Towards High Availability and Performance Database Clusters for Archived Stream

Zhengbing Hu
Huazhong Normal University,Department of Information Technology,Wuhan,China
Email:kievpastor@yahoo.com

Kai Du
National University of Defense Technology, Changsha, China
Email: keyes.du@gmail.com

***Abstract*— Some burgeoning web applications, such as web search engines, need to track, store and analyze massive real-time users' access logs with high availability of 24*7. The traditional high availability approaches towards general-purpose transaction applications are always not efficient enough to store these high-rate insertion-only archived streams. This paper presents an integrated approach to store these archived streams in a database cluster and recover it quickly. This approach is based on our simplified replication protocol and high performance data loading and query strategy. The experiments show that our approach can reach efficient data loading and query and get shorter recovery time than the traditional database cluster recovery methods.**

***Index Terms*—High availability, Archived Stream, Database Clusters**

I. INTRODUCTION

Some burgeoning applications have appeared which needs the high availability and extra high performance of data insertion operations. The records of web behavior, such as the records of personal search behavior in search engines, online stock transactions or call details, are the classical archived streams [11]. For instance, Google can improve the users' search experiences based on Personalized Search [3]. In this way, the data to be collected includes the user's web access path, access time of each page, and so on. This information should be written into a large database in a real-time mode and queried repeatedly when the user uses the search engine again. Another example is the record of call details. Every day a telecom company needs to store the users' call details such as the call's start and end time, called number. All of these archived streams applications have the following common characteristics:

- A round-the-clock Internet company needs a high availability of 24*7. However high availability is a great challenge for a large-scale Internet company like Google since a large number of equipments are needed to archive streams which always leads to low reliability of the whole system.
- High-rate data streams need a high performance and near real-time record insertion method. Google processes about 4200 requests every second [4] and needs a high performance insertion program to record all the users' behavior.
- The recorded data can be viewed as historical data because it will not be updated any more but only be queried repeatedly after being stored. Furthermore the strict data integrity is always not needed as in a classical DBMS since the loss of a small part data will not influence the statistic query result to the massive historical data.

We call these applications as *log-intensive* web applications. [11] is the first one which optimizes querying on the live and archived streams, but doesn't study the insertion performance and system's availability. [14] studies the availability of an updatable data warehouse filled with less-update data. It optimizes the 2PC and 3PC [2] to reduce the disk IO cost by eliminating the force-write logs. This improves the runtime and recovery performance. However it is still based on the general-purpose 2PC which is not efficient enough to our high-rate insertion-only archived streams. So this paper will design an efficient algorithm based on reducing disk IO and a simpler consistency protocol to archive the high-rate streams.

The first contribution of this paper is to study how to optimize the insertion operations. Writing no online-log and archived-log in databases and committing data in bulk is adopted in our approach. The second one is to provide a simple consistency protocol based on the no-update feature of the data. The third one is to design an efficient recovery method to recover failure replicas and bring them online quickly.

The organization of this paper is as follows: Section II is the problem statement and related work. Section III describes the system architecture, efficient transaction processing and simplified consistency protocol. Section IV introduces the recovery approach; Section V is the experiments; Section VI is the conclusion.

II. PROBLEM STATEMENT AND RELATED WORK

doi:10.4304/jcp.4.12.1332-1339

Let's consider the classical *log-intensive* applications: when the users are accessing the web sites, all the users' behavior may be stored and a group of record items $<R_1, R_2, \ldots R_n>$ are generated at all times. These record items must be real-time stored and be queried by subsequent web access. A high available and efficient system needs to be built for these applications. Database clusters are an effective way to complete this. A database cluster *C* is *m* database servers $N_1, N_2, \ldots N_m$, each having its own processors and hard disks, and running a "black-box" DBMS [1]. The "Read One Write All Available" policy [2] is always adopted. It means that when a read request is received, it is dispatched to *any one* node in the available nodes set $ANS=\{ N_j \mid N_j$ is available node and $1 \le j \le m\}$; when a write request is received, the data items $<R_1, R_2, \ldots R_n>$ are inserted into *all* nodes of the available nodes set *ANS*.

DEFINITION 1. (Failure or Available Node) When a database node meets an instance or media failure, and it can't provide query or insertion function to any data, this node is a *failure node*. Otherwise the node is an *available node*.

DEFINITION 2. (K-safety Fault Tolerance Model) K-safety [Jim] means that if up to K sites can fail, and the system can still continue to service. The minimum number of sites required for K-safety is K + 1, namely in the case where the K + 1 workers store the same replicated data. In a database cluster of $N_1, N_2, \ldots N_m$ with wholly replicated data, it is a *m-1*-safety model.

Based on definition 1, we could divide the cluster $C =\{N_1, N_2, \ldots N_m\}$ into two sets: available nodes set: $ANS =\{ N_j \mid N_j$ is available node and $1 \le j \le m\}$ and failure nodes set: $FNS = C - ANS = \{ N_k \mid N_k$ is failure node and $1 \le k \le m \}$. Then what should be completed is as the following: 1) Efficiently insert the record items $<R_1, R_2, \ldots R_n>$ into all nodes in *ANS* and query any node in *ANS*. 2) Efficiently recover the nodes in *FNS*. 3) Bring the recovered node in *FNS* online and change both *ANS* and *FNS*.

In [8], bulk loading the data into databases is adopted to optimize the insertion performance, however they didn't touch upon the topic of high availability and the recovery procedure. About 2) and 3), there is much related work on it. The primary/secondary replica protocol [20] in commercial databases [21,22,23]ships updates logs from the primary to the secondary and this way can't support up-to-date read queries on the secondary replicas and decrease the insertion performance because of the frequent IO access in our *log-intensive* web applications. The 2PC [2] can keep all replicas up-to-date, but has poor runtime performance for its runtime force-writes logs and poor recovery performance based on complex ARIES [7, 14].

In order to avoid force-writes, ClustRa [13] uses a neighbor write-ahead logging technique, in which a node redundantly logs records to main memory both locally and on a remote neighbor which in fact writes distributed logs; HARBOR[14] avoids the logs thoroughly by revising the 2PC protocol, but the revised 2PC protocol is still too complex to these insertion-intensive and no-update applications. [15, 16] is not based on 2PC and propose a simple protocol, but it needs to maintain an undo/redo log.

The object of this paper is to design an efficient integrated approach to solve the problem of high availability and high performance for these real-time *log-intensive* web applications. The basic idea is to insert the data in bulk[1] without online log in databases and set a consistency fence for every table in the data processing phase. And in the recovery phase, based on this simplified data processing mode, a recovery method is designed to get recovery data from other normal nodes or the coordinator.

III. TRANSACTION PROCESSING

In this section, we first introduce the framework to solve the three problems addressed in Section II. Then how to process the two types of transactions-- insertion and query is described. At last a simple replication protocol is discussed.

A. System Framework: Transaction Types and Unique External Timestamp

As is discussed in Section I, in the log-intensive workloads, all the transactions can be classified as two types: insertion and query transactions since there are no update transactions. The insertion means inserting high-rate data into databases. The query means querying the massive non-update history data.

In order to implement the high performance insertions, we adopt the following: 1) Buffer and insert the data into a database in bulk. The experiments show that bulk insertions always outperform standard single insertions by an order of magnitude. 2) Write no online logs in databases for insertions. Since the IO cost is always the bottleneck of databases, the regular force-write logs should be cut for the sake of performance. 3) Insert multiple objects in parallel. Eliminating the dependency of the insertions on different objects could be reached by simply canceling the foreign key constraints. 4) Recovery methods based on no-log must be developed.

According to 1), a coordinator is added upon a database cluster to buffer and insert data in bulk in Fig.1. For every object or table, an *insertion* thread is always running and it inserts the buffered data into all available nodes. Since the coordinator processes the same data more easily than any underlying database, it consumes less CPU and IO. Thus only one thread for one table is enough to a balanced system. For every query request, a *query* thread dynamically starts and stops with that query. By the way, the *insertion* threads will refresh the meta-info *TF* and *ANS* (introduced in Section III.B) about the databases and the *query* threads will read this on time.

Another two mechanisms are designed to implement fault tolerance and consistency protocol. The first is the

[1] The statistical query result of the massive log data will not be influenced by the small part of data buffered in bulk. It is a basic assumption of the *log-intensive* applications in Section 1.

unique external timestamp. Since a record data item usually has a time field *log_time*, we can construct a unique id for every record by adding a field *log_number* which can differentiate the different records with the same *log_time* value. Thus every record has a virtual combined unique identifier *log_id* through binding *log_time* and *log_number*. The allied timestamp is also used in [14]. However it is generated in the database core when the insert transaction is committed which will destroy the autonomy of the underlying databases.

B. Insertion Processing

In this section, the insertion process of high-rate data will be firstly described. Then how to load the data efficiently through three means will be discussed.

The data insertion processing is illustrated in Fig 2. The data to be loaded into databases is buffered into the input buffer B-in, and when B-in is full, it will be changed into output buffer B-out (①② in Fig.2). Then the data in *B-out* will be written into multiple database replicas simultaneously(③ in Fig.2). After the insertion thread receives all the messages of replicas(④ in Fig.2), it refreshes the *Time Fence (TF)* and *Available Node Set (ANS*, is the same as *A* in Section II*)*(⑤ in Fig.2). Only if the insertion thread meets a database replica failure, it will write *B-out* into local log files (⑥ in Fig.2). Thus before the failed replica is recovered, a group of *insertion log* files will be generated and maintained.

The *Time Fence (TF)* is the *log_id* of the latest record inserted into the database. Every table has a *TF*. It is used to synchronize the query threads and insertion threads.

Buffering the tuples, writing data in bulk, and writing to databases without logs, are the three key methods to improve the insertion performance. The object of buffering the tuples in *B-in* and *B-out* is to decrease the times of access to the underlying databases. In order to raise the concurrency of loading, we maintain one *B-in* and *B-out* for every table. Writing data to databases in bulk is a subsequent step of buffering. Loading data directly into database data files without common online logs is the third step. This step remarkably improves the loading performance by eliminating disk IO access of writing logs. This leverages the databases' high performance data loading technology which is implemented in common databases like Oracle's Direct Path Loading in OCI [9] and DB2's LOAD [10].

However this direct loading technology will commit the data directly and this will lose the consistency of a distributed transaction. How to guarantee the consistency of an insertion operation on multiple replicas will be described in detail in Section III.B.

From the above analysis, we could draw a conclusion that no logs are generated both on the coordinator node and database nodes in processing data insertion just as [14]. Since the volume of the log is at least larger than the data in a database, this method reduces at least 50% IO of the normal fashion. It is a more efficient approach than the [15] which needs to store logs both on middleware and database nodes.

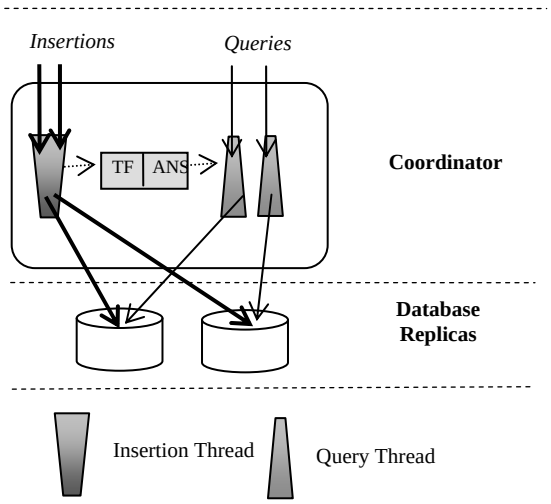

Figure 1. System Framework

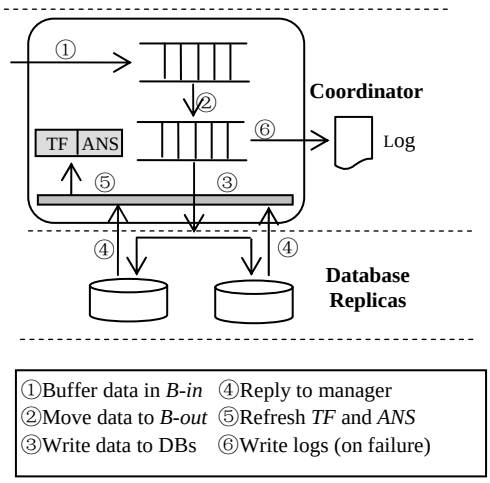

Figure 2. System Framework

C. Query Processing

The process of queries includes two steps. Step one is rewriting the SQL. In order to synchronize the result sets of every database replicas, an extra condition of *log_id* should be added according to the *TF* of every table which is set by the insertion thread. The revising rule is as Table 1. Thus all query threads have a strict uniform logical view about the data in the several replicas even though the same data may be not inserted synchronously by an insertion thread. ***TF[table_a]*** means table_a's *TF***.**

Step two is dispatching the revised SQL to an available replica in the *ANS*. This can be done in terms of some load balance policy like current requests number, CPU usage ratio and so on.

Since "Read One Write All Available" is adopted, the direct loading in Section III.B will lose the atomicity because several database replicas always can't commit an insertion at the same time. However, from the rewriting rule we can see that only until all replicas (except the failure replicas) have committed the data, the *TF* will be refreshed and the insertion is really committed. So the query will have a synchronized view of the data on every available node.

TABLE I.
REWRITE QUERY RULE

Original	Rewritten
SELECT tuples FROM table_a WHERE original_predicates;	SELECT tuples FROM table_a WHERE original_predicates *AND* ***log_time*** < ***TF[table_a].log_time*** *AND* ***log_number*** <***TF[table_a].log_number;***

D. Replication Protocol

Replication protocol is to keep copies (replicas) consistent despite updates [5, 6]. The traditional two-phase commit protocol (2PC) [24] or its variation [14] can be used to synchronize the data, but it is too complex and expensive for its communication overhead in our high-rate log-intensive workloads. Recently some efficient eager replication protocols [17,18]based on group communication primitives have started to appear. Most of these new protocols [18] can partly solve the problem of throughput and scalability but haven't improved the latency. All these general-purpose replication protocols seem too complex for the simple transaction semantics of *log-intensive* workload and always inefficient for their storing SQL queue or log and complex locks.

In the *log-intensive* workload, the atomicity and consistency of an insertion transaction is guaranteed by every table's *TF*. When a *table_a*'s insertion thread receives the replies of every replica, it must wait until it attains an exclusive (write) lock of *table_a*'s *TF*. After that it can refresh the *table_a*'s *TF* and the *ANS*. Before a query thread revises a query SQL, it must wait until it attains a share (read) lock of *table_a*'s *TF*. Thus the committed data will not be seen until all replicas have committed it. This simply guarantees insertion transactions' atomicity because no query will see the data before the *TF* is changed. The consistency of one copy serializability [19] also is kept by this mean which can be proved like the following:

Theorem 1 The Time Fence (*TF*) approach is one copy serializable.

Proof: Assume three transactions R1(a), R2(a)and W(a), the first two are queries on table a, the last is insertion on table a. Then the Theorem can be revised like this: if a transaction serialization is R1(a) ,W(a),R2(a), then the result set of R1(a) is older than R2(a). This can be proved based on how to refresh table a's TF. According to the transaction serialization, we have $TF_{R1(a)} \leq TF_{W(a)} \leq TF_{R2(a)}$. Then taking into account the revising rule of the query SQL, the added predicates of R1(a) is older than R2(a), so the result set of R1(a) is older than R2(a). □

This *TF* approach is optimized than the revised 2PC in [14] for two reasons: 1) 2PC needs four network communications while *TF* only needs two. 2) 2PC takes into account the complex problem like concurrency control, deadlock, rollback which will not occur in archived streams applications. The reason is that the log-intensive web applications don't need so strict ACID semantic and complex transaction model in classical OLTP applications, but need a high-rate massive simple processing model.

IV. RECOVERY APPROACH

The recovery approach is based on the insertion data log files (generated in ⑥ in Fig.2). We design a recovery algorithm on a granularity of tables. This algorithm is constituted of a recovery manager thread rm_thread and many recovery threads recovery_thread(node_id, table_id). The rm_thread always runs on the background and monitor which failure database needs to be recovered. If it finds some one, it will create one recovery_thread for every table on that database. After a recovery_thread recovers a table, it will inform the rm_thread. The recovery procedure of every recovery_thread can be divided into three phases in Section IV.A.

A. Phase 1: Recover From the Latest Save Point

When an insertion is pushed to a replica, the data will be directly written in pieces into the data files of the database. When the database meets an instance failure, one part of data of the insertion request is stored in the

TABLE II.
RECOVERY ALGORITHM

Recovery Algorithm

rm_thread
start:
for every recovering replica with *node_id* **do**
  for every table with on *node_id* **do**
  if no recovery_thread(*node_id*, *table_id*)
  then create recovery_thread(*node_id*, *table_id*)
  end of for
end of for
if recovery_thread(*node_id*, *table_id*) is ok
then
  inform insertion_thread(*table_id*) that table(*node_id*, *table_id*) is recovered
  if on every node_id is recovered
  then remove the table_id's log files
else goto start
 	
recovery_thread(*node_id*, *table_id*)
phase 1: recover the oldest log file of *table_id to node_id*
phase 2: recover the other log files of *table_id to node_id*
phase 3: inform **rm_thread** that recovery of *table_id* on *node_id* is ok and catch up the current insertion

database while other in the memory is lost. In order to save the stored data and avoid duplicating it, we should get the *log_id* of the latest stored data. We call this *log_id* as "*the latest save point(LSP)*". The LSP can be got in this standard SQL clause:

SELECT MAX(log_time), MAX(log_number) INTO *LSP*.log_time, *LSP*.log_number FROM table_a;

Just as mentioned in Section III.B, we can leverage the oldest insertion log file of the logs group. The pseudo code is just like:

LOAD DIRECT FILE= the oldest file of *table_id*
WHERE log_time ≤ *LSP*.log_time
AND log_number < *LSP*.log_number;

Thus all the data left in the oldest insertion log file is loaded into the recovering database. Then the other insertion log files can be directly loaded into the database.

From the above procedure, we can find that both the recovery of multiple tables in one database and the recovery of multiple failure databases can be done in parallel.

B. Phase 2: Catch Up with Data Logs

This phase is a subsequent step of phase 1 and simpler. The pseudo code is:

LOAD DIRECT FILE=other files of table_id;

In this phase, we can optimize the recovery by merging several small files into big files. This can improve the recovery performance due to decreasing the access times to the recovering database. The size of every big file is determined by the load of network, disk, cpu of two sides. The effect of merging will be shown in Section V.

C. Phase 3: Catch Up with Current Insertion

After loading all the log files of *table_id*, the recovery_thread will inform the rm_thread and the insertion_thread(*table_id*). The insertion_thread will push the current insertion to the database of *node_id*. After the insertion_thread has completed this insertion, it will refresh the TF of *table_id* and added the recovered database into the ANS of *table_id*. From that time on, the insertion and query transaction can send to the table of *table_id* of the recovered database.

If all recovering databases have recovered on this table of *table_id*, the insertion_thread(*table_id*) will no longer write log files.

D. Recover From Media Failure

When a database meets a media failure, such as some data files can't be read or written, the recovering procedure is more complex than the former sections. It can be implemented like the following two steps:

1) Recover the data files based on partitions. Partition [12] is a database technology to divide massive data into small segments[2]. First of all, two types of partitions should be defined: *current* and *historical* partitions. The *current* partition is the partition which the data is loaded into, and other partitions are *historical* partitions which aren't changed any more. The *historical* partitions should be recovered in preference to the current. The recovery procedure for them is to drop the local bad partition, export the corresponding partition from a remote normal node[3], and import the partition. Exporting remote partitions can be done without locks since the historical partitions will not be changed. For the current partition, locking the remote partition is needed before exporting it since it is inserted. This will decrease the loading performance sharply. So we devise a new method to get the historical part of the current partition by querying the partition of a remote node:

SELECT * FROM table_a on remote node
WHERE log_time≤*first_file*.log_time
AND log_number<*first_file*.log_number
AND log_time≥*last_par*.log_time
AND log_number>*last_par*.log_number;

Here the *first_file* is the first log file written by the insertion_thread(table_a), and the *last_par* is the latest normal partition before the current partition.

2) Recover the missing update data with the instance failure recovery procedure. After recovering the *historical* and *current* partitions, the remaining recovery is as the same as in Section IV.A-C.

V. EXPERIMENTS

In this Section, we first give the experimental setting in Section V.A. In Section V.B we analyze the relation of insertion performance with the bulk size and the number of concurrent users. In Section V.C we experimentally compare the performance of our proposed recovery method against ARIES and analyze the time of the three recovery phases. In Section V.D we discuss the transaction performance during failure and recovery.

A. Experimental Setting

A database cluster with three nodes is built and every one has the same data. A coordinator node is added. All the four nodes have two CPUs of Xeon 2G, 4G RAM, two 70G SCSI disks and are installed on Redhat AS 3.0. The three database nodes are installed with Oracle 10.1.0.4. And all the codes are written in GNU C++.

The experimental data comes from the access records of some commercial search engine. Every record data has about 329 bytes and 20MB data includes 63636 records. Every record item has *log_time* and *log_number* fields and other columns.

B. Runtime Performance

The runtime transaction performance can be discussed in two types: one user and many concurrent users. From another aspect, writing logs or not is another key factor which influences the insertion performance. Since the query processing will not write logs, its performance variation will not be shown in the figures.

In Fig.3, we can find two conclusions: 1) The optimized loading's performance is fifty to hundred fold

[2] The record data with a good even distributed feature on the time dimension can take advantage of the query process and recovery performance of partitions on time.

[3] We can assume that all nodes have the same partitions which is generally used in real world projects.

of the standard INSERT SQL's and 2PC which is used in [14]. 2) As we predict, writing less log means better performance. As the results show, when a database node writes online log or both online and archived log, the processing time in average relative to no log is about 1.43:1.14:1. 3) The insertion processing time is proportional to the size of the data. When the bulk size is 80MB, the insertion performance is the best which is 5.88MB/s or 18,700 records/s.

In Fig.4, the bulk size is 80MB and the time is the average processing time of multi-users. Three scenes are simulated: writing online log on databases and the coordinator (it happens when a database node failed), writing online log on databases and writing no log. The ratio is 1.28:1.11:1. It shows that the number of concurrent users has a weak influence on the relation of the performance of three conditions.

C. Recovery Performance

In Fig.5, we compare the classical ARIES recovery method and ours. The results show that when the recovered data size is less than 4.5MB, the ARIES is better, but after that point our method gets a better performance. When the recovered data size is small, the startup cost of our method is greater than ARIES and later the complexity of ARIES leads to its long recovery time. In Fig.6, the time of three recovery phases is shown. The startup time in phase 1 and the catching up time in phase 3 is a constant time, while the insertion time in phase 2 is proportional to the data to be recovered.

D. Performance during Failure and Recovery

The transaction processing performance during the databases' failure and recovery is another problem needed to be discussed. In Fig.7, the x-axis is the time, the left y-axis is the insertion performance whose criterion is MB/s, and the right y-axis is the query performance whose criterion is the completed transactions per second.

Before the 10th second, the system runs in the normal state. At the 10th second, one of the three databases fails, the coordinator detects this and the DBA restarts the database at the 15th second. During this period, the insertion performance decreases a little about 13% because the log files need to be stored on the coordinator's disk, and the query performance decreases about 31% because 1/3 of the three nodes can't process the query requests. From 15th second to 25th second, the recovery phase 1 and 2 complete, and the performance is just as the 15th second because the recovery will not decrease the online performance. From 26th to 27th second, the phase 3 completes, and the performance return to the normal level.

From Fig.7, we can find that there is no sharp performance degradation because other transactions will not be interrupted when one database fails.

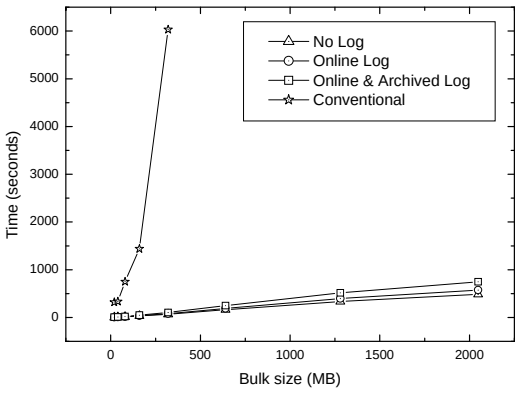

Figure 3. Insertion performance and bulk size.

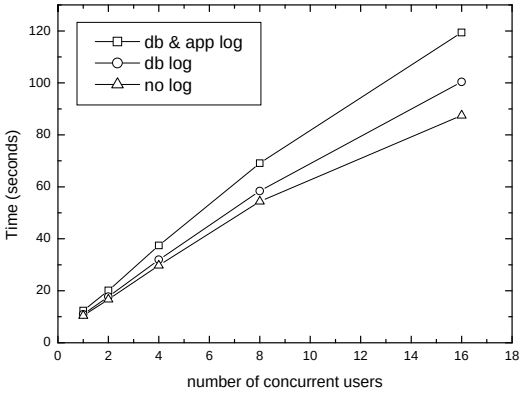

Figure 4. Insertion performance and # of users.

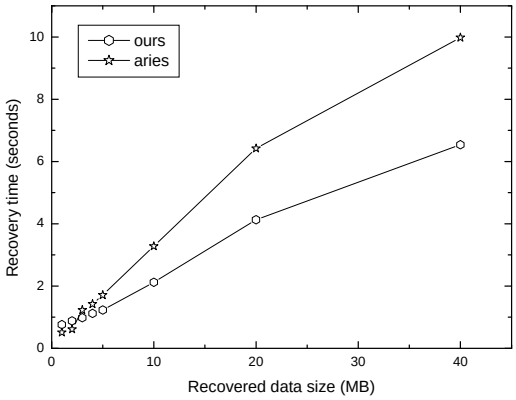

Figure 5. Recovery performance and recovered size.

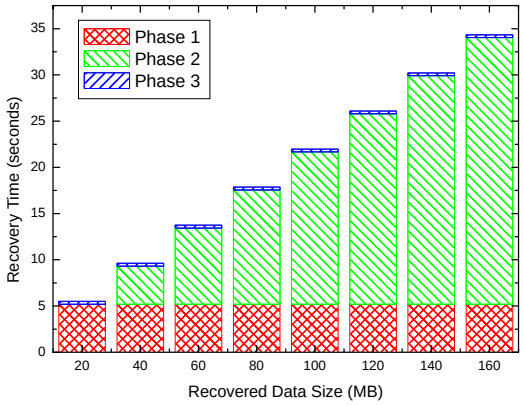

Figure 6. Decomposition of recovery time

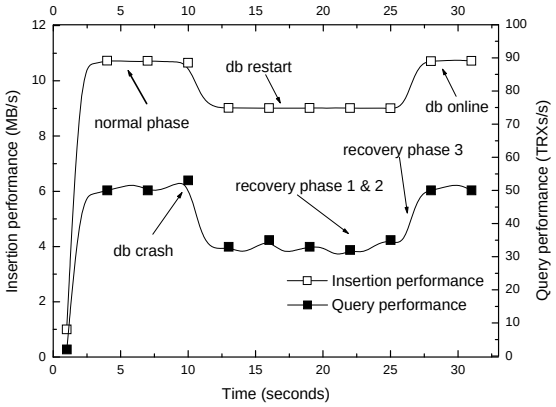

Figure 7. Transaction processing performance during failure and recovery

In this paper we have studied the problem of how to store and recover high-rate archived streams in a database cluster. According to the *log-intensive* applications, we present an optimized data insertion method based on reducing the disk IO access cost and a simple and efficient consistency protocol. The experiments results show that our approach can reach efficient data loading and query and get shorter recovery time than the traditional database cluster recovery methods.

ACKNOWLEDGMENT

The authors wish to thank Prof. V.P. Shirochin.This research was supported by China Postdoctoral Science Foundation(20070420908) and by the Project-sponsored by SRF for ROCS, SEM (2008890).

REFERENCES

[1] S. Ganarski, H. Naacke, E. Pacitti, P. Valduriez: Parallel Processing with Autonomous Databases in a Cluster System, CoopIS, 2002.
[2] J. Gray, A. Reuter. Transaction Processing: Concepts and Techniques. Morgan Kaufman, 1992.
[3] Google personalized search. http://www.google.com/psearch
[4] http://news.com.com/Google,+eBay+Strategic+bedfellows/2100-1024_3-6110304.html
[5] J. Gray and P. Helland and P. O'Neil and D. Shasha: The Danger of Replication and a Solution, ACM SIGMOD, 1996.
[6] Marta Patino-Martinez1,2, Ricardo Jim ′ enez-Peris, Bettina Kemme, Gustavo Alonso. Consistent Database Replication at the Middleware Level. ACM Transactions on Computers, Vol. V, No. N, Month 2004, Pages 1 – 43.
[7] C. Mohan, D. Haderle, B. Lindsay, H. Pirahesh, and P. Schwarz. ARIES: a transaction recovery method supporting fine-ranularity locking and partial rollbacks using write-ahead logging. ACM TODS, 17(1):94–162, 1992.
[8] Y. Dora Cai, Ruth Aydt, Robert J. Brunner. "Optimized Data Loading for a Multi-Terabyte Sky Survey Repository". Super Computing 2005.
[9] http://www.lc.leidenuniv.nl/awcourse/oracle/appdev.920/a96584/oci16m51.htm
[10] http://publib.boulder.ibm.com/infocenter/db2luw/v8//topic/com.ibm.db2.udb.doc/admin/t0004590.htm
[11] Sirish Chandrasekaran, Michael Franklin. Remembrance of Streams Past:Overload-Sensitive Management of Archived Streams. VLDB 2004.
[12] www.psoug.org/reference/partitions.html
[13] S.-O. Hvasshovd,Torbjrnsen, S. E. Bratsberg,and P. Holager. The clustra telecom database: High availability, high throughput, and real-time response. In VLDB, pages 469–477. ACM Press, 1995.
[14] Edmond Lau, Samuel Madden. An Integrated Approach to Recovery and High Availability in an Updatable, Distributed Data Warehouse. vldb06
[15] R. Jim′ enez-Peris, M. Patino-Martinez, and G. Alonso. An algorithm for non-intrusive, parallel recovery of replicated data and its correctness. In SRDS, 2002.
[16] B. Kemme. Database Replication for Clusters of Workstations. PhD dissertation, Swiss Federal Institute of Technology, Zurich, Germany, 2000.
[17] Matthias Wiesmann, Fernando Pedone, Andr′ e Schiper, Bettina Kemme,Gustavo Alonso. Transaction Replication Techniques: a Three Parameter Classification. SRDS 2000.
[18] A. Sousa,J. Pereira, L. Soares, A. Correia Jr., L. Rocha, R. Oliveira, F. Moura.Testing the Dependability and Performance of Group Communication Based Database Replication Protocols. Dependable Systems and Networks (DSN) 2005.
[19] P. Bernstein, V. Hadzilacos, and N. Goodman. Concurrency Control and Recovery in Database Systems. Addison-Wesley, 1987.
[20] B. Liskov, S. Ghemawat, R. Gruber, P. Johnson, and L. Shrira. Replication in the harp file system. In SOSP, pages 226–238. ACM Press, 1991.
[21] Microsoft Corp. Log shipping. http://www.microsoft.com/technet/prodtechnol/sql/2000/reskit/part4/c1361.mspx.
[22] Oracle Inc. Oracle database 10g Oracle Data Guard. http://www.oracle.com/technology/deploy/availability/htdocs/DataGuardOverview.html.
[23] Sybase, Inc. Replicating data into Sybase IQ with replication server. http://www.sybase.com/detail?id=1038854.
[24] C. Mohan, B. Lindsay, and R. Obermarck. Transaction management in the R* distributed database management system. ACM TODS,11(4):378–396, 1986.

Zhengbing Hu was born in 1978. He received B.E., M.E. and P.h.D degree in National Technical University of Ukraine. His current research interests include Network Security, Intrusion Detection System, Artificial Immune System, Data Minging etc..

Kai Du was born in 1978. He received B.E. and M.E. PhD degree in National University of Defense Technology, China. His current research interests include large-scale data management, data reliability, distributed computing.