Improving Performance of Parallel Database Systems by Reorganizing Database On-Line

Jiahong Wang, Masatoshi Miyazaki
Faculty of Software and Information Science
Iwate Prefectural University

Abstract

The performance of parallel database systems may be restricted due to data skew. Redistributing data is an effective approach to coping with data skew. However, during most typical types of redistribution in a typical parallel database system, the area being redistributed is unavailable (off line). Considering that a highly available (24-hour) system calls for the ability to redistribute data on line, in this paper, we address a practical subject: improving system performance by reorganizing database on-line.
Index Terms: Data redistribution, data skew, parallel databases, two-phase locking.

1 Introduction

In recent years, the transaction processing rates that need to be supported have been growing beyond those that can be supported by the conventional (mainframe) database system. Some applications thereby become infeasible with the conventional database system because the response time is poor. Especially, with the exponential growth in the use of such application types as online transaction processing, online analytical processing, decision support, data warehousing, and data mining, it is often required for a database system to provide very high transaction throughput with rapid response times, uninterrupted operations with outstanding system reliability, a way to grow incrementally, the ability to process very large databases, and the analysis of massive amounts of information. These requirements cannot be satisfied with the conventional system, and call for the use of parallel database systems.

A well-known architecture for parallel database processing is the shared-nothing architecture (Fig. 1). Examples of related commercial products are IBM DB2 Parallel Edition [3], Sybase MPP [5], and Tandem NonStop SQL [7]. In a shared-nothing system, there are multiple processors connected by an interconnection network. Each processor accesses its private memory. Data can be partitioned or divided into several smaller so-called partitions. Partitions are distributed across disk drives attached directly to each processor. Retrieval and update requests are decomposed automatically into sub-requests and executed in parallel among the applicable nodes. That is, each processor actually controls its own data partition and works independently on its assigned piece of the sub-request.

Shared-nothing systems have great potential to serve the ever-increasing demands for high performance data processing. This potential, however, may not be reached if the data is not spread evenly over the nodes of the system, i.e. if data skew occurs. For example, node 3 in Fig. 1 becomes the bottleneck of the system. At the initial placement time, data can be easily spread over the nodes of the system evenly. For a created online database, however, as a result of the insert, delete, and update activity, some partitions grow and others shrink. As a result, data skew occurs. In fact, there is no one data distribution scheme that is optimal for the lifetime of a database system. In order to sustain the performance of the system, data has to be frequently moved across nodes to keep data distributed evenly. In addition, when a new node is added to the system, we have also to redistribute the data.

Data redistribution, however, typically requires taking a database off line, which can be unacceptable for a highly available system (a system to be fully available 24-hours-per-day, 7-days-per-week). This paper addresses a very practical subject, i.e., redistributing data on line (concurrently with users' reading and writing of data in the database). An approach for online data redistribution is proposed, and its performance is studied.

2 Related Work

One straightforward approach to coping with data skew is perhaps to re-balance the data when bulk loading new data into the database, or by bulk moving some data out of the database first, and then, bulk loading them back again, as Sybase MPP does [5]. This approach, however, is basically an off-line approach, and requires system managers to determine how to balance the data and perform the actual operation.

Another approach is to lock the data (a database table) to be redistributed in exclusive mode, invalidate all transactions that involve the data, and then, perform data redistribution, as IBM DB2 Parallel Edition does [3]. This approach, however, is extremely expensive since users cannot perform any operations, e.g., update and query, on the data being redistributed for a long time.

Different from the above two approaches, Tandem NonStop SQL redistributes data on line by moving a partition of data from one disk to another [7]. The index modification, however, is not considered, and no performance results are provided.

Two techniques for on-line index modification in shared nothing parallel databases are proposed in [1]. These two techniques, however, do not address the issue of redistributing data in a partition that satisfy some condition. For example, consider that hash partitioning strategy [3, 4, 6] is used to assign each tuple of a database table to a physical partition according to a calculation based on the partitioning key. If data skew occurs, we have to change the hash function to rebalance data. The change of the hash function incurs that some tuples have to be moved from one partition to others. In this case, it is not the partition but some tuples in it that need to be moved.

We adopted a similar strategy to that presented in [7] and proposed in [1]. Compared with [7], we focus on on-line index modification and its performance. Compared with [1], we focus on redistributing a batch of tuples in a partition of a database table that satisfy some condition. Especially, the index reorganization in our approach takes the characteristics of the data into consideration.

3 System Model

Consider a shared-nothing parallel database system shown in Fig. 1. Users' interaction with the

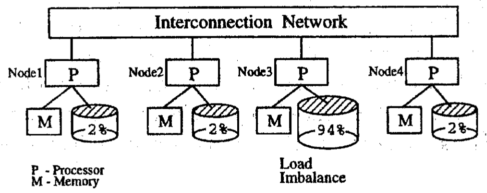

Figure 1: Shared-nothing architecture and data skew

system is through a coordinator. The coordinator runs on the same node as the application does, or in the case of a remote application, the node to which that application is connected. All nodes can be used as a coordinator node.

The key to the performance of a shared-nothing system is its data partitioning. Partitioning divides the data up physically among the nodes so that it can be accessed and managed separately, minimizing contention. Tuples in a table are assigned to a partition by key value, which can be one or more columns in the table. A global catalog provides system-wide schema information, ensuring that each request is only sent to the partitions with data involved. Three commonly used partitioning methods are as follow: hash partitioning that assigns each record to a physical partition according to a random encoding of the partitioning key; range partitioning that allows related tuples of information to be stored together in the same partition; and schema partitioning that assigns all tuples of a table to a single partition.

At the coordinator node, an individual request for information (SQL query) is analyzed using partitioning information from the global catalog. Parallel queries are generated and sent to each node involved and executed there in parallel.

When a node is over-loaded, tuples in a partition at that node (called source node) are required to be moved to other nodes (called destination nodes), so as to balance the load of the nodes. The data redistribution is performed by the cooperation of the tasks at both source node and destination nodes (called source redistributor and destination redistributor respectively). Source redistributor reads tuples from disks at source node, packages the tuples, and send destination redistributor the packets. The destination redistributor stores the received tuples into disks at destination nodes and reorganizes the indexes.

4 Data Redistribution

For the sake of simplicity, in the following description, we assume that the tuples that satisfy the redistribution condition (e.g., redistributing the tuples with ages larger than 30) are required to be moved from a *source partition* at the *source node* to a *destination partition* at only one *destination node*. Note that the proposed approach can also handle the case of one source node and multiple destination nodes.

The proposed approach consists of two parts: the part for the source redistributor (the process that performs the redistribution at source node) and the part for the destination.

Part 11: for the source node (during redistribution)

1. Record the current LSN (Log Sequence Number) for the log at source node, let it be LSN_1.
2. Scan sequentially each file page of the source partition to unload the tuples that satisfy the redistribution condition. If a tuple is found in a page, the corresponding TID (Tuple IDentifier) and the LSN in the tuple are obtained, and a tuple <TID, LSN, $Flag_{index1}$, $Flag_{index2}$,...> is inserted into a so-called *mask table*, with all the $Flag_{index}$s being set to be true. Unloaded tuples are appended with the corresponding TIDs and LSNs, and the resulting tuples are sent to the destination node.
3. Scan sequentially the source node log since LSN_1 was recorded, extract the log data that concern the tuples that satisfy the redistribution condition, and send the destination node these log data to bring the unloaded data at destination node up to date (i.e., to perform the first pass of compensation).
4. Record again the current LSN for the log at source node, let it be LSN_2.
5. Block users' new data requests that will update the source partition and make sure all outstanding ones to be completed. Force the modified pages of the source partition in database buffer into disks. Note that users continue to be able to read the source partition.
6. Perform step 3 with the log data since LSN_2 was recorded.
7. Block all user access to both the source and the destination partitions, and make sure all outstanding ones to be completed. This is done by requesting eXclusive locks on both of these two partitions. When both of the X locks are granted, modify the higher level of meta data (i.e., the partitioning method for the related table) to switch users' future access to redistributed data to the destination node.
8. Resume users' access to source partition.

Part 12: for the source node (after redistribution)

for each data access request to the source partition do the following:

```
if it is an access via an index (assume it is index $x$) with a specified key
then get the corresponding index page that contains the key
    if the index page has been processed to reflect the reorganization
    then perform the normal processing.
        if the index page is modified,
        then erase the mark on the index page to tell that it should be re-processed.
    else get TID of the tuple to be accessed
        if this TID is in mask table,
        then reject the access request,
            delete the corresponding item in the index page, and
            set $\mathrm{Flag}_x$ of the tuple in mask table to false.
            if all the $\mathrm{Flag}_{\mathrm{index}}$s of the tuple become false,
            then the tuple can be deleted.
        delete all such index items in the index page
        that have TIDs belonging to the mask table.
        put a mark on the index page telling that it has been processed.
else use the mask table to examine every data access request and
    to reject the request to the data with TID being in the mask table.
```

Note that in order to shorten the time of data redistribution, we resume users' access to source partition as soon as data has been copied to the destination node completely, and leave the copied data and the corresponding indexes unprocessed. The copied data and the corresponding indexes will be deleted on-line later (more research is needed to examine the various issues involved). It is thereby possible that transactions will access the index entries corresponding to the copied data, and the copied data themselves. There are two methods that can be considered to cope with this problem. The one is to lock all the copied data exclusively before resuming users' access to source partition. The other is as stated above in Part12 of the source redistributor. That is, in order to prevent the transactions from accessing the copied data, an in-memory mask table is maintained regarding the validity and availability of the copied data. A transaction that needs data from the mask table can abandon its attempts to read/modify the data. Compared with the second method, the first one is straightforward and can be easily implemented. It is illustrated (see section 5) that, however, the second one can provide far higher transaction throughput.

Part 21: for the destination node (during redistribution)

Without losing generality, it is assumed that there exists only one index (named index i) for the database table to be redistributed. Set variable j to 1, and initialize a buffer named Buf_i and disk file File_i and File_{ij}). For each inbound message do the following:

Concurrent task one:

1. Extract the key, TID, and LSN parts in each tuple in the message, construct the mapping record $<\mathrm{TID}_D$, TID, LSN, key$>$ and append it to Buf_i. See the next step about TID_D.
2. Write all the tuples in the message into the corresponding database table. While writing the tuples, get the current TID (named TID_D), and complete the mapping table with TID_Ds.
3. If Buf_i becomes full, write it into File_{ij}, increase j, and initialize new disk file File_{ij}.

Concurrent task two:

1. Use mapping records in Buf_i to reorganize the index.
2. Move the processed mapping records into File_i.

Part 22: for the destination node (after redistribution)

- Read all the File$_{ij}$s in parallel, and
- Reorganize the index in a normal bulk method [1, 7]

Note that the copy of data at the destination node is not available for transactions until after completion of the Pare 21. It is possible that transactions at the destination node will access the index entries corresponding to the copied data and the copied data themselves. As in the case of source redistributor, here an in-memory mask table is also needed. A transaction that needs data from the mask table can abandon its attempts to read/modify the data.

In concurrent task two, the mapping records can be used to reorganize the index in three ways. The first is a so-called FIFO way, where mapping records are used by the order with which they enter the mapping table. FIFO has been proved to be ineffective [1]. The second is a so-called bulk way [1], where Buf$_i$ is first sorted and then inserted into the index. The third is what we are considering. We think that in most of cases, key values of a table generally obey some distribution, and have a mean. By simulation we found that if we sort Buf$_i$ and pick up first the mapping records with keys approximating to the mean, the amount of data wrote into File$_{ij}$ will be decreased largely, and accordingly, the time cost of Part 21 can be shorted greatly.

5 Performance Study

In this section we give the performance results of our simulation study of the proposed approach. Considering that the proposed approach consists of two components (i.e., the source and the destination redistributors), and speeding up any of them will speed up the whole redistribution work, we studied the performance of each of them separately, so that the performance of the proposed approach can be studied more accurately. This paper only reports the performance results of the source redistributor. The performance study of destination redistributor is being done.

The simulation parameters are set as follows. There are four nodes and two relations in the system. A relation consists of 4000 pages of data. Relation 1 is evenly-distributed over all the four nodes (1000 pages per node). In the case of no data skew, relation 2 is also evenly-distributed over all the four nodes. In the case of data-skew, 1500, 500, 1000, and 1000 pages are located at node 1, 2, 3, and 4 respectively.

Figure 2 shows transaction throughputs of the simulated system in the following three kinds of

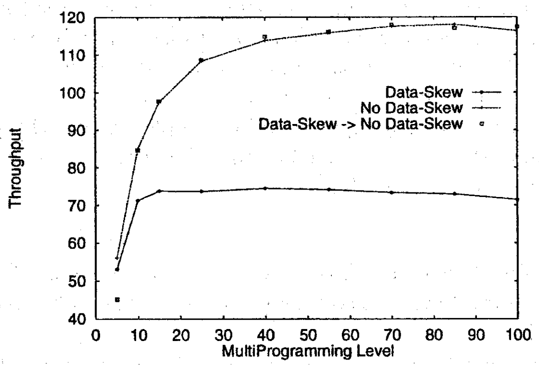

Figure 2: Throughputs (1) in the case of data skew, (2) in the case of no data skew, (3) during and after data redistribution. Data is redistributed at mpl of 5, and data skew is eliminated thereafter.

simulation environment respectively: (1) there exists data skew; (2) there does not exist data skew; (3) data redistribution is performed to eliminate data skew. This figure tells that data redistribution would degrade throughput. This is because during redistribution, resource contention for disks and network communication is increased largely. After data redistribution, however, throughput is well increased due to the eliminated data skew.

As stated in section 4, in order to shorten the time of data redistribution, users' access to source partition is resumed as soon as data has been copied to the destination node, and leave the copied data and the corresponding indexes undeleted. It is thereby possible that transactions will access these data and index entries. There are two methods to cope with this problem. The one is to lock all the copied data exclusively before resuming users' access to source partition. The other is to use an in-memory mask table to prevent the transactions from accessing the copied data. From figure 3 we see that the second method (see the results identified with "No Locking") can provide far higher transaction throughput than the first one (see the results identified with "Locking") due to the reduced data contention. We thereby adopted the second method in our approach.

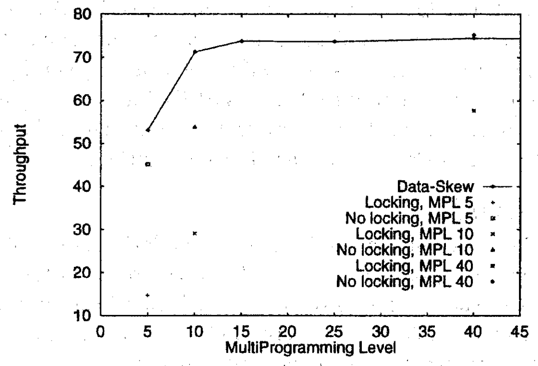

Figure 3: Throughputs in the case of data skew, and throughputs during data redistribution in the case of (1) locking and (2) no locking. Data is redistributed at mpl of 5, 10 and 40 respectively.

6 Conclusion

The performance of parallel database systems may be restricted due to the unbalanced data distribution among nodes. A method for coping with this problem is to redistribute data so as to rebalance data load among nodes. In most of the current redistribution methods for parallel database systems, however, the area being redistributed is unavailable (off line). In this paper, we have presented an approach to solving the important problem of on-line data redistribution for parallel database systems. The alternative schemes that can be considered have also been examined, and the performance has been studied by simulation.

This is an ongoing research project, and we are implementing the proposed approach on a physical database system (the Berkeley DB Package, Version 2.4.14) to examine its alternatives and the performance implication more deeply.

References

[1] K.J. Achyutuni, E. Omiecinski and S.B. Navathe, Two Techniques for On-Line Index Modification in Shared Nothing Parallel Databases, in: *proc. the 1996 SIGMOD Conf., (Montreal, Canada, 1996)* 125-136.

[2] M. Stonebraker, The Case for Shared Nothing, *Database Eng. Bull.*, 1 (1986) 4-9.

[3] DB2 Parallel Edition V1.2 Parallel Technology, *IBM Corp., Apr. 1997.*

[4] *Sybase MPP for the IBM RISC System/6000 Scalable POWERparallel*

[5] Sybase Technical News, *Vol.6, No.9, Dec. 1997.*

[6] *Extended Parallel Option for Informix Dynamic Server: Technical Brief*

[7] J. Troisi, NonStop Availability and Database Configuration Operations, *Tandem Systems Review, 10(3): 18-23, July 1994.*