Finding Consistency in an Inconsistent World: Towards Deep Semantic Understanding of Scale-out Distributed Databases

Neville Carvalho, Hyojun Kim, Maohua Lu, Prasenjit Sarkar,
Rohit Shekhar, Tarun Thakur, Pin Zhou, Remzi H. Arpaci-Dusseau*
*Datos IO, University of Wisconsin–Madison**

Abstract

We present a new problem in data storage: how to build efficient backup and restore tools for increasingly popular Next-generation Eventually Consistent STorage systems (NECST). We show that the lack of a *concise, consistent, logical view* of data at a point-in-time is the key underlying problem; we suggest a *deep semantic understanding* of the data stored within the system of interest as a solution. We discuss research and productization challenges in this new domain, and present the status of our platform, Datos CODR (Consistent Orchestrated Distributed Recovery), which can extract consistent and deduplicated backups from NECST systems such as Cassandra, MongoDB, and many others.

1 Introduction

For much of the history of commercial computing infrastructure, data was stored in a simple and centralized manner. An excellent example is found in traditional systems such as Sun's Network File System [2, 19], in which hundreds of clients connected to a small number of servers; while clients may cache copies of files (in memory or local disks [12]), a single logical copy of data is persistently stored in a well-known format on server-side disks.

This centralization not only eased the design and implementation of said file services themselves [12, 19], but also (importantly) sparked the creation of an entire ecosphere of data management tools and techniques: snapshots to find data at particular points in recent history [11], replication via RAID [18] to tolerate disk failures, backups to protect against more insidious forms of data loss [13, 24], and long-term archives and cross-site replicas to enable restoration after serious data disasters [14]. Such tools are critical as they enable enterprises to store and manage data to meet their availability, reliability, and performance needs. These tools are more easily implemented in the centralized design, as information can be accessed in a single place (the server).

However, the central server no longer rules the world of storage. One critical change is the rise of *diversity*; instead of a single storage solution (e.g., an NFS filer), organizations host data in a vast collection of traditional storage systems (such as EMC [7] and NetApp [11] block and

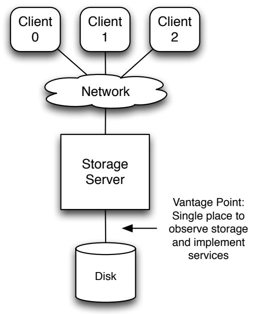
Client 0
Client 1
Client 2
Network
Storage Server
Vantage Point: Single place to observe storage and implement services
Disk

Figure 1: **Traditional Centralized Storage.**

file servers), in traditional SQL databases [21], in scalable distributed file systems such as GFS [8] and HDFS [20], and in a new class of NoSQL distributed key-value storage systems such as MongoDB, Cassandra, and Redis [1, 5]. Each system enables different applications and workflows (e.g., traditional query processing on SQL databases, advanced analytics on HDFS, and scalable web services on a NoSQL store), and as a result each are increasingly central to the successful operation of modern businesses and organizations. As an example, consider Apple's cloud services, which includes one of the world's largest Cassandra clusters as well as serious deployments of MongoDB, CouchBase, and HBase [3].

Coincident with the surge of diversity is another key trend, the *distributed eventually-consistent* nature of storage services; modern systems no longer store data on disks (or SSDs) within a single machine, but rather spread data across many machines in replicated fashion; the replication is implemented in an eventually-consistent manner [6, 23], meaning that (some) replicas are updated lazily; only after a long (and sometimes hard to determine) period of time will all replicas reflect the effects of a particular update. Eventual consistency makes it easier to build scalable, available storage; as a result, these systems have gained widespread (and increasing) utilization.

Unfortunately, the combination of diversity and eventually-consistent distribution, while solving numerous application needs, has muddied the waters for the rest of the data administration and management toolchain. How can a tool create a consistent backup, if it cannot readily discern what the current state of the storage system is? How can an efficient deduplicated backup be re-

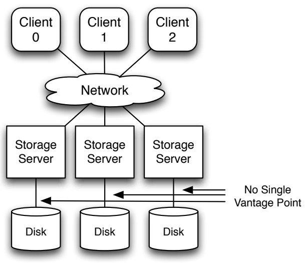

Figure 2: **Modern Distributed Storage.**

alized, if the replicas are not created in an easily understood format? How can these necessary features be realized underneath a large and diverse set of storage systems? Simply put, the techniques and approaches developed for centralized storage no longer function in the diverse and eventually-consistent storage era we have today.

In this paper, we discuss the new challenges that arise in this new era of *Next-generation Eventually-Consistent STorage (NECST)*, particularly as it relates to the entire set of needs found in NECST storage management. We first describe why there is a need for data lifecycle management, and then present detailed examples of the new problems that arise in the NECST environment. The core problem, as we outline, is simple: that tools and systems cannot readily obtain an *efficient, consistent, and logical view* of data beneath these complex, diverse, and distributed NECST systems. Finally, we put forth a vision that describes how to rectify many of these difficult issues and realize a new era of distributed storage management. Specifically, we believe that the key to success centers upon a *deep semantic understanding* of the data being stored within these new storage systems. Only by monitoring and inspecting I/O traffic and reconstructing its meaning (i.e., whether a quorum has been reached, or exactly how a particular data item has been replicated) can critical NECST management functions be implemented in an efficient and scalable manner.

At Datos IO, we are have built a first-generation storage management platform for NECST systems: CODR (Consistent Orchestrated Distributed Recovery). While we have made significant progress, we also believe there are interesting research challenges left to be addressed; our hope is that this paper spurs further innovation in this critical problem domain.

2 The Need: Travel Back In Time

Before delving into examples, we first answer the simple question: why are tools and systems for managing the life cycle of data needed? The main reason is simple: enterprise organizations of any size and across any industry vertical have a fundamental need to restore and access particular versions of data from different points in time.

The reasons to go back in time are manifold. One common set of problems arise from operational errors (a.k.a. "fat fingers") [10, 17], where an operator mistake leads to corruption or loss of data. Even with excellent protection in place (e.g., highly redundant RAID [18]), operator mistakes can easily lose data; having the ability to recover an older version after such a mistake is critical.

Another typical scenario is disaster recovery [14], where an entire site's data has been lost and must be restored. Being able to quickly create a consistent snapshot of data, and then copy said snapshot to a safe remote site, enables quick recovery in the event of a large-scale catastrophe (e.g., a fire or flood).

One more interesting use case is found in *cloning* [22], in which a (read-only) snapshot of a data set is created. The snapshot can then be used for offline data analytics, allowing updates to the main store to proceed unimpeded.

Of course, being able to access older or different versions of data implies that large amounts of old data must be kept in some format. Fortunately, new compression technologies, such as deduplication [16, 24], have become commonplace. Thus, with space-efficient backup and archival technology, all of the above use cases for data management can be realized without excessive costs.

3 Background

As described earlier, the world of storage is changing. No longer content with simple block-based storage systems [15], industry-standard network file systems [19], or even traditional SQL databases [21], users have started exploring a range of storage systems that pave new ground in how data is stored and accessed. We now describe how these modern distributed storage systems operate, focusing on two important systems: MongoDB and Cassandra.

MongoDB is a cross-platform document-oriented database that eschews the traditional table-based relational database structure and instead uses dynamic schemas and the JSON document format. The design simplifies application development and deployment as schema changes are no longer onerous. MongoDB provides high availability with replica sets where each replica set consists of two or more eventually consistent copies of the data. Each replica set member may act in the role of primary or secondary replica at any time, where the primary replica performs all writes and reads by default; the secondary replicas can also perform reads. When a primary replica fails, the replica set automatically conducts an election process to determine which secondary should become the primary. MongoDB builds sharding on top of replication; the data is split into ranges (based on a shard key) and distributed across multiple (typically three) replica sets.

Apache Cassandra is similar to MongoDB in the adoption of a distributed database model with sharding and replication, as well as the use of eventual consistency to propagate writes through the system. However Cassan-

System	Type	Updates	Shard	Replicate
MongoDB	Document	In-place	Replica set	M-S
Cassandra	Wide-column	Append	Single node	M-M
CouchDB	Document	Append	Single node	M-M
Redis	Key-Value	Append	Single node	M-S
DynamoDB	Doc/K-V	Append	Single node	M-M

Table 1: **Comparison of Different Systems.** *In the right column, M means master, S means Slave.*

dra is different in MongoDB in terms of its data model and the mechanism by which sharing and replication is implemented. Cassandra's data model is a partitioned row store where rows are organized into tables. The first component of a table's primary key is the partition key; within a partition, rows are clustered by the remaining columns of the key. Unlike MongoDB, every node can service reads and writes leading to a masterless replication model. Sharding is based on the partition key of a row.

4 The Problem: Inconsistency

We now discuss some of the specific new problems that arise in trying to create tools that can monitor, snapshot, backup, and restore modern distributed storage systems. All of the problems relate to one fundamental underlying issue: the difficulty of obtaining a *space-efficient, logical,* and *consistent view* of data within the storage system.

Quorum Updates: Quorum-based replication is an age-old method of building distributed data stores [9]. One problem that is common in quorum-based replication schemes is determining the order of updates across replicas, which is essential in deciding which values should comprise a snapshot. For example, many NoSQL databases allow write requests to return success if a quorum of nodes merely acknowledge the receipt of the write request. While this optimization improves performance (i.e., writes can proceed at the speed of memory), it leads to complications in capturing the state of the system. For example, if two write requests to the same database object arrived at two different nodes at roughly the same time, it is difficult to determine a strict ordering between the two write requests. The lack of ordering thus makes it challenging to determine the latest value of the database object at any given point in time.

Let us examine a specific example of a Cassandra cluster with a certain number of nodes operating with a particular quorum consistency level. Imagine two nodes (A and B) that perform a write on the same column of a row at the same timestamp. Because Cassandra resolves order using timestamps, the exact order of these two writes is not known. One could break the non-determinism using a lexicographic order on the value of the column, but this is not necessarily deterministic from the perspective of the application. A backup or archival tool must be able to understand such a situation and create a backup or archive that is meaningful despite this type of difficulty.

Resharding and Migration: Sharding (or partitioning) data items across nodes is essential in building distributed storage [4]; re-sharding them, to distribute load and avoid hotspots, is also essential. Unfortunately, capturing a consistent version of a distributed database is greatly complicated when data is constantly migrating between the nodes of the clustered system. For example, in a MongoDB cluster, documents within a certain key range may move between different shards as the partitioning dictates that the accesses to different shards be balanced across the cluster.

Consider MongoDB, which uses a rebalancing approach to distribute data of a sharded collection evenly across a sharded cluster. When a shard has too many chunks as compared to other shards, MongoDB automatically balances the chunks across shards. The balancing procedure in MongoDB for sharded clusters is entirely transparent to the user and application layer. In such a situation, if a system administrator attempts to create a consistent snapshot of the database cluster, the administrator may discover that the same database key is present in multiple shards. Thus, any data management system must be cognizant of such intricacies and handle it accordingly.

Recovery After Reconfiguration: Even if one were to capture a consistent version of a clustered system, recovery presents another challenge by itself, due to the changing nature of clustered systems. For example, system topology may have changed between the time of backup and and the time of restore (due to node failure or addition). The change in topology implies that the partitioning strategy in the current cluster will be different from that when the version was taken. Reconciling the differences in partitioning strategy is hard if one attempts to restore the clustered system quickly without incurring the expense of a subsequent repair.

Here we present a specific example. Suppose we have a Cassandra cluster with three nodes. The primary keys are partitioned across the three nodes in the system. Assuming a lexicographic partition, we can assume that database entries with keys starting with [A–I] go to Node 1, those with keys starting with [J–S] go to Node 2 and the remaining keys starting with [T–Z] go to Node 3. Let us assume that we are backing up data in this system every day. After a week, we add a fourth node to the Cassandra cluster, following which the entries in the cluster will be re-partitioned across all the four nodes. After the re-partitioning is complete, the database entries with keys in the ranges [A–G], [H–N], [O–T], [U–Z] reside in Nodes 1, 2, 3 and 4 respectively. Due to an operational error following this re-partitioning but before the cluster is backed up, all data is lost. At this point, we have to rely on one of the backups that were taken in the week before the fourth node was added. When any one of these backups is restored to the cluster, entries with the original range splits

[A–I], [J–S], [T–Z] go to Nodes 1, 2 and 3, and no data goes to Node 4. However, this is inconsistent with the current partitioning strategy; consequently, a query for a database entry starting with Z will be redirected to Node 4 which has no data and the query will fail.

Deduplication Difficulty: In traditional systems, data replication is simple to understand at the physical level; for example, in a mirrored RAID system, each block and its replicas are bitwise identical. Thus, when performing deduplication, similar blocks are readily identified.

Unfortunately, in modern clustered storage systems, data copies are not always exactly identical, thus creating a new challenge for data management: how can data management tools create efficient backups or archives if they cannot discern one copy of a data item from another?

Let us examine a specific scenario to understand the problem better. In a multi-master system with multiple replicas like Cassandra, the data stored in each node is quite different. For example, in a Cassandra cluster with 5 nodes n1–n5, configured with 3 replicas, the contents in 5 nodes after inserting 5 rows with key k1–k5 to the cluster (assuming k1–k5 are evenly distributed to the 5 nodes) might be:

```
n1: k1 row, k4 row, k5 row
n2: k1 row, k2 row, k5 row
n3: k1 row, k2 row, k3 row
n4, k2 row, k3 row, k4 row
n5: k3 row, k4 row, k5 row
```

As we can see, none of the replicas are bitwise identical. One may argue that the traditional variable chunk deduplication might work in this scenario if the average chunk size is set to the average row size. However, those techniques require the average row size to be large and the variation of row sizes to be small to avoid an explosion of metadata. Even if the above requirements are met, the contents for the same row (key) in different nodes are still likely different due to the nature of weak consistency in a distributed system.

5 Deep Semantic Understanding

As we have seen above, gaining a consistent, logical view of storage in the NECST environment presents us with new and interesting challenges: quorum updates, resharding and migration, recovery to different topologies, and deduplication under non-identical replicas all complicate storage management (and related tools) considerably. Simply put, the architecture of NECST storage engines forces us to rethink how the entire storage ecosystem around NECST will be realized.

We believe the key to progress on this new and important problem lies in gaining a *deep semantic understanding* of the replicated data that comprises modern storage systems. This understanding consists of numerous facets of how such systems operate, including:

- **Quorum reconciliation.** Unlike traditional storage, where it is relatively easy to tell when an update has taken place, the simple task of knowing when an update has been committed to the storage system is challenging. NECST systems demand that tools and systems that are interested in what is stored within them understand the basics of how quorums are formed, and exactly how and when a data item is safely replicated within the system. By having a comprehensive understanding of the NECST replication protocol, a backup tool can determine the order of updates and form a coherent view of storage.
- **Redundant-copy detection.** Unlike traditional striped or mirrored systems, in which redundancy is easily observed, NECST systems may encode data copies in a non-bitwise-identical fashion. Thus, a NECST backup or archival system must be able to meticulously comb through the NECST system to determine where logically identical copies reside, so as to be able to coalesce them and thus achieve storage-efficient backup.
- **Configuration-oblivious backup and restore.** Distributed systems have frequent configuration changes, scaling up to meet new demands or down when a failure occurs and a system is removed from operation. NECST tools must be able to store, and then recreate, data despite the fact that its configuration has changed.

Deep semantic understanding thus enables a data-management platform to comprehend exactly what data is being stored and how it is updated, enabling the efficient creation of compact backups, snapshots, and archives. However, deep semantic understanding alone is not enough.

Specifically, NECST tools must also meet a number of performance-oriented goals to be effective, adding as little overhead on the main storage data path as possible. Such *data-path minimality* is critical for deployment. Further, NECST tools must scale. The *sine qua non* of modern storage systems are their ability to add nodes to increase capacity and performance; thus, new storage-management tools must be able to keep pace.

Alternate Approaches: Initially, we evaluated simple approaches to quorum reconciliation, redundant-copy detection, and configuration-oblivious backup/restore. For example, we used a distributed key-value store to keep track of every row in a database snapshot; on scanning all the rows in the database snapshot, quorum reconciliation and redundant copy detection were performed using the contents of the key-value store. However, at large scale, the contents of the store cannot be placed in memory without great cost. Alternatively, if stored on disk, access would harm the SLAs that the backup system has promised.

Similarly, one could implement a configuration-oblivious restore by simply using the insert APIs of the target database cluster. This approach impacts the foreground workload and compromises the speed of restore.

Finally, instead of realizing a single platform underneath a range of modern storage systems, each system could provide its own versioning, backup, and related tools. We feel this approach is not viable for the following two reasons. First, it complicates management, requiring administrator knowledge of many tools (one per storage system) instead of one. Second, it is a waste of human effort. Much of the code needed to implement these features would be similar across systems, and building said features within each is thus a waste of time and effort.

6 Datos IO CODR

At Datos IO, we have built a first-generation platform to meet the storage management needs of the NECST era, called CODR (Consistent Orchestrated Distributed Recovery). CODR currently provides backup and restore features for a number of distributed storage systems, such as Cassandra and MongoDB among others. CODR introduces the concept of versioning, where a *version* is defined to be a cluster consistent snapshot of a scale-out distributed database.

Backup proceeds in numerous phases. First, CODR first takes a full snapshot of the database of interest; after this, CODR tracks changes applied to the database and generates incremental versions for the changes.

To capture database contents from the nodes of NECST, CODR can either leverage native local-snapshot support on each node (as with Cassandra) and stitch together a coherent whole from said parts, or obtain the contents by other means, such as extracting contents from files within the underlying local file system (as with MongoDB). The latter requires care; files may change during extraction. These approaches are highly efficient, greatly reducing overheads as compared to querying the data store directly.

The complexity of tracking changes between versions is also different across NECST systems, depending on whether the database is append-only or update-in-place. In general, append-only systems present fewer challenges.

Full and incremental snapshots are transferred, in parallel, to a backup storage system, which can be a single node in smaller deployments, or a cluster in larger-scale settings. At the backup store, CODR must process the collection of local snapshots to realize a version. CODR achieves this end by running an integrated quorum and semantic-deduplication algorithm, resulting in a single, space-efficient copy of the data.

CODR currently supports configuration-oblivious restore via deep indexing of each snapshot. A parallel copy moves the data to the appropriate nodes and induces a lazy database refresh to complete the restore.

7 Conclusions

We have described the burgeoning world of NECST storage, in which eventually-consistent storage systems are an important component in the enterprise datacenter. At Datos IO, we have built a data-management platform for these new and increasingly important storage systems. We believe there are a wide range of interesting and challenging research problems in this space, and hope that others may join us in realizing a world in which NECST storage management is as easy and effective tomorrow as classic storage management is today.

References

[1] Malik, Lakshman. Cassandra: A Decentralized Structured Storage System. LADIS '09.

[2] R. Arpaci-Dusseau, A. Arpaci-Dusseau. Operating Systems: Three Easy Pieces. Arpaci-Dusseau Books, 2014. www.ostep.org.

[3] Asay. Apple's Secret NoSQL Sauce Includes a Hefty Dose of Cassandra. Tech Republic, 2015.

[4] Bronson, Amsden, Cabrera, Chakka, Dimov, Ding, Ferris, Giardullo, Kulkarni, Li, Petrov, Puzar, Song, Venkataramani. TAO: Facebook's Distributed Data Store for the Social Graph. USENIX '13.

[5] Chodorow. MongoDB: The Definitive Guide. O'Reilly, 2013.

[6] DeCandia, Hastorun, Jampani, Kakulapati, Lakshman, Pilchin, Sivasubramanian, Vosshall, Vogels. Dynamo: Amazon's Highly Available Key-Value Store. SOSP '07.

[7] EMC. Symmetrix Enterprise Information Storage Systems.

[8] Ghemawat, Gobioff, Leung. The Google File System. SOSP '03.

[9] Gifford. Weighted Voting for Replicated Data. SOSP '79.

[10] Gray. A Census of Tandem System Availability Between 1985 and 1990. Tandem TR 90.1, 1990.

[11] Hitz, Lau, Malcolm. File System Design for an NFS File Server Appliance. USENIX Winter '94.

[12] Howard, Kazar, Menees, Nichols, Satyanarayanan, Sidebotham, West. Scale and Performance in a Distributed File System. TOCS 6(1), February 1988.

[13] Hutchinson, Manley, Federwisch, Harris, Hitz, Kleiman, O'Malley. Logical vs. Physical File System Backup. In OSDI '99.

[14] Keeton, Santos, Beyer, Chase, Wilkes. Designing for disasters. FAST '04.

[15] Lee, Thekkath. Petal: Distributed Virtual Disks. ASPLOS'96.

[16] Muthitacharoen, Chen, Mazieres. A Low-Bandwidth Network File System. A Low-Bandwidth Network File System. SOSP '01.

[17] Oppenheimer, Ganapathi, Patterson. Why Do Internet Services Fail? USITS '03.

[18] Patterson, Gibson, Katz. A Case for Redundant Arrays of Inexpensive Disks (RAID). SIGMOD '88.

[19] Sandberg. The Design and Implementation of the Sun Network File System. USENIX Summer '85.

[20] Shvachko, Kuang, Radia, Chansler. Hadoop Distributed File System. In MSST '10.

[21] Stonebraker, Rowe. The Design of POSTGRES. TKDE '86.

[22] Subramanian, Sundararaman, Talagala, A. Arpaci-Dusseau, R. Arpaci-Dusseau. Snapshots in a Flash with ioSnap. EuroSys '14.

[23] Terry, Theimer, Petersen, Demers, Spreitzer, Hauser. Managing Update Conflicts in Bayou. SOSP '95.

[24] Zhu, Li, Patterson. Avoiding the Disk Bottleneck in the Data Domain Deduplication File System. FAST '08.