論文 / 著書情報
Article / Book Information

Title	A Compound Parallel Btree for High Scalability and Availability on Chained Declustering Parallel Systems
Authors	Min Luo, Akitsugu Watanabe, Haruo Yokota
出典 / Citation	IEICE TRANSACTIONS on Information and Systems, Vol. E94-D, No. 3, pp. 587-601
発行日 / Pub. date	2011, 3
URL	http://search.ieice.org/
権利情報 / Copyright	本著作物の著作権は電子情報通信学会に帰属します。 Copyright (c) 2011 Institute of Electronics, Information and Communication Engineers.

PAPER Special Section on Data Engineering

A Compound Parallel Btree for High Scalability and Availability on Chained Declustering Parallel Systems*

Min LUO[†a)], **Akitsugu WATANABE**[†], *Nonmembers*, *and* **Haruo YOKOTA**[†], *Fellow*

SUMMARY Scalability and availability are the 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 (HA) while preserving scalability. However, independent treatment of the primary and backup data in each node requires long failover times. We propose a novel method for the compound treatment of chained declustered replicas using a parallel Btree, termed the Fat-Btree. In the proposed method, a single Fat-Btree provides access paths to both the primary and backup data of all processor elements (PEs), which greatly reduces failover time. Moreover, these access paths overlap between two neighboring PEs, which enables dynamic load balancing without physical data migration by dynamically redirecting the access paths. In addition, this compound treatment improves memory space utilization to enable index processing with good scalability. Experiments using PostgreSQL on a 160-node PC cluster demonstrate the effectiveness of the high scalability and availability of our proposed method.

key words: *parallel Btree, scalability, availability, chained declustering*

1. Introduction

The explosive growth of digital information together with the demand for its 24H on-line availability have generated interest in research on databases with high performance and high availability (HA), both of which are improved by the replication of distributed database systems on a shared-nothing parallel infrastructure. However, adding more processor elements (PEs) to a shared-nothing parallel database system to increase system performance also increases the probability of faults in the system. Therefore, replication in parallel database systems has played an increasingly important role in ensuring availability while maintaining scalability. Here, scalability enables the database to grow while supporting an ever-increasing rate of throughput; availability enables the database to continue providing a full service without losing data when some parts of the system fail. Efficient accessing methods with skew-balancing ability and data replication with synchronization strategies are very important for achieving scalability and availability, respectively [1].

Manuscript received June 5, 2010.
Manuscript revised October 4, 2010.

[†]The authors are with the Department of Computer Science, Tokyo Institute of Technology, Tokyo, 152–8552 Japan.

[*]A major part of this paper was presented at 21st International Conference on Database and Expert Systems Applications, Bilbao, Spain.

a) E-mail: luomin@de.cs.titech.ac.jp

DOI: 10.1587/transinf.E94.D.587

It has been proved that: consistency, availability and partition tolerance cannot be provided simultaneously in distributed systems, because providing guarantees for two properties will violate the guarantee for the third property [2]. Thus, in this work, we relax the requirement of partition tolerance since network partition is less common in parallel distributed databases, for which availability and consistency are more essential properties. However, since the publication of [3], in which it is argued that systems with strong consistency have unstable behavior when scaled up, research interest has become focused on consistency and availability, leaving the scalability issue seldom addressed [2], [4]. Existing replication databases fail to scale up, particularly at a high update rate, because of the high cost of data synchronization [5]. In addition, well-known massive data centers for cloud-based applications, such as PNUTS [6] at Yahoo, Dynamo [7] at Amazon and BigTable [8] at Google, have adopted the strategy of sacrificing strong consistency for availability and scalability. This strategy benefits scalability, but the advantage of replication in gaining higher throughput is lost, because the replicas are not ready to be queried most of the time. Moreover, the advantage of replication in gaining higher availability may also be lost in the long run if strong consistency cannot be ensured [4]. Therefore, in this trade-off, a sacrifice of consistency or scalability does not have a dependable effect, and this has long been an obstacle to developing efficient replication techniques.

To address this problem, efficient data access methods guaranteeing consistency between replicas play an important role. Basically, there are two main methods of efficient data accessing in distributed systems: distributed-hash-table (DHT)-based [9] and B-tree-based [10] methods. DHT-based methods can efficiently support exact-match queries but they destroy the semantics and locality of data objects and incur a high cost in the case of range queries. In contrast, B-tree-based methods with value-range partitioning schema can efficiently support range queries but suffer limited access-skew-balancing ability, data skews in the case of repeated updates and the overhead of intermediate-node management.

To provide fast and scalable data accessing for range partitioning, many B-tree-based parallel index structures have been introduced to access the data stored in a shared-nothing environment [11]. However, these parallel indexes do not provide the dynamical access-skew-balancing ability required for high scalability. In addition, they do not con-

sider index management on replica data for higher system availability. In this work, we provide dynamical access-skew-balancing ability within a parallel B-tree structure without introducing a high intermediate-node management overhead in a replicated system. By further overcoming the remaining problem in parallel B-tree structures mentioned above, our B-tree-based solution becomes superior to existing hash-based methods since it preserves efficient range-query ability. Moreover, the availability in a replicated system is also improved by maintaining replica data using our index.

To increase availability in current parallel B-tree systems, data replication and an efficient failover process are required. In addition, the consistency between primary and replica data should be maintained to ensure immediate failover and data availability. However, as Gray et al. pointed out [3], traditional replication schemes do not scale well. The reconciliation rate grows as the square of the number of replicas and the deadlock rate increases as the cube. Therefore, the chained declustering method [16] is adopted in our proposed database infrastructure to achieve a low degree of replication while realizing high availability and scalability compared with other methods [17], [18].

In this paper, 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.

This paper makes the following contributions.

- To our knowledge, this is the first proposal to manage both primary and backup data within a single directory structure. Our proposed parallel B-tree provides dynamical load balancing and immediate failover ability with low intermediate-node maintenance cost. It also enables efficient data access and range queries.
- To our knowledge, this is the first range-partition-based indexing method to implement HA capability in a chained declustering system. Although the capability of HA in chained declustering has been declared for decades [17], [18] and range-partition-based indexes are highly suitable for chained declustering, no valid index has been proposed for the efficient accessing of replicas to improve availability and scalability.
- An efficient load-balancing algorithm is proposed to flatten access skew dynamically without data migration to enable high scalability, as well as an efficient failover algorithm to realize high availability without the cost of "promotion" backups.
- An efficient synchronization method for primary and replica data and their corresponding index are provided. The data consistency in our system is ensured through serialization criteria based on a variant of the concurrency control method that we previously proposed [23], while the updates are synchronized on primary and replica data in the same order in an all-or-nothing manner by adopting the nWAL-based commit protocol [24].
- We evaluated our method on a PC cluster that consists of 160 nodes, each with an HDD and processor. The experimental results demonstrate all the above points.

The rest of this paper is organized as follows. First, the choice of data placement, the concept of the parallel B-tree and the method of nWAL are described in Sect. 2. Then, a basic infrastructure that straightforwardly combines these techniques with independent access paths for the primary and backup data is discussed in Sect. 3. In Sect. 4, we propose a method of handling both primary and backup data by a single Fat-Btree to support efficient failover and load balancing, and the implementation details are described in Sect. 5. Experiments and results are reported in Sect. 6. In Sect. 7, we describe some related research. The conclusions and future work are discussed in the final section.

2. Background

We briefly review three technologies for constructing scalable and available shared-nothing parallel databases: data placement strategies, parallel indexing structures and a synchronization method. Then, we give the assumptions and conditions employed in this paper before we describe our approach.

2.1 Data Placement Strategies

The shared-nothing configuration is generally used to achieve high performance in parallel database systems. This is because it consists of a set of independent PEs that do not share memory or disks so that the computational and I/O resources can be maximized. It is simple to speed and scale the system up to hundreds of PEs [21].

Chained declustering [16] is a technique that offers both HA and good load balancing on shared-nothing parallel systems. In chained declustering, PEs in one relation-cluster maintain two physical copies of the relation, a primary copy and a backup copy. These two copies are declustered across the cluster by the same partitioning strategy. Because the corresponding fragments of primary and backup copies are stored on different PEs, no data is lost after a failure.

The selection of the data-partitioning strategy determines the availability and throughput of chained declustering data placement. Basically, there are three types of partitioning strategies [22]: hash, round-robin and value-range partitioning. Hash partitioning is ineffective for range queries and does not scale up when the number of PEs increases [18]. Round-robin partitioning produces no skew but is ineffective for queries because it requires brute-force searches. Value-range partitioning can treat range queries and match queries efficiently but has a risk of skew during repeated updates.

Although each partitioning strategy has at least one significant limitation, overcoming the current shortcomings of one of the strategies will make it outperform the others.

2.2 Parallel Indexing Structures

As mentioned in Sect. 1, there are two main methods of distributed data accessing. DHT-based methods uniformly map nodes and data objects into a single ID space and each node is responsible for a specific range of the ID space. It balances the load among nodes and ensures an upper bound of required time *log(n)* (n is the number of nodes) to locate data objects [29]. Considerable effort has been devoted to supporting range-query applications in DHT-based systems [25]–[28] however, their efficiency is still limited as all systems require additional structures, which introduce an extra overhead. On the other hand, B-tree-based parallel indexing with efficient range-query capability, has been proposed for dynamic data management, high throughput and efficient data skew handling via index node migration [19]. However, it suffers from a high cost of updating the index structures and limited access-skew-balancing ability. Because both strategies have their advantages and disadvantages, the disadvantage of the B-tree-based parallel index can be reduced if its shortcomings can be overcome.

To reduce the update cost in parallel B-tree indexing, an update-conscious parallel B-tree structure, a Fat-Btree, has been proposed [12]. The formal definition of a Fat-Btree is given using the following notation. Each node of a tree is distinguished by its identifier i, and the level of node i is denoted by $L(i)$. When node i is the parent of node j, $P(j) = i$ and $L(j) = L(i) + 1$. Therefore, let S_i be a set of PEs storing node i. Then, a parallel B-tree structure satisfying the condition $S_i \supseteq S_j$ if $i = P(j)$ is called a Fat-Btree structure. As shown in the four-PE Fat-Btree given as an example in Fig. 1, multiple copies of index nodes close to the root node with relatively low update frequency are replicated on several PEs, while leaf nodes with relatively high update frequency are distributed across the PEs. Thus, the maintenance cost of the Fat-Btree is much lower than that of other parallel Btree structures. In addition, the Fat-Btree has a higher cache hit rate [12] and more efficient concurrency control protocols than other methods [30].

2.3 nWAL-Based Synchronization Methods

Most replication databases employ an atomic commit protocol (ACP), such as the two-phase commit (2PC), to ensure data consistency and transaction atomicity when there is any change that spans database sites. However, these protocols have high latency because of the message exchange overhead, forced log writes and so forth. Although these drawbacks have been mitigated to some extent, ACPs cannot yield satisfactory performance for chained replication systems without fully exploiting the features in this environment. For instance, the WAL [31] is widely used in database systems to ensure atomicity and durability. However, transactions have to be suspended for the forced log-write into stable storage before the new version can replace the previous one. In the case of replicated databases, this inefficiency is solved by the neighbor write-ahead log (nWAL) [20].

Unlike ordinary WALs, the nWAL stores the log in the neighboring hosts' memory without a forced log-write or disk I/O. When a log-write becomes necessary in the distributed system, the log of the transaction is simply transferred into the memory of its replica nodes through the network. Furthermore, transactions are not committed until the neighboring node sends back a message to confirm the success of receiving the nWAL thus, the system always has the log saved before any transaction is committed. Although the overhead incurred in the nWAL includes the message exchange delay and the time required to store log records in the main memory, current high-speed networks can deliver the log record messages several orders faster than disk or memory I/O.

Moreover, for chained declustered replication systems, the nWAL is naturally suitable for guaranteing the consistency of primary and backup data. Because the transactions are committed until the backup has successfully received the nWAL, the backup can always be made consistent with the latest version of its primary data by using the nWAL whenever the primary data fails. Therefore, no particular synchronization messages in the replicated system are required. In addition, when the failure of either primary or backup data occurs during the commit process, no voting phase is required for a faster commit. We have proposed the Backup-Assist-1.5-Phase (BA-1.5PC) [24] commit protocol, which efficiently performs nWAL logging and handles transaction failures in a chained declustering environment. It guarantees the consistency of distributed transactions, i.e., all sites participating in a transaction either unanimously commit or unanimously abort the transaction.

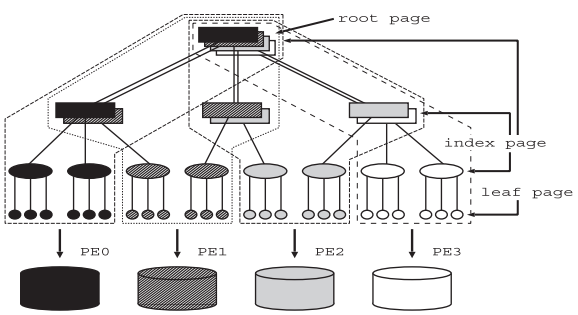

Fig. 1 A fat-btree.

2.4 Assumptions and Conditions in This Paper

For the methods of ensuring system availability we discuss in this paper, we employed the following assumptions and conditions.

1). Failures can be detected by existing failure detection techniques, the discussion of which is beyond the scope of this paper.

2). All member PEs know the IP address of all other PEs and the relationship in their logical neighborhood, and it is assumed that membership management methods, such as

group communication work well in our system. The parallel B-tree index in our approach is in charge of data location management and isolated from the process of membership management.
3). Primary and backup versions of the same data do not fail simultaneously.

3. A Straightforward Infrastructure

We present a straightforward infrastructure for our distributed replication database based on existing techniques and discuss its its advantages and disadvantages.

As mentioned in Sect. 1, the synchronization cost is the most serious obstacle to achieving high performance and scalability in replication databases unless the number of replica copies is reduced. The chained replication system only maintains one replica for each piece of data, thus easily achieving higher throughput, availability and stronger consistency than other replication schemes [17], [18]. Therefore, we treat the chained replication scheme as the basic configuration in this work.

To provide a global access path for the primary data in a chained replication system, a parallel index structure is required. We adopt the Fat-Btree because of its low cost of concurrency control and high flexibility in compound treatment, as discussed in the next section. Alternative indexes are discussed later in Sect. 7.

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 occurs, it is sufficient to maintain an independent Btree for the backup in each PE.

3.1 Implementation of IndepIndexCDR

We implemented a straightforward configuration with an independent index for chained declustered replication, named IndepIndexCDR, by using PostgreSQL and a 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 generated within each update transaction asynchronously from its primary.

An update operation in the Fat-Btree has to acquire the modification latches from the root index node to its target leaf index node. For the concurrency control in the Fat-Btree, a later update that conflicts with previous uncommitted updates in latch acquiring is postponed/restarted to acquire the corresponding index-nodes latches repeatedly. This nodes latching processing by using a concurrency control method [23] described briefly as follows.

For example, we assume that a transaction T_1 starts to be executed at PE_1: it starts a process requiring the latch from the root node. The IX latch is first required in the upper-level index node and is released immediately once the IX latch of the lower-level index node is received successfully. This latch requiring-releasing process continues from the root index node to the leaf index node where the target data locate, if all the intermediate index nodes in this process are not full. Then the update occurs at the leaf nodes after the X latch is acquired. Once there is a full intermediate index node in the process, the X latch is required on the parent of the full node to guarantee that the insertion of a new index pointer will not cause node-splitting to the upper-level index nodes. Note that, an intermediate index node may be replicated in the neighboring PEs, the transaction will restart if the X latch of this kind of index node are not received on all the neighboring PEs. Otherwise, T_1 will update the data on both PE_1 and PE_2 with the commit protocol BA-1.5PC [24] mentioned in Sect. 2.3 to guarantee the consistency of data.

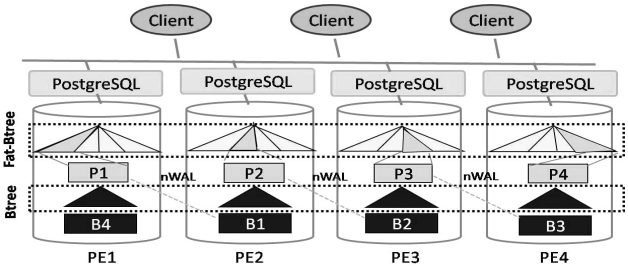

Fig. 2 Configuration of IndepIndexCDR.

Figure 2 illustrates the system configuration. Each PE has an PostgreSQL instance to manage the local data. When a postmaster of the PostgreSQL receives a request from clients, it first consults the Fat-Btree to obtain the location of target data from its local sub-FatBtree. If the data is found in the local PE, the request is executed directly; otherwise, it is forwarded to the appropriate PE depending on the access path in the intermediate index nodes of the Fat-Btree between adjacent PEs. Note that for an update request, an nWAL message is sent to the backup node before the result is returned to the user. After the backup node receives the nWAL, it updates the backup data asynchronously using the local Btree index on the backup data. In the case of failover or load balancing, the backup is able to take over the load from the primary after it finishes the transactions remaining in its nWAL.

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 [17], [18], in which only the head and tail PEs accept query and update requests, IndepIndexCDR can access any data item in primary storage from any PE through 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 Sect. 6.3.

3.3 Availability in IndepIndexCDR

Almost all the replication databases claim failover capability, but the recovery time is seldom discussed. A metadata server is usually required in many systems, which introduces a possible bottleneck and a single point of failure, while no central node is required in IndepIndexCDR. In addition, the nWAL ensures no data loss whenever a failure occurs.

However, this structure has its weak points. As the backup data are not indexed within a parallel index, they are not directly accessible from other PEs to share the workload if the primary data is overloaded. In the case of failover in IndepIndexCDR, the surviving backup data is also not accessible via other nodes through the parallel index. To make the backup accessible through the parallel index, additional processes are required, such as merging the Btree of backup data with the subFat-Btree of the primary data, or dumping the backup into the primary to build a Fat-Btree for both data. These "promotion" processes will obviously increase the failover time since they are very time-consuming.

4. Compound Treatment

Here, we propose our method of overcoming the above disadvantages of IndepIndexCDR while retaining its advantages.

As mentioned above, this system can be improved if the primary and backup are managed by a single parallel index. Fortunately, chained declustered replication places the continuous fragments in a range of partitions as a primary for the current PE and a backup for its neighbor. As shown in Fig. 3, there are two subFat-Btrees to manage the primary and backup data on each PE. Because the data are declustered and range-partitioned, they are coupled so that they exist in the Fat-Btree structure without any intersection. Since the compound subFat-Btree on one PE also has the overlapping intermediate paths to its neighboring subFat-Btrees, similarly to the original Fat-Btree, it provides an access path from the root node to all primary and backup data located in any PEs. Therefore, an independent B-tree for the backup data is no longer required. We name this configuration compound-index chained declustered replication or *CompIndexCDR*.

Figure 4 shows an example of this configuration. The upper part shows a global view of the intermediate nodes in the B-tree index for all the data over the range 1–60, which are evenly stored in four PEs. Using the original Fat-Btree, some of the intermediate nodes will be replicated in several PEs because they overlap. To help readers visualize this, we have girdled 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 intermediate nodes may have pointers to copies of their leaf nodes located in other PEs. For example, the copy of node "1, 10" in PE_2 has a pointer to the leaf nodes "1, 7" and "10, 16" in PE_1 and the leaf node "10, 16" in PE_2. These overlapped intermediate paths allow any data to be traced in the system from the root index.

As shown in the lower part of Fig. 4, each PE has two subFat-Btrees, one for its backup (left) and one for its primary (right). For the same volume of data, they may have a similar index structure and intermediate nodes in their primary and backup PE, such as PE_1's primary index and PE_2's backup index. Also, each index node may have pointers to its child nodes located in PE_1's primary index and PE_2's backup index.

The purpose of these duplicated pointers is to maintain the overlapped paths enabling the access of backup data by the compound Fat-Btree. An example of overlapped paths for data in the range 31–40 is shown in Fig. 5. 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 a path to the same data replicas lo-

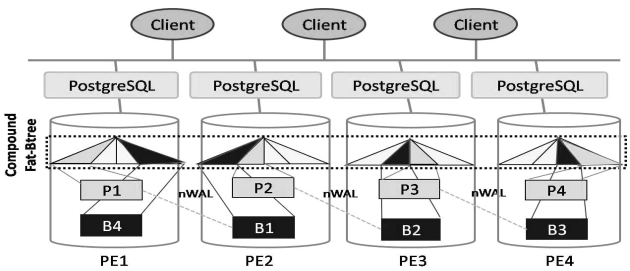

Fig. 3 Configuration of CompIndexCDR.

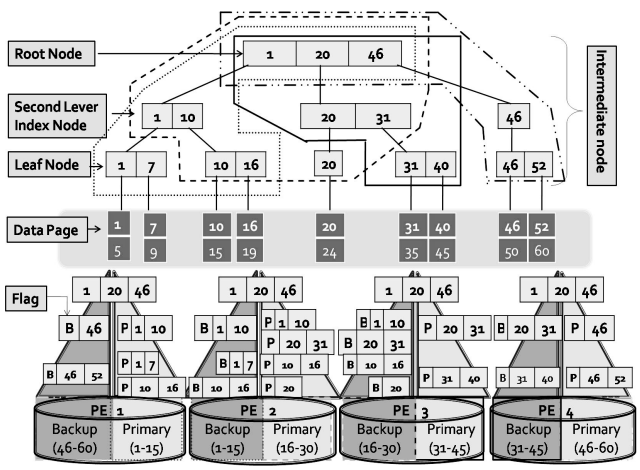

Fig. 4 A compound fat-Btree model.

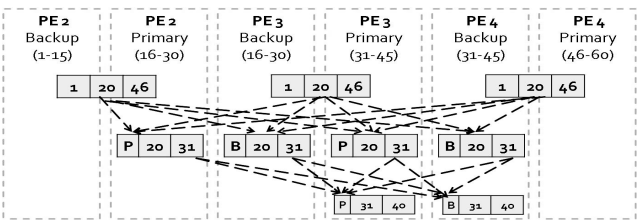

Fig. 5 Overlapped paths in the compound fat-Btree.

cated either in the primary or backup's leaf nodes. To distinguish these pointers, we store them with different flags, such as P and B, in the intermediate index nodes where they are located. Ordinarily, transactions are carried out using the pointers that are marked with flag P to access the primary data. Note that the transactions started by the nWAL manager are local synchronization updates: they only use the index pointers marked by flag B. In addition, these pointers are kept available during SMOs by referencing all the SMOs in any participating primary PE to its backup PE. We describe the intermediate-path maintenance process later in Sect. 4.3.

Our method does not introduce any additional overhead for keeping the backup index up to date, even though indexing the backup for efficient synchronization is very common in other replication databases. We also show that the doubled index size of CompIndexCDR does not reduce the throughput in Sect. 6.3.

4.1 Availability in CompIndexCDR

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

For example, if PE_3 fails, the new primary location information need only be modified in PE_2 and PE_4, which have intermediate paths with PE_3. In this case, PE_2 and PE_4 will recovery data records 31–45 failed on PE_3 by redirecting the requests for these data to their backup copies stored on PE_4. Then, the flags of the intermediate nodes that covered this range of data records will switch in two steps:

1. Step-A: switch the flags of the lowest level of intermediate nodes that is corresponding and closest to these data between P and B;
2. Step-B: modify the flags of the upper level of intermediate nodes having child nodes that are switched in Step-A by the following rules:
   - if all the child nodes of this upper node are modified in Step-A, then switch the flag between P and B;
   - if the original flag in the upper node is the same as that of the modified leaf node in Step-A, then do nothing;
   - otherwise, modify the flag to P/B, which means that some of its child nodes are stored as primaries while others are stored as backups.

According to this process, the node (B, 31, 40) in PE_4 is first changed into (P, 31, 40) because the data records 31–45 in the backup of PE_4 have to act as primary data. On the other hand, the node (P, 20, 31) in PE_2 has its intermediate path to the (B, 31, 40) in PE_4 as shown in Fig. 5. Because it also has part of child nodes located at the local primary without flag switching, its flag becomes (P/B, 20, 31). Similarly, the node (B, 20, 31) in PE_4 is switched as (P/B, 20, 31). Because the "31" in both nodes have primary pointers to the new primary index node (P, 31, 40) in PE_4, requests for the data in the range of 31–45 will be forwarded to there. Note that, in contrast to the load balancing we describe later, there is no new backup data generation in the system, thus the "update" on the new primary does not write to the backup. However, after the failed PE_3 is exchanged with a new one, the new backup's location will be registered.

We verified that this failover process is much quicker than that of IndepIndexCDR, and a comparative experiment is presented in Sect. 6.4.

Table 1 Load before balancing.

	PE_1	PE_2	PE_3	PE_4
Access frequency	α	2α	α	α
Amount of data in primary	P_1	P_2	P_3	P_4
Amount of data in backup	B_4	B_1	B_2	B_3

Table 2 Load after balancing.

	PE_1	PE_2	PE_3	PE_4
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	$\frac{3}{4}B_4$	$B_1 + \frac{3}{8}P_2$	$\frac{5}{8}B_2 + \frac{1}{2}P_3$	$\frac{1}{2}B_3 + \frac{1}{4}P_4$

4.2 Scalability in CompIndexCDR

Load skew in a value-range partition greatly degrades system scalability [21], and balancing the skew by data migration takes a long time and destroys the efficiency of load balancing [35]. A declustering replication scheme [16] only provides the capability of balancing skew without data migration. As far as we know, no practical implementation of this capability has been provided. Moreover, the previous works on chained declustering required a centered metadata server, which decreases system availability. In contrast, in CompIndexCDR, no center node exists and the backup is able to share the workload with its primary PE by a simple modification.

Let us consider an example of the four nodes in Table 1. The load can be balanced without data migration if the query frequency for data on PE_1, PE_3 and PE_4 is α, while the query frequency for data on PE_2 is no more than 2α. 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, because many methods have been proposed for obtaining this information of any PE without any centered nodes, and [37] is one of them which is suitable for the Fat-Btree.

First, we assume that the workload consists of "all-read" transactions. The balanced placement is shown in Table 2. Our load-balancing process involves switching the primary and backup locations on these nodes by exchanging the flags from P to B, or B to P, in the corresponding adjacent PEs. Thus, requests are forwarded to the new primary location by following the index pointers with flags switched to P.

For simplicity without losing generality, we give expressions used to calculate the amount of load transfer. We assume a chained declustering system with n nodes, where PE_0 has a skew load of $X(\leq 2)$ while all the other nodes have the same load of l. An evenly balanced load in the system will be $\frac{x-1+n}{n}$, thus PE_0 will transfer a load of (x-$\frac{x-1+n}{n}$) to its neighbor and retain the average load $\frac{x-1+n}{n}$ in itself. A similar process will be carried out across the cluster, in which PE_i will transfer a load of $\frac{(x-1)\times(n-i-1)}{n}$ to its neighbor $(1 \leq i \leq (n-1))$. Note that the balanced access load on the primary of each PE is the same, in Table 2, $\frac{5}{8}P_2$ in PE_2 and $\frac{3}{8}B_2$ in PE_3 are hot data with an access frequency of 2α; thus, the skew is balanced with each PE having the same load of $\frac{5}{4}\alpha P$.

Regarding "all-update" transactions, our method still balances the skew with the same placement as that in Table 2. Note that each piece of data still retains a copy of its backup by changing the "old primary" into a "new backup" in its neighbor, for example, $\frac{3}{8}P_2$ in PE_2. Similarly in Sec. 4.1, all the "updates" on $\frac{3}{8}P_2$ will be forwarded through an intermediate path to PE_2 or PE_3, then the "updates" are carried out on $\frac{3}{8}B_2$ at PE_3, and the nWAL manager of PE_3 sends the log to the backup PE_2. Because the backup data are always maintained in the system, future failures are still tolerable.

It is obvious that update transactions will update both the primary and the backup; thus, the amount of transactions on each node in our system will not change even if the placement in Table 2 is adopted when the transactions are "all-update". However, because the backup is updated asynchronously in our system, the overhead on the backup part can be alleviated if we decelerate its update rate and preserve unexecuted updates as nWAL record until the overhead is remitted, which has been verified to be the most efficient method for data synchronization in chained declustering systems [36]. Therefore, small load skews (i.e., a skew of 2α 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 in these cases, the proposed method is able to halve the skew immediately and reduce the data migration cost. This is because, even without further load transfer to evenly balance the skew across the cluster, the skewed node is able to share half of the workload with its neighboring replicas. In addition, during data migration for load balancing, the performance of the skewed node will be further decreased; however by sharing the migration load with replicas, this migration cost is also reduced on the skewed node.

4.3 Index Maintenance in CompIndexCDR

In CompIndexCDR, ensuring the index consistency between primary and backup becomes more important. In contrast to IndepIndexCDR, maintaining index consistency seems difficult in CompIndexCDR, because there are more overlapped intermediate index nodes and intermediate paths requiring the consistency maintenance of any neighboring nodes. However, this task can been divided into two phases. In the first phase: potentially conflicting transactions that may cause the modification of the same intermediate index nodes are avoided from being executed concurrently, neither in the primary nor the backup. In the second phase, the possible modifications on overlapped index nodes are propagated and synchronized in all participating PEs.

In the first phase, it appears that the backup index should start the same IX/X-latch-requiring process that occurs on the primary index at same time to avoid the concurrent execution of any transactions that may cause conflicts on the intermediate index nodes at the backup PE. However, this is not necessary because the primary and backup contain the same data and their corresponding subFat-Btree index structures are also the same. Any conflicting transactions in the X-latch-requiring process on the primary index also conflict on the backup index when these transactions are carried on the backup data for primary-backup data synchronization. Therefore, the original concurrency control method [23] is still effective in CompIndexCDR. In addition, the same index structures on the primary and the backup data is guaranteed by executing those conflicting transactions in same order and using the atomic commit protocol [24].

In the second phase when a transaction successfully obtains all the X latches required, it sends the nWAL and starts the modification locally as a *host* PE. All the participating PEs, including the backup, split their specified node and send back the pointer of the newly created node to the *host* PE. The *host* PE manages the sets of pointers received and propagates new pointer information to update them in the participating PEs and backup index. Note that in the original Fat-Btree, interleaved index modification occurs at a very low rate. Only the intermediate index nodes that have a pointer to a neighboring node require this synchronization. In contrast, in CompIndexCDR, as shown in Fig. 5, the intermediate paths are maintained on each level of the index nodes, thus causing higher synchronization rates between the primary and backup indices during updates, which means that the primary and backup are not able to be executed asynchronously.

However, the maintenance of index nodes at every level is not necessary in practice. To support failover in CompIndexCDR, maintaining only the first-level intermediate path is sufficient to transfer the load between the primary and the backup, as shown in Fig. 5. In addition, high-level intermediate nodes have a lower index modification and synchronization cost during updates on the primary and backup. Maintaining a deeper interleaved path is only meaningful for higher load-balancing flexibility. Because at a greater depth, the parent index node at primary points to a greater *fanout* of child index nodes for the same amount of backup data, thus more meticulous load-balancing ability is possible by redirecting the skewed load to the backup by modifying a precise number of flags of intermediate paths. Theoretically,

if there are sufficient tuples, for the depth h of the intermediate paths we maintained, in an N-node system, the number of available paths (index pointers) available from a h node level in the primary to a level $h+1$ node in the backup is $\frac{(fanout^{h+1})}{N}$. In addition, the primary and backup only have to execute index node synchronization concurrently when the current update causes the SMO of index nodes higher than h. Otherwise, the backup can be updated asynchronously since there is no need to synchronize the SMOs with the primary when the same update is executed. Therefore, during the X-latch-requiring process, when the primary only requires the X latch on the index nodes below level h, it will not force its backup to commit the same transaction before itself.

In our CompIndexCDR implementation, we maintain two levels of intermediate paths with the index fanout number equal to eight. The smallest transformable load unit between the primary and backup is about $\frac{1}{8}$, even when N is as large as 64. This unit is transferred by modifying any one of the flags of an intermediate path between the primary and backup. The experimental results in Sect. 6 show that the cost of our method of index maintenance does not significantly decrease system throughput, while achieving high scalability and availability.

5. Details of System

We provide some detailed descriptions of our system here.

5.1 System Architecture and Query Processing Flow

In CompIndexCDR, the database is distributed by storing the data pages into the leaf nodes of a Fat-Btree, which is managed by multiple PEs. Each PE is responsible for processing the corresponding tuples stored in it. As shown in Fig. 6, the data page located in each node is stored at the leaf nodes of the local subFat-Btree. The buffer manager (Buf.Mgr.) is in charge of importing and exporting these data pages between the memory and the Fat-Btree. The postmaster is in charge of receiving and replying to clients' requests. The Pgsql backend (PgsqlBE) and communication manager (Comm.Mgr.) are in charge of fetching the local data page and the communication with other PEs, respectively. The primary/backup register (P-B Reg.) is in charge of recording the primary and backup locations and intermediate-index-node flag management during failover or load balancing. The Fat-Btree manager (FBT.Mgr.) is in charge of traversing the local Fat-Btree and controlling Comm.Mgr., PgsqlBE and P-B Reg.

The query process is listed below, assuming that PE_m receives the request and PE_n stores the target data, as in Fig. 6.

1. Step 1. The postmaster on PE_m extracts the key upon receiving the request and traces this key value in the index by using FBT.Mgr. and forwards the request to the target PE by using Comm.Mgr.
2. Step 2. When the client's request arrives at PE_n, PgsqlBE is launched to process it. If the request is not "select", goto step3; else goto step4.
3. Step 3. FBT.Mgr makes an inquiry of P-B Reg. to check if there are changes in the primary and backup addresses of the target page and to choose the intermediate index to be used. The nWAL is sent to the backup node by the local Comm.Mgr and waits for the reply. The Comm.Mgr on backup node registers the nWAL in the local nWAL manager and replies to PE_n. PE_n waits until it receives the response, then goto step4. The nWAL manager in the backup node starts synchronizing those corresponding intermediate index nodes.
4. Step 4. PgsqlBE first accesses the buffer, which caches the most recently accessed data page using Buf.Mgr, to search for the target page. If it fails, the target data page is fetched by scanning the local Fat-Btree.
5. Step 5. The target page is loaded into the buffer to obtain the target tuple from the page. The result is returned to the client through the local Postmaster.
6. Step 6. Write back the page.

The pages in the buffer that have been modified are marked with checkpoints, and Buf.Mgr modifies the Fat-Btree when these pages are removed from the memory.

5.2 Compound Fat-Btree Handling

As its characteristic of range partition, our compound Fat-Btree doubles the data storage range in each PE and indexes the overlapped data with adjacent PEs. As shown in Fig. 7, in contrast to the intermediate nodes of the previous Fat-

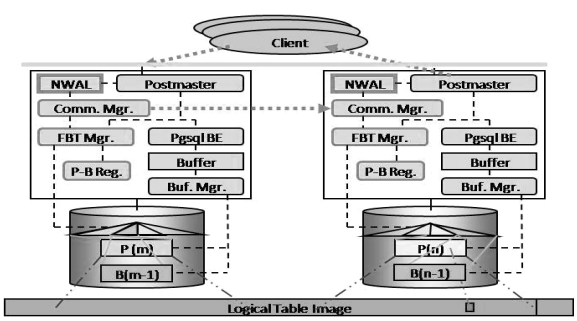

Fig. 6 Details of CompIndexCDR system.

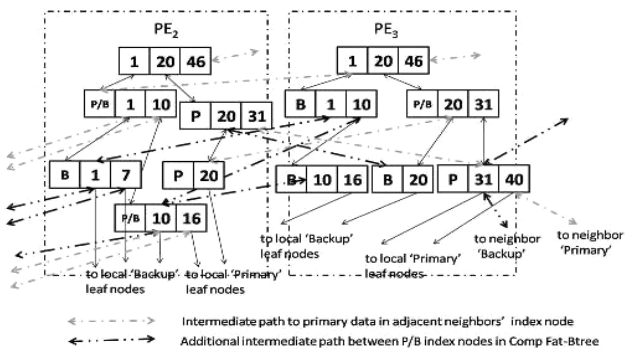

Fig. 7 Details of compound fat-Btree between PEs.

Btree, shown as gray single-dot-dashed lines, the compound Fat-Btree must also maintain new links, as shown by black double-dot-dashed line, to form an intermediate path from the primary data to its backup in other PEs. For example, index node (20) at (P, 20, 31) in PE_2 now has two pointers, one to its local leaf node, the other links to the replica leaf node locates in PE_3 and points to the backup data there; however, index node (31) need not maintain this extra pointer, the original intermediate path in Fat-Btree is sufficient.

"P-B Reg" is used to manage the flags. The logic primary and backup access paths are changed by exchanging "the flag" in these pointer records. Note that the lowest granularity of the new primary and backup is not modified tuple by tuple but as records for several continuous data ranges under some intermediate index nodes. By modifying "the flag" in these intermediate index nodes, a range of data stored as the "primary" or "backup" in each PE can be dynamically exchanged. However, so as not to disarray the replication placement, all the PEs that contains the replicas of the modified intermediate index node should reset "the flag" simultaneously, such as the neighbor PEs pointed by the pointers on the edges of Fig. 7. Therefore, each data has only one copy of primary and backup in the system.

5.3 Load Balancing and Failover Algorithm

Because of the limitation to the length of this paper, we provide the pseudo-code for the load-balancing and failover algorithms here.

The load-balancing algorithm in CompIndexCDR is shown in Fig. 8. First, a primitive method is used to find the skews in the system. This method continuously forwards a list that contains the volume of recent queries in every PE. Each PE places its recent query volume into its corresponding entry in the list and forwards the list to the next PE. If the volume on any PE exceeds some threshold above the average amount of queries in the list, the PE starts load balancing.

For example, consider the balancing of the skewed load as in Table 1. PE_2 receives the load list and calculates the average load from the records in the list as $\frac{1+2+1+1}{4}$. We assume that the threshold here is $\frac{1}{2}$. Because $(\frac{1}{2}+\frac{5}{4})$ is less than its local load of 2, PE_2 starts load balancing. In this case, because $2 < \frac{2(n-1)}{(n-2)}$ (n = 4), the skew can be balanced without data migration. Then a new primary placement *pp*[n] is calculated using the mathematical expression provided in Sect. 4.2, and the same result as instructed in Table 2 is obtained. Finally, all the PEs in *pp*[n] are notified to switch the role of the corresponding data between the primary and backup by sending the corresponding data range in the related "P-B Reg.".

The failover algorithm in CompIndexCDR is shown in Fig. 9. As stated in Sect. 2.4, we assume that the failure can be detected in the system and that all the remaining nodes are notified once the failure is found. The replica node of the failed node acknowledges the failure message and starts the failover process. By referring to the load transform expression in Sect. 4.2, the new amount of primary data is calculated and modified by a similar process using the load-balancing algorithm above. The current PE sends the failover message to start the process on its replica PE, with the parameter *j* transferred. The last PE (another neighbor of the failed PE), will stop the process since the value of *j* that it receives is larger than *N*.

```
Dynamic Load Balancing
begin
 Let op, lp and rp be the target, left and right PE, respectively;
 Let pdm[i] be the amount of primary data 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 + Threshold then
   Calculate the new primary placement pp[n];  // if this skew can be balanced 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. 8 Algorithm for handling skews.

```
Automatic Failover
begin
Let PE_f, PE_{f+1} be the failed node and the right PE(replica), respectively;
Let pdm[i] be the amount of primary data on node i;
Let N be the number of remaining nodes in the system;
Failure is discovered and broadcast to all nodes; / / with failed node ip info.;
PE_{f+1} acknowledges the failure of its primary and starts the failover process on itself;
  Calculate the amount of data it needs to transfer:
      using expression in Sec. 4.2 with the parameter j=0 for new pdm[f];
  if (j<N);  Transfer the failover msg to its replica PE_{f+2} with j+=1;
  Exchange the local primary data with the process in Dynamic Load Balancing (lines 13-28);
PE_{f+2} acknowledges the failure msg from PE_{f+1} and start the failover process on itself;
  Calculate the amount of data it needs to transfer:
      using expression in Sec. 4.2 with the value of j it received for the new pdm[f+1];
  if (j<N);  Transfer the failover msg to its replica PE_{f+2} with j+=1;
  Exchange the local primary data with the process in Dynamic Load Balancing (lines 13-28);
.....
PE_{((f+n-1)mod n)} acknowledges the failure msg from PE_{f+1} and starts the failover process on itself;
  Calculate the amount of data it needs to transfer:
      using expression in Sec. 4.2 with the value of j it received for the new pdm[(f+n-1)mod n];
  if (j<N);  Transfer the failover msg to its replica PE_{f+2} with j+=1; // the msg will not be sent
  Exchange the local primary data with the process in Dynamic Load Balancing (lines 13-28);
end
```

Fig. 9 Algorithm for failover.

6. Experiments

It is difficult to provide direct comparisons with former studies because, to the best of our knowledge, no previous work exists that manages both the primary and backup within a single directory or analyzes the failover efficiency issue in a chained declustering database or even partial-replicated databases. Therefore, we compare our work with the well-known Postgres-R (PG-R), MySQL+proxy and HBase replication DBMSs instead. The naive straightforward structure IndepIndexCDR is also evaluated to demonstrate the higher scalability and availability of our proposed approach.

6.1 Experimental Setup

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.

6.2 Comparison with Other Full/Partial Replication Databases

We first compare the performance of CompIndexCDR with multiple-replication DBMS PG-R, MySQL+proxy and HBase to assess the performance of our system.

PG-R [39] was the first significant research prototype with fully functional database replication mechanisms based on an open-source database. MySQL+proxy is another full-replication database with an independent proxy node to provide load-balancing ability. HBase is an open-source, distributed database modeled after Google's Bigtable; it provides BigTable-like capabilities and has attracted much interest among the distributed database community. Although these systems use different configurations for their replication strategies, what we focus on here is a comparison between full/partial replication with our approach.

Here, full replication, means that the data stored in the the whole distributed system are fully copied into every individual node in the system such as the data distribution in Postgre-R and MySQL+proxy. Because each node has a full copy of data, *read* transactions on each node are able to be responded to immediately without remote communication. However, such systems suffer from a heavy update cost of the synchronization of all the copies in the system.

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

Figure 10 (a) shows the effects of the update cost discussed above. In these experiments, we adopted exactly the same configuration as that in [39]. The five or ten nodes each store 8,000 tuples, and 40 clients are evenly distributed to query the systems with different workloads as shown in the figure. Although Postgres-R (PG-R) has better "all-read" performance than CompIndexCDR, its worse "all-update" performance makes PG-R have the lowest overall performance. This is because even a moderate number of updates saturate seach node with the "writeset" [39].

In contrast to PG-R, MySQL+proxy adopts a centered proxy node for load balancing and read/write splitting. The queries are carried out only on slave nodes, and updates are started from master nodes and then propagated to all the slave nodes asynchronously for data consistency. This centered metadata management method is an easy and efficient method for load balancing; however, it limits the scalability as shown in Fig. 10 (b). In this two- and four- node configuration, we adopt MySQL5.0+proxy0.8 with each node containing 10,000 tuples; increasing the number of client threads obviously saturates the proxy node's memory and the CPU capability with the load transfer and split overhead, thus resulting in very limited scalability.

According to the above experimental results, the full-replication system is only useful for query-intensive applications. It is also inefficient when adopting centered metadata management methods for transactions or consistency control.

The recently released HBase-0.20.2 is adopted to make the third comparison. It is a type of partial-replication system in which the whole data stored in the system are split into several fragments and each node contains part of these fragments with some degree of data replication. It runs on our cluster system which is based on Hadoop-0.20.1 and jdk1.6.0.18. 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 that in our system. We choose HBase for this comparison because it is one of the few open-source systems that also support partial partitions similarly to CompIndexCDR. Although HBase is designed for superior unstructured data retrieval in (key, value) pairs without range-query support, it is still meaningful to compare the read/write scalability with row-based relational DBMS like our system, because both sys-

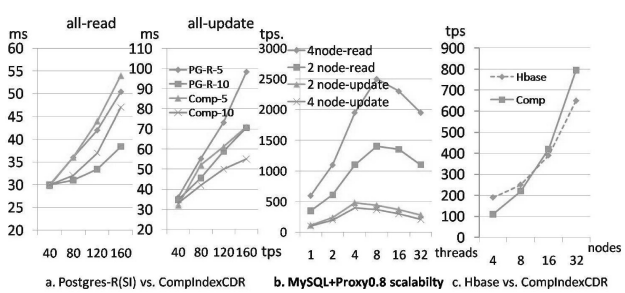

Fig. 10 Performance comparison of CompIndexCDR with postgres-R and Hbase.

tems should efficiently support exact queries and updates. The number of the "datanode" in our experiment is limited to 32 because the software in current computation clouds typically uses significantly fewer datanodes. For example, the default number of datanodes in Amazon applications is 20 [38].

In the experiment, the dataset contains 10,000 tuples with 4 KBytes of data in each tuple with two columns. There is one client on each node to perform "all-read" transactions to access tuples with their key value. The results are shown in Fig. 10 (c). We do not provide "all-update" results here because they are almost the same as those shown in Fig. 10 (c). The figure shows that the column-oriented DBMS with BigTable has better performance for a small number of nodes. However, our system outperforms it with increasing number of nodes owing to its better scalability.

We can conclude from these experiments that for applications that involve even a moderate ratio of updates and require high scalability, CompIndexCDR outperforms the full-replication PG-R, MySQL+proxy and partial-replication HBase systems. The full-replication solution is only superior for "all-read" cases. For HBase which also adopts partial replication, the scalability decreased more rapidly than our system with increasing number of nodes. This is because it is based on the centered namenodes for metadata management in the HDFS.

6.3 Throughput Comparison

We compare the throughput and verify the scalability of our two structures taking into account their different index-traversing costs. The cost mainly consists of intranode and internode traversing costs. The former is the cost of traversing a local index; this cost is greater for larger datasets. The latter is the cost of locating remote data as well as maintaining the distributed index; the low internode cost is one of the strengths of our index. Because the throughput only reflects the total traversing cost and huge datasets have an increased intranode cost in both systems, a small dataset will show the difference in the internode cost more clearly. In addition, a small dataset will cause a high number of index-latching conflicts and require more intermediate-node synchronization operations. To evaluate the performance of CompIndexCDR more clearly, we initialize each PE with a small dataset of 10,000 tuples, with each tuple having 134 bytes.

In the experiments, each PE receives requests from its client nodes simultaneously; the key in each request is generated randomly from the whole data range. For the case of 64 PEs, these queries are as follows: key = random(1, 640000); select* from table where id = key; update table set value += 1 where id = key. We do not use complex queries here because we focus on the index cost. In addition, each client sends 20 transactions in series to a PE and has a private thread on the target PE to process his requests.

Figures 11 (a), (b) show the scalability of our systems with 32 clients on each node. Figure 11 (a) illustrates the all-read case. The performances of these two structures are

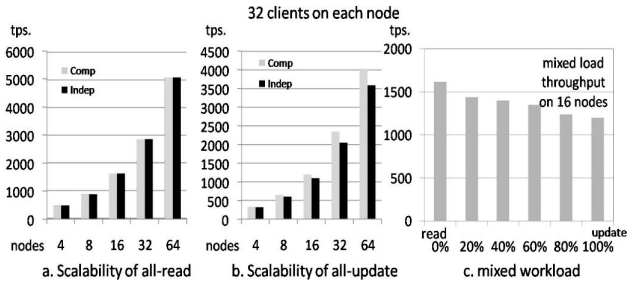

Fig. 11 Comparison of scalability.

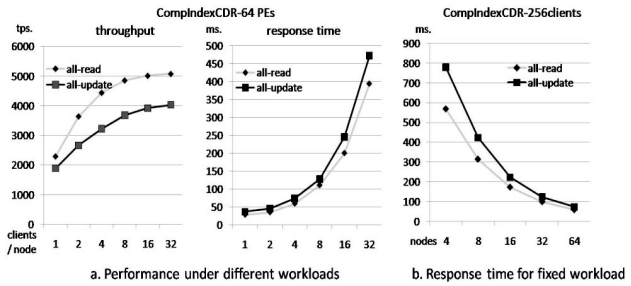

Fig. 12 Performance in the case of different and fixed workloads.

almost the same and their scalability is reasonably 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 increased number of remote transactions in the larger system. Figure 11 (a) also shows that although the index size of CompIndexCDR is twice that of IndepIndexCDR, their throughputs are almost the same. The reason for this may be that the backup is not accessed; thus, both systems have similar memory hit rates. Figure 11 (b) shows that CompIndexCDR has better scalability for an "all-update" workload. This may be because the backup in IndepIndexCDR is updated by an independent B-tree, while it is accessed within one compound index in CompIndexCDR, therefore the index is not switched in memory. We perform a mixed read/write workload experiment on a 16-node configuration. The result in Fig. 11 (c) verifies that the mixed transaction performance is generally between those of "all-read" and "all-update" throughput. The throughput decreases when the percentage of updates increases in the case of a mixed workload. We do not provide further mixed workload results in other experiments, since this experimental result can be used as a reference.

Figure 12 (a) shows the throughput and response time of CompIndexCDR in the case of 64 PEs with various numbers of clients per node. We omit the results for other sizes, because they are almost the same. As the figure shows, the throughput of CompIndexCDR increases with the number of clients. When each node has 32 clients, the growth of throughput is moderate at a stable value that is defined by the limitations of the CPU and I/O performances in our cluster system. On the other hand, the response time also increases with the number of clients, which is normal in DBMSs and

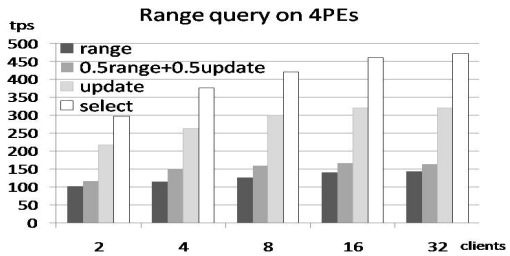

Fig. 13 Performance of range query.

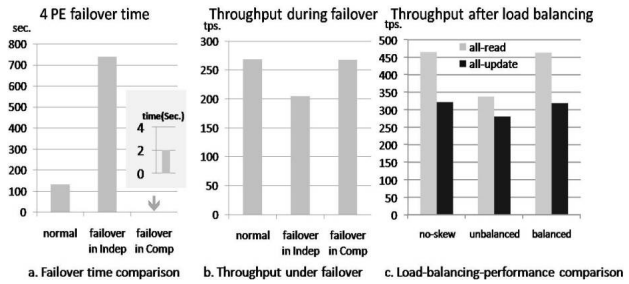

Fig. 14 Response time and failover and load-balancing performances.

is usually solved by adding more PEs to distribute the workload. Figure 12 (b) shows the trend of the response time as the number of PEs increases. For a fixed workload with a total of 256 clients, the response time is effectively reduced because these requests are evenly distributed after new PEs are added.

The range-query performance in the current system is examined using the results shown in Fig. 13. In this four-node configuration experiment, each node has from 2 to 32 clients. Every client starts a range query to retrieve 30 tuples, on average one at a time, or an exact update request immediately after his previous request is responded to. As shown by the all-range query results, although its throughput is less than the exact query, it fetches tuples about ten times quicker than exact query. However, when range queries and updates are executed concurrently in equal proportions, the throughput does not exhibit the same ratio of integration because of the time lag between conflict read/write operations. In addition, we note that the gap lock is currently not yet implemented in our concurrency control method; thus, phantom records are not protected in the case of querying a range of tuples located in multiple leaf nodes concurrently with update transactions. We plan to resolve the above problems by implementing a gap lock or multiple-version-based snapshot isolation to improve concurrency in our future work.

6.4 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 the data for IndepIndexCDR, and we adopt the failover described in Sect. 4 for CompIndexCDR. In Fig. 14 (a), "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. "Failover in Indep" is much larger than "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 s, which is much shorter than the time for IndepIndexCDR. We also examined the throughput during failover. Figure 14 (b) shows that the throughput decreases only slightly during failover.

We do not provide a comparison of failover performance between our system and Postgres-R or MySQL+proxy. Because these two systems adopt the full-replication scheme. Postgres-R maintains all the replication locations in each PE whose location is transparent to users by adopting a *JDBC driver* for data connection; MySQL+proxy maintains the replication locations using the centered node *Proxy* to balance and transfer user requests. Thus, both the JDBC driver and the Proxy Postgres-R provide failover ability by transferring the connections from failed nodes to any remaining nodes. These methods work but introduce the SPoF of the centered nodes in both systems. Here we compare IndepIndexCDR with CompIndexCDR to evaluate the performance for failover without the participation of centered nodes. We assume there is only one node failure, and failover is viewed as the complete construction of a Fat-btree on surviving replicas in IndepIndexCDR and the redirection of the access path in CompIndexCDR.

6.5 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%, i.e., the same skew as that in Sect. 4.2. Figure 14 (c) shows that this skew decreases the throughput in "all-read" and "all-update" by 25% and 15%, respectively. After using the balancing method in Sect. 4.2, the throughput in "all-read" is almost the same as that in "no-skew", and the "all-update" is also much improved, which verifies the effectiveness of load-balancing as argued in Sect. 4.2.

We do not provide a comparison of the load-balancing performance of our system with that of other database systems, because the Postgres-R used for comparison in this paper does not yet support load balancing. Many full-replication DBMSs including MySQL+proxy and HBase, provide load-balancing ability with good performance. However, they use either a center node proxy, which is verified to cause a bottleneck and an SPoF that itself reduces system scalability and availability, or data migration, which also introduces a high cost during the balancing process. Unfortunately, all other open-source partial-replication DBMSs, as far as we know, do not support "load-balancing".

7. Related Work

To reduce the synchronization cost in a synchronous replication system, different database isolation techniques other than serializability have been introduced [32], [33] to increase the concurrency of transactions; different commit protocols other than 2PC have been proposed [24], [34] to shorten the commit time and reduce the messages traffic; and the "write set" concept has been introduced to reduce the synchronization cost of a given update in multiple slaves [34]. Moreover, partial replication has also been introduced to reduce the synchronization cost and increase the efficiency of disk space usage [5], [34]. However, synchronous replication databases, for example, the well-known Postgres-R, still have a high synchronization cost as shown in Fig. 10; although partial-replication databases, which require prior knowledge of the data distribution over the replicas, maintain at least one site containing the whole database, even a few updates cause many updates which force other transactions to be aborted [40]. The asynchronous replication strategy, which trades consistency for performance, is an alternative. However, it causes data loss during a failover.

To rapidly recovery damaged data, the primary-backup approach has been studied. Teradata's interleave declustering strategy can provide an immediate load and space balance after failure [41], but there is a high probability of data loss when the clustering unit becomes large and it is only able to work with data sets that are hash-partitioned. Gamma [42] uses the chained declustering strategy to handle range partitioning and reconcile HA with balancing loads during failure. Hot mirroring [43] and HP's AutoRAID [44] suggest the possibility of a hybrid method based on parity-based and primary-backup approaches.

To provide an efficient access method for a declustering database, various parallel index structures have been proposed [11]–[13]. In all these structures, every PE can store data, perform similarity queries simultaneously, and autonomously split and distribute data over several PEs at any time, thus the system scalability is improved. In addition, hotspots or centred nodes are avoided, thus the system availability is also improved.

Considerable work has been carried out on providing dynamic skew handling with index structures. In [45], two alternatives are studied for performing the necessary index modifications, called one-at-a-time (OAT) page movement and BULK page movement. However, both techniques depend on conventional B-tree algorithms, which may lead to a considerable SMO cost [13]. [14] presents a fast convergence technique for handling access load skew in range-partitioned databases over share-nothing parallel systems. The RING method [15] using a circulating aBtree is an elegant solution for reducing the quantity of data migration during load balancing.

Unfortunately, all these works mainly focus on only one or two targets among reducing the synchronization cost, rapid recovery, efficient access and dynamic skew handling, and they do not discuss the access method and the usage of the replicas to promote scalability and availability.

In addition, other parallel indices, which may be alternatives to our compound treatment concept, are not as suitable as the index used in our method; for example, the "FirstTierIndex" of the "aB+tree" [13], [15] must be updated in all the PEs immediately any data reallocation. Otherwise, many requests will need to be redirected in the system, resulting an expensive overhead when the system is scaled up. "Generalized Hyperplane Tree" in [11] exploits the P2P paradigm and presents an instance of "Address Search Tree (AST)" in every node in P2P systems to record the location where the data resides; thus, when the roles of primary data and its replicas are switched during skew balancing as in our system, the new location of that primary data must be notified to all the "ASTs" in every node. Otherwise, the requests will still be sent to the original (skewed) node.

On the other hand, DHT-based structures with range-query ability provide some other alternatives to our compound treatment configuration. Mercury [48], which is a DST-based variant, adopts circular overlays and stores data continuously in them to support multiattribute range queries. However, besides to predecessors and successors in its own overlay, it needs to maintain extra links to other overlays. Therefore, we did not choose it for compound treatment because load balancing may become much more complicated for link management when duplicated overlays are introduced. The skip-list-based overlay Skip Graph [49], is another choice. Unlike the original DHTs, a skip graph does not require randomized hash functions and is therefore capable of range queries, but the load balance between nodes becomes a serious problem. A proposed solution is to either increase the number of virtual servers or use an additional Skip Graph to track the load on each node. Our proposed method for the compound management of the primary and backup using a single-directory structure may also be effective in improving its load-balancing ability. However, the Skip Graph focuses on the skewed storage load, and its performance under a skewed accessing load is unclear; thus, we did not apply our strategy based on the Skip Graph.

8. Conclusions and Future Work

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

CompIndexCDR has been proved to outperform other replication database solutions, including Postgres-R(SI), MySQL+proxy and Hbase, both in scalability and throughput. Moreover, it achieves better availability and scalability than ordinary DBMS, which treat the backup separately

from the primary. We constructed IndepIndexCDR to make this comparison, and found that compound treatment has many benefits without diminishing throughput and scalability. In particular, failover costs much less time and has little effect on ordinary queries, and load skew is able to be dynamically and rapidly handled with less or no data migration. Furthermore, CompIndexCDR retains the virtues of IndepIndexCDR, such as no centralized server, low synchronization cost and range-query ability.

As the CPUs and I/O performance of the PC cluster we used in the experiment were slower than current up-to-date systems, we consider that CompIndexCDR can achieve better performance very easily if the hardware is upgraded. Thus, we conclude that CompIndexCDR is a very good choice for parallel databases which require scalability and availability.

In this work, we have not considered the time required for restoring a backup after a failure by the data on the neighbor. The adaptive overlapped declustering method proposed in [46] is a good means of reducing the restoration time. As future work, we plan to combine adoptive overlapped declustering with CompIndexCDR. In addition, we maintained only one copy of replicas in our system under the assumptions that there is only one failure on the same data at the same time. To achieve higher reliability, multi-replica solutions should be adopted. However, as discussed in this paper, the high synchronization cost of multireplica schemes should be avoided. We plan to provide multiversion management for these replicas to decrease the synchronization cost by delaying the updates on part of the replicas.

Acknowledgements

Part of this research was sponsored by CREST of Japan Science and Technology Agency (JST), and MEXT via Grants-in-Aid for Scientific Research #19024028 and #22240005.

References

[1] M. Luo, A. Watanabe, and H. Yokota, "Compound treatment of chained declustered replicas using a parallel Btree for high scalability and availability," Database and Expert Systems Applications (DEXA'10), LNCS, vol.6262/2010, pp.49–63, 2010.

[2] S. Gilbert, "Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services," SIGACT News, vol.33, no.2, pp.51–59, 2002.

[3] J. Gray, P. Helland, and P. O'Neil, "The dangers of replication and a solution," Proc. ACM SIGMOD, pp.172–182, 1996.

[4] H. Yu and A. Vahdat, "The costs and limits of availability for replicated services," ACM Trans. Compt. Syst., vol.24, no.1, pp.70–113, 2006.

[5] E. Pacitti, M.T. Ozsu, and C. Coulon, "Preventive multi-master replication in a cluster of autonomous databases," Proc. Int. Conf. Euro-Par, pp.318–327, 2003.

[6] A. Silberstein, "Efficient bulk insertion into a distributed ordered table," Proc. ACM SIGMOD, pp.765–778, 2008.

[7] P. Bernsteini, "Data management issues in supporting large-scale web services," IEEE Data Eng. Bull., 2006.

[8] F. Chang and Dean, "Bigtable: A distributed storage system for structured data," Proc. 7th Conf. on USENIX Symp. Operating Systems Design and Implementation, vol.7, pp.205–218, 2008.

[9] S. Rhea, D. Geels, T. Roscoe, and J. Kubiatowicz, "Handling churn in a DHT," Proc. Annual Conf. on USENIX Annual Tech. Conf. (ATEC '04). USENIX Association, Berkeley, CA, USA, 2004.

[10] R. Bayer and E. McCreight, "Organization and maintenance of large ordered indices," Mathematical and Information Sciences Report No.20, Boeing Scientific Research Laboratories, 1970.

[11] P. Zezula, G. Amato, V. Dohnal, and M. Batko, "Parallel and distributed indexes," in Similarity Search: The Metric Space Approach, pp.161–181, Springer USA, 2006.

[12] H. Yokota, Y. Kanemasa, and J. Miyazaki, "Fat-Btree: An update conscious parallel directory structure," Proc. 15th Int. Conf. Data Engineering, p.448, 1999.

[13] M.L. Lee, M. Kitsuregawa, B.C. Ooi, K. Tan, and A. Mondal, "Towards self-tuning data placement in parallel database," ACM SIGMOD Record, vol.29, no.2, pp.225–236, 2000.

[14] H. Feelifl, M. Kitsuregawa, and B.C. Ooi, "A fast convergence technique for online heat-balancing of Btree indexed database over shared-nothing parallel systems," Proc. 11th Int. Conf. Database and Expert Systems Applications, pp.846–858, 2000.

[15] H. Feelifl and M. Kitsuregawa, "Ring: A strategy for minimizing the cost of online data placement reorganization for Btree indexed database over shared-nothing machines," Proc. 7th Int. Conf. Database Systems for Advanced Applications, pp.190–199, 2001.

[16] H. Hsiao and D.J. DeWitt, "Chained declustering: A new availability strategy for multiprocessor database machines," Proc. 6th Int. Conf. Data Engineering, pp.456–465, 1990.

[17] H. Hsiao and D.J. DeWitt, "A performance study of three high availability data replication strategies," Distrib. & Parallel Databases, vol.1, no.1, pp.53–80, 1993.

[18] R.V. Renesse and F.B. Schneider, "Chain replication for supporting high throughput & availability," Proc. 6th Symp. Operation Systems Design & Implementation (OSDI), vol.6, p.7, 2004

[19] B. Seeger and P. Larson, "Multi-disk B-trees," ACM SIGMOD Record, vol.20, no.2, pp.436–445, 1991.

[20] S.-O. Hvasshovd, Recovery in Parallel Database Systems, 2nd ed., Springer, 1999.

[21] D. Dewitt and J. Gray, "Parallel database systems: The future of high performance database systems," Commun. ACM, vol.35, no.6, pp.85–98, 1992.

[22] G. Copeland, W. Alexander, and E. Boughter, "Data placement in Bubba," ACM SIGMOD Record, vol.17, no.3, pp.99–108, 1988.

[23] T. Yoshihara, D. Kobayashi, and H. Yokota, "Mark-opt: A concurrency control protocol for parallel B-tree structures to reduce the cost of SMOs," IEICE Trans. Inf. & Syst., vol.E90-D, no.8, pp.1213–1224, Aug. 2007.

[24] X. Ouyang, T. Yoshihara, and H. Yokota, "An efficient commit protocol exploiting primary-backup placement in a distributed storage system," Proc. 12th Pacific Rim Int. Symp. Dependable Computing, pp.238–247, 2006.

[25] S. Ratnasamy, J.M. Hellerstein, and S. Shenker, "Range queries over DHTs," Intel Research Berkeley, MA, Tech. Rep. IRB-TR-03-009, June 2003.

[26] A. Gupta, D. Agrawal, and A. El Abbadi, "Approximate range selection queries in peer-to-peer," Proc. Conf. Innovative Data Systems Research (CIDR), 2002.

[27] K. Aberer, "P-grid: A self-organizing access structure for p2p information systems," Proc. 9th Int. Conf. Cooperative Information Systems, Springer, pp.179–194, 2001.

[28] J. Gao and P. Steenkiste, "An adaptive protocol for efficient support of range queries in DHT-based systems," ICNP 04: Proc. 12th IEEE Int. Conf. Network Protocols, pp.239–250, IEEE Comput. Soc., Washington, DC, USA, 2004.

[29] H. Zhuge, X. Chen, X. Sun, and E. Yao, "HRing: A structured P2P overlay based on harmonic series," IEEE Trans. Parallel Distrib. Syst., vol.19, no.2, pp.145–158, 2008.

[30] T. Yoshihara, D. Kobayashi, and H. Yokota, "A concurrency control protocol for parallel B-tree structures without latch-coupling for ex-

plosively growing digital content," Proc. 11th Int. Conf. Extending Database Technology: Advances in Database Technology (EDBT), vol.261, pp.133–144, 2008.
[31] T. Haerder and A. Reuter, "Principles of transaction-oriented database recovery," ACM Comput. Surv., vol.15, no.4, pp.287–317, Dec. 1983.
[32] S. Elnikety, W. Zwaenepoel, and F. Pedone, "Database replication using generalized snapshot isolation," Proc. 24th IEEE Symp. Reliable Distributed Systems, pp.73–84, 2005.
[33] A. Fekete, "Allocating isolation levels to transactions," Proc. 24th ACM SIG-(MOD/ACT/ART) Symp. Principles of Database Systems, 2005.
[34] B. Kemme and G. Alonso, "Don't be lazy, be consistent: Postgres-R, a new way to implement database replication," Proc. 26th Int. Conf. Very Large Databases, pp.134–143, Sept. 2000.
[35] R. Arnan, E. Bachemat, T.K. Lam, and R. Michel, "Dynamic data reallocation in disk arrays," ACM Trans. Storage, vol.3, no.1, p.2, 2007.
[36] C.N. Fischer, "Performance evaluation of chained declustering strategies for disk array systems," Proc. UK National Simulation, pp.14–18, Cambridge, U.K., 1999.
[37] A. Watanabe and H. Yokota, "Directory traverse cost based skew handling for parallel data access," Syst. Comput. Jpn., vol.34, no.14, pp.13–22, 2002.
[38] Amazon Web Service, http://aws.amazon.com/elasticmapreduce/, 2010.
[39] S. Wu and B. Kemme, "Postgres-R(si): Combining replica control with concurrency control based on snapshot isolation," Proc. 21st Int. Conf. Data Engineering, pp.422–433, 2005.
[40] J.M. Bernabe-Gisbert, V. Zuikeviciute, F.D. Munoz-Escoi, and F. Pedone, "A probabilistic analysis of snapshot isolation with partial replication," Proc. Symp. Reliable Distributed Systems, pp.249–258, 2008.
[41] Teradata, "DBC/1012 database computer system manual release 2.0," Document No.C10-0001-02, Teradata Corp., 1985.
[42] D.J. Dewitt, H. Hsiao, and R. Rasmussen, "The gamma database machine project," IEEE Trans Knowl. Data Eng., vol.2, no.1, pp.44–62, 1990.
[43] K. Mogi and M. Kitsuregawa, "Hot mirroring: a method of hiding parity update penalty and degradation during rebuilds for RAID 5," ACM SIGMOD Record, vol.25, no.2, 1996.
[44] J. Wilkes, R. Golding, C. Staelin, and T. Sullivan, "The HP AutoRAID hierarchical storage system," ACM Trans. Comput. Syst., vol.14, no.1, 1996.
[45] K.J. Achyutuni and E. Omiecinski, "Two techniques for on-line index modification in shared nothing parallel databases," ACM SIGMOD Record, vol.25, no.2, 1996.
[46] A. Watanabe and H. Yokota, "Adaptive overlapped declustering: A highly available data-placement method balancing access load and space utilization," Proc. 21st Int. Conf. Data Engineering, pp.828–839, 2005.
[47] A. Watanabe and H. Yokota, "Adaptive lapped declustering: A highly available data-placement method balancing access load and space utilization," Proc. Int. Conf. Data Engineering, pp.828–839, 2005.
[48] A.R. Bharambe, M. Agrawal, and S. Seshan, "Mercury: Supporting scalable multi-attribute range queries," Proc. ACM SIGCOMM, pp.353–366, Portland, USA, 2004.
[49] J. Aspnes and G. Shah, "Skip graphs," Proc. 14th ACM-SIAM Symp. Discrete Algorithms (SODA), Jan. 2003.

Min Luo received his B.E. degree from Wuhan University, China, in 2004, and was an M.E.-Ph.D. student at the State Key Laboratory of Software Engineering, Wuhan University, China, from 2004 to 2007. He is currently a Ph.D. student at Tokyo Institute of Technology. He is engaged in research on data engineering and storage systems.

Akitsugu Watanabe received his B.E. and M.E. degrees from Tokyo Institute of Technology in 2000 and 2002, respectively, and is currently a Ph.D. student at the Department of Computer Science, Graduate School of Information Science and Engineering. His research interests include data engineering, information storage systems, and dependable computing.

Haruo Yokota received his B.E., M.E., and Dr.Eng. degrees from Tokyo Institute of Technology in 1980, 1982, and 1991, respectively. He joined Fujitsu Ltd. in 1982, and was a researcher at ICOT for the Japanese 5th Generation Computer Project from 1982 to 1986, and at Fujitsu Laboratories Ltd. from 1986 to 1992. From 1992 to 1998, he was an Associate Professor at Japan Advanced Institute of Science and Technology (JAIST). He is currently a Professor at the Department of Computer Science in Tokyo Institute of Technology. His research interests include the general research areas of data engineering, information storage systems, and dependable computing. He is an associate editor of the VLDB Journal, the chair of ACM SIGMOD Japan Chapter, a trustee member of IPSJ and the Database Society of Japan (DBSJ), a fellow of IPSJ, and a member of JSAI, IEEE, IEEE-CS, ACM, and ACM-SIGMOD.