



Article

The Time Machine in Columnar NoSQL Databases: The Case of Apache HBase

Chia-Ping Tsai ¹, Che-Wei Chang ^{2,*} , Hung-Chang Hsiao ^{3,*} and Haiying Shen ⁴

¹ Apache HBase and Kafka Project Management Committees, Wilmington, DE 19801, USA; chia7712@gmail.com

² Department of Computer Science and Information Engineering, Chinese Culture University, Taipei 11114, Taiwan

³ Department of Computer Science and Information Engineering, National Cheng Kung University, Tainan 70101, Taiwan

⁴ Department of Computer Science, University of Virginia, Charlottesville, VA 22908, USA; hs6ms@virginia.edu

* Correspondence: che.wei.alfred.chang@gmail.com (C.-W.C.); hchsiao@csie.ncku.edu.tw (H.-C.H.)

Abstract: Not Only SQL (NoSQL) is a critical 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. Particularly, our study focuses on columnar NoSQL databases. We propose and evaluate two solutions to address the data recovery problem in columnar NoSQL and implement our solutions based on Apache HBase, a popular NoSQL database in the Hadoop ecosystem widely adopted across industries. Our implementations are extensively benchmarked with an industrial NoSQL benchmark under real environments.



Citation: Tsai, C.-P.; Chang, C.-W.; Hsiao, H.-C.; Shen, H. The Time Machine in Columnar NoSQL Databases: The Case of Apache HBase. *Future Internet* **2022**, *14*, 92. <https://doi.org/10.3390/fi14030092>

Academic Editors: Michael Sheng and Petros Patias

Received: 18 February 2022

Accepted: 10 March 2022

Published: 15 March 2022

Publisher's Note: MDPI stays neutral with regard to jurisdictional claims in published maps and institutional affiliations.



Copyright: © 2022 by the authors. Licensee MDPI, Basel, Switzerland. This article is an open access article distributed under the terms and conditions of the Creative Commons Attribution (CC BY) license (<https://creativecommons.org/licenses/by/4.0/>).

Keywords: NoSQL; backup; recovery; Hadoop; HBase; MapReduce; big data

1. Introduction

Not Only SQL (NoSQL) *databases* (NoSQL DBs or NoSQL for short) are critical technologies that provide random access to big data [1,2]. Examples of such databases include Cassandra [3], Couchbase [4], HBase [5] and MongoDB [6]. 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.

In general, NoSQL DBs are scalable. In a cluster computing environment that consists of a set of computing nodes, a table in a NoSQL DB is horizontally partitioned into a number of regions, and each region can be assigned to any computing node in the cluster. When the cluster expands by installing additional computing nodes, some partitions of a table may be further segmented, and newly created regions are reallocated to the newly added computing nodes to utilize available resources.

Compared with the sequential access pattern supported by present solutions (e.g., the *Hadoop distributed file system* [7] or HDFS) for big data, the NoSQL technology is attractive if real-time, random accesses to the large data set are required. Specifically, typical NoSQL DBs pipeline database tables that are stored in the main memory to their underlying file systems as data items are streamed into systems over time. Files that represent data items in tables are structured in a sophisticated manner such that any query to the data items can be fetched rapidly. More precisely, a file may be sorted based on an attribute of its stored

data items and may contain an index structure to facilitate a search for data items present in the file. A bloom filter may be included in the file, which helps to identify whether a data item (probably) exists in the file or not, thereby improving its access efficiency.

In addition to efficiency, NoSQL DBs are concerned with reliability and availability by having update operations to database tables recorded in log files. By replaying the log files, a NoSQL DB restores its prior state immediately preceding the crash of any database server. Database table files and log files can be replicated to distinct storage nodes that assemble the file system, thereby leveraging the overall system reliability and availability.

While NoSQL technology receives wide attention due to its high scalability, reliability, and availability, existing NoSQL DBs lack features (or the delivery of immature features) compared with conventional *relational databases* (e.g., MySQL [8]), which may be 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 evolve over time, 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 columnar NoSQL databases such as Cassandra [3] and HBase [5]. (The terminologies, Col-NoSQL and columnar NoSQL, are interchangeable in this paper.) 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:** As mentioned, existing Col-NoSQL DBs can be recovered to a prior state. However, the restored state is the most recent state immediately before the failure. The objective of failure recovery in present Col-NoSQL is to leverage the availability of DBs to offer 7×24 services.
- **Analysis for historical data:** Col-NoSQL DBs are often 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:** Col-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 Col-NoSQL DBs are isolated from the tests. Restoring the past data items in Col-NoSQL database tables on the fly into standalone ones means that testing environments becomes an issue.
- **Auditing:** State-of-the-art Col-NoSQL DBs such as HBase and Phoenix [9] 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 Col-NoSQL DBs. Two solutions are proposed based on the HBase Col-NoSQL, each having unique features to accommodate distinct usage scenarios. We focus on the Hadoop HBase as it is a part of the Hadoop ecosystem, and receives support from the active Hadoop development community. Specifically, HBase is extensively adopted by industry, serving as a database backend for big data [10]. (HBase is the second-most-used wide-column database storage engine as of this paper's writing in 2022 [10]).

- **Mapper-based:** The mapper-based approach operates when the MapReduce framework [11,12] is available. Specifically, the framework shall be present in the target HBase cluster. The mapper-based approach takes advantage of parallelism for transforming and loading the data set for the destination cluster to minimize the delay of recovery. However, the mapper-based solution would place an additional burden on memory utilization in the target cluster side.
- **Mapreduce-based:** The mapreduce-based approach is yet another solution that depends on the MapReduce framework to extract, transform, and load designated data items into the destination HBase cluster. This approach is guaranteed to operate reliably, thereby ensuring that the recovery process is complete. Our proposed solutions not only bear scalability in mind, but also take the quality of recovered tables into our design considerations.

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 their implementations in detail. 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) [13], to investigate the performance of our proposals. Notably, while we discuss our solutions based on HBase as proof of concept, our solutions can be also adopted by other wide-column, distributed Col-NoSQL DB engines as we rely on the proposed notation of *shadow tables* (discussed later in this paper), an independent Col-NoSQL DB involved in the data recovery process.

The remainder of the paper is organized as follows. We specify the system model and research problem in Section 2. Section 3 details our proposed solutions, namely, the mapper-based and the mapreduce-based approaches. We evaluate our proposals in real environments and discuss the performance benchmarking results in Section 4. Section 5 enumerates state-of-the-art solutions relevant to the data recovery problem. Our paper is summarized in Section 6.

2. System Model and Research Problem

2.1. System Model

Figure 1 depicts the system model we consider in this study. Our model consists of a number of off-the-shelf commodity servers. Among k servers in Figure 1, each performs two functions, namely, the region server and the datanode. The k servers with the datanode 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 datanode functions form an HBase cluster.

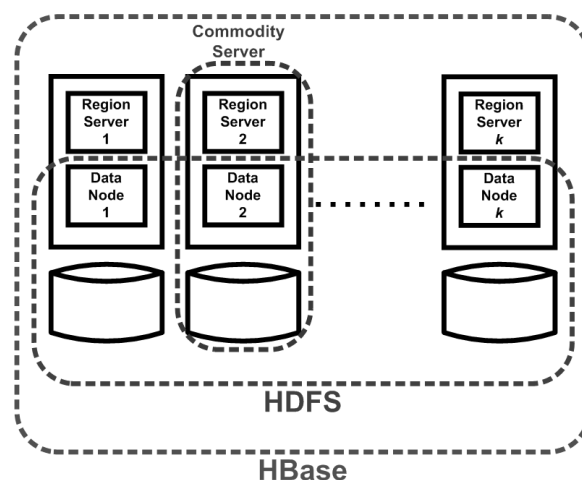


Figure 1. System model [5].

We note that when a region server flushes its memstores to HFiles, the HFiles remain in their file system locally. Depending on the replication factor specified in the HDFS, more than one replica for each chunk of an HFile may be populated at different datanodes for high availability and durability. On the other hand, when fetching a key-value pair, the region server that hosts the associated region checks its local HFiles if the requested data is not present in its main memory (including the memstore and blockcache). Consequently, locally stored HFiles accelerate data accesses.

Each key-value pair stored in a table in HBase can be associated with a *timestamp*. HBase demands that the local clocks of region servers are synchronous, and presently relies on the *network time protocol* [14]. 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.

Finally, in Figure 1, we do not depict the components such as the *ZooKeeper* used to provide the coordination service and the *Master* for region location lookup and reallocation. The name node, which implements the file namespace in the HDFS, is also not depicted.

2.2. 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 $T = \{kv(t) : 0 \leq t \leq t_c\}$, 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 $T_s = \{kv(t) : kv(t) \in T \text{ and } t_1 \leq t \leq t_2\}$.

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 $\mathcal{R}$ be a region of T if $\mathcal{R} \subseteq T$ and $\mathcal{R}$ 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 $R = \{\mathcal{R}_1, \mathcal{R}_2, \dots, \mathcal{R}_k\}$ such that

- $\mathcal{R}_i$ is a region of T for any $1 \leq i \leq k$;
- $\mathcal{R}_i \cap \mathcal{R}_j = \emptyset$ for any $1 \leq i \leq k, 1 \leq j \leq k$ and $i \neq j$;
- $R = \bigcup_{1 \leq i \leq k} \mathcal{R}_i = T$.

Definition 6. An HBase cluster that consists of r region servers, denoted by $S = \{S_1, S_2, \dots, S_r\}$ is valid if any $\mathcal{R}_i \subset R$ is allocated to a unique S_j at any time instance, where $1 \leq i \leq k$ and $1 \leq j \leq r$.

Here, $\mathcal{R}_i$ may be reallocated from one region server to another due to failure recovery or load balancing in HBase.

Notably, 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, where a region is a horizontal partition of a database table. The scaleout feature by horizontally partitioning is widely supported by distributed database storage engines. These regions are allocated to and served by different region servers,

which results in the possibility that the set of key-value pairs in T_s may appear in distinct region servers.

We now 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

$$T_s = \{kv(t) : kv(t) \in T \text{ and } t_1 \leq t \leq t_2\}. \quad (1)$$

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.

Notably, S_1 and S_2 follow the system model as discussed in Section 2.1. S_1 and S_2 may not be identical in terms of cluster size (i.e., $|S_1| \neq |S_2|$) and the specification of region servers in both clusters. However, S_1 can also be equal to S_2 in our problem setting, or the server sets of S_1 and S_2 are partially overlapped, that is, $S_1 \neq S_2$ and $S_1 \cap S_2 \neq \emptyset$. Specifically, the DR problem can be used to state the failure recovery problem in a typical HBase setting [15], that is, a serving HBase cluster can be recovered in real-time if some region servers in the cluster fail. More precisely, for such a problem, we let $S_2 \subset S_1$ ($S_1 \neq S_2$), and $t_1 = 0$ and $t_2 = t_c$ for each table in S_1 . Another scenario with regard to the DR problem is set to $S_1 \cap S_2 = \emptyset$ and $0 < t_1 < t_2 < t_c$ for some tables in S_1 when one intends to analyze partial historical data in S_1 over S_2 while letting S_1 remain to accept data solely. Yet another example is that an enterprise that is operating HBase may extend its infrastructure by deploying S_2 as a pilot run.

For ease of discussion, the DR problem specified in Definition 9 is simplified. However, our operational implementation that addresses the problem is full-fledged, which allows users to designate the granularity of recovery in the cell level of a table. Parameters that specify the objects to be recovered include row keys, columns, qualifiers, and timestamps in HBase. Filters by ranges and partial strings over the candidates for recovered data are additionally supported. Moreover, users may demand the restoration of historical data to their familiar HBase cluster environments because an HBase application may have been optimized for a particular cluster environment. Consequently, our full-fledged implementation allows recovery of historical data in tables to a predefined system configuration by the location of a region (i.e., the region server) and the size of a region (i.e., the number of rows or the number of bytes in a region).

3. Architectures and Approaches

In this section, we first present our design considerations for resolving the DR problem. Section 3.1 addresses pragmatic issues when implementing solutions to the DR problem. Section 3.2 discusses two solutions that address the problem. We then, in Section 3.3, qualitatively discusses the merits of our proposals, considering pragmatic concerns.

3.1. Design Considerations

Solutions to resolve the DR problem consider the pragmatic issues in the following aspects:

- *Time for recovery*: Recovering requested data objects may take time. Specifically, the time required for recovery includes time periods for performing the table restoration and deployment processes as defined in Definitions 7 and 8, respectively. As the data to be recovered potentially are large (in terms of the number of bytes or rows), the table restoration and deployment processes shall be performed simultaneously to reduce the recovery time. On the other hand, we may have several computational entities to perform the recovery in parallel.
- *Fault tolerance*: As the recovery may take quite some time, entities involved in the recovery may fail, considering that failure is the norm. Solutions for recovery shall tolerate potential faults and can proceed subsequently once servers used to perform the recovery process fail. (We will specify clearly in Section 3.2 for our proposed architectures regarding computational entities in the recovery.)
- *Scalability*: As the size of data to be recovered can be very large, solutions designed for the DR problem shall be scalable in the sense that when the number of computational entities involved in addressing the problem increases, time for the table restoration and deployment processes shall be reduced accordingly. However, this situation depends on the availability of the entities that can join the recovery.
- *Memory constraint*: Servers that are involved in the recovery are often resource constraints, particularly the memory space. If the size of data to be recovered is far greater than any memory space available to any server for recovery, then several servers may be employed in the recovery to aggregate the available memory space or pipeline the computation of the data to be recovered and retain the computed data on the fly during the recovery.
- *Software compatibility*: Recovery solutions shall be compatible with existing software packages. Specifically, our targeted software environment is the Hadoop HBase, which is an open-source project of the Hadoop ecosystem. The functionality and performance of HBase are enhanced and improved over time. Solutions to the problem shall be operational for HBase versions that will be developed later.
- *Quality of recovery*: Once a table with designated data objects is recovered, the table deployed in the targeted HBase cluster may serve upcoming read and write requests. The concern is not only the integrity for the set of recovered data objects in the recovered table, but also the service rate of the table that will be delivered.

3.2. Proposals

3.2.1. Architecture

The overall system architecture is shown in Figure 2. 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.

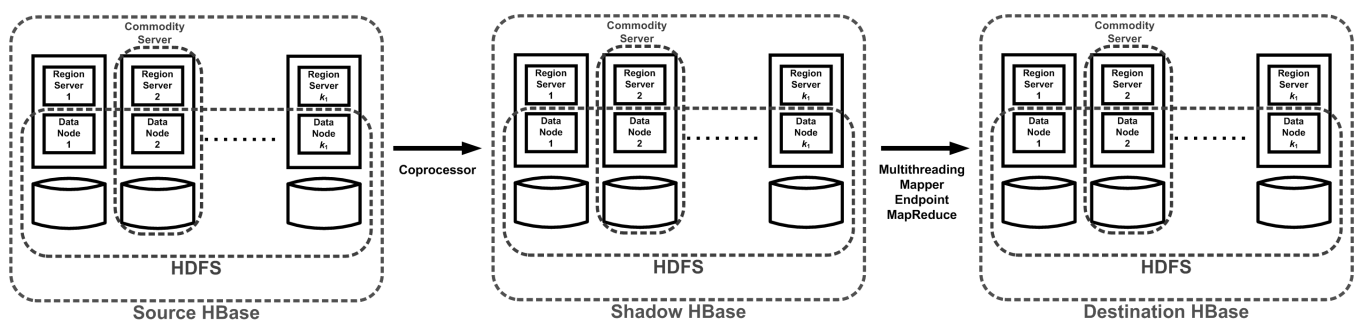


Figure 2. Overall system architecture.

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 Figure 2, we rely on the *observer* mechanism provided by HBase. (We refer interested readers for the background of Apache HBase in Appendix A.) The mechanism allows embedding third-party codes into the HBase write flow such that any normal update in the HBase can additionally perform housekeeping 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 extract the historical data in the shadow HBase cluster and transform the retrieved data to the destination HBase cluster. Our implementations are compatible with the Hadoop software stack. Specifically, the *mapper-based* and *mapreduce-based* approaches are based on the Hadoop MapReduce framework.

3.2.2. Application Programming Interfaces (APIs)

When designated tables in past time periods are restored to the destination cluster, then applications access data items in these tables through standard HBase APIs [16]. However, we also offer APIs such that application programmers can manipulate data items for operations replayed during the restoration period. For example, an application user may be interested in how a cell value changed over time for a given period $[t_1, t_2]$ of a specified key in a table. The major API for browsing historical data items in our proposals is

Review CreateReview(Filter) (2)

where *Filter* specifies the table name, columns/qualifiers, and time period, among others, of parameters for data items to be investigated. *CreateReview* streamlines the matched data to the calling entity, and *Review* is a container that accommodates the matched data items. The sample code shown in Algorithm 1 illustrates the use of the API.

Algorithm 1: A code example of extracting and loading data items during the replay

```

input : Filter
output: Review

1 History history = new History();
2 Review review = history.CreateReview(Filter);
3 for KeyValue KV:review do
4   | ...

```

3.2.3. Data Structure

Before we discuss potential solutions to the DR problem, we introduce the data structures on which these solutions rely. In particular, we propose *shadow tables*. A shadow table is essentially a table in the HBase. A data table created by a user in the HBase is basically associated with a shadow table unless the data table will not be recovered in the future. Here, 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.

Table 1 shows the data structure of a shadow table. We assume that the data table T to be recovered contains a single column. 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 .

Table 1. Schema in a shadow table D_T (Here, $\oplus$ denotes the string concatenation operator.)

Row Key						Value
...						...
Prefix	Batch Time Sequence	ID of Key-Value Pair in a Batch	Region ID	Operation Type	Original Row Length	Original Row Key $\oplus$ Original Value
...						...

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.

3.2.4. Approaches

Mapper-based Approach: The mapper-based approach is based on the Hadoop MapReduce framework [12]. Given k regions to be recovered in D_T , in the shadow cluster we fork k map tasks to gather the requested historical data in D_T and then to output these data as HFiles in the destination HBase cluster. Figure 3a depicts the mapper-based architecture.

In our implementation, each map task (or mapper) scans D_T with filters by invoking standard HBase APIs. It collects the data in its memory until the size of the data reaches a predefined threshold (i.e., the default value of 128 MBytes). The data is then immediately flushed as an HFile stored in the HDFS of the destination cluster. Once the flush is successfully conducted, the mapper continues to gather the remaining historical data from D_T and then repeats its flush operation.

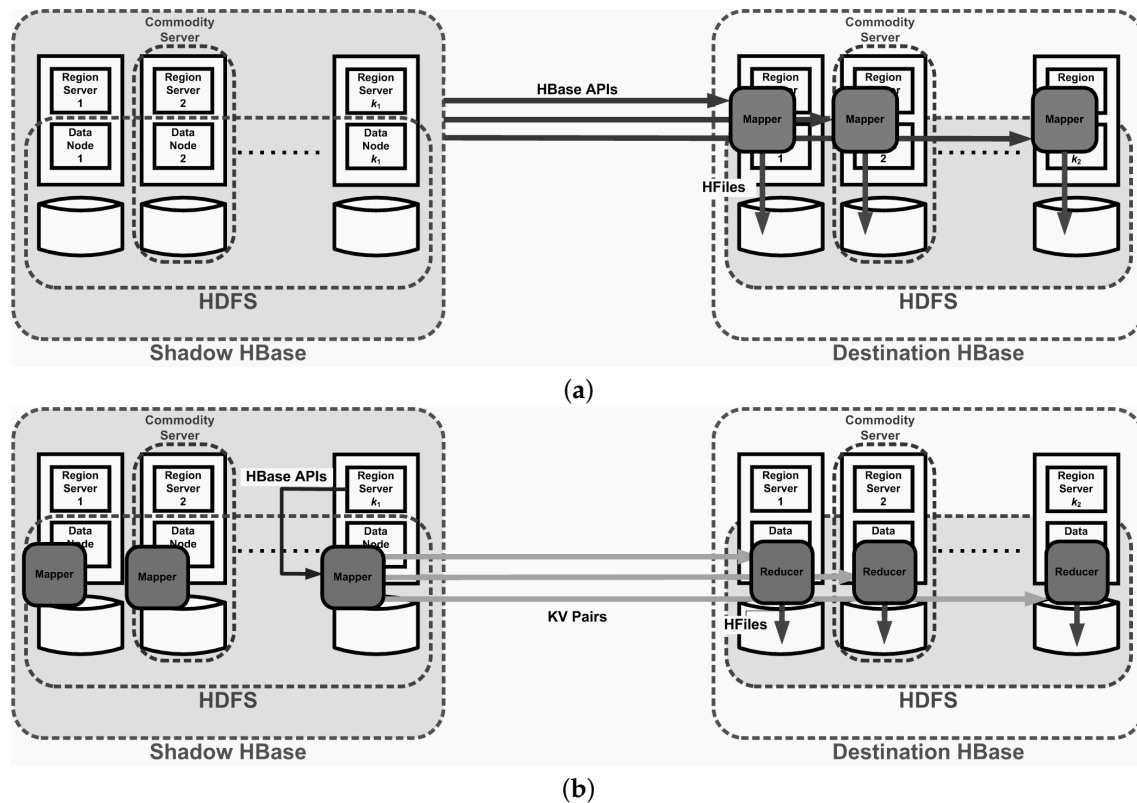


Figure 3. (a) The mapper-based architecture and (b) the mapreduce-based architecture (Gray areas are major components that are involved in the recovery process).

Each mapper manages its buffer pool. When a mapper that is performing its table restoration and deployment processes fails, in our mapper-based solution, the *jobtracker* in the MapReduce framework forks another mapper to redo the task that was previously performed by the failed mapper from the beginning. The *jobtracker* process in the MapReduce framework assigns map and reduce tasks to datanodes in the HDFS. In addition to map and reduce tasks performed by datanodes, they also report the status of their assigned tasks through their local *tasktracker* processes. In the event of failures, $>k$ mappers may possibly join the recovery process such that some of them output identical HFiles. Specifically, the *speculative task execution* [11] offered by the MapReduce framework may be enabled in the MapReduce framework. The speculative execution allows the *jobtracker* to assign “slow tasks” to extra mappers and reducers. The speculative execution may not only introduce extra burden to the shadow cluster, but also generate more redundant HFiles in the destination cluster. Consequently, we discard redundant HFiles if found in our implementation.

Mapreduce-based Approach: Due to memory space constraints, a node in the mapper-based approaches flushes its memory buffers if the memory space occupied by these buffers reaches a predefined threshold. As a result, the HFiles that represent a region may be of poor quality. Specifically, x HFiles are introduced for a region (having the key range R) hosted by a region server in the destination cluster, and y of them ($y < x$) manage the key

range $r \subset R$. Subsequently, a read operation that fetches a key-value pair in r would need to access y HFiles in the worst case, thereby lengthening the delay of the read operation.

Our proposed mapreduce-based approach intends to minimize the number of HFiles that represent any region. We depend on the MapReduce framework, where map/reduce tasks generate/retrieve their data to/from the HDFS, thereby facilitating the compilation of HFiles. By sorting and storing the key-value pairs requested for T in a global fashion through the disks of the datanodes that constitute the file systems (i.e., the HDFS) in the shadow and destination clusters, we are able to generate k HFiles for a region such that any two HFiles of the k 's manage disjoint key ranges.

The MapReduce framework essentially realizes a sorting network. In the network, *mappers* and *reducers* are deployed in datanodes of the HDFS. Each mapper m is given an input and then outputs several sorted *partitions* according to intermediate keys which can be specified by the map task the mapper performs. (Each piece of data generated by a map task is represented as an intermediate key and value pair.) The partitions are stored in the m 's local disk due to memory space constraints. Several partitions in the local disk of m are then merged, sorted (according to intermediate keys), and stored in the local disk again to minimize the bandwidth required to deliver the merged group to reducers. Each reducer r fetches designated grouped data from the mappers and then sorts the gathered data according to intermediate keys in these groups. r executes its reduce task once it has merged and sorted all designated groups in the remote mappers. A reduce task is then performed given the merged and sorted data.

Figure 3b depicts our proposed solution based on the MapReduce framework. k_1 mappers and k_2 reducers are deployed in the shadow and destination HBase clusters, respectively. Notably, in our design, we have k_1 (k_2) region servers, and each datanode in the shadow (destination) cluster provides the region server function.

Each mapper m in our proposal reads requested data from D_T in the shadow cluster. m decodes the fetched key-value pairs and converts each retrieved key-value pair by having the original row key (i.e., the row key shown in Table 1) of the decoded key-value pair as an intermediate key. The original row key is stored in the value field of the shadow table in Table 1. We extract the original row key from the value field by referring to the original row length recorded in the table. Clearly, the original value is designated as an intermediate value.

As k_1 mappers are allocated to k_1 region servers in the shadow cluster and each region server manages a region of D_T in the cluster (that is, a total of k regions that represent D_T), each mapper pipes its partitions, and each partition is sorted based on its row keys and stored in its local disk. Here, each partition is designated to a reducer in the destination cluster. Partitions assigned to the same reducer r are further merged and sorted as local files that will be fetched later by r . (We can indicate in the MapReduce framework the number of local files to be generated after partitions are merged and sorted.)

Reducer r performs its reduce task when all local files, designated to r , are gathered from the mappers. r merges and sorts these files based on intermediate keys (i.e., the row keys). r then ejects HFiles in its HDFS, given the size of each HFile (default size of 128 MBytes).

To illustrate our mapreduce-based architecture and approach, let D_T contain 3 regions, and thus $k_1 = 3$. Assume that the destination HBase cluster has $k_2 = 4$ datanodes and thus 4 region servers. As each mapper m among the $k_1 = 3$ ones in the shadow cluster is responsible for a region of D_T , m decodes requested rows from D_T , and assigns the decoded original row key for each of these rows as an intermediate key and the decoded original value as an intermediate value. A partition consists of a set of sorted intermediate key-value pairs. m may generate several partitions. Assume 8 partitions, denoted by $p_0, p_1, p_2, \dots, p_7$, are introduced by m , where each p_i maintains the intermediate key-value pairs with keys in $[R_i, R_{i+1})$ for $i = 0, 1, 2, 3$ since we allocate $k_2 = 4$ reducers ($\{r_0, r_1, r_2, r_3\}$), each assigned to a unique datanode in the destination cluster. We partition the intermediate key space to $k_2 = 4$ reducers, each r_i responsible for compiling HFiles for the intermediate key space

in $[R_i, R_{i+1})$. Note that depending on the intermediate key distribution, the 8 partitions $\{p_0, p_1, p_2, \dots, p_7\}$ may be merged and sorted into four groups such as $\{p_0, p_1\}$, $\{p_2\}$, $\{p_3\}$, and $\{p_4, p_5, p_6, p_7\}$ such that the intermediate key ranges of $\{p_0, p_1\}$, $\{p_2\}$, $\{p_3\}$, and $\{p_4, p_5, p_6, p_7\}$ are $[R_0, R_1)$, $[R_1, R_2)$, $[R_2, R_3)$, and $[R_3, R_4)$, respectively. The merged and sorted groups for $\{p_0, p_1\}$, $\{p_2\}$, $\{p_3\}$, and $\{p_4, p_5, p_6, p_7\}$ would then be fetched by r_0 , r_1 , r_2 and r_3 , respectively. Among the reducers, each gathers the merged, sorted, and grouped files from $k_1 = 3$ mappers. r_i ($i = 0, 1, 2, 3$) then dumps HFiles where each is sorted based on the intermediate keys (i.e., the row keys of the recovered table T).

The mapreduce-based solution tolerates faults. If nodes in the shadow and destination clusters fail, then the map and reduce tasks are assigned to and recomputed by active nodes such that the entire recovery process can succeed. In addition, our mapreduce-based solution evidently introduces HFiles that manage disjoint key space in the destination HBase cluster. This capability shortens delays of read operations sent to the destination HBase cluster.

3.3. Discussion

Table 2 compares the two proposals discussed in Section 3.2.4.

Table 2. Comparisons of our proposed solutions.

	Recovery Time	Fault Tolerance	Scalability	Memory Constraint	Software Compatibility	Recovery Quality
Mapper-based	fast	yes	yes	single memory buffer	yes	poor
Mapreduce-based	medium	yes	yes	disk	yes	yes

Recovery time: The mapper-based and mapreduce-based approaches rely on the Hadoop MapReduce framework, which incurs overhead (in terms of processor cycles and memory space) in managing the framework execution environment.

Fault tolerance: The mapper-based and mapreduce-based solutions tolerate faults. When a datanode in the HDFS that executes a map or reduce task fails, the Hadoop MapReduce framework requests another active datanode to redo the task.

Scalability: The mapper-based and mapreduce-based solutions linearly scale with the increased number of datanodes. Additional servers involved in the computation share the load for the table restoration and deployment processes.

Memory space limitation: Our proposals bear available memory space limitation in mind. In the solutions based on the MapReduce framework, each mapper in the mapper-based approach and each mapper/reducer in the mapreduce-based approach effectively exploit their secondary storage space to buffer requested key-value pairs to be sorted and flushed as HFiles, thus relieving the memory burden.

Software compatibility: Our proposed solutions are compatible with existing versions of HBase while our implementation is based on HBase. Our solutions based on the MapReduce framework not only rely on the map and reduce interfaces, but also invokes normal HBase and HDFS APIs.

Recovery quality: Unlike the mapreduce-based solution, the mapper-based approach may introduce a large number of HFiles for any region, thereby lengthening the delay of acquiring key-value pairs in the persistent storage. This situation is mainly due to the memory constraint imposed on the mapper-based approach. By contrast, the mapreduce-based approach guarantees that a single HFile is introduced for each disjoint key space and a region.

4. Performance Evaluation

4.1. Experimental Setup

We conduct full-fledged implementations for our proposals presented in Section 3. This section benchmarks the two proposals. Our experimental hardware environment is the IBM BladeCenter server HS23 [17], which contains 12 2-GHz Intel Xeon processors.

Our proposed architecture consists of three clusters, namely, the source, shadow, and destination, for historical data recovery. The configurations for the three clusters are shown in Table 3. We note that in the IBM server, virtual machines (VMs or nodes) are created for our experiments. (Virtualizing Hadoop in production systems can be found in [18].) The maximum number of VMs, based on VMware [19], we can create is approximately 50, given our hardware capacity of IBM BladeCenter. As discussed in Section 3, 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, disjointed 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, one is the master, and the remaining 25 ones operate both datanode and region server functions. (Table 3 further details the memory space allocated to the nodes in the source, shadow, and destination clusters.) 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.

Table 3. Configurations of source, shadow, and destination clusters (here, (x, y) indicates the number of x nodes and y the Gbytes of memory space per node).

	Source (Shadow)	Destination
HDFS		
Namenode	(1, 8)	(1, 8)
Datanode	(25, 1)	(25, 1)
HBase		
Master	(1, 8)	(1, 8)
Region server	(25, 1)	(25, 1)

Regarding the parameter setting for Hadoop, we allocate one jobtracker and 25 (50) tasktrackers for the mapper-based (mapreduce-based) approach. The chunk that assembles a HDFS file is in size of 64 MBytes. The replication factor for each chunk is one. Each mapper/reducer has 400 MBytes memory space for computation. In our experiments, the speculative execution offered by MapReduce is disabled, that is, each map/reduce task is uniquely performed by a single datanode.

In the HBase, the memory sizes allocated to memstore and blockcache are 128 and 400 Mbytes, respectively. In particular, in the blockcache, each block takes 64 Kbytes. We disable the option for compressing HFiles in the experiments. Bloom filters are also disabled for HFiles. (In the HBase, HFiles can be compressed to minimize the disk space they occupy. On the other hand, as a region may be represented by several HFiles, the delay in searching for a designated key-value pair in these HFiles can be reduced by filtering through their associated bloom filters).

We adopt the YCSB [13] to assess our 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 (i.e., the workload A for update heavy provided YCSB). By contrast, in the destination cluster, YCSB is used to generate read requests by having each read select a key-value pair with row key picked uniformly at random (i.e., the workload C for read only). Due to the storage space

available to our experimental environment, the maximum total size of key-value pairs we can store in the source HBase is up to 300 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 in our experiments is ≈ 2500 bytes. That is, the average total number of key-value pairs in the table of our experiments is $\approx \frac{3 \times 10^9}{2500} = 1.2 \times 10^6$, $\approx \frac{30 \times 10^9}{2500} = 12 \times 10^6$, and $\approx \frac{300 \times 10^9}{2500} = 120 \times 10^6$ for data sets with 3, 30, and 300 Gbytes, respectively.

The performance metrics investigated in this paper mainly include the delays and overheads of performing the recovery process. Notably, the performance results that will be discussed later are averaged over 20 runs. For the metrics we gathered, the confidence interval is up to the 95th percentile.

4.2. Performance Results

4.2.1. Overall Performance Comparisons

Figure 4 shows the averaged latency and standard deviation for the entire recovery process with the mapper-based and mapreduce-based approaches. Here, the size of an HBase table stored in the source cluster is 30 Gbytes. 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 mapper-based approach outperforms the mapreduce-based solution as the mapper-based approach introduces few disk accesses without data shuffling due to the mapreduce framework.

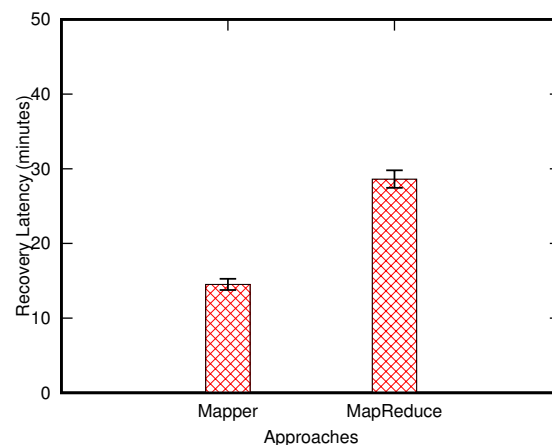


Figure 4. Latency of the recovery process (the table restored in the source cluster is in size of 30 Gbytes).

The mapper-based and mapreduce-based approaches are scalable and fault tolerant without memory constraints because of the Hadoop MapReduce framework. However, these approaches take considerable time for the recovery process at the expense of temporarily buffering intermediate data in disks. Disk operations take time. In particular, the mapreduce-based approach introduces an additional time penalty to shuffle intermediate data between mappers and reducers. Reducers again rely on their local disks to buffer data. Consequently, the mapper-based and mapreduce-based solutions impose considerable burdens on the disks and the network.

Figure 5 shows the recovery quality that are resulted from our two proposals. The mapreduce-based approach outperforms others clearly in terms of the averaged delay of a read operation. Its read latency is approximately 5 milliseconds, while other three approaches take at least 40 milliseconds on average to fetch a value given a key.

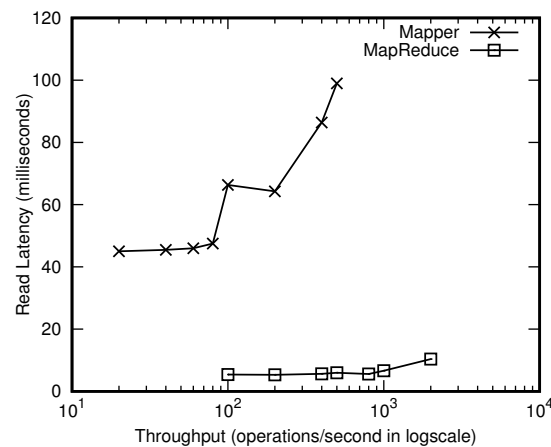


Figure 5. Mean delay of read operations.

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. Figure 6 shows the overhead 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 (i.e., Table 1) in the shadow cluster.

Figure 6 illustrates such overheads. In Figure 6a, our proposal introduces an average of 2 times of delay (i.e., the ratio of $\frac{w/o}{w/}$) to write/update a key-value pair in the source cluster. (In Figure 6, w/o and w/ denote a write operation performed without and with the presence of the shadow cluster, respectively.) As we discussed in Section 3, 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 ≈ 0.6 milliseconds compared with ≈ 0.3 milliseconds required by a naive write.

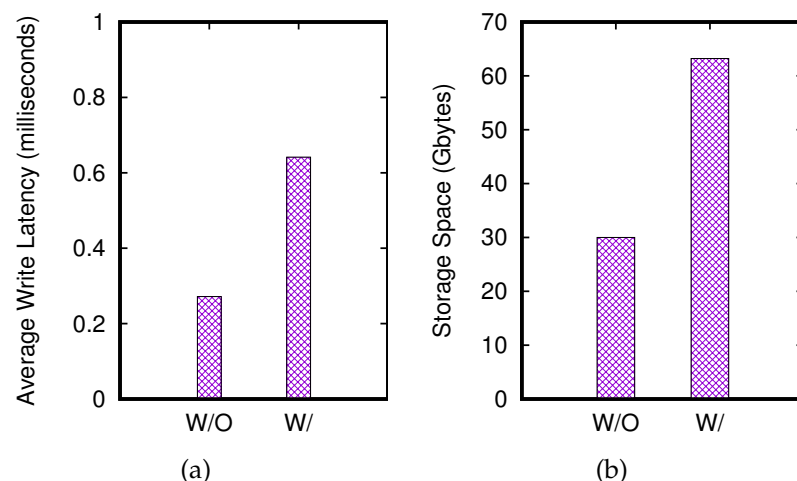


Figure 6. Overheads, where (a) is the delay for write operations, and (b) the storage space required.

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 Figure 6b) 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 Table 1 is space efficient as the row key of Table 1 takes an additional 25 bytes for each key-value pair stored in the shadow HBase cluster.

4.2.2. Effects of Varying the Data Set Size

We vary the size of data set from 3 Gbytes to 300 Gbytes, and investigate the performance of our proposed solutions. The performance results are shown in Figure 7. The experimental results clearly indicate that the recovery delays due to our approaches are lengthened proportionally to the size of the data set. We observe that the mapper-based and mapreduce-based approaches increase their recovery delays linearly with the size of the data set. This situation occurs because when the size of a data set is enlarged, the number of key-value pairs increases accordingly. When the increasing number of key-value pairs needs to be manipulated, the numbers of read and write operations for these key-value pairs are linearly enlarged.

Second, for the distributed approaches (i.e., the mapper-based and mapreduce-based solutions), our designs and implementations are effective and efficient in the sense that each node is allocated a nearly identical data size, and all nodes perform their recovery processes concurrently such that no node lags behind.

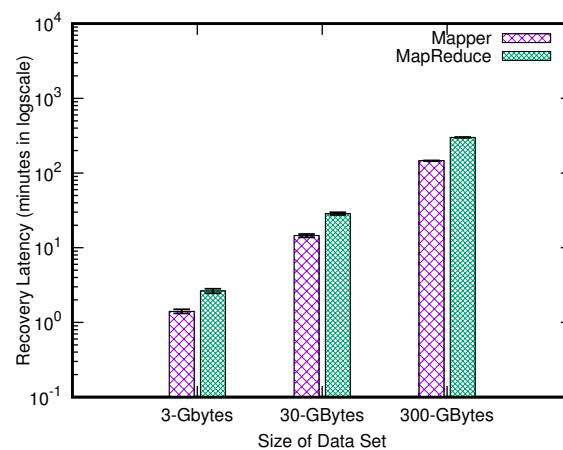


Figure 7. Latency of the recovery process, where the table restored in the source cluster is in size of 3 to 300 Gbytes.

The delays of read operations in the destination cluster for different sizes of data sets are not shown since the performance results are similar to those discussed in the prior section.

4.2.3. Effects of Varying the Cluster Size

We investigate the effect of varying the sizes of the source, shadow, and destination clusters. Let (a, b, c) represent the numbers of nodes in the source, shadow, and destination clusters equal to a , b , and c , respectively. Three configurations are studied, including $(15, 15, 15)$, $(20, 20, 20)$, and $(25, 25, 25)$. Here, the experiment is mainly designed to understand whether the mapper-based and the mapreduce-based approaches scale well for the recovery process.

Figure 8a depicts that the mapper-based and mapreduce-based approaches perform well when the cluster size increases. Specifically, to measure the scalability of the three proposals, we measure

$$\frac{\text{recovery delay from our implementation}}{\text{ideal recovery delay for scaleout}}. \quad (3)$$

Given a baseline configuration C_1 , let t_1 be the recovery delay measured for C_1 . Then, the ideal recovery delay (t_2) for a scaleout HBase cluster, C_2 , is computed as $t_2 = t_1 \times \frac{\text{size of } C_1}{\text{size of } C_2}$. For example, if $C_1 = (15, 15, 15)$ and $C_2 = (25, 25, 25)$, then we estimate $t_2 = t_1 \times \frac{15}{25}$. Clearly, a recovery solution scales well if the metric defined in Equation (3) is close to 1.

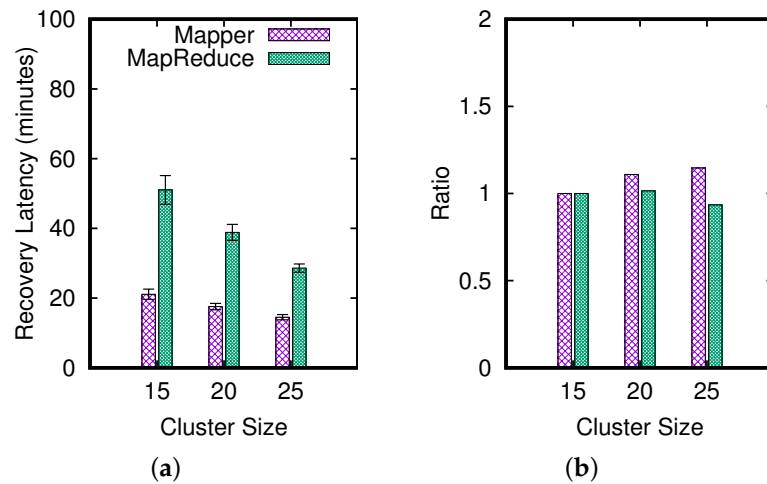


Figure 8. Effects of varying the cluster size (the ratio of 20-node source/shadow/destination cluster to 10-node one).

Figure 8b depicts the metric defined in Equation (3). The experimental results show that the mapper-based and mapreduce-based solutions scale well and effectively parallelize the recovery task to nodes in the system.

5. Related Works

We discuss in this section a number of potential tools offered by HBase for backup and recovery for data items (i.e., key-value pairs) in any table. These tools include *Snapshots*, *Replication*, *Export*, and *CopyTable*.

In the HBase, HFiles can be manually copied to another directory in the HDFS (to a remote HDFS) to create another HBase cluster that reclaims the stored HFiles. With such an approach, users cannot specify the data items (based on columns/qualifiers) to be restored. In addition, providing any point-in-time backup and thus recovery is impractical for such a backup approach. More precisely, to support any point-in-time backup and recovery, the users should copy HFiles upon every newly inserted/updated data item. If so, the introduced HFiles may overwhelm the storage space as most data items in the files are redundant.

Existing tools offered by HBase have provided mechanisms for backup and/or recovery for data items stored in database tables. Snapshots [20], contributed by Cloudera [21] to HBase, allows users to perform the backup periodically (or aperiodically on demand). However, Snapshots currently clones entire HBase tables. It offers no flexibility for users to designate which data items in a table should be replicated. In addition, when Snapshots performs the backup, the serving HBase cluster should be offline.

Replication [22] is a mechanism for synchronizing data items in a table between two HBase clusters. With replication, a newly added/updated key-value pair in a table of an HBase cluster (say, A) is streamlined to another HBase cluster (say, B). This step is achieved when a region server in A logs an update operation, and the logged operation is sent synchronously to the log of some of B's region servers. Any region server in B replays its logged operations as any normal update operation locally introduced to B. The replication scheme performs the backup in the cell level and synchronizes tables that are deployed in different locations consistently. Although the replication scheme allows the serving HBase

cluster, A , to remain online, and users can determine which data items should be replicated in a cell level, it does not offer point-in-time recovery.

Export [23] is another tool provided by HBase that can copy parts of data items in an HBase table as HFiles. Similar to our mapper-based and mapreduce-based solutions, Export is realized based on the MapReduce framework. However, unlike our proposed solutions, Export is not designed to provide point-in-time recovery. Unlike Export, CopyTable [24] offered by HBase not only allows users to perform backup for any interested data items in a table, but restores the backup directly as an HBase table without specifying the metadata with regard to the newly copied table. (The metadata describes critical information with regard to the structure of an HFile. For example, it includes the length of a block, where a block that contains a number of key-value pairs is a fundamental entity in the HBase when fetching key-value pairs from HFiles into the blockcache of a region server.) However, similar to Export, CopyTable supports no point-in-time recovery.

Backup and recovery are critical features required by typical enterprise database users. Existing commercial products implement these features for relational databases. For example, the Oracle database technology provides incremental backup and point-in-time recovery [25]. Incremental backup provided by Oracle mainly stores persistent files that represent database tables, and the backup entity is an entire database table. By contrast, our proposed solutions are developed for entities of cells in database tables. Both Oracle database recovery and our solutions support point-in-time recovery. However, our solutions can restore database tables without disabling the online database service, whereas the Oracle needs the database service to be shutdown. On the other hand, our proposals are built for big data such that the entire database recovery process is fault tolerant.

Research works discussed the recovery issue can also be found in [26,27] later in the literature. Compared with [26,27], our study in this paper thoroughly investigates potential design considerations to the data recovery issue for Col-NoSQL, including recovery time, fault tolerance, scalability, memory space constraint, software compatibility and recovery quality.

Our proposed solutions recovering database tables comprise the table restoration and deployment processes. The table deployment process assumes that keys accessed are distributed uniformly at random. However, our deployment process may implement solutions such as that in [28,29] to improve the performance for non-uniform accesses to database tables based on application knowledges. On the other hand, we may further consider the energy issue such as the study in [30] in our proposals. Moreover, solutions such as [31,32] improving the resource utilization in clouds are orthogonal to our study.

Studies for backup and recovery of file systems can be found in the literature (e.g., [33]). Such studies explore the issues and solutions with regard to the backup and recovery of operations to a file system. In a file system, the operations include CREATE, OPEN, DELETE, etc. to files and directories. Our study aiming at enhancing Col-NoSQL technology is orthogonal to the backup and recovery of a file system.

A recent work, *Delta Lake*, sharing the similar objective of our study presented in this paper, suggests time travel operations for historical data [34]. The time travel operations in *Delta Lake* manipulate the log typically available in a database engine for failure recovery.

6. Summary and Future Works

We have presented the data recovery issue in Col-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 proposed solutions has its pros and cons, which reveals its unique operational scenario.

Our implementations of the two 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. We are currently enhancing our solutions to

support access control lists such that only authenticated users can access authorized data items in the history.

Author Contributions: Project administration, C.-W.C. and H.-C.H.; Software, C.-P.T.; Writing—original draft, H.-C.H.; Writing—review & editing, C.-W.C., H.-C.H. and H.S. All authors have read and agreed to the published version of the manuscript.

Funding: This research received no external funding.

Data Availability Statement: Not Applicable, the study does not report any data.

Acknowledgments: Hung-Chang Hsiao was partially supported by Taiwan Ministry Science and Technology under Grants MOST 110-2221-E-006-051, and by the Intelligent Manufacturing Research Center (iMRC) from The Featured Areas Research Center Program within the framework of the Higher Education Sprout Project by the Ministry of Education (MOE) in Taiwan.

Conflicts of Interest: The authors declare no conflict of interest.

Appendix A. Background of Apache HBase

This section presents an overview of the Apache Hadoop HBase [5]. HBase is a *column-based* NoSQL database management system. Clients in HBase can create a number of tables. Conceptually, each table contains any number of *rows*, and each row is represented by a unique *row key* and a number of *column values*, where the value in a column of a row, namely, the *cell*, is simply a byte string. HBase allows clients to expend the number of columns on the fly even though the associated table has been formed and existing rows of data are present in the table.

A table in the HBase can be horizontally partitioned into a number of *regions*, where the regions are the basic entities that accommodate consecutive rows of data. Regions are disjointed, and each row can be hosted by a single region only. The HBase is designed to be operated in a cluster computing environment that consists of a number of *region servers*. In default, if the number of regions of a table is c and the number of region servers available to the table is n , then the HBase approximately allocates $\frac{c}{n}$ regions to each region server, and each among c regions can only be assigned to one of the n region servers.

When inserting a row of data (or a cell data in a row), the HBase first identifies the region server r , which hosts the region responsible for the data item to be updated. Precisely, if the data contained is assigned with the row key k , then the starting key (denoted by l) and the ending key (denoted by u) managed by the region responsible for the data item must be $l \leq k \leq u$. Once the region server r is located, then the written data is sent to r , which is buffered in r 's main memory, namely, *memstore* in the HBase, to be flushed in batches into the disk latter.

In the HBase, the data in a region is represented as *HFiles* stored in the underlying *Hadoop Distributed File Systems* (HDFS) [7]. The basic format of an HFile is a set of *key-value* pairs. By a key-value pair, we mean that (1) the *key* of a row data is the row key plus the column name and (2) the *value* is the cell value that corresponds to that key. We note that key-value pairs are elementary elements that represent data stored in the HBase. In this paper, we explicitly differentiate the *row key* from the key that uniquely identifies a key-value pair, where the latter contains the former. For example, consider a record representing customer information that includes the name of a customer and his/her contact address. Then, we may create an HBase table with unique IDs that differentiate among customers. These IDs form the set of row keys, and in addition to a unique ID of a row, each row contains two extra fields that consist of the name and the contact address of a customer. To access the cell of such a table, we specify the key-value pair as (x, y) , where x is the key such as *customerID:name* and y can be *John*, for example.

Ideally, if an HBase table is partitioned into k regions, and each region is included in the HDFS as an HFile, then k HFiles are generated and maintained even with failures, and each HFile maintains consecutive keys of key-value pairs stored in the file. (The HDFS guarantees the reliability of storing an HFile with replication. Specifically, an HFile may

be split into c disjoint chunks, and each chunk has three replicas by default, which are maintained at distinct locations, i.e., *datanodes* in the HDFS.) However, in practice, a region server may introduce several HFiles for a particular region such that pipelining the data to the files can be performed rapidly. More precisely, a region server may manage several memstores for a particular region in its main memory, and it streamlines these memory blocks to the file system in a FIFO manner. Consequently, for a region, >1 HFiles may be present in the HDFS, which results in a lengthy delay in accessing key-value pairs in a region. In the HBase, there is a process called *compaction* that merges >1 HFiles into a single one. As HFiles are often generated over time and the compaction process takes considerable time, the compaction is typically designed to reduce an excess number of HFiles that represent a region instead of merging all the HFiles into a single one.

To read a data item stored in the HBase, we shall clearly specify the key of the data item. Basically, the key is x of the corresponding key-value pair (x, y) . However, HBase also supports accesses with partial string matching for x by providing scan and filter operations. To simplify our discussion, we are given x to fetch y . The region server, say r , that manages the region that comprises the key of y checks if its memstore has a latest copy of y . If not, r looks up its *blockcache*, which implements an in-memory caching mechanism for frequently accessed data objects. If the cache does not contain the requested data item (i.e., cache miss), then r goes through the HFiles of the region stored in the HDFS until it finds the requested copy. To speed up the access, *bloom* filters may be used to discard the HFiles that do not store y . As the HBase allows a client to query historical data (e.g., the latest three versions of y), and these versions may appear to distinct HFiles, manipulating several files is thus required.

Write-ahead log (WAL) is a critical mechanism in the HBase for failure recovery. When issuing a key-value pair to the HBase, the region server that manages the associated region logs operations for later playback in case the region server crashes. Precisely, when the region server receives a newly inserted (or updated) key-value pair, it first updates the memstore of the region responsible for the key-value pair and then appends the update in the end of WAL. Each region server maintains a single WAL file that is shared by all regions hosted by the server. A WAL file is essentially an HDFS file. Thus, each of the chunks that assemble the file can be replicated to distinct datanodes, thereby ensuring the durability of key-value pairs committed for clients.

Finally, HBase offers the *observer* mechanism. With observers, third-party codes can be embedded into the HBase write flow such that any normal update in the HBase can additionally perform housekeeping routines (e.g., checkpoints and compactions) without modifying the HBase internals, thereby leveraging the compatibility of our proposed solutions with different HBase versions.

References

1. Deka, G.C. A Survey of Cloud Database Systems. *IT Prof.* **2014**, *16*, 50–57. [CrossRef]
2. Leavitt, N. Will NoSQL Databases Live Up to Their Promise? *IEEE Comput.* **2010**, *43*, 12–14. [CrossRef]
3. Apache Cassandra. Available online: <http://cassandra.apache.org/> (accessed on 17 February 2022).
4. Couchbase. Available online: <http://www.couchbase.com/> (accessed on 17 February 2022).
5. Apache HBase. Available online: <http://hbase.apache.org/> (accessed on 17 February 2022).
6. MongoDB. Available online: <http://www.mongodb.org/> (accessed on 17 February 2022).
7. Apache HDFS. Available online: http://hadoop.apache.org/docs/r1.2.1/hdfs_design.html (accessed on 17 February 2022).
8. MySQL. Available online: <http://www.mysql.com/> (accessed on 17 February 2022).
9. Apache Phoenix. Available online: <http://phoenix.apache.org/> (accessed on 17 February 2022).
10. DB-Engines. Available online: https://db-engines.com/en/ranking_trend/wide+column+store (accessed on 17 February 2022).
11. Dean, J.; Ghemawat, S. MapReduce: Simplified Data Processing on Large Clusters. In Proceedings of the 6th Symposium Operating System Design and Implementation (OSDI'04), San Francisco, CA, USA, 6–8 December 2004; pp. 137–150.
12. Apache Hadoop. Available online: <http://hadoop.apache.org/> (accessed on 17 February 2022).
13. Cooper, B.F.; Silberstein, A.; Tam, E.; Ramakrishnan, R.; Sears, R. Benchmarking Cloud Serving Systems with YCSB. In Proceedings of the ACM Symposium Cloud Computing (SOCC'10), Indianapolis, IN, USA, 10–11 June 2010; pp. 143–154.
14. The Network Time Protocol. Available online: <http://www.ntp.org/> (accessed on 17 February 2022).
15. HBase Regions. Available online: <https://hbase.apache.org/book/regions.arch.html> (accessed on 17 February 2022).

16. HBase APIs. Available online: <http://hbase.apache.org/0.94/apidocs/> (accessed on 17 February 2022).
17. IBM BladeCenter HS23. Available online: <http://www-03.ibm.com/systems/bladecenter/hardware/servers/hs23/> (accessed on 17 February 2022).
18. Webster, C. *Hadoop Virtualization*; O'Reilly: Sebastopol, CA, USA, 2015.
19. VMware. Available online: <http://www.vmware.com/> (accessed on 17 February 2022).
20. HBase Snapshots. Available online: <https://hbase.apache.org/book/ops.snapshots.html> (accessed on 17 February 2022).
21. Cloudera Snapshots. Available online: http://www.cloudera.com/content/cloudera-content/cloudera-docs/CM5/latest/Cloudera-Backup-Disaster-Recovery/cm5bdr_snapshot_intro.html (accessed on 17 February 2022).
22. HBase Replication. Available online: <http://blog.cloudera.com/blog/2012/07/hbase-replication-overview-2/> (accessed on 17 February 2022).
23. HBase Export. Available online: https://hbase.apache.org/book/ops_mgt.html#export (accessed on 17 February 2022).
24. HBase CopyTable. Available online: https://hbase.apache.org/book/ops_mgt.htm#copytable (accessed on 17 February 2022).
25. Oracle Database Backup and Recovery. Available online: http://docs.oracle.com/cd/E11882_01/backup.112/e10642/rcmintro.htm#BRADV8001 (accessed on 17 February 2022).
26. Matos, D.R.; Correia, M. NoSQL Undo: Recovering NoSQL Databases by Undoing Operations. In Proceedings of the IEEE International Symposium Network Computing and Applications (NCA), Boston, MA, USA, 31 October–2 November 2016; pp. 191–198.
27. Abadi, A.; Haib, A.; Melamed, R.; Nassar, A.; Shribman, A.; Yasin, H. Holistic Disaster Recovery Approach for Big Data NoSQL Workloads. In Proceedings of the IEEE International Conference Big Data (BigData), Atlanta, GA, USA, 10–13 December 2016; pp. 2075–2080.
28. Zhou, J.; Bruno, N.; Lin, W. Advanced Partitioning Techniques for Massively Distributed Computation. In Proceedings of the ACM SIGMOD, Scottsdale, AZ, USA, 20–24 May 2012; pp. 13–24.
29. Wang, X.; Zhou, Z.; Zhao, Y.; Zhang, X.; Xing, K.; Xiao, F.; Yang, Z.; Liu, Y. Improving Urban Crowd Flow Prediction on Flexible Region Partition. *IEEE Trans. Mob. Comput.* **2019**, *19*, 2804–2817. [[CrossRef](#)]
30. Wu, W.; Lin, W.; He, L.; Wu, G.; Hsu, C.-H. A Power Consumption Model for Cloud Servers Based on Elman Neural Network. *IEEE Trans. Cloud Comput. (TCC)* **2020**, *9*, 1268–1277. [[CrossRef](#)]
31. Ye, K.; Shen, H.; Wang, Y.; Xu, C.-Z. Multi-tier Workload Consolidations in the Cloud: Profiling, Modeling and Optimization. *IEEE Trans. Cloud Comput. (TCC)* **2020**, preprint. [[CrossRef](#)]
32. Christidis, A.; Moschogiannis, S.; Hsu, C.-H.; Davies, R. Enabling Serverless Deployment of Large-Scale AI Workloads. *IEEE Access* **2020**, *8*, 70150–70161. [[CrossRef](#)]
33. Deka, L.; Barua, G. Consistent Online Backup in Transactional File Systems. *IEEE Trans. Knowl. Data Eng. (TKDE)* **2014**, *26*, 2676–2688. [[CrossRef](#)]
34. Delta Lake. Available online: <https://databricks.com/blog/2019/02/04/introducing-delta-time-travel-for-large-scale-data-lakes.html> (accessed on 17 February 2022).