Parallel Data Placement

Sung-Soo Kim
Data Management Research Section
Electronics and Telecommunications Research Institute
128 Gajeong-ro, Yuseong-gu
Daejeon, South Korea

sungsoo@etri.re.kr

ABSTRACT

In this technical report, we describe the techniques for data placement in the parallel database systems. Many data-intensive applications require support for very large databases (e.g., hundreds of terabytes or petabytes). Examples of such applications are e-commerce, data warehousing, and data mining. Very large databases are typically accessed through high numbers of concurrent transactions (e.g., performing on-line orders on an electronic store) or complex queries (e.g., decision-support queries). The first kind of access is representative of On-Line Transaction Processing (OLTP) applications while the second is representative of On-Line Analytical Processing (OLAP) applications. Supporting very large databases efficiently for either OLTP or OLAP can be addressed by combining parallel computing and distributed database management.

1. INTRODUCTION

In this technical report, we assume a *shared-nothing architecture* because it is the most general case and its implementation techniques also apply, sometimes in a simplified form, to other architectures. Data placement in a parallel database system exhibits similarities with data fragmentation in distributed databases [1]. An obvious similarity is that fragmentation can be used to increase parallelism. In what follows, we use the terms *partitioning* and *partition* instead of horizontal fragmentation and horizontal fragment, respectively, to contrast with the alternative strategy, which consists of *clustering* a relation at a single node. The term *declustering* is sometimes used to mean partitioning. Vertical fragmentation can also be used to increase parallelism and load balancing much as in distributed databases. Another similarity is that since data are much larger than programs, execution should occur, as much as possible, where the data reside.

However, there are two important differences with the distributed database approach. First, there is no need to maximize local processing (at each node) since users are not associated with particular nodes. Second, load balancing is much more difficult to achieve in the presence of a large number of nodes. The main problem is to avoid resource contention, which may result in the entire system thrashing (e.g., one node ends up doing all the work while the others remain idle). Since programs are executed where the data reside, data placement is a critical performance issue.

2. DATA PLACEMENT

Data placement must be done so as to maximize system performance, which can be measured by combining the total amount of work done by the system and the response time of individual queries. We have seen that maximizing response time (through intra-query parallelism) results in increased total work due to communication overhead. For the same reason, inter-query parallelism results in increased total work. On the other hand, clustering all the data necessary to a program minimizes communication and thus the total work done by the system in executing that program. In terms of data placement, we have the following trade-off: maximizing response time or inter-query parallelism leads to partitioning, whereas minimizing the total amount of work leads to clustering. This problem is addressed in distributed databases in a rather static manner. The database administrator is in charge of periodically examining fragment access frequencies, and when necessary, moving and reorganizing fragments. An alternative solution to data placement is full partitioning, whereby each relation is horizontally fragmented across all the nodes in the system. There are three basic strategies for data partitioning: round-robin, hash, and range partitioning (Figure 1).

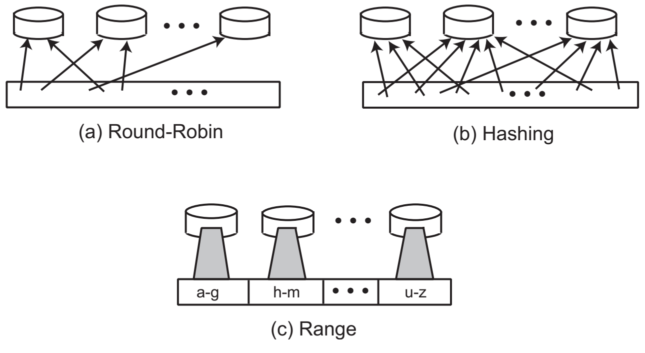

Figure 1: Different Partitioning Schemes.

1. *Round-robin partitioning* is the simplest strategy, it ensures uniform data distribution. With n partitions, the i-th tuple in insertion order is assigned to partition (i mod n). This strategy enables the sequential access to a relation to be done in parallel. However, the direct access to individual tuples, based on a predicate, requires accessing the entire relation.

2. *Hash partitioning* applies a hash function to some attribute that yields the partition number. This strategy allows exact-match queries on the selection attribute to be processed by exactly one node and all other queries to be processed by all the nodes in parallel.

3. *Range partitioning* distributes tuples based on the value intervals (ranges) of some attribute. In addition to supporting exact-match queries (as in hashing), it is well-suited for range queries. For instance, a query with a predicate A between A_1 and A_2 may be processed by the only node(s) containing tuples whose A value is in range [A_1,A_2]. However, range partitioning can result in high variation in partition size.

Compared to clustering relations on a single (possibly very large) disk, full partitioning yields better performance. Although full partitioning has obvious performance advantages, highly parallel execution might cause a serious performance overhead for complex queries involving joins. Furthermore, full partitioning is not appropriate for small relations that span a few disk blocks. These drawbacks suggest that a compromise between clustering and full partitioning (i.e., *variable partitioning*), needs to be found.

A solution is to do data placement by variable partitioning. The degree of partitioning, i.e., the number of nodes over which a relation is fragmented, is a function of the size and access frequency of the relation. This strategy is much more involved than either clustering or full partitioning because changes in data distribution may result in reorganization. For example, a relation initially placed across eight nodes may have its cardinality doubled by subsequent insertions, in which case it should be placed across 16 nodes.

In a highly parallel system with variable partitioning, periodic reorganizations for load balancing are essential and should be frequent unless the workload is fairly static and experiences only a few updates. Such reorganizations should remain transparent to compiled programs that run on the database server. In particular, programs should not be recompiled because of reorganization. Therefore, the compiled programs should remain independent of data location, which may change rapidly. Such independence can be achieved if the run-time system supports associative access to distributed data. This is different from a distributed DBMS, where associative access is achieved at compile time by the query processor using the data directory.

One solution to associative access is to have a global index mechanism replicated on each node. The global index indicates the placement of a relation onto a set of nodes. Conceptually, the global index is a two-level index with a major clustering on the relation name and a minor clustering on some attribute of the relation. This global index supports variable partitioning, where each relation has a different degree of partitioning. The index structure can be based on hashing or on a B-tree like organization. In both cases, exact match queries can be processed efficiently with a single node access. However, with hashing, range queries are processed by accessing all the nodes that contain data from the r queried elation. Using a B-tree index (usually much larger than a hashed index) enables more efficient processing of range queries, where only the nodes containing data in the specified range are accessed.

3. GLOBAL AND LOCAL INDEXES

Figure 2 provides an example of a global index and a local index for relation EMP(ENO, ENAME, DEPT, TITLE) of the engineering database example we have been using in this book.

Suppose that we want to locate the elements in relation EMP with ENO value E50. The first-level index on set name maps the name EMP onto the index on attribute ENO for relation EMP. Then the second-level index further maps the cluster value E50 onto node number j. A local index within each node is also necessary to map a relation onto a set of disk pages within the node. The local index has two levels, with a major clustering on relation name and a minor clustering on some attribute. The minor clustering attribute for the local index is the same as that for the global index. Thus *associative routing* is improved from one node to another based on (relation name, cluster value). This local index further maps the cluster value E5 onto page number 91.

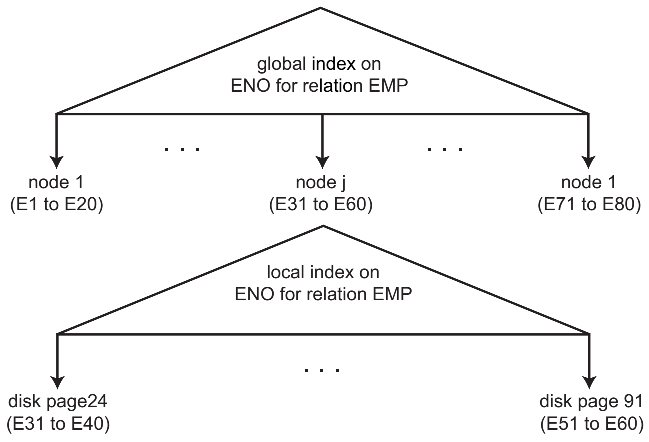

Figure 2: Example of Global and Local Indexes.

Experimental results for variable partitioning of a workload consisting of a mix of short transactions (debit-credit like) and complex ones indicate that as partitioning is increased, throughput continues to increase for short transactions. However, for complex transactions involving several large joins, further partitioning reduces throughput because of communication overhead.

A serious problem in data placement is dealing with skewed data distributions that may lead to non-uniform partitioning and hurt load balancing. Range partitioning is more sensitive to skew than either round-robin or hash partitioning. A solution is to treat non-uniform partitions appropriately, e.g., by further fragmenting large partitions. The separation between logical and physical nodes is also useful since a logical node may correspond to several physical nodes.

4. INTERLEAVED PARTITIONING

A final complicating factor is data replication for high availability. The simple solution is to maintain two copies of the same data, a primary and a backup copy, on two separate nodes. This is the mirrored disks architecture promoted by many computer manufacturers. However, in case of a node failure, the load of the node with the copy may double, thereby hurting load balancing. To avoid this problem, several high-availability data replication strategies have been proposed for parallel database systems [Hsiao and DeWitt, 1991]. An interesting solution is Teradatas interleaved partitioning that further partitions the backup copy on a number of nodes. Figure 3 illustrates the interleaved partitioning of relation R over four nodes, where each primary copy of a partition, e.g., R1, is futher divided in three partitions, e.g., r11, r12, and r13, each at a different backup node. In failure mode, the load of the primary copy gets balanced among the backup copy nodes. But if two nodes fail, then the relation cannot be accessed thereby hurting availability. Reconstructing the primary copy from its separate backup copies may be costly. In normal mode, maintaining copy consistency may also be costly.

Node	1	2	3	4
Primary copy	R_1	R_2	R_3	R_4
Backup copy		$r_{1.1}$	$r_{1.2}$	$r_{1.3}$
	$r_{2.3}$		$r_{2.1}$	$r_{2.2}$
	$r_{3.2}$	$r_{3.3}$		$r_{3.1}$

Figure 3: Example of Interleaved Partitioning.

5. CHAINED PARTITIONING

A better solution is Gammas chained partitioning, which stores the primary and backup copy on two adjacent nodes (Figure 4). The main idea is that the probability that two adjacent nodes fail is much lower than the probability that any two nodes fail. In failure mode, the load of the failed node and the backup nodes are balanced among all remaining nodes by using both primary and backup copy nodes. In addition, maintaining copy consistency is cheaper. An open issue is how to perform data placement taking into account data replication. Similar to the fragment allocation in distributed databases, this should be considered an optimization problem.

Node	1	2	3	4
Primary copy	R_1	R_2	R_3	R_4
Backup copy	r_4	r_1	r_2	r_3

Figure 4: Example of Chained Partitioning.

6. COCLUSION

Parallel database systems strive to exploit multiprocessor architectures using softwareoriented solutions for data management. Their promises are high-performance, high-availability, and extensibility with a good cost/performance ratio. Furthermore, parallelism is the only viable solution for supporting very large databases within a single system.

Parallel data management techniques extend distributed database techniques in order to obtain high-performance, high-availability, and extensibility. Essentially, the solutions for transaction management, i.e., distributed concurrency control, reliability, atomicity, and replication can be reused. However, the critical issues for such architectures are data placement, parallel query execution, parallel data processing, parallel query optimization and load balancing. The solutions to these issues are more involved than in distributed DBMS because the number of nodes may be much higher. Furthermore, parallel data management techniques use different assumptions such as fast interconnect and homogeneous nodes that provide more opportunities for optimization.

7. REFERENCES

[1] M. T. zsu and P. Valduriez. *Principles of Distributed Database Systems*. Springer Publishing Company, Incorporated, 3rd edition, 2011.