4K－1

Load Balancing in Parallel Database Systems

Jiahong Wang[†], Masatoshi Miyazaki[†], Hisao Kameda[‡], and Jie Li[‡]
[†]Iwate Prefectural University, [‡]University of Tsukuba

1 Problem

Many parallel database systems have the shared-nothing architecture (Fig. 1). Examples are IBM DB2 Parallel Edition [4] and Sybase MPP [5]. In a shared-nothing system, there are multiple processors connected by an interconnection network. Each processor accesses its private memory. Database is divided into several smaller partitions, and these partitions are distributed across disk drives attached directly to each processor. Retrieval and update requests are decomposed into sub-requests and executed in parallel among the applicable nodes.

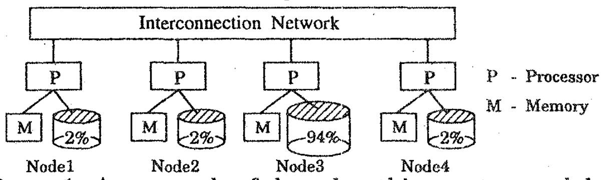

Figure 1: An example of shared-nothing system and data skew. In this system, there exists 2% of the data at nodes 1, 2, and 4 respectively, and 94% of the data at node 3. Node 3 becomes a bottleneck of the system.

When a shared-nothing system is initially built, data can be evenly distributed over the nodes of the system. After a period of use, however, as a result of insert, delete, and update activities, some partitions grow and others shrink, and data skew occurs (Fig. 1). Numerous previous studies have shown that data skew degrades system performance significantly [1]. In order to sustain system performance, data has to be moved across nodes so as to keep system data load balanced. In addition, when a new node is added to the system, system data load has to be rebalanced too. Moving data, including dropping and rebuilding indexes, however, generally requires taking a database off line for a long time, which can be unacceptable for a highly available system (a system to be fully available 24-hours-per-day and 7-days-per-week). This paper addresses a very practical subject: moving data *on line*, i.e., redistributing data concurrently with users' reading and writing of the data.

2 Solution

System model: It is assumed that each node has the same software architecture (e.g., file organization). This is the general case for shared-nothing systems.

The relational database system is considered. Each relation table is divided into several partitions. One partition can not be allocated to more than one node, but more partitions can exist at one node. A partition and all its indexes are organized as a *partition object*, which is fully self-describing so that it can be moved from node to node. By the movement of partition objects, the data load on the system remains balanced. Since a partition and its indexes are moved as a whole and each node has the same software system architecture, no index rebuilding is required for the moved data.

The storage of a partition is divided into units called partition pages. Hierarchical index structure such as B^+-tree is assumed, and the storage of an index is divided into units called index pages.

On-line data redistribution: Assume that a partition object P with a single index I is required to be moved from node S to D. This is performed by the following two phases. In the meantime, P is available to users.

PHASE 1: Moving the index I: the following four concurrent actions occur:

1.Send D the index pages one by one until the end of I is reached. When the end of I is reached, send system coordinator a message indicating that I has been sent.

2.For each dirty index page that is to be forced into the disk from system buffer pool of S, if this page has been sent to D, then send D again a copy of this page along with the corresponding page number.

3.Initialize an index file at D for the received index pages, and enter each page received into this index file, with the page number remaining unchanged.

4.Upon receiving the message, system coordinator switches to D the newly-arrived transactions that access P, and start the phase of moving the tuple data.

PHASE 2: Moving the tuple data: the following three concurrent actions occur:

1.Send D the relation pages one by one until the end of the relation file is reached. When a page is sent, check if there comes a page-requesting message. If there does, then send D the requested page.

2.Initialize a relation file at D for the received relation pages, and enter each page received into this relation file, with the page number remaining unchanged.

3.For each page access request for P at D, check if the page has been in D, if it does, then fetch it from D directly, else fetch it from S remotely by sending a page request to S and waiting for the requested page.

3 Performance Study

Experimental testbed: In order to evaluate the proposed approach, we built a functional shared-nothing system (Fig. 2). In the front-end node, there exists a workload generator for simulating a multi-user application environment, a redistributor for managing the data redistribution, and a client version of the Postgresql database management system [3]. In each data processing node there exists a modified server version of the Postgresql in which the proposed approach is embedded. Data is moved from node 1 (S) to node 2 (D).

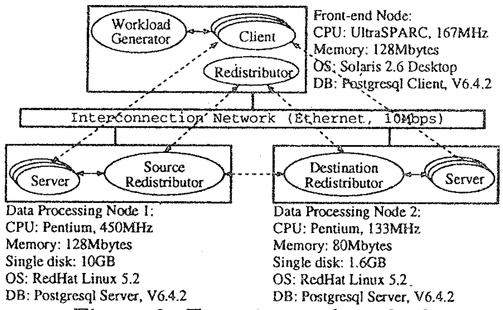

Figure 2: Experimental testbed.

The wisconsin benchmark relation [2] is used. Each tuple is 148 bytes long. One partition object (500000 tuples of data with a b-tree index of 10117120 bytes) of the relation is considered, which is initially placed on data processing node 1. Note that there was no point in having more than one relation or partition object in the experiment because only one partition object of a single relation is moved at a time.

The workload generator maintains MPL (MultiProgramming Level) clients. Each client starts a transaction, which is executed by the corresponding server. Each transaction accesses exactly one tuple at random. A transaction accesses one index to locate the page of the tuple, and then retrieves the page from disk and accesses the tuple. For read-only (RO) transactions, the content of the tuple is returned to the client. For read-write (R-W) transactions, the tuple is updated and written back into the disk. When a transaction is completed, the corresponding client exits, and a new client is generated immediately. The ratio of occurrence probability of RO to RW transactions is 0.7:0.3.

Performance results: One of the performance metrics is transaction throughput that is defined as the number of committed transactions per second. The other is a so-called normalized loss metric used to estimate the loss of the system in terms of the number of potential transactions that could not execute due to contention caused by the data redistribution activity: $loss = (T_{normal} - T_{redis}) * RT_{redis}$ and the normalized loss is the loss for the proposed approach divided by the loss for the off-line approach. Here T_{normal} is the average transaction throughput in the case that no data redistribution occurs, RT_{redis} is the time to complete the data redistribution, and T_{redis} is the average transaction throughput within RT_{redis}.

Figure 3 gives average transaction throughputs at different MPLs in the case that data redistribution is and is not performed respectively. Figure 4 gives results of the normalized loss for the proposed approach and the off-line approach. The line "Normalized loss in the case of off-line" says that, if we perform data redistribution off line, no transactions can be executed during data redistribution period, and the system losses 100% of the transactions. The line "Normalized loss" says that, if we perform data redistribution on line, 40% of performance improvement can be achieved.

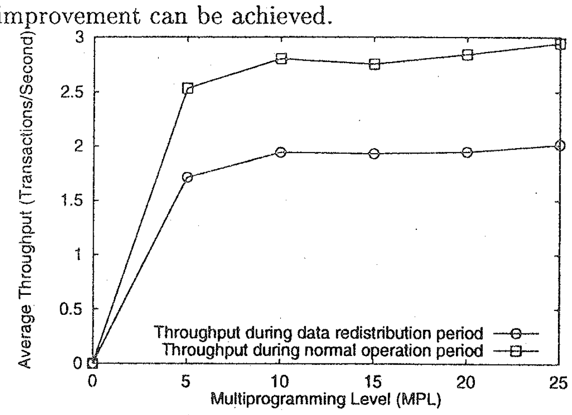

Figure 3: Average transaction throughput.

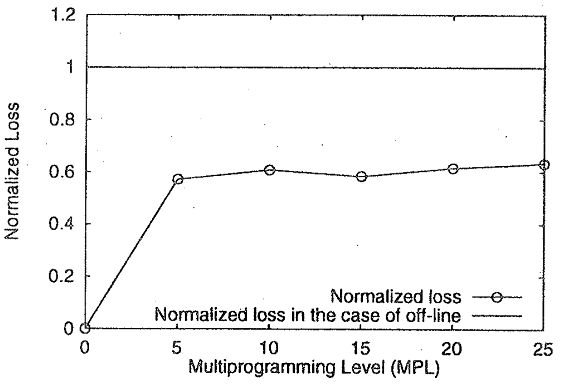

Figure 4: Normalized loss.

4 Conclusion

We have proposed an effective approach for rebalancing data load of the shared-nothing system, which allows concurrent read and write operations by transactions while data is rebalanced, so that a very large or highly available (24-hour) system need not go off line for data load rebalancing. Our goal is to increase concurrency, decrease overheads, and decrease time requirement as far as possible, which was achieved by the following: (1) When data is moved, no locks are acquired on both the data and its indexes. (2) No time-consuming log scan is required, which is necessary for the previous on-line data redistribution approaches. (3) The data and its indexes are just moved from the source node to the destination node (as if the ftp utility was used), and re-used there. Indexes at the source node can be dropped simply, and the conventional time-consuming index rebuilding at the destination node is not required. (4) While data and its indexes are moved, users can use the data normally, as if no data movement occurred at all.

References

[1] D.J. DeWitt and J. Gray, Parallel Database Systems: The Future of Database Processing or a Fad?, *Comm. ACM*, 35(6): 85-98, 1992.
[2] J. Gray, The Benchmark Handbook for Database and Transaction Processing Systems, Ed., *Morgan Kaufmann Publishers, Palo Alto, CA*, (1993).
[3] M. Stonebraker and G. Kemnitz, The POSTGRES Next-Generation Database Management System, *Comm. of the ACM*, 34(10): 78-92, 1991.
[4] DB2 Parallel Edition V1.2 Parallel Technology, *IBM Corp., Apr. 1997*.
[5] Sybase Technical News, *Vol.6, No.9, Dec. 1997*.