Journaling and Recovering in NoSQL

Chia-Ping Tsai†, Hung-Chang Hsiao‡,†, Yu-Chang Chao†, Cheng-Lung Chu‡
†Department of Computer Science and Information Engineering
National Cheng Kung University, Tainan 70101, Taiwan
‡Cloud Platform Technology Department
Cloud Service Application Center
Industrial Technology Research Institute South Campus, Tainan 709, Taiwan
E-mail: hchsiao@csie.ncku.edu.tw

Abstract—Not Only SQL (NoSQL) is a recent technology that is scalable and provides flexible schemas, thereby complementing existing relational database technologies. Although NoSQL is flourishing, present solutions lack the features required by enterprises for critical missions. In this paper, we explore solutions to the data recovery issue in NoSQL. Data recovery for any database table entails restoring the table to a prior state or replaying (insert/update) operations over the table given a time period in the past. Recovery of NoSQL database tables enables applications such as failure recovery, analysis for historical data, debugging, and auditing. In this paper, we first identify the design and implementation issues with regard to the data recovery problem for NoSQL databases, including time length of recovery, scalability, memory constraint, and software compatibility. Two solutions are then proposed and evaluated to address the data recovery problem in NoSQL; each solution has its pros and cons. We implement our proposals based on Apache HBase, a popular NoSQL database in the Hadoop ecosystem. Our implementations are extensively benchmarked with an industrial NoSQL benchmark under real environments.

I. INTRODUCTION

Not Only (NoSQL) databases (NoSQL DBs or NoSQL for short) are emerging technologies that provide random access to big data [1]. Examples of such databases include Cassandra [2], Couchbase [3], HBase [4] and MongoDB [5]. NoSQL DBs typically organize database tables in a columnar manner. Essentially, a table comprises a number of columns. Each column, a simple byte string, is managed and stored in the underlying file system independently. Sparse tables in NoSQL can efficiently utilize memory and secondary storage space. The columnar-based data structure simplifies the change of a table schema in case new columns are inserted on the fly.

While NoSQL technology receives wide attention due to its high scalability, reliability, and availability, existing NoSQL DBs lack features (or deliver immature features) compared with conventional relational databases (e.g., MySQL [6]), which may be highly required by enterprises. For instance, most NoSQL DBs are incapable of performing transactional operations such that a set of accesses to more than one table cannot be handled atomically. In addition, several NoSQL DBs lack the SQL interface. Thus, they are not easy to use. Moreover, some NoSQL DBs (e.g., HBase) provide no secondary indices for any specified columns, thus reducing the efficiency of manipulating the columns without indexing.

NoSQL DBs remain in their early stage and their features need to be improved or expanded to address the needs of enterprises in dealing with big data. In this paper, we are particularly interested in the data recovery issue in NoSQL. By data recovery for a database table, we mean to restore a database table to any prior time instance (or point-in-time recovery for short) or to replay operations on a database table given a time period in the past. Recovering a database table can have a number of applications:

- **Failure recovery**: The objective of failure recovery in present NoSQL is to leverage the availability of DBs to offer 7 × 24 services. Existing NoSQL DBs can be recovered to a prior state. However, the restored state is the most recent state immediately before the failure.
- **Analysis for historical data**: NoSQL DBs can be operated in a manner that enable DBs to operate online to accommodate any newly inserted/updated data items even with the existence of other DBs that act as backups which clone parts of past data items stored in the tables maintained by the serving DBs. Analysis is then performed on the backups.
- **Debugging**: NoSQL DBs are improved over time. Rich features are incrementally released along with the later versions. Enterprises may prefer to test newly introduced features. Therefore, with their prior data sets in standalone environments, serving NoSQL DBs are isolated from the tests. Restoring the past data items in NoSQL database tables on the fly into standalone, testing environments thus becomes an issue.
- **Auditing**: State-of-the-art NoSQL DBs such as HBase and Phoenix [7] have enhanced security features such as access control lists (ACL). Users who intend to manipulate a database table are authorized first, and then authenticated before they gain access. However, even with authorized and authenticated users, information in accessed database tables may be critical. Preferably, a database management system can report when a user reads and/or writes a database table in certain values, thus tracing the user’s access pattern.
We present architectures and approaches to the data recovery problem in NoSQL DBs. Two solutions are proposed based on the HBase NoSQL, each having unique features to accommodate distinct usage scenarios. We focus particularly on the Hadoop HBase as it is a part of the Hadoop ecosystem and receives support from the active Hadoop development community.

This paper is the first attempt in the literature to discuss the data recovery issue for NoSQL. While tools such as Snapshots [8], Replication [9], Export [10], and CopyTable [11] in HBase mainly provide the backup service, they cannot resolve the research problem presented in this paper. We conduct a full-fledged implementation for our proposed solutions with regard to data recovery in the HBase. This paper not only presents the design issues for the proposed approaches, but also discusses in detail their implementations. In addition, the two proposed solutions are benchmarked in a real environment that consists of 50 virtual machines. We consider the industrial benchmark, the Yahoo Cloud Serving Benchmark (YCSB) [12], to investigate the performance of our proposals.

II. SYSTEM MODEL AND RESEARCH PROBLEM

A. System Model

Our model consists of a number of off-the-shelf commodity servers. Among k servers, each performs two functions, namely, the region server and the datanode. (Datanodes are the fundamental entities that assemble the Hadoop distributed file systems [13].) The k servers with the data node functions form the distributed file system HDFS, while their region server functions serve operations (such as GET, PUT, and SCAN) for key-value pairs in HBase. These k servers that constitute the region server and the data node functions form an HBase cluster.

Each key-value pair stored in a table in HBase can be associated with a timestamp. The timestamp is either specified by a client or by HBase itself (i.e., the region server responsible for the region that accommodates the key-value pair). A cell in a table can maintain several versions of values (three by default) in terms of timestamps.

As HBase is designed to store big data, we assume reasonably that the main memory space available to each region server is far less than the space provided by the file system. That is, each region server has a small memory operation space to buffer data items for reads and writes, and to perform routine processes such as compactions.

B. Research Problem

Consider any table T in HBase. T may consist of a large number of key-value pairs (denoted by kv), each associated with a timestamp (t). We define T as follows:

**Definition 1.** A table T consists of key-value pairs T = \{kv(t) \mid 0 \leq t \leq t_c\}, where 0 is the initial clock time and t_c is the present system clock.

**Definition 2.** Given two time instances t_1 and t_2 (where t_1 \leq t_2 \leq t_c), historical data (key-value pairs) with respect to T are T_s = \{kv(t) \mid kv(t) \in T \text{ and } t_1 \leq t \leq t_2\}.

Data and key-value pairs are interchangeable in this paper. Clearly, T_s \subseteq T is a subset of key-value pairs in T for a time period between t_1 and t_2. T may be partitioned into a number of regions. These regions are allocated to and served by different region servers in the HBase cluster, which results in the possibility that the set of key-value pairs in T_s may appear in distinct region servers.

**Definition 3.** A row key is a prefix string of kv’s key that uniquely differentiates distinct key-value pairs in T.

**Definition 4.** Let R be a region of T if R \subseteq T and R contains key-value pairs with consecutive row keys in alphabetical order.

**Definition 5.** Let R be the complete set of regions with respect to T, and \( R = \{R_1, R_2, \ldots, R_k\} \) such that:
- \( R_i \) is a region of T for any \( 1 \leq i \leq k \).
- \( R_i \cap R_j = \emptyset \) for any \( 1 \leq i \leq k \), \( 1 \leq j \leq k \) and \( i \neq j \).
- \( R = \bigcup_{1 \leq i \leq k} R_i = T \).

**Definition 6.** An HBase cluster that consists of r region servers, denoted by \( S = \{S_1, S_2, \ldots, S_r\} \) is valid if any \( R_i \subseteq R \) is allocated to a unique \( S_j \) at any time instance, where \( 1 \leq i \leq 2 \) and \( 1 \leq j \leq r \).

Here, \( R_i \) may be reallocated from one region server to another due to failure recovery or load balancing in HBase.

We are now ready to define the research problem, namely, the data recovery problem, studied in this paper.

**Definition 7.** Given T, the current system clock \( t_c \) and any two time instances \( t_1 \leq t_2 \) (where \( t_2 \leq t_c \)), then the table restoration process is to find a subset of T, denoted by \( T_s \), such that
\[
T_s = \{kv(t) : kv(t) \in T \text{ and } t_1 \leq t \leq t_2\}. \tag{1}
\]

**Definition 8.** Given \( T_s \) and an HBase cluster \( S \), the table deployment process partitions key-value pairs in \( T_s \) into a number of regions and assigns these regions to region servers in S such that S is a valid HBase cluster.

**Definition 9.** Given two HBase clusters \( S_1 \) and \( S_2 \) (where \( S_1 \) is a valid, source cluster and \( S_2 \) is the destination cluster), two time instances \( t_1 \) and \( t_2 \) (where \( t_1 \leq t_2 \leq t_c \)) and a table \( T \) in \( S_1 \), the data recovery (DR) problem includes:
- a table restoration process that finds \( T_s \subseteq T \) in \( S_1 \) for the time period between \( t_1 \) and \( t_2 \);
- a table deployment process that partitions \( T_s \) into \( S_2 \) such that \( S_2 \) is a valid HBase cluster.

III. ARCHITECTURES AND APPROACHES

A. Architecture

The overall system architecture is shown in Fig. 1. Conceptually, our proposals involve three HBase clusters, namely, the source, the shadow, and the destination HBase clusters. The source cluster is the HBase cluster, which serves normal requests from clients. The destination HBase cluster is the target cluster designated by users for storing and serving recovered data. In contrast, the shadow HBase cluster buffers historical data that may be requested for recovery.
We note that the three clusters are conceptually independent. Logically, each cluster has its own HBase and HDFS infrastructures. Computational (or storage) entities in the three clusters may overlap, depending on usage scenarios. Even any two clusters among the three HBase instances may physically share common HBase and/or HDFS infrastructures. For example, the source, the shadow, and the destination clusters have identical HBase and HDFS infrastructures.

As depicted in Fig. 1, we rely on the observer mechanism provided by HBase. The mechanism allows embedding third-party codes into the HBase write flow such that any normal update in the HBase can additionally perform bookkeeping routines (e.g., checkpoints and compactions) without modifying the HBase internals, thereby leveraging the compatibility of our proposed solutions with different HBase versions. Specifically, with the observer mechanism, we integrate the data write flow in the HBase such that all updates (including PUT and DELETE) to the source HBase cluster are mirrored in the shadow cluster. (We do not instrument read operations such as GET and SCAN due to clients as these operations do not vary the state of a table in the HBase.)

The shadow HBase cluster stores historical update operations to tables in the source HBase cluster. We suggest two solutions, each having its advantages and disadvantages, to transform the retrieved data to the destination HBase cluster. We differentiate a data table from its shadow table where the former is used to serve normal requests from clients while the latter tracks the update operations to the data table. Shadow tables are solely adopted for recovery.

We assume that the data table \( T \) to be recovered contains a single column in the following discussions. Consider any key-value pair, \( kv \), of a data table \( T \) and its shadow table \( D_T \). Each key-value pair \( kv \) in \( T \) has a corresponding row (denoted by \( D_{kv} \)) in \( D_T \), and each row in \( D_T \) consists of a row key and a value. The row key comprises six fields as follows:

- **Prefix**: As \( D_T \) can be quite large (in terms of the number of its maintained key-value pairs), the prefix field is used to partition and distribute \( D_T \) into the region servers in the destination HBase cluster. If the number of region servers in the destination cluster is \( k \), then the prefix is generated uniformly at random over \([0, k - 1] \).
- **Batch Time Sequence**: In the HBase, clients may put their key-value pairs into \( T \) in batches to amortize the RPC overhead per inserted key-value pair. In \( T \), each \( kv \) of the key-value pairs in a batch is labeled with a unique timestamp, namely, the batch time sequence, when it is mirrored to a shadow table (i.e., \( D_{kv} \) in \( D_T \)). The time sequence basically refers to the local clock of the region server that hosts \( kv \), thereby enabling the proposed approaches to recover data within a designated time period, as will be discussed later.
- **ID of a Key-Value Pair in a Batch**: To differentiate distinct key-value pairs in a batch for \( T \), we assign each key-value pair in the batch an additional unique ID in \( D_T \). This step ensures that when a user performs two updates to the same (row) key at different times, if the two updates are allocated in the same batch, then they are labeled with different IDs in the batch. Consequently, our recovery
solution can differentiate the two updates in case we want to recover any of them. On the other hand, in the HBase, if two updates go to the same key, the latter update will mask the former when they are issued by an identical client in the same batch. The ID of a key-value pair in a batch helps explicitly reveal the two updates.

- **Region ID**: The ID associated with $D_{kv}$ in $D_T$ for the corresponding $kv$ in $T$ indicates the region server that hosts $kv$ for $T$. Region IDs are critical when we aim to recover the distribution of regions in the HBase source cluster.

- **Operation Type**: The field denotes the operation for $kv$ in $T$. Possible operations include PUT and DELETE provided by the HBase.

- **Original Row Length**: As a value of a key-value pair, $kv$, in $T$ can be any byte string whose length varies. The original row length field indicates the number of bytes for the string length taken by the value of the key-value pair $kv$ in $T$.

The value in a row of $D_T$ contains only

- **Original Row Key Concatenated by Original Value**: The original row key represents the row key of $kv$ in $T$, and the original value is the value of $kv$. In addition to the value of $kv$, we also need to recover the row key of $kv$ such that $T$ with designated data objects is recovered.

### D. Approaches

1) **Multithreaded-based Approach**: The idea of the centralized approach is to employ multiple threads in a server to gather the historical data designated by users from the shadow cluster and then to deploy the data to the destination cluster. (Recall that we implement an observer mechanism to simultaneously stream data from the source cluster to the shadow cluster.) However, the centralized server that streams the data is equipped with fixed-size memory space. Therefore, the server itself shall handle the memory management to ensure that the data will not run out of the memory available to the server, thereby preventing the server crashing.

Given $D_T$ and a time period $[t_1, t_2]$, our centralized server approach manages a set of “read threads” to fetch qualified data from $D_T$ and a set of “write threads” to push the retrieved key-value pairs to the destination cluster.

In our implementation (as shown in Fig. 2(a)), $k$ read threads are created if the number of regions that have the qualified key-value pairs (i.e., batch time sequences are present in $[t_1, t_2]$) in $D_T$ is $k$. Note that we let $D_T$ be partitioned into $k$ regions in the shadow cluster by assigning each key-value pair (gathered from the source cluster) a randomly selected prefix field such that each of the $k$ read threads goes to each region, thereby balancing the load of the region servers in the shadow cluster.

Each read thread retrieves the key-value pairs from $D_T$ by using standard HBase APIs to realize our implementation in the HBase with different versions. Each read thread takes advantage of the blockcache in each region server to accelerate their fetches. (A blockcache is a mechanism implemented by each region server, which stores key-value pairs accessed recently. It implements the LRU replacement policy when it runs out of its memory space. When a key-value pair, requested by a client, enters the blockcache, it piggybacks several key-value pairs whose keys are numerically closest to the key of the requested data item.) Specifically, because the rows of a shadow table are organized in alphabetical order of batch time sequences, key-value pairs in $T$ of the source HBase cluster will be stored consecutively in $D_T$ if they are introduced in the time proximity. Consequently, each read thread fetches its designated data items from $D_T$ efficiently as the blockcache managed by each region server in the shadow cluster prefetches key-value pairs generated over time proximity due to our design for shadow tables.

Multiple read threads simultaneously fetch key-value pairs from the shadow HBase cluster. Key-value pairs gathered in the memory are sorted according to their original row keys. When the total size of key-value pairs is accumulated up to a predefined threshold, the buffer that stores the set of key-value pairs is closed and waits in the memory space to be flushed into the HDFS. In our current implementation, the buffers that hold the set of key-value pairs to be flushed are chained in a FIFO queue.

We maintain a pool of write threads to handle the flush operations. A write thread is invoked each time to flush the buffer in the front of the FIFO queue as an HFile. When the thread finishes its flushing, it returns to the pool and is made available to flush other buffers in the queue. The write thread pool is established in advance to reduce the time overhead in creating the threads.

Note that in the centralized, multithreaded approach, the size of a flushed HFile is at 128 MBytes, which is the default size of an HFile in the HBase. The buffer pool may possibly contain a large number of buffers waiting to be flushed when the total service rate of write threads is lower than the service rate aggregated by read threads. If any read thread finds that the buffer pool reaches a predefined threshold, then the read thread is frozen and resumed later once the buffer pool is available to accommodate new buffers.

Once all read threads finish their reads and no buffer exists in the buffer pool, we can then load the HFiles introduced by the write threads in the destination HBase cluster. HBase provides a bulk load mechanism so that the region servers can be immediately aware of the regions that correspond to these HFiles. For simplicity, we create $d$ regions in the destination cluster if the cluster consists of $d$ region servers, each managing the identical size of row key spaces. However, our approach is flexible and allows users to indicate how a table is partitioned into the destination cluster.

2) **Endpoint-based Approach**: HBase provides a mechanism, namely, the endpoint, that allows clients to develop and install program codes in a region server such that these codes can access data stored locally in the server. Notably, the endpoint mechanism differs from the observer mechanism as discussed in Section III-A in that while the latter is mainly invoked on the read and write path flows in a HBase region.
server, the former can be explicitly activated anytime by clients. Based on the endpoint mechanism, we implement the table restoration process in the shadow cluster to restore data stored locally in the region servers of the cluster.

When performing the recovery, an endpoint function deployed in a region server is activated (refer to Fig. 2(b)). As each region server has installed our endpoint function, each endpoint function in a region server $S$ is responsible for restoring the requested data in $D_T$ hosted by $S$. The endpoint function streams its gathered key-value pairs into a buffer, and the buffer sorts the collected key-value pairs. Once the buffer reaches 128 Mbytes, it is written immediately to the destination cluster’s HDFS.

Our endpoint-based approach does not implement the threading mechanism as in the multithreading-based solution. Unlike the multithreading-based approach, multiple endpoint functions are operated in different region servers concurrently. Each behaves as a single thread that pipelines its buffered key-value pairs to the destination cluster, thus aggregating computational resources in the region servers.

IV. PERFORMANCE EVALUATION

A. Experimental Setup

We conduct full-fledged implementations for our proposals presented in Section III. This section benchmarks the two proposals. Our experimental hardware environment is the IBM BladeCenter server HS23 [15], which contains 12 2-GHz Intel Xeon processors. The total server memory space available in our experiments is 128 Gbytes.

Our proposed architecture consists of three clusters, namely, the source, shadow, and destination, for historical data recovery. We note that in the IBM server, virtual machines (VMs or nodes) are created for our experiments. The maximum number of VMs, based on VMware [16], we can create is approximately 50, given our hardware capacity of IBM BladeCenter. As discussed in Section III, our proposed solutions allow overlap among the source, shadow, and destination clusters. Our experiments allocate a total of 54 nodes. These nodes are partitioned into two separate, disjoint groups, where one group serves the source and shadow clusters, and the other is occupied solely by the destination cluster. For the former, the source (shadow) cluster comprises one namenode for the HDFS and one master node for the HBase. Here, we let the namenode and zookeeper share the same VM. The HDFS and HBase in the source (shadow) cluster manage 25 datanodes and 25 region servers, respectively. The datanodes and region servers of both HDFS and HBase share the same set of 25 VMs. That is, in the source (shadow) cluster, each of the 25 VMs hosts one datanode in the HDFS and one region server. By contrast, the destination cluster consists of 27 VMs, where one is the namenode and zookeeper, one is the master, and the remaining 25 ones operate both datanode and region server functions. By default, the maximum cluster size of the source (shadow or destination) cluster includes approximately 25 nodes as discussed. We also investigate a relatively smaller cluster...
(i.e., 15 nodes) allocated to each of the source, shadow, and destination clusters; this will be discussed later in this section.

We adopt the YCSB [12] (ver. 0.14) to assess our two proposed approaches. In our experiments, we rely on YCSB to introduce the data set to the source cluster. The key-value pairs inserted in the source cluster have row keys that are selected from a Zipf distribution. The total size of key-value pairs we store in the source HBase is 30 Gbytes. The length of a value in a key-value pair generated by YCSB is randomized. The average total length of a key-value pair is our experiments is \( \approx 2,500 \) bytes.

B. Performance Results

Fig. 3. (a) Latency of the recovery process, (b) delay overhead, and (c) space overhead

Fig. 3(a) shows the latency of the entire recovery process with the multithreading-based and endpoint-based approaches. In the experiment, we recover the entire table, and the latency measured for the recovery includes those required by the table restoration and deployment processes. The experimental results indicate that the endpoint-based approach outperforms the multithreading-based solution. Specifically, the endpoint-based approach introduces few disk accesses and network transfers, thus performing faster; the multithreading-based approach clearly introduces the performance bottleneck due to the centralized server that performs the recovery process.

In addition to the performance results for the recovery process discussed so far, we are also interested in the overhead incurred due to our proposed architecture, namely, the source, shadow, and destination clusters, for HBase table recovery. Figs. 3(b) and 3(c) show the overheads introduced by streamlining data to the shadow cluster. The overhead includes delay in streaming key-value pairs to the shadow cluster and the storage space required to store the historical key-value pairs in the shadow cluster. In Fig. 3(b), our proposal introduces an average of 2 times of delay (i.e., the ratio of \( w/o \) ) to write/update a key-value pair in the source cluster. (In Figs. 3(b) and 3(c), \( w/o \) and \( w/o \) denote a write operation performed without and with the presence of the shadow cluster, respectively.) As we discussed in Section III, we augment the write flow path such that when a write operation is performed, it simultaneously streams the operation to the shadow cluster. Our instrumentation is transparent to users who issue the write operation. Concurrently issuing the write operation to the shadow cluster takes extra RPC overhead, which is fair by our present implementation. That is, a write operation in our implementation takes only \( \approx 0.6 \) milliseconds compared with \( \approx 0.3 \) milliseconds required by a naive write.

Obviously, the shadow cluster introduces additional storage space overhead as it stores data items and operations over the data items in the history. For a 30-Gbyte data set, our shadow cluster requires an additional 34 Gbytes of disk space (see Fig. 3(c)) compared with 30 Gbytes in the source cluster, that is, with our proposed solutions, the required total disk space is roughly twice that demanded by the original source HBase cluster. This finding indicates that our design of the data structure presented in Section III-C is space efficient as the row key in the shadow table takes an additional 25 bytes for each key-value pair stored in the shadow HBase cluster.

V. SUMMARY AND FUTURE WORKS

We have presented the data recovery issue in NoSQL. Our study is the first in the literature to identify the data recovery problem for NoSQL. We not only discuss the design and implementation issues with regard to the problem, but also propose two solutions that address the problem. We discuss our implementations for the two solutions based on Apache HBase in detail. Experimental results are also extensively illustrated. We conclude that each of our two proposed solutions has its pros and cons, which reveals its unique operational scenario.

Our implementations of the proposed solutions are full-fledged. We consider the software upward compatibility to ensure that our implementations can be simply integrated into later versions of HBase. Although our implementations are mainly based on HBase ver. 0.94, they can be simply ported to HBase ver. 0.98 with the security feature. More precisely, we are currently enhancing our solutions to support access control lists such that only authenticated users can access authorized data items in the history. On the other hand, we seek to enhance the reliability of our endpoint-based solution since region servers involving the recovery process may fail and the present HBase does not support any failure recovery mechanism for the endpoint scheme.

REFERENCES