for warehousing and analyzing large amounts of data. Therefore partitioning of data and parallel query processing are covered in significant detail. Query optimization is also of importance, for the same reason. However, parallel query optimization is still not a fully solved problem; exhaustive search, as is used for sequential query optimization, is too expensive in a parallel system, forcing the use of heuristics.

The description of parallel query processing algorithms is based on the shared-nothing model. Students may be asked to study how the algorithms can be improved if shared-memory machines are used instead.

Exercises

18.9 For each of the three partitioning techniques, namely round-robin, hash partitioning, and range partitioning, give an example of a query for which that partitioning technique would provide the fastest response.

Answer:

Round robin partitioning:

When relations are large and queries read entire relations, round-robin gives good speed-up and fast response time.

Hash partitioning

For point queries on the partitioning attributes, this gives the fastest response, as each disk can process a differnt query simultaneously. If the hash partitioning is uniform, entire relation scans can be performed efficiently.

Range partitioning For range queries on the partitioning attributes, which access a few tuples, range partitioning gives the fastest response.

18.10 What factors could result in skew when a relation is partitioned on one of its attributes by:

a. Hash partitioning?

b. Range partitioning?

In each case, what can be done to reduce the skew?

Answer:

a. Hash-partitioning:
Too many records with the same value for the hashing attribute, or a poorly chosen hash function without the properties of randomness and uniformity, can result in a skewed partition. To improve the situation, we should experiment with better hashing functions for that relation.

b. Range-partitioning:
Non-uniform distribution of values for the partitioning attribute (including duplicate values for the partitioning attribute) which are not taken into account by a bad partitioning vector is the main reason for skewed partitions. Sorting the relation on the partitioning attribute and then dividing it into n ranges with equal number of tuples per range will give a good partitioning vector with very low skew.

18.11 Give an example of a join that is not a simple equi-join for which partitioned parallelism can be used. What attributes should be used for partitioning?

Answer: We give two examples of such joins.

a. $r \bowtie_{(r.A=s.B) \wedge (r.A<s.C)} s$
Here we have an equi-join condition which can be executed first, and the extra conditions can be checked independently on each tuple in the join result. Partitioned parallelism is useful to execute the equi-join,

b. $r \bowtie_{(r.A \geq (\lfloor s.B/20 \rfloor) * 20) \wedge (r.A < ((\lfloor s.B/20 \rfloor)+1) * 20)} s$
This is a query in which an r tuple and an s tuple join with each other if they fall into the same range of values. Hence partitioned parallelism applies naturally to this scenario, even though the join is not an equi-join.

For both the queries, r should be partitioned on attribute A and s on attribute B. For the second query, the partitioning of s should actually be done on $(\lfloor s.B/20 \rfloor) * 20$.

18.12 Describe a good way to parallelize each of the following:

a. The difference operation

b. Aggregation by the **count** operation

c. Aggregation by the **count distinct** operation

d. Aggregation by the **avg** operation

e. Left outer join, if the join condition involves only equality

f. Left outer join, if the join condition involves comparisons other than equality

g. Full outer join, if the join condition involves comparisons other than equality

Answer:

a. We can parallelize the difference operation by partitioning the relations on all the attributes, and then computing differences locally at each processor. As in aggregation, the cost of transferring tuples during partitioning can be reduced by partially computing differences at each processor, before partitioning.

b. Let us refer to the group-by attribute as attribute A, and the attribute on which the aggregation function operates, as attribute B. **count** is performed just like **sum** (mentioned in the book) except that, a count of the number of values of attribute B for each value of attribute A) is transferred to the correct destination processor, instead of a sum. After partitioning, the partial counts from all the processors are added up locally at each processor to get the final result.

c. For this, partial counts cannot be computed locally before partitioning. Each processor instead transfers all unique B values for each A value to the correct destination processor. After partitioning, each processor locally counts the number of unique tuples for each value of A, and then outputs the final result.

d. This can again be implemented like **sum**, except that for each value of A, a **sum** of the B values as well as a **count** of the number of tuples in the group, is transferred during partitioning. Then each processor outputs its local result, by dividing the total sum by total number of tuples for each A value assigned to its partition.

e. This can be performed just like partitioned natural join. After partitioning, each processor computes the left outer join locally using any of the strategies of Chapter 12.

f. The left outer join can be computed using an extension of the Fragment-and-Replicate scheme to compute non equi-joins. Consider $r \supset\!\bowtie s$. The relations are partitioned, and $r \bowtie s$ is computed at

each site. We also collect tuples from r that did not match any tuples from s; call the set of these dangling tuples at site i as d_i. After the above step is done at each site, for each fragment of r, we take the intersection of the d_i's from every processor in which the fragment of r was replicated. The intersections give the real set of dangling tuples; these tuples are padded with nulls and added to the result. The intersections themselves, followed by addition of padded tuples to the result, can be done in parallel by partitioning.

g. The algorithm is basically the same as above, except that when combining results, the processing of dangling tuples must done for both relations.

18.13 Describe the benefits and drawbacks of pipelined parallelism.

Answer:

- **Benefits:** No need to write intermediate relations to disk only to read them back immediately.
- **Drawbacks:**
  a. Cannot take advantage of high degrees of parallelism, as typical queries do not have large number of operations.
  b. Not possible to pipeline operators which need to look at all the input before producing any output.
  c. Since each operation executes on a single processor, the most expensive ones take a long time to finish. Thus speed-up will be low despite the use of parallelism.

18.14 Suppose you wish to handle a workload consisting of a large number of small transactions by using shared-nothing parallelism.

a. Is intraquery parallelism required in such a situation? If not, why, and what form of parallelism is appropriate?

b. What form of skew would be of significance with such a workload?

c. Suppose most transactions accessed one *account* record, which includes an *account_type* attribute, and an associated *account_type_master* record, which provides information about the account type. How would you partition and/or replicate data to speed up transactions? You may assume that the *account_type_master* relation is rarely updated.

Answer:

a. Intraquery parallelism is probably not appropriate for this situation. Since each individual transaction is small, the overhead of parallelizing each query may exceed the potential benefits. Interquery parallelism would be a better choice, allowing many transactions to run in parallel.

b. Partition skew can be a performance issue in this type of system, especially with the use of shared-nothing parallelism. A load imbalance amongst the processors of the distributed system can significant reduce the speedup gained by parallel execution. For example, if all transactions happen to involve only the data in a single partition, the processors not associated with that partition will not be used at all.

c. Since *account_type_master* is rarely updated, it can be replicated in entirety across all nodes. If the *account* relation is updated frequently and accesses are well-distributed, it should be partitioned across nodes.

18.15 The attribute on which a relation is partitioned can have a significant impact on the cost of a query.

a. Given a workload of SQL queries on a single relation, what attributes would be candidates for partitioning?

b. How would you choose between the alternative partitioning techniques, based on the workload?

c. Is it possible to partition a relation on more than one attribute? Explain your answer.

Answer:

a. The candidate attributes would be

i. Attributes on which one or more queries has a selection condition. The corresponding selection condition can then be evaluated at a single processor, instead of being evaluated at all processors.

ii. Attributes involved in join conditions. If such an attribute is used for partitioning, it is possible to perform the join without repartitioning the relation. This effect is particularly beneficial for very large relations, for which repartitioning can be very expensive.

iii. Attributes involved in group-by clauses; similar to joins, it is possible to perform aggregation without repartitioning the corresponding relation.

b. A cost-based approach works best in choosing between alternatives. In this approach, candidate partitioning choices are generated, and for each candidate the cost of executing all the queries/updates in a workload is estimated. The choice leading to the least cost is picked. One issue is that the number of candidate choices is generally very large. Algorithms and heuristics designed to limit the number of candidates for which costs need to be estimated are widely used in practice.

Another issue is that the workload may have a very large number of queries/updates. Techniques to reduce this number include the following (a) combining repeated occurrences of a query that only differ in constants, replacing them by one parametrized query along with a count of number of occurrences and (b) dropping queries which are very cheap anyway, or not likely to be affected by the partitioning choice.

c. It is possible to partition a relation on more than one attribute, in two ways. One is to involve multiple attributes in a single composite partitioning key. The other way is to keep more than one copy of the same relation, partitioned in different ways. The latter approach is increases update costs, but can speed up some queries significantly.