RECODS: Replica Consistency-On-Demand Store

Yuqing Zhu #1, Philip S. Yu *2, Jianmin Wang #3

#School of Software, Tsinghua University, Beijing 10084, China
[1]zhu-yq@mails.thu.edu.cn, [3]jimwang@tsinghua.edu.cn

*Department of Computer Science, University of Illinois at Chicago, Chicago IL 60607, USA
[2]psyu@uic.edu

***Abstract*—Replication is critical to the scalability, availability and reliability of large-scale systems. The trade-off of replica consistency vs. response latency has been widely understood for large-scale stores with replication. The weak consistency guaranteed by existing large-scale stores complicates application development, while the strong consistency hurts application performance. It is desirable that the best consistency be guaranteed for a tolerable response latency, but none of existing large-scale stores supports maximized replica consistency within a given latency constraint. In this demonstration, we showcase RECODS (REplica Consistency-On-Demand Store), a NoSQL store implementation that can finely control the trade-off on an operation basis and thus facilitate application development with on-demand replica consistency. With RECODS, developers can specify the tolerable latency for each read/write operation. Within the specified latency constraint, a response will be returned and the replica consistency be maximized. RECODS implementation is based on Cassandra, an open source NoSQL store, but with a different operation execution process, replication process and in-memory storage hierarchy.**

I. Introduction

The occurrence of NoSQL stores and cloud storage attracts attentions from both industry and academia, as the inevitable arrival of Big Data. Large-scale stores have been widely employed in a wide range of online services over the past few years. Web search, social networking and recommendation, stock trading, webstore, and gaming represent a few prominent examples of such services. Scalability, availability and reliability are the three properties desired by these services. Most of these services also require a deployment across multiple data centers. The geographical distribution lends data reliability and availability to these services. Replication is one key technique to achieve these favorable properties.

As unavailable services easily drive away users, many large-scale stores choose to guarantee only eventual consistency for online service, e.g., Dynamo [1]. But the unknown consistency status on eventual consistency also greatly complicates application development. Thus some researchers pointed out that, it is not fair that NoSQL stores guarantee only eventual consistency [2], [3]. Multiple levels of consistency need to be provided to cloud applications. Cloud applications need to be provided with appropriate consistency for best performance [4], [5]. It is most desirable that applications be guaranteed the best consistency within their tolerable latency [6].

Among applications requiring strong consistency, some can tolerate long writes, and some long reads. With RECODS, the strong consistency can be guaranteed through either consistent write or consistent read, i.e., write/read with infinite latency bounds. Applications tolerating long writes can specify consistent writes and fast reads, while those tolerating long reads can specify fast writes and consistent reads. Applications not caring consistency can specify their maximum tolerable latency, so that future operations can have a stronger possibility to read a consistent values. Herein, application developers can actively choose the desirable latency vs consistency trade-off with RECODS.

Consider the following use cases of dynamic web pages. Studies report that web users leave a website if the web page does not display in three seconds. After subtracting all the time required for foreground processing, the time left for background data access becomes less than 100 milliseconds. Due to the importance of the time limit, weaker consistency and stale data become tolerable. With RECODS, developers need only specify a latency bound with 100 milliseconds for the related data access operation. This will guarantee that the page be displayed before users become impatient and leave. On uploading data, users are more patient to wait. Therefore, the application can set a larger latency constraint such that the uploaded data is written with consistency. Later, when the uploaded data is requested for display, the read will take shorter latency for a consistent value. In some cases, if the consistency of some data is very important, e.g., recent posts by close friends, the application can specify an infinite latency bound to get a consistent value. Without RECODS, it is very complicated to develop an application that meets the above requirements.

In the remainder of this demonstration proposal, we first sketch RECODS' system design, focusing on the operation execution process, the replication process and the storage structures. Then, we proceed to describe our demo scenario. We will demonstrate how RECODS can guarantee replica consistency on demand through an Olympic medal chart application.

II. RECODS Design

RECODS consists of connected nodes with computing and storage resources. Nodes are distributed over one or more data centers. The system architecture is *symmetric*. Nodes and replicas can be equally accessed. If an operation request arrives at a receiving node without the corresponding data, this request is forwarded to the closest node holding a replica. Operation results are forwarded back to the receiving nodes and then returned to the requester. The operation latency is computed

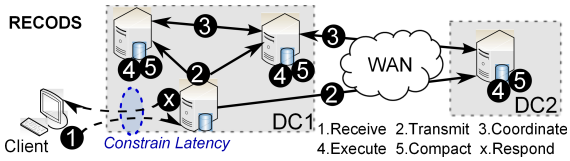

Fig. 1. RECODS's staged replication in a multi-DC environment.

from the time the request arrives at the receiving node till the time the receiving node responds to the requester. Figure 1 illustrates the above process.

There are three basic assumptions behind RECODS design. First, network communication and disk access times take up a major portion of the cross-node replication time. Second, multiple-copy existence and disk storage are two important measures for durability guarantee. Third, the execution of multiple operations can also cause a long latency, especially for read-test-write operations and compactions.

With the above assumptions, RECODS contributes in the following aspects. RECODS breaks the replicated operation execution process into six major stages (Figure 1) to allow flexible consistency control. Stage is the basic unit to guarantee durability and failure tolerance. The more stages are taken for an operation, the better consistency is guaranteed. The stages are further decomposed into minute steps, each of which is related to limited processing time. The small granularity makes latency prediction and bounding possible. Based on the latency given by user, an operation execution process is constructed by finding a subset of steps that meets the latency requirement and maximizes replica consistency. To support the execution process for on-demand consistency control, RECODS implements multi-level storage structures over Cassandra to enable the control and to hold writes durably for each stage. In the following, the RECODS design is briefly introduced. A more detailed description of RECODS exists [7].

A. Operations with Latency Bound Specification

The key API mainly supports three read/write operations, i.e. ***read***, ***write***, and ***readTestWrite***. The *readTestWrite* operation cannot be easily implemented in NoSQL stores guaranteeing eventual consistency, but it is naturally supported by RECODS. We assume the API operates over a flexible table model similar to that of Cassandra [8] and BigTable [9].

The basic data unit for these operations is column, which is referenced by *table*, *row key*, *column family*, and *column*. Here the column may also contain a super column name (as in Cassandra), if the column family is declared to be one with super columns. Besides the reference to the data unit, an operation must also be specified with the expected latency bound ***tBound***. The operation will return a response within the latency *tBound*.

The *write* and *readTestWrite* operations can be specified with an *ordered* parameter. If *ordered* is evaluated to *true*, the operation is an in-order operation. In-order operations are executed in the same arrival order with other in-order operations. If *ordered* is false or not given, the operation has a lower processing priority than in-order operations, so its execution order is not guaranteed. The *readTestWrite* operation must also be specified with two data units and three values. If reading the first data unit returns a value equal to the first given value, the second data unit is set to the second value; otherwise, the second data unit is set to the third value. Note that, the two data units must reside in the same row and column family, i.e., the same tablet. In the following, we also refer to a *readTestWrite* as a write.

Write Bound. Writes always get accepted and responded in RECODS. If a write is specified with an infinite response latency, all previous ordered writes on the same data unit are guaranteed to be applied before this new write, leading to a following instantaneous read returning a recent value. The minimum processing latency for a write is the time RECODS can guarantee the durability of the write, e.g., recording the write in a node-local log or forwarding the write to an adequate number of nodes. A response latency smaller than the minimum processing latency would be ignored. A write with an intermediate response latency will lead to partial processing of this write and/or previously unprocessed writes. A larger response latency leads to a more recent value for a following instantaneous read, or a shorter processing latency for a following consistent read with an infinite latency specification.

Read Bound. A read is guaranteed to be forwarded to its corresponding replica in RECODS. If a read is specified with an infinite response latency, all previous ordered writes on the same data unit are guaranteed to be applied before this new read. That is, the read returns the most recent value by ordered writes. The minimum processing latency for a read is the time that the receiving replica processes the read locally. If there are previous unexecuted writes on the corresponding data unit, RECODS returns NULL for the read, if given a response latency smaller than or equal to the minimum processing latency; otherwise, RECODS returns what the replica reads locally. A read with an intermediate response latency will lead to partial processing of previously unexecuted writes before processing the read. A larger response latency leads to a more recent value returned by the read.

B. Staged Replication Process

There are six stages, including *reception*, *transmission*, *coordination*, *execution*, *compaction* and *acquisition*. In the *reception* stage, writes are received by one of the replica node. Writes are transmitted to all other replicas in the transmission stage. Since a write can be submitted to any replica, writes must go through a *coordination* stage before execution. In this way, all replicas are guaranteed to execute the same write sequence in the *execution* stage, thus avoiding conflict resolution. The processing of a write can also go through the *compaction* stage that compacts previously appended results of write executions. The compaction stage is not directly related to replica consistency, but this stage helps to speed up the *acquisition* stage for a later read.

RECODS replication strategy allows *update-anywhere*, *eager synchronous* **and** *lazy asynchronous* replication simultaneously. Figure 2 demonstrates the RECODS replica control process for a write. The processing flow of a write can

contain at most five stages, though it must always contain the reception stage. A consistent write with eager replication goes through all the five processing stages. The response latency is bounded by processing the maximum number of stages possible within the time bound. Taking fewer stages leads to a less consistent write with lazy asynchronous replication. Read request processing can also contain at most five stages (except for the reception stage). The acquisition stage is required for the read processing to return the requested value. The other four stages handle writes previously received but not yet processed. Though only the acquisition stage actually processes the read request for the result, the first four stages can lead to a more consistent result returned by the read. With the best consistency status, a consistent read needs only to process the acquisition stage.

RECODS orders a chosen set of stages as in Figure 2. Though the processing can actually take the stages in any order, the order in Figure 2 guarantees the output from the previous stage is taken as input by later stages. For example, the reception stage always precedes the other stages in the processing; and the coordination stage always follows the transmission stage. Notice that, a response can be returned to the requester at the end of any stage. Writes for transmission and coordination stages are logged before response. The logging is for the purpose of durability and failure tolerance. A new write can be processed whenever the preceding request is responded. The better the consistency status is, the fewer the unexecuted writes are left. In other words, the fewer stages and writes a consistent read needs to process, indicating a faster response. On the other hand, the better the consistency status is, the fewer writes are left unprocessed; thus, the more recent value can be returned by a read within the same latency bound.

C. Stepwise Bounding of Latency

In order to bound latency, the processing time needs to be predicted. RECODS borrows the idea from Riemann integral in latency prediction. It decomposes stages further into minute steps, which costs no more than millisecond processing time. The step processing time is then approximated and predicted by linear functions. Statistics are collected on the processing of each step to enable approximation. RECODS computes the step subset for execution based on the latency estimation and the stage boundary. The stage boundary must be complied for the sake of durability and failure tolerance. Besides that the steps for read/write processing can be flexibly decided, the number of writes to be processed can be decided individually for each step. The longer latency is specified, the more steps and the more writes can be chosen; and vice versa. Thus, RECODS not only computes the step subset for each operation execution process, but also the corresponding set of writes. RECODS computes the path after receiving the operation and before processing it.

As the selection and processing of stages are in the order specified in Figure 2, we can observe that the more stages means the more executable or executed writes. Executing more writes, or producing more executable writes, increases the

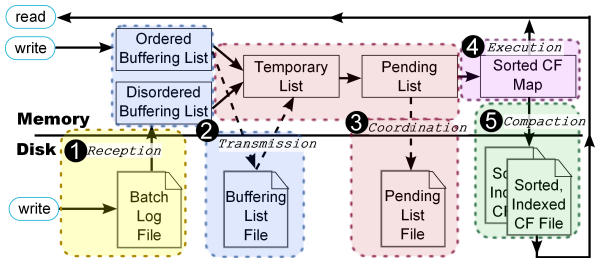

Fig. 2. RECODS's staged processing flow for a write and the supporting storage structures on one node. Stages are ordered by the numbers. Response is allowed at the end of any stage.

probability that a following read returns a consistent value. The only way to increase this probability is to decrease the number of unexecuted writes, and shorten the time for keeping unexecuted writes for the most consistent replica. Summarizing the above analysis, we have the following principles for the step and the write subset computation: (1) the path either covers or bypasses a stage to guarantee durability and failure tolerance; (2) the step subset is maximized and its execution time is within the given bound; (3) the set of writes processed within a step is maximized; (4) the path for a write covers the reception stage, and that for a read the acquisition stage. To directly improve replica consistency status, the numbers of executed and executable writes are maximized first, thus the maximum number of writes to be executed is computed from stage execution, coordination, transmission, to stage compaction.

D. Storage Structures

Figure 2 demonstrates RECODS' multi-level storage structures. *Batch Log File(bl-file)* for the *reception* stage stores writes in disk. The order to execute writes in bl-file is not guaranteed. bl-file enables RECODS to receive writes even under network partition, guaranteeing availability. After network partition is fixed, writes in bl-file are sent to other replicas, following the write execution process in normal cases, except that these writes are not guaranteed with execution order. That is, the *ordered* parameter is set to *false* for writes received under network partition.

Without network partition, the received writes can be transmitted to all reachable replicas in the *transmission* stage. Before transmission, writes requiring to be executed in order are stored in *Ordered Buffering List (ob-list)*, while those not requiring order are in *Disordered Buffering List (db-list)*. The *ordered* parameter signifies whether a write is an ordered one or a disordered one. The ob-list and db-list constitute the *Buffering List (b-list)*. Thresholds can be set for b-list. If the thresholds are reached, newly received ordered writes are logged in the *Buffering List File (b-file)*, while disordered writes are appended to the bl-file. When there are not enough writes for coordination in b-list, writes in b-file are first retrieved and then those in bl-file.

The *coordination* stage exploits *Temporary List (t-list)* to temporarily store writes under coordination. Coordinated writes are stored in *Pending List (p-list)*, pending for execution. If not all replicating nodes participate in the coordination stage, the coordinated writes are also logged in the *Pending*

Fig. 3. Olympic Medal Counts Displayed with Different Latency Specifications.

List File (p-file). Temporarily unavailable nodes can later request the file for a catch-up.

In the *execution* stage, an executed write updates one or more columns of the corresponding row in a certain column family. The updated column values are stored in its *Sorted Column Family Map (cfMap)*. Usually coordinated writes in p-list are executed immediately since the execution takes a relatively short time and directly improves consistency status.

Either when the *compaction* stage is initiated or a cfMap reaches its size threshold, a cfMap is flushed out as a *Sorted, Indexed Column Family File (cfFile)*. There may be more than two cfFiles for one column family simultaneously if a large number of writes pour in. The cfFiles can overlap in row keys. A cfFile compaction, similar to the leveled compaction of Cassandra, is initiated when the number of cfFiles reaches the configured limit, or when a compaction stage is initiated actively. Actively initiating compaction at spare time can effectively avoid the sudden surge of resource consumption and the huge performance degradation [10] due to the passive large-scale compaction.

In the *acquisition* stage, read requests are first directed to the cfMap, and then the cfFiles. The cfMap is sorted by row key, and the cfFiles are indexed to fasten the reading process.

The execution process can thus be summarized as follows. A replica puts a newly received write in the b-list and send it if given a tight latency bound; or, the replica returns the accessible value on read. Given enough time, the replica can execute coordinated writes in p-list, compact the executed writes, request the leader for a write sequence coordination process, or transmit disordered writes in bl-file. A replica with spare resources, e.g., CPU cycles, network bandwidth and memory, can similarly take some or all of the stages to actively improve its consistency status for future requests.

III. Demo Scenario

We will demonstrate RECODS through the Olympic medal chart scenario. Note that the demonstrated Olympic medal chart scenario will have exceptionally frequent and concurrent read/write requests arriving at multiple nodes in RECODS. Three computers or laptops will be set up for the demonstration. One displays in realtime the throughput, latency, and read/write ratio information of data accesses in the backend RECODS. The other two displays the application web pages as a comparison of different access patterns. Figure 3 demonstrates the two web page interfaces users will see. A user can first select a country as his/her focused country. The read and write latencies can thus be specially set for this country to guarantee an up-to-date record in the medal chart. A default read/write latency can be set for other countries' records. This default latency is usually small so that the records get displayed quickly, since the user is less concerned whether these records are up-to-date or not. In the demo, to check the consistency with regard to latency, we will demonstrate that the larger the read latency bound is, the more recent value the read returns. The web page on the left of Figure 3 has China as the focused country while USA is set on the right web page. Records for the two countries are up-to-date respectively on the corresponding web pages. The default latency for the left web page is larger than that on the right, thus records for other countries on the left are more up-to-date than those on the right. Participants to the conference will be invited to set the read/write latencies for the web pages reporting the Olympic medal counts.

References

[1] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels, "Dynamo: Amazon's highly available key-value store," *SIGOPS Operating Systems Review*, vol. 41, no. 6, pp. 205–220, 2007.

[2] E. Brewer, "Pushing the cap: Strategies for consistency and availability," *Computer*, vol. 45, no. 2, pp. 23 –29, feb. 2012.

[3] R. Ramakrishnan, "Cap and cloud data management," *Computer*, vol. 45, pp. 43–49, 2012.

[4] M. L. Kersten, S. Idreos, S. Manegold, and E. Liarou, "The researcher's guide to the data deluge: Querying a scientific database in just a few seconds." *PVLDB*, vol. 4, no. 12, pp. 1474–1477, 2011.

[5] D. Florescu and D. Kossmann, "Rethinking cost and performance of database systems," *SIGMOD Record*, vol. 38, no. 1, pp. 43–48, 2009.

[6] T. Hoff, "10 ebay secrets for planet wide scaling," November 2009, http://highscalability.com/blog/2009/11/17/10-ebay-secrets-for-planet-wide-scaling.html.

[7] Y. Zhu, P. S. Yu, and J. Wang, "Latency bounding by trading off consistency in nosql store: A staging and stepwise approach," 2012, http://arxiv.org/abs/1212.1046.

[8] A. Lakshman and P. Malik, "Cassandra: A decentralized structured storage system," *SIGOPS Oper. Syst. Rev.*, vol. 44, no. 2, pp. 35–40, Apr. 2010.

[9] F. Chang, J. Dean, S. Ghemawat, W. C. Hsieh, D. A. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. E. Gruber, "Bigtable: a distributed storage system for structured data," in *Proceedings of OSDI 2006*. Berkeley, CA, USA: USENIX Association, pp. 15–15.

[10] S. Patil, M. Polte, K. Ren, W. Tantisiriroj, L. Xiao, J. Lopez, G. Gibson, and A. Fuchs, "Ycsb++: Benchmarking and performance debugging advanced features in scalable table stores," in *Proceedings of SoCC 2011*. New York, NY, USA: ACM, 2011.