T2R2 東京科学大学 リサーチリポジトリ
Science Tokyo Research Repository

論文 / 著書情報
Article / Book Information

Title	Compound Treatment of Chained Declustered Replicas Using a Parallel Btree for High Scalability and Availability
Author	Min LUO, Akitsugu WATANABE, Haruo YOKOTA
Journal/Book name	Database and Expert Systems Applications, Springer LNCS., vol. 6262/2010, , pp. 49-63
発行日 / Issue date	2010, 8
DOI	http://dx.doi.org/10.1007/978-3-642-15251-1_4
権利情報 / Copyright	The original publication is available at www.springerlink.com.
Note	このファイルは著者（最終）版です。 This file is author (final) version.

Powered by T2R2 (Science Tokyo Research Repository)

Compound Treatment of Chained Declustered Replicas Using a Parallel Btree for High Scalability and Availability

Min LUO[†], Akitsugu WATANABE[†], Haruo YOKOTA[†,‡]

[†]Department of Computer Science, Tokyo Institute of Technology
2–12–1 Ookayama, Meguro-ku, Tokyo 152–8552, Japan
[‡]Global Scientific Information and Computing Center, Tokyo Institute of Technology
2–12–1 Ookayama, Meguro-ku, Tokyo 152–8550, Japan

Abstract. Scalability and availability are key features of parallel database systems. To realize scalability, many dynamic load-balancing methods with data placement and parallel index structures on shared-nothing parallel infrastructure have been proposed. Data migration with range-partitioned placement using a parallel Btree is one solution. The combination of range partitioning and chained declustered replicas provides high availability while preserving scalability. However, independent treatment of the primary and backup data in each node results in long failover times. We propose a novel method for compound treatment of chained declustered replicas using a parallel Btree, called the Fat-Btree. In the proposed method, the single Fat-Btree provides access paths to both primary and backup data in all processor elements, which greatly reduces failover time. Moreover, it enables dynamic load balancing without physical data migration, and improves memory space utilization for processing the index. Experiments using PostgreSQL on a 160-node PC cluster demonstrate the effect.

1 Introduction

The explosive growth of digital information, together with the high performance and availability requirements, has driven a continuing interest in the research on database systems in shared-nothing parallel environments in which replication plays an important role in availability and scalability with load balancing among the different processing nodes. However, existing replicated-database systems have a weakness in scaling up under frequent update requests because of the high costs of synchronizing the replicas. Well-known approaches for cloud applications in large data centers, such as PNUTS, Dynamo and BigTable, sacrifice strong consistency for scalability. However, they lose opportunities of using the high system throughput for applications requiring stricter consistency, because the replicas are not consistent most of the time in these approaches. Moreover, the advantage of higher availability gained from the replication may also be lost in the long run because of the inconsistency [26]. Therefore, this trade-off between scalability, availability and consistency has long been an obstacle in efficient replication techniques.

To attack this problem, efficient data access methods guaranteeing consistency between the replicas play an important role. The data access methods must also be capable of handling sophisticated failover and dynamic load-balancing for availability and scalability. Many parallel index structures have been introduced to access the data stored in a shared-nothing environment [16]. For instance, the Fat-Btree [24] and the aB$^+$-tree [13] are range-partition-based parallel indexes providing dynamic data allocation for high scalability. However, these parallel indexes are mainly focused on throughput or latency rather than availability.

Chained declustering data placement [9] adopts a low degree of replication to realize availability and scalability. From the availability point of view, only low degrees of replication are required. The range-partition-based index method is well suited to chained declustering. However, as far as we know, no valid indexing methods have been proposed to consider the use of replicas in chained declustering to enhance system availability and scalability.

We propose a database infrastructure for indexing range-partitioned data with a low degree of replication to achieve high scalability and availability without sacrificing data consistency. We first consider a parallel index structure on a range-partitioned chained replication database as a straightforward configuration for the infrastructure. We then propose a novel method of compound replica treatment utilizing the Fat-Btree index. It reduces the management cost of the index structure, balances load without data migration and enables shorter failover times. We also adopt the neighbor write-ahead log protocol (nWAL) [10] adapted to the proposed configuration, to reduce the synchronization cost between the replicas without data loss.

The key innovative points of this work are: a) it is the first proposal for managing both primary and backup within one directory structure; b) it has an efficient automatic load-balance algorithm for dynamic load skew without data migration; and c) it has an efficient failover algorithm for higher system availability without the cost of "promotion" backups. To the best of our knowledge, no previous work provides practical dynamic load-balancing or failover utilities in chained replication database systems, although they have long been claimed [8, 20]. We have evaluated our method on a PC cluster with 160 nodes. The experimental results demonstrate all the above-mentioned effects.

2 Background

We briefly review three technologies in shared-nothing parallel databases: data placement strategies, parallel indexing structures and synchronization methods.

2.1 Data Placement Strategies

Chained declustering [9] offers high availability, scalability and load balancing on shared-nothing parallel database systems [3]. Two physical copies of the relation are declustered by the same partitioning strategy, and the corresponding

fragments of these two copies are stored on different Processing Elements (PEs), so data on a failed node will not be completely lost during the failure.

Basically, there are three types of partitioning strategies in chained declustering: hash, round-robin and value-range. Hash partitioning is ineffective for range queries and does not scale up [20], round-robin partitioning produces no skew, but is ineffective for queries, while value-range partitioning can treat range queries efficiently, but has a risk of load skew after repeated updates. Thus, each partition strategy has limitations, and it is important to choose an appropriate strategy and provide solutions to overcome the limitations.

2.2 Parallel Indexing Structures

To meet the demands of handling large amounts of data, parallel indexing structures have been proposed to provide dynamic data management, high throughput and efficient load skew handling via the index nodes for these systems [17].

The Fat-Btree is a parallel Btree structure that was proposed to provide dynamic data management, high throughput and efficient skew handling [24]. In a Fat-Btree, the leaf pages of the parallel Btree are distributed among PEs. Each PE has a subtree of the whole Btree containing the root node and intermediate index nodes between the root node and leaf nodes allocated to that PE. In the Fat-Btree structure, there are multiple copies of index nodes close to the root node, but they have a relatively low update frequency; on the other hand, leaf nodes have a relatively high update frequency but are not duplicated. Thus, the nodes with higher update frequencies have lower synchronization overhead. Therefore, the maintenance cost is much lower than other parallel Btree structures, such as Copy-Whole-Btree and Single-Index-Btree [24]. Moreover, the Fat-Btree has a higher cache hit rate [24] and more efficient concurrency control protocols LCFB [25] than conventional parallel Btrees [17, 14, 19, 25].

2.3 Synchronization with nWAL

Write-ahead log (WAL) [7] is widely used in database systems to ensure atomicity and durability. However, transactions must be suspended to wait for the forced log-write to stable storage before the new version replaces the previous one. In replication databases, this inefficiency is solved by the neighbor WAL (nWAL) protocol [10], which transfers the log of the transaction on PE_i into the memory of its replica nodes PE_{i+1} through the network before the transaction commits. Moreover, nWAL is naturally suitable for maintaining data consistency in range-partitioned chained replication systems, because the backup can always be synchronized with the latest version of its primary by using the nWAL in the memory. Thus, no extra synchronization messages are required.

3 A Straightforward Configuration

We present a straightforward configuration for our distributed replication database with existing techniques.

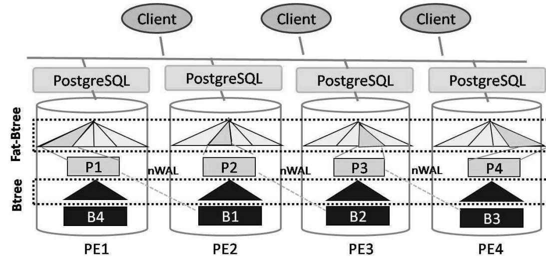

Fig. 1. Configuration of IndepIndexCDR

As mentioned in Section 1, the synchronization cost is the most serious obstacle to achieving high performance and scalability in replication databases. Considerable work has been done to reduce the costs [4, 6, 15, 11]. However, it is hard to achieve satisfactory results unless the number of replica copies is reduced. As we know, the chained replication system only maintains one replica for each piece of data, thus easily achieving higher throughput, availability and stronger consistency compared with other replication schemes [8, 20]. Therefore, we treat the chained replication scheme as the base configuration.

To provide a global access path for the primary data in a chained replication system, some parallel index structure is required. We adopt the Fat-Btree mainly because of its low cost in concurrency control [25] and high flexibility in the compound treatment discussed in the next section. Alternative indexes are discussed later in Section 6.

On the other hand, the backup parts may also be indexed to speed up synchronization in most distributed databases. Because they are not required to serve queries when no failure or skew happens, it is sufficient to maintain an independent Btree for the backup in each PE. Therefore, we adopt the independent Btree to index backups and name this configuration independent-index chained declustered replication *IndepIndexCDR*.

3.1 Implementation of IndepIndexCDR

We implemented IndepIndexCDR using PostgreSQL and the Fat-Btree. Each PE has an instance of PostgreSQL with a part of the Fat-Btree (a subFat-Btree) to access its primary. Following the chained declustering strategy, the backup is located in its primary's right-hand neighbor and it receives the nWAL from its primary synchronously, which may be applied to the backup asynchronously after the primary transaction commits. Figure 1 illustrates this configuration.

Because of space limitations, we omit the detailed data access process in IndepIndexCDR, which is almost the same as that in [24]. Note that for an update request, an nWAL message is sent to the right node before the result is returned to the user. The backup is then updated asynchronously by using the local Btree index on the backup.

3.2 Scalability in IndepIndexCDR

Because of the low degree of replication in IndepIndexCDR, the number of synchronization transactions is greatly reduced. Moreover, unlike the original chained replicated structure in [8, 20], in which only the head and tail PEs accepted query and update requests, IndepIndexCDR can access any data item in primary storage from any PE by the Fat-Btree; thus, the overhead for handling client requests is dispersed over all the PEs. Therefore, IndepIndexCDR should have good scalability. We will report a quantitative experimental evaluation of this later in Section 5.3.

3.3 Availability in IndepIndexCDR

We are not concerned with failure detection techniques here; we assume that a failure can be detected, and only consider the failover efficiency.

Almost all the replication databases claim failover capabilities, but the recovery time is seldom discussed. In addition, the metadata server that is required in many systems introduces a possible bottleneck and single point of failure, while no central node is required in IndepIndexCDR. Although a record of the neighbor in the backup circle is required in each PE, it is only updated when the adjacent PE fails, which is a low-probability event; thus, the cost may be negligible. In addition, the nWAL ensures no data loss whenever a failure occurs.

However, this structure has its weak points. As the backup are not indexed within the parallel index, they are not directly accessible from other PEs to share the workload if the primary is overloaded. To make the backup accessible, additional processes are required, such as merging the Btree with the subFat-Btree on the primary, or dumping the backup into the primary to build the Fat-Btree for these data. Obviously, the "promotion" process is very time consuming.

4 Compound Treatment

We now propose our solution, which overcomes the disadvantages while inheriting the advantages of IndepIndexCDR.

As mentioned, the above configuration can be improved if both primary and backup are managed by one parallel index. Fortunately, the chained declustered replication places continuous logical fragments in the range partitions as a primary for the current PE and a backup for its neighbor. As shown in Fig. 2, because the data are declustered and range partitioned, they can be coupled into the Fat-Btree structure without any intersection. Because the compound subFat-Btree on each PE also has overlapping intermediate paths to the subFat-Btrees located on its neighbors, as in the original Fat-Btree, it provides the access path from the root node to any data located in any PE either in primary or backup. Therefore, the independent Btree for the backup is no longer required. We name this configuration the compound-index chained declustered replication *CompIndexCDR*.

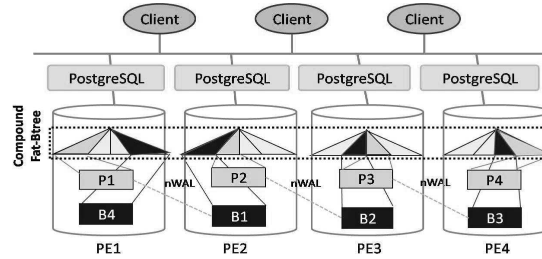

Fig. 2. Configuration of CompIndexCDR

Figure 3 shows an example of this configuration. The upper part shows a global view of the intermediate nodes in the Btree index for all the data over the range (1–60), which are evenly stored in four PEs. By using the original Fat-Btree, some of the nodes will be replicated in several PEs because they are overlapped. To help readers visualize this, we circle the nodes that have a copy in each PE. We use a dotted line for PE_1, a dashed line for PE_2, a solid line for PE_3, and a dashed-dotted line for PE_4. Note that the intermediate nodes that have more than one copy in the PEs have pointers to the leaf nodes located in other PEs. For example, the copy of node "1, 10" in PE_2 has the pointer to the leaf nodes "1, 7" and "10, 16" located in PE_1 and the leaf node "10, 16" located in PE_2. Thus, these overlapped intermediate nodes make an access path for tracing any data from the root among the PEs.

As shown in the lower part of Fig. 3, each PE has two subFat-Btrees for its backup (left) and primary (right), respectively. For the same volume of data, they may have similar index structure and intermediate nodes in their primary and backup PE, such as the index of PE_1's primary and PE_2's backup, while their leaf nodes have pointers to different data pages that are located in PE_1's primary and PE_2's backup, respectively.

To maintain the overlapped paths in this compound Fat-Btree, we also maintain pointers in the parent nodes to all the copies of their child nodes. An example of the overlapped paths for data in the range (31–40) is shown in Fig. 4. In this figure, the root node "1, 20, 46" has paths to all the copies of the second-level node "20, 31" and all the copies of this second-level node have the paths to the primary's and backup's leaf nodes. To distinguish these pointers, we store them with different flags, such as "P" and "B" in the intermediate node. Ordinarily, transactions are carried out by using the pointers that are marked with flag "P" in each intermediate node to access the primary data. nWAL will synchronize the backup after an update. Because the transactions started by the nWAL manager are local synchronization updates, they will use the index pointer marked by flag "B". Note that these pointers are kept available during structure modification operations (SMOs) [25], by referencing all the SMOs in any primary PE to its backup PE. Although the index structure may vary after the SMOs in each PE, the access paths still exist in the intermediate nodes.

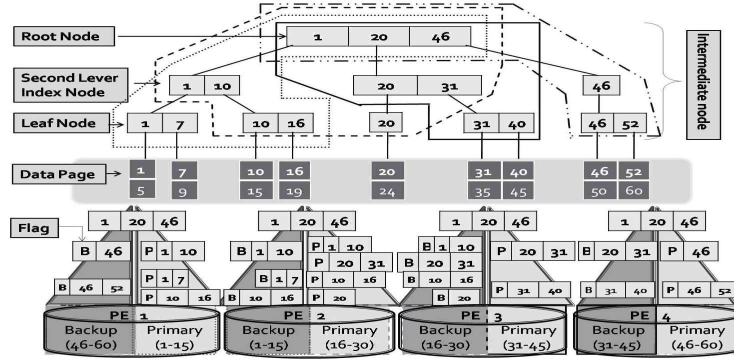

Fig. 3. A Compound Fat-Btree Model

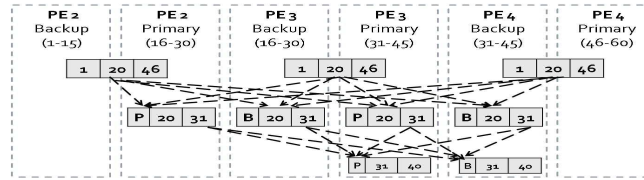

Fig. 4. Overlapped Paths in the Compound Fat-Btree

Our method does not introduce any additional overhead for keeping the backup's index up to date, even if indexing the backup for efficient synchronization is very common in other replication databases. We show that the double-index-sized CompIndexCDR does not reduce the throughput in Section 5.3.

4.1 Availability in CompIndexCDR

The time-consuming "promotion" process described in Section 3 is no longer necessary in CompIndexCDR. In practice, we only switch the flag values between "P" and "B", or modify the indexes of the adjacent PEs that have overlapped intermediate nodes with the failed one with value "P/B".

For example, if PE_3 fails, the new primary location information must be modified only in PE_2 and PE_4, which have intermediate nodes with PE_3. In this case, PE_2 and PE_4 will receive data records (31–45). Then, the flags of those intermediate nodes that covered this range of data records will switch in two steps: Step-A, switch the flags of the corresponding leaf nodes between "B" and "P"; Step-B, modify the flags of the other intermediate nodes having child nodes that are switched in Step-A, following the rules: if all the child nodes of an intermediate node are modified in Step-A, then switch the flag; else if the original flag in the intermediate node is the same as that of the modified leaf node in Step-A, then do nothing; else modify the flag to "P/B", which means

some leaf nodes of this intermediate node are stored as backups while others can be accessed as primaries. After this process, node (P, 20, 31) in PE_2 will be (P/B, 20, 31) and node (B, 20, 31) in PE_4 will be (P/B, 20, 31), while the "31" in both nodes will have the primary point to the leaf node in PE_4. Thus, requests for the data located in this data range will be forwarded to PE_4's backup. If the original PE fails, there is no need to modify the new backup's location or the intermediate nodes on PE_3, and the "update" on the new primary does not write to backup, which is different from the load balancing described later. However, after the failed PE is replaced, the new backup's location will be registered.

We verified that this failover process is much quicker than that of IndepIndexCDR; a comparison experiment is presented in Section 5.4.

4.2 Scalability in CompIndexCDR

Load skew in a value-range partition may greatly degrade system scalability [3]. Ordinarily, data migration is required to handle the skew; this may take a long time and destroy the efficiency of load balancing [2].

The declustering replication scheme [9] provides the ability to balance skews without data migration. However, although there have been many studies on this topic, so far as we know, no one has provided an implementation of this capability in practice. In addition, the metadata servers introduced in these studies may decrease system availability. Our system, CompIndexCDR has no central node, and backups can share the workload with the primary PEs by some simple configuration rules.

For the above example of four nodes, the load can be balanced without data migration if, for example, the query frequency on PE_1, PE_3 and PE_4 is α, while the query frequency on PE_2 is 2α, as shown in Table 1.We also assume that P_i and B_i ($i \in [1, 4]$) in the table represent the same amounts of data. We assume that the query frequency is known merely for convenience; in practical terms, many methods have already been proposed to find the access probability on any PE, for example, [21] is one of them and is suitable for the Fat-Btree.

Table 1. Before Load Balancing

	PE_1	PE_2	PE_3	PE_4
Access rate	α	2α	α	α
Primary	P_1	P_2	P_3	P_4
Backup	B_4	B_1	B_2	B_3

Table 2. After Load Balancing

	PE_1	PE_2	PE_3	PE_4
Access rate	$\frac{5}{4}\alpha$	$\frac{5}{4}\alpha$	$\frac{5}{4}\alpha$	$\frac{5}{4}\alpha$
Primary	$P_1+\frac{1}{4}B_4$	$\frac{5}{8}P_2$	$\frac{1}{2}P_3+\frac{3}{8}B_2$	$\frac{3}{4}P_4+\frac{1}{2}B_3$
Backup	$\frac{3}{4}B_4$	$B_1+\frac{3}{8}P_2$	$\frac{5}{8}B_2+\frac{1}{2}P_3$	$\frac{1}{2}B_3+\frac{1}{4}P_4$

We first assume the workloads are "all-read". The balanced placement is shown in Table 2. Because of space limitations, we omit the detailed description of the balancing process. A general algorithm for load balancing is given in Fig. 5.

Next, we assume the workloads are "all-update". The method still balances the skew by using the same placement in Table 2. Note that each piece of data still keeps a copy of its backup by changing the "old primary" into "new backup" in the neighbor. For example, $\frac{3}{8}P_2$ on PE_2. Similarly to Section 4.1, all the "updates" on $\frac{3}{8}P_2$ will be forwarded through the intermediate path to PE_2 or

```
Dynamic Load Balancing
begin
 Let op, lp, and rp be the target, left, and right PE, respectively;
 Let pdm[i] be the primary data amount on node i;
 Receive a list of loads ll from lp;
 Update ll[op] to reflect the current workload of the PE;
 Calculate the average load al from ll;
 if ll[op] > al + Threashold then
   Calculate the new primary placement pp[n];  // if this skew can be balance without data migration
   for (i = 0; i < n; i++)        // n is the number of PEs
   {
     broadcast the modification msg to node[i] to change primary;
     if pdm[i] > pp[i] then
     {
        Demote_Primary(the leftmost data page in pdm[i], pdm[i] - pp[i]);
        set the 'Flag' for the new 'backup' at pdm[i] - pp [i];
        broadcast the modification msg to node[i+1]{or node[0] if i = n}to change backup there;
        Promote_Backup(the leftmost data page in pdm[i+1], pdm[i] - pp[i]);
        set the 'Flag' for the new 'primary' at pdm[i] - pp [i];
     }
     else
     {
        Promote_Backup(the rightmost data page in pdm[i], pp [i] - pdm[i]);
        set the 'Flag' for the new 'primary' at pp [i] - pdm[i];
        broadcast the modification msg to node[i-1] {or node[n] if i = 0}to change primary there;
        Demote_Primary(the rightmost data page in pdm[i], pp[i] - pdm[i]);
        set the 'Flag' for the new 'backup' at pp [i] - pdm[i];
     }
   }
   Reset list ll ;
 if op is the rightmost PE then
   Send ll to the leftmost PE;
 else
   Send ll to rp;
end
```

Fig. 5. Algorithm for Handling Skews

PE_3, then the "updates" are carried out on $\frac{3}{8}B_2$ at PE_3, and the nWAL manager on PE_3 will send nWAL log messages to the backup PE_2. Because the data backups are always maintained in the system, future failures are still tolerable.

It is obvious that for update transactions, each node still has the same number of transactions to be executed even if the placement in Table 2 is adopted. However, as the backups are updated asynchronously, the backup overhead can be alleviated and the unexecuted updates can be preserved temporarily as nWAL records until the overhead is remitted. Therefore, small load skews (i.e., a 2α skew always arises after a node fails) can easily be balanced without data migration. Nevertheless, data migration is still necessary when the skew is more than $\frac{2(n-1)}{n-2}\alpha$ (n is the number of PEs). However, even so, the proposed method is able to halve the skew immediately, and reduce the data migration cost.

5 Experiments

It is hard to provide direct comparisons with former studies because, to the best of our knowledge, no previous work exists that manages both primary and backup within one directory or analyzes the failover efficiency issue in a

Table 3. Experimental environment

Blade servers:	Sun Fire B200x Blade Server
CPU:	AMD Athlon XP-M 1800+ (1.53 GHz)
Memory:	PC2100 DDR SDRAM 1 GB
Network:	1000BASE-T
Gigabit Ethernet Switch:	Catalyst 6505 (720 GB/s backbone)
Hard Drives:	TOSHIBA MK3019GAX (30 GB, 5400 rpm, 2.5 inch)
OS:	Linux 2.4.20
Java VM:	Sun J2SE SDK 1.5.0 03 Server VM

chained declustering database. Instead, we compare our work with the well-known Postgres-R (PG-R) and HBase replication DBMSs, as well as with the naive-structure IndepIndexCDR that treats primary and backup separately, to demonstrate the scalability and availability of our proposed approach.

5.1 Experimental Environment

The experimental system was implemented on a 160-node PC cluster system, and the experimental environment is summarized in Table 3. We used continuous integers to act as the primary key in each PE in these experiments.

5.2 Comparison with Postgres-R and HBase

At first, we compare the performance of CompIndexCDR with multi-replication DBMS PG-R and HBase to assess our system performance.

PG-R [23] is the first significant research prototype to provide fully functional database replication mechanisms based on an open-source database. HBase is an open-source, distributed database modeled after Google's BigTable; it provides BigTable-like capabilities and has recently attracted much interest in the distributed database community. We chose these two systems as the criteria for our comparisons. Although the configurations of these systems are different in replication strategies, what we focus on here is the comparison between full/partial replication with our approach.

We use the same experimental configuration as in [23], which gives the most recent performance results of PG-R, to make the first comparison. In this experiment, both PG-R and CompIndexCDR contain five or 10 nodes and each node stores 8000 tuples. The experiment is performed with 40 clients that are evenly distributed on PEs. Figure 6-a shows the experimental results.

For the "all-read" result, PG-R clearly outperforms our system because of its full replication scheme. However, as its authors declared [23], PG-R does not win by much and will lose in scalability even for a moderate number of updates when the system scales. In contrast, CompIndexCDR has a limitted update overhead and share the load along the cluster. We verify its scalability with up to 64 nodes in the next section.

We adopted the recently released HBase-0.20.2 to make the second comparison, which runs on our cluster system based on Hadoop-0.20.1 and jdk1.6.0.18.

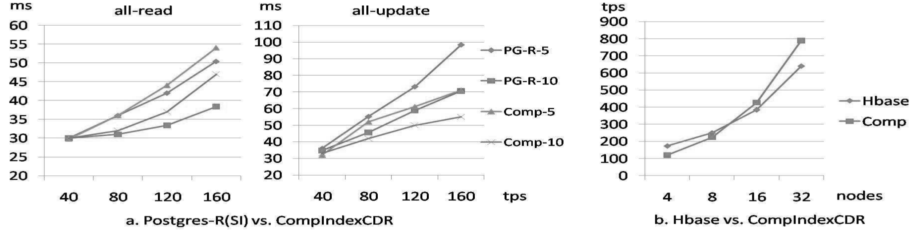

Fig. 6. Performance Comparison with Postgres-R and Hbase

We used the default settings in HBase and Hadoop throughout the experiments, except for changing the replica degree to two, which is the same as in our system. The "Datanode" number in our experiments is only up to 32 because the software in current computation clouds typically uses significantly fewer instances. For example, Amazon limits the number of nodes to 20 by default [27].

In this experiment, the dataset contains 10,000 tuples, and each tuple is 4 KBytes with four columns, and we have one client on each node to perform all-read transactions. The results are shown in Fig. 6-b. We do not provide all-update results here because they are almost the same as the all-read results. The figure shows that the column-oriented DBMS with BigTable has better performance for a small number of nodes. However, as our system has better scalability, it soon outperforms Hbase as the system grows.

We can conclude from these experiments that, for scalability and throughput, CompIndexCDR outperforms both the full-replication PG-R and partial-replication HBase. A further examination of the availability follows.

5.3 Throughput Comparison

We compare the throughput and scalability of our two structures to reflect their different index-traversing costs. The cost mainly consists of intranode and internode traverse costs. The first one is the cost for traversing a local index; larger datasets increase this cost. The second one is the cost for locating remote data as well as maintaining the distributed index; the low cost of this part is one of the strengths of our index. Because the throughput only reflects total traversing cost, and huge datasets increase the intranode cost in both systems, we show the difference in the internode costs more clearly by initializing each PE with a small dataset of 10,000 tuples, each of 134 bytes.

In the experiments, each PE receives the requests from their client nodes simultaneously; the key in each request is generated randomly from the whole data range. As for the 64 PE case, these queries will be: key = random(1, 640000); select* from table where id = key; update table set value += 1 where id = key. Because we focus on the index cost, we do not use complex queries. In addition, each client sends 20 transactions to a PE serially and has a private thread on the target PE to process its requests.

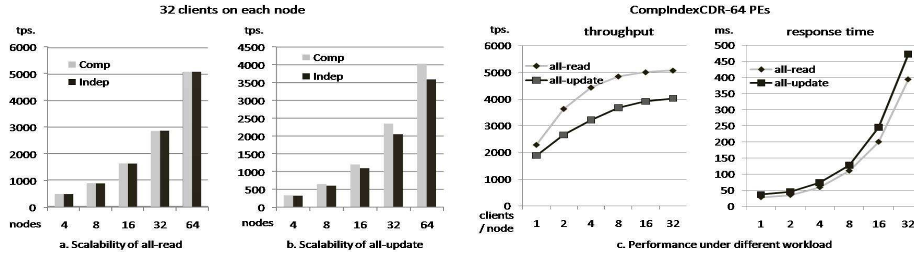

Fig. 7. Scalability Comparison

Figure 7-(a,b) shows the scalability of our systems with 32 clients on each node. Figure 7-a illustrates the all-read case. The performances of these two structures are almost the same and their scalability is quite good. The throughput increases by about 80% when the number of PEs is doubled. The missing 20% may result from the increasing amount of network communication caused by the growing number of remote transactions in the larger system. Figure 7-a also shows that although the index size of CompIndexCDR is twice that of IndepIndexCDR, their throughputs are almost the same. The reason may be that the backup is not accessed, so both systems have similar memory hit rates. Figure 7-b shows that CompIndexCDR has a better scalability for the all-update workload. This may be because the backup in IndepIndexCDR is updated by an independent Btree, while it is accessed within one compound index in CompIndexCDR, therefore the index is not switched in memory. We do not provide experiments for mixed read/write workloads, because they simply produce intermediate results between all-read and all-update workloads.

Figure 7-c shows the throughput and response time of CompIndexCDR at the scale of 64 PEs with various client numbers per node. We omit the results at other sizes, because those results are almost the same as this one. As the figure shows, the throughput of CompIndexCDR increases with the client numbers. When each node has 32 clients, the growth is moderate at a stable value that is defined by the limitations of the CPU and I/O performance in our cluster system. On the other hand, the response time also increases with the client numbers, which is normal in DBMSs and is usually solved by adding more PEs to scatter the workload. Figure 8-a shows the trend of response times as the number of PEs increases. For a fixed workload of 256 clients in total, the response time is effectively reduced because those requests are evenly scattered after new PEs are added into the system.

5.4 Failover Time Comparison

In this experiment, we used four PEs, each with 10,000 tuples and each serving two users. When a PE fails, we dump the backup into the primary and rebuild the Fat-Btree for those data for IndepIndexCDR, and we take the failover as

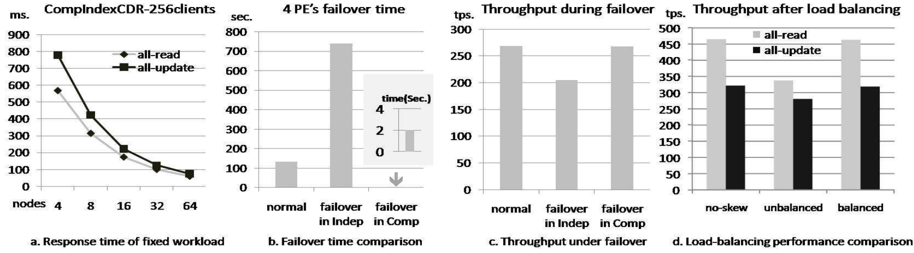

Fig. 8. Response Time and Failover, Load Balancing Performance

described in Section 4 for CompIndexCDR. In Fig. 8-b, "Normal" stands for the time required to initialize 10,000 tuples in one PE. "Failover in Indep" stands for the time taken by IndepIndexCDR to prepare a new primary. It is much higher than the value for "Normal" because the PE must process ordinary requests while making the new primary. As the graph shows, the failover in CompIndexCDR takes less than 2 seconds, which is much better than the time for IndepIndex-CDR. We also examined the throughput during failover. Figure 8-c shows that the throughput decreases only slightly during failover.

5.5 Load Balance Comparison

In this experiment, we used four PEs, each with 10,000 tuples and serving 32 users 40% of the requests access the data in one PE and the other three PEs share the remaining 60%, the same skew as in Sec. 4.2. Figure 8-d shows that this skew decreases the throughputs in "all-read" and "all-update" by 25% and 15%, respectively. After using the balancing method in Sec. 4.2, the "all-read" is almost the same as that of "no-skew" and the "all-update" is also much improved, which verifies the effectiveness as argued in Section 4.2.

6 Related Work

The primary/backup approach has been studied to provide system availability. The interleave declustering strategy provides an immediate load and space balance, but at the cost of reliability. Chained declustering strategy handles the range partition and reconcile high availability with load balancing without loss in reliability. Hot mirroring and RAID suggest the possibility of a hybrid approach, combining parity-based and primary-backup approaches.

On the other side, various parallel index structures have been proposed[16, 13, 24] to provide efficient access method for declustering database. In addition, [1] provides two alternatives for performing the necessary index modifications (OAT), however, they both lead to considerable SMOs cost [13]. Fat-Btree [24] vastly reduces the SMOs cost and improves the dynamic skew handling method. [5, 12] provide load balancing solutions with less data migration.

Unfortunately, all the above proposals mainly focused on only one or two targets among reducing synchronization cost, rapid recovery, efficient access and dynamic skew handling. They did not explore access methods or the use of replicas to promote scalability and availability. Note that, other parallel indexes that may be alternative in our method are not suitable for the compound treatment as ours. For example, the “FirstTierIndex” of the “aB$^+$-tree” [13] must be updated immediately in all PEs after data reallocation. “GHT” in [16] requires an “AST” in all PEs to record the primary/backup switch, and [12] require much more modification of adjacent nodes’ indexes in skew balancing.

7 Conclusions and Future Work

We proposed a compound index treatment of a chained, replicated declustering scheme CompIndexCDR for shared-nothing parallel database systems. So far as we know, this is the first research to support rapid recovery, dynamic skew handling and efficient data access, as well as reducing synchronization cost by the compound management of primary and backup in a chained replication system.

As the CPU and I/O performance of the PC cluster we used in our experiments are slower than those of current systems, we believe CompIndexCDR can achieve much better performance once the hardware is upgraded. Thus, we conclude that CompIndexCDR is a very good choice for highly scalable and available parallel databases.

In this work, we have not considered the time for restoring a backup after a failure. The adaptive overlapped declustering method [22] is a good way to reduce the restoration time. In future, we plan to combine adaptive overlapped declustering with CompIndexCDR. We also plan to consider the management of multi-replica schemes to improve availability and performance.

Acknowledgments

Part of this research was sponsored by CREST of Japan Science and Technology Agency (JST), and MEXT via a Grant-in-Aid for Scientific Research #19024028.

References

1. K. J. Achyutuni, E. Omiecinski, S. B. Navathe, Two techniques for on-line index modification in shared nothing parallel databases, In Proceedings of the ACM SIGMOD Int’l Conf. on Management of data, p.125-136, June 04-06, 1996
2. R. Arnan, E. Bachemat, T. K. Lam, R. Michel, Dynamic data reallocation in disk arrays, ACM Transactions on Storage (TOS), v.3 n.1, p.2-es, March 2007
3. D. Dewitt, Jim Gray, Parallel database systems: the future of high performance database systems. Communications of the ACM, v.35 n.6, p.85-98, June 1992
4. S. Elnikety, W. Zwaenepoel, F. Pedone, Database replication using generalized snapshot isolation, SRDS ’05 p.73-84, Oct. 26-28, 2005

5. H. Feelifl, M. Kitsuregawa, B. C. Ooi, A fast convergence technique for online heat-balancing of btree indexed database over shared-nothing parallel systems, In Proceedings of 11th Int'l Conf. DEXA '00, p.846-858, Sep. 04-08, 2000
6. A. Fekete. Allocating isolation levels to transactions. In Proceedings of 24th ACM SIG-(MOD/ACT/ART) symposium on principles of database systems, June, 2005
7. T. Haerder, A. Reuter, Principles of transaction-oriented database recovery, ACM Computing Surveys (CSUR), v.15 n.4, p.287-317, Dec. 1983
8. H.-I. Hsiao, D. J. DeWitt, A performance study of three high availability data replication strategies, Distributed & Parallel Databases, v.1 n.1, p.53-80, Jan. 1993
9. H.-I. Hsiao, D. J. DeWitt, Chained declustering: A new availability strategy for multiprocessor database machines, In Proceedings of ICDE '90, p.456-465, 1990
10. S.-O. Hvasshovd. Recovery in Parallel Database Systems. Morgan Kaufmann Publishers, 2nd edition, 1999
11. B. Kemme, G. Alonso, Don't be lazy, be consistent: postgres-r, a new way to implement database replication, In Proceedings of VLDB '00, p.134-143, Sep. 2000
12. H. Feelifl, M. Kitsuregawa, RING: a strategy for minimizing the cost of online data placement reorganization for btree indexed database over shared-nothing machines, In Proceedings of the 7th Int'l Conf. on DASFAA '01, p.190-199, April, 2001
13. M. L. Lee, Towards self-tuning data placement in parallel database, Proceedings of the ACM SIGMOD Int'l Conf. on Management of data, p.225-236, May, 2000
14. J. Miyazaki, H. Yokota, Concurrency control and performance evaluation of parallel B-tree structures. IEICE Trans. INF. & SYST. E85-D(8), p.1269-1283, 2002
15. X. Ouyang, H. Yokota, An efficient commit protocol exploiting primary–backup placement in a distributed storage system, In Proceedings of the 12th Pacific Rim International Symposium on Dependable Computing, p.238-247, Dec. 18-20, 2006
16. P. Zezula, G. Amato, V. Dohnal, M. Batko, Similarity Search The Metric Space Approach, Chapter 5. Parallel and Distributed Indexes. Springer US, 2006.
17. B. Seeger, P. Larson, Multi-disk B-trees, In Proceedings of the 1991 ACM SIGMOD international conference on Management of data, p.436-445, May 29-31, 1991
18. D. Taniar, J. W. Rahayu, Global parallel index for multi-processor DB systems. Information Sciences, Volume 165, Issues 1-2, 3 Sep. 2004, p. 103-127.
19. T.Yoshihara, D. Kobayashi, H. Yokota, Mark-opt: A concurrency control protocol for parallel B-tree structures to reduce the cost of SMOs. IEICE transactions on information and systems, ISSN 0916-8532, 2007, vol. 90, p. 1213-1224
20. R. V. Renesse, F. B. Schneider, Chain replication for supporting high throughput & availability, In Proceedings of the 6th USENIX Symposium OSDI '04, p.7, 2004
21. A. Watanabe, H. Yokota, A directory traverse cost based skew handling for parallel data access, Transactions of IEICE (D-I), Vol. J85-D-I, p.877-886, 2002.9
22. A. Watanabe, H. Yokota, Adaptive overlapped declustering: a highly available data-placement method balancing access load and space utilization. In Proceedings of the 21st Int'l Conf. on Data Engineering, p. 828-839, 2005.
23. S. Wu, B. Kemme, Postgres-r(si): Combining replica control with concurrency control based on snapshot isolation. In Proceedings of ICDE '05, p. 422-433, 2005.
24. H. Yokota, et al. Fat-Btree: An update conscious parallel directory structure. In Proceedings of the 15st Int'l Conf. on Data Engineering, p. 448, 1999.
25. T. Yoshihara, D. Kobayashi, H. Yokota, A concurrency control protocol for parallel b-tree structures without latch-coupling for explosively growing digital content. In Proceedings of the 11th Int'l Conf. on EDBT '08, vol. 261. p. 133-144, 2008
26. H. Yu, A. Vahdat, The costs and limits of availability for replicated services, In Proceedings of the 18th ACM SOSP '01, Oct. 21-24, 2001
27. Amazon Web Service LLC: http://aws.amazon.com/elasticmapreduce/, 2009.