BARNS: Towards Building Backup and Recovery for NoSQL Databases

Atish Kathpal, *NetApp* Priya Sehgal, *NetApp*

Abstract

While NoSQL databases are gaining popularity for business applications, they pose unique challenges towards backup and recovery. Our solution, BARNS addresses these challenges, namely taking: a) *cluster consistent* backup and ensuring *repair free restore*, b) *storage efficient* backups, and c) *topology oblivious* backup and restore. Due to eventual consistency semantics of these databases, traditional database backup techniques of performing quiesce do not guarantee cluster consistent backup. Moreover, taking crash consistent backup increases recovery time due to the need for repairs. In this paper, we provide detailed solutions for taking backup of two popular, but architecturally different NoSQL DBs, Cassandra and MongoDB, when hosted on shared storage. Our solution leverages database distribution and partitioning knowledge along with shared storage features such as snapshots, clones to efficiently perform backup and recovery of NoSQL databases. Our solution gets rid of replica copies, thereby saving ~66% backup space (under 3x replication). Our preliminary evaluation shows that we require a constant restore time of ~2-3 mins, independent of backup dataset and cluster size.

1. Introduction

Recently, we see an increased adoption of NoSQL databases like MongoDB, Cassandra, etc., to address scale and agility challenges of modern applications such as analysis of ever increasing data related to customers, operations and IoT. While these databases are mostly deployed on commodity servers with direct attached disks, many enterprises are turning to high performance and scalable shared storage [1]. More recently, many storage vendors have partnered with NoSQL companies specially with improvement in performance of all-flash storage [3][4]. Shared storage also helps achieve consolidation, better data management and enables independent scaling of compute and storage.

According to IDC, one of the top infrastructural challenges faced by NoSQL deployments is data protection and retention [1]. Although NoSQL databases perform replication for high availability and read performance, in case of logical data corruption, accidental deletion, or ransomware attacks [21], the damage spreads to all the replicas. Ganesan, et. al [26] also highlight the fact that redundancy does not imply fault tolerance. Moreover, according to Gartner, enterprises are looking at the ability to repurpose backup for various use cases such as DevOps, test/dev, analytics and cloud onramping [2]. Hence, there is a need for an efficient and scalable solution for backup and recovery of NoSQL databases.

Unlike traditional RDBMS, backup and recovery of NoSQL databases has various challenges, which are highlighted by Carvalho, et. al [7] and discussed in depth in Section 2. These include taking cluster-wide consistent backup resulting in repair free recovery, redundant copy removal from backup, and topology oblivious backup and restore.

In this paper, we present BARNS - a backup and recovery solution for NoSQL databases, hosted on shared storage. Unlike copy-based backup solutions [18, 19, 20], we leverage light-weight snapshots, which perform copy-on-write, and clone (writeable snapshot) features of shared storage. Our key contributions include:

(a) We present two different approaches, based on the type of NoSQL database: master-less, e.g., Cassandra [10], and master-slave, e.g., MongoDB [6].

(b) Leverage NoSQL database knowledge and techniques to achieve cluster-wide consistency during backup, get rid of logically identical copies and be resilient to topology changes during backup and recovery.

BARNS achieves ~66% space savings by logical reduction of two copies (assuming 3x replica) for both MongoDB and Cassandra. While repair operation (usually required after restoring) is dependent upon the dataset and cluster size in Cassandra, our repair-free restore solution requires only a constant restore time of ~2mins.

2. Background and Related Work

Backup in relational DBs is a well-researched topic [13, 14], but is relatively new in distributed NoSQL databases. While designing a backup and recovery solution for NoSQL databases, we need to consider the category it belongs to: (a) master-less, or (b) master-slave. In case of master-less, e.g., Cassandra, Couchbase, data is scattered across all the nodes, typically through consistent hashing [22]. Generally, no node is an exact replica of another. In master-slave, e.g., MongoDB, Redis, a primary is responsible for accepting all updates and propagating to all secondary nodes. On reaching consistency, primary and secondary nodes hold the same data.

Key challenges in backup and restore for NoSQL DBs are as follows:

Cluster-wide consistent backup: Relational DBs that typically run on a single or handful of nodes, expose APIs to quiesce database [5], which makes it simple to take application consistent backup. However, in case of distributed NoSQL databases, which are *elastic* in nature, *eventually consistent* and do not support cross-node, cross table transactions, it is impractical to perform quiesce of all the nodes to take backup. Moreover, node-level quiesce does not guarantee that data across nodes would be consistent, due to eventual consistency, thereby requiring repair during restore. One can take uncoordinated backup or snapshot of individual nodes resulting in crash consistent backup, such as in OpsCenter [12]. While such a backup is quick, repair operation is inevitable during restore. DataStax recommends running *nodetool repair* command after restoring from a Cassandra snapshot [24]. This impacts recovery time objective (RTO) adversely, making both quiesce-based solution and crash consistent backup unattractive. Thus, there is a need to design a backup and recovery solution for NoSQL databases, which performs quick, cluster consistent backup and provides repair free recovery.

Removal of redundant copies: NoSQL DBs perform replication for high availability and load distribution, but these are not required in backup copy. One might argue that fixed or variable length deduplication, which is generally enabled on shared storage can get rid of replicas. However, these replicas do not de-duplicate due to reasons such as data distribution and layout, compression and encryption [7]. For example, Cassandra distributes and replicates data in such a way that different combination of rows are stored in the data files across the nodes, making it impossible to achieve chunk level deduplication. MongoDB's WiredTiger storage engine appends unique internal metadata, which diminishes opportunity to de-duplicate replicas of same document.

Topology changes: Topology of NoSQL cluster may change across backup and restore. For instance, a node may go down during backup or an additional node may be introduced during restore and vice-versa. In these scenarios, it is important to reconcile the differences in partitioning strategies to avoid repair during restores.

There exist open source tools such as *mongodump* [23], which enable users to take node-level dumps of DB contents and later restore back using *mongorestore* [17]. Such tools require custom scripting to implement cluster level backups and restore, which must address challenges listed above. Solutions like those from Datos IO [18], Talena [19], and MongoDB Ops Manager [20] address these challenges by post processing incremental data streams. These solutions do not integrate with underlying storage features of snapshots and clones, and instead copy data out of the cluster to implement backup workflows. Exact internal details of these solutions are not known as these are proprietary offerings.

In this paper, we present backup and recovery solutions for Cassandra and MongoDB when hosted on shared storage. Our work differs from above solutions, as we do not require deep semantic understanding of replicated data. Instead, we leverage in-built database features or commands and shared storage features to address the challenges discussed previously. Since we refrain from looking into the data stream, our solution provides key differentiating features: (a) perform backup and recovery on data encrypted by databases, (b) compatible with different versions of Cassandra and MongoDB, and (c) extend this solution to other databases similar to Cassandra and MongoDB (using their APIs or commands).

3. BARNS

The aim of BARNS is to honor the consistency semantics of NoSQL databases and achieve space efficient, topology oblivious backup and recovery. Instead of performing recovery at restore-time, BARNS does most of the recovery related work when it creates the backup. We reuse database functionality to create a cluster-consistent backup and leverage light-weight snapshot (performs copy-on-write) and clones (writeable snapshot) features present in most modern storage systems, to gain space-efficiency. We do not stop foreground IOs while taking backup.

3.1. Cassandra Backup/Recovery

Cassandra offers tunable consistency for both reads and writes. To achieve strong consistency, it is recommended to perform write and read operations on quorum nodes [16]. However, we found experimentally that the values with the latest timestamp always win and replicate to other nodes during reads, irrespective of the consistency level. Consider a four node Cassandra cluster with nodes named A, B, C, and D and consisting of keyspaces with a replication factor of 3. Suppose a quorum write request comes to A for a key K1 with value V1, which hashes into B and is replicated to C and D, but both C and D are unreachable. On receiving acknowledgement from only B, A fails the quorum write. However, the latest write to K1 is still present with B. As Cassandra does not provide a way to rollback writes, when nodes C, D become available, the latest value of K1 is propagated to both C and D due to *read repair* or *manual repair*. Thus, our solution must ensure that we meet such consistency semantics in backup. Moreover, Cassandra employs consistent hashing to distribute data to various nodes making it difficult to decide the minimal number of data nodes to backup that can capture the entire data.

The core idea is to perform merge sort on the data files across all nodes. This is like running Cassandra's *compaction* across all nodes. Cassandra's compaction process merges keys based on timestamp, combines columns, evicts tombstones and consolidates data files [11]. Traditionally, compaction is meant to reclaim space and resolve conflicts in data files only for a single node. We leverage this mechanism to achieve cluster-wide consistent and space efficient backup, and a repair-free recovery. To achieve topology oblivious backup and recovery, we save cluster configuration, i.e., health and token assignment of each node.

Backup Workflow

Whenever a write comes to a Cassandra node, it stores the data in an in-memory structure, memtable, and appends it to an on-disk commit log. The memtable is flushed to SSTables on reaching a threshold. We need to backup the SSTables and commit logs per node. Let us consider a simple deployment model of Cassandra on shared storage with one to one mapping between Cassandra nodes and a logical storage container, say an iSCSI LUN. Each data LUN stores commit log and SSTables belonging to a node. BARNS takes Cassandra backup in two phases:

(1) *Light weight backup (LWB)* – In this phase, we capture the topology of the Cassandra cluster: health and token assignment of every node. We take un-coordinated snapshot of data LUNs of only healthy nodes and save the mapping of these snapshots with the token assigned to the individual nodes in a *backup metadata* (bkp_meta) and assign a backup name to it. We save only the token information for dead/unhealthy nodes in bkp_meta. A sample of bkp_meta for a two-node cluster with one unhealthy node is shown below.

```
{"backup_name": 1488869633.586644,
 "cluster_name": "barns",
 "members": [
   {"lun": "/vol/cass1/lun_cass1",
    "snap-name": "17-03-02_01:02:18",
    "stateStr": "Healthy",
    "tokens": "<list of tokens> " }
   {"lun": "/vol/cass2/lun_cass2",
    "snap-name": "",
    "stateStr": "UnHealthy",
    "tokens": "<list of tokens> " }]}
```

If SSTables and commit logs of a single node reside on different LUNs, we need to ensure that the snapshot of both these LUNs is coordinated, e.g., using NetApp® Consistency Group [15]. This helps achieve consistent snapshot per node. However, the snapshot of LUNs across the nodes is un-coordinated as we take care of consistency in post-processing phase.

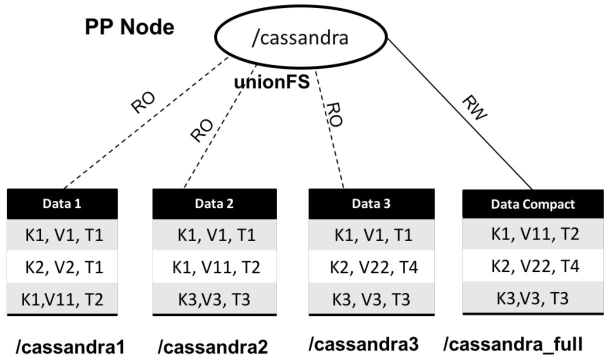

Figure 1: pp-compact phase

(2) *Post-processing phase (PP)* – This phase is responsible for resolving consistency conflicts and removing redundant copies of data by post processing the snapshots taken in LWB phase. We use a different set of Cassandra node(s) for performing this phase, referred to as *pp_node*. This phase is divided in two sub-phases - (a) *pp-flush:* flush commit logs to SSTable, and (b) *pp-compact:* perform compaction across all SSTables. In pp-flush, we clone the snapshots taken in LWB. These clones are discovered and mounted on pp_node at different mount points, say `/cassandra1`, `/cassandra2`, etc. and configured with appropriate tokens as retrieved from bkp_meta. Cassandra instances are started against these mount points. Once Cassandra is running, we initiate commit log flush operation by issuing - **nodetool flush** command, on each clone. At the end of pp-flush, we have new SSTables with higher version numbers created for all the keyspaces on all the clones. We delete all the empty commit logs and stop the cluster.

As shown in Figure 1., the main goal of pp-compact sub-phase is to perform compaction across all the clones mounted during pp-flush. Before running compaction, we need to organize the data stored in multiple mount points in such a way that a single Cassandra instance can access all the SSTables. Hence, we rename SSTable filenames (version number portion) of a keyspace such that it is unique across the various mount points. Further, to store the results of the compaction we create a special LUN, say fullback_lun and mount on say `/cassandra_full` on pp-node. To unify the view for Cassandra, we mount all the previous mount points containing the renamed SSTables (as read-only) and the mount point dedicated to store results i.e., `/cassandra_full` (as read-write) using unionfs-fuse [9] on `/cassandra`. We configure Cassandra such that it uses unionfs mount point, `/cassandra`, as its data directory and takes charge of the tokens of all nodes present in bkp_meta. With the virtue of unionfs, a single Cassandra process can access all the SSTables stored across multiple mount points. Finally, we call Cassandra compaction command, **nodetool compact**, which merges all the SSTables and stores the compacted result on `/cassandra_full`. At the end of pp-compact, fullback_lun contains cluster consistent, space efficient backup of the production cluster. We create a snapshot of fullback_lun and discard all the previous clones created during pp-flush and snapshots taken during LWB. The backup metadata is now updated with the full backup LUN's snapshot name and all the tokens of the initial Cassandra cluster.

Restore Workflow

We clone the snapshot corresponding to the fullback_lun and mount it on all the Cassandra nodes of the restore cluster. We create a different clone per restore node. The tokens from backup metadata are distributed equally to all the restore nodes. Once Cassandra starts on each of the restore nodes, its data directory contains data of all the peers. However, individual Cassandra instances ignore the extra data and only take ownership of the data based on token ring assignment. Thus, restore is oblivious of the number of nodes that were a part of the production cluster during backup. Since the clone provided by most shared storage appliances performs copy-on-write, our recovery solution is also space efficient.

Limitations: Our solution only provides full backup of Cassandra, and is CPU and memory intensive during post processing. In future, we plan to augment our solution with incremental backup. Moreover, since Cassandra allows maximum 1536 tokens to be assigned to a node, a single pp-compact instance will not be able to scale beyond 6 nodes (assuming each node has 256 tokens). To solve this problem, we propose to run multiple instances of pp-compact, each responsible for compacting 6 nodes. This will result in more than one full backup LUN, impacting storage efficiency and requiring some repair across the full backups. We do not flush in-core commit log, resulting in loss of some cached, but acknowledged writes. We can resolve this problem by configuring Cassandra to sync the commit log before acknowledging the client.

3.2. MongoDB Backup/Recovery

Backup Workflow

Unlike Cassandra, MongoDB is a master-slave NoSQL DB. A sharded and replicated MongoDB cluster consists of several shards (partitions) with each shard consisting of a replica set (RS). A RS contains a set of nodes: one primary or master and multiple secondary members. These are complete logical replicas of each other. All updates to a RS first go to primary and eventually propagate to secondary nodes. Hence, an intuitive solution is to take snapshot of primary node LUNs of each RS in the cluster. However, there exist error scenarios that need to be handled. Moreover, given the master-slave architecture, we need to ensure that data within a single shard or RS is consistent during backup. BARNS takes MongoDB backup in two phases:

(1) *Light weight backup (LWB)* - In this phase, we first pause any background inter-shard data migrations by calling "**stop balancer**" API, as recommended by MongoDB [25]. Next, we query the cluster topology and mark the data LUNs (consisting of journal and data files) of all live nodes for backup. If a replica set (RS) has less than quorum number of healthy nodes, we must fail the backup, because the RS cannot elect a primary or serve IOs in such state and needs manual intervention. In the normal case, BARNS triggers an un-coordinated storage snapshot of all LUNs marked for backup. Lastly, we persist backup metadata, which includes a backup name, list of all RS in the cluster and a mapping of each member LUN in the RS to its respective snapshot name (like Cassandra backup metadata).

(2) *Post process (PP)* - We post-process the snapshots taken during LWB in a separate sandbox environment, to bring backup to a cluster consistent state. Each RS can be post processed independently, in parallel or sequentially depending upon resource constraints. For each RS, we mount the member snapshots on post process node(s) and start MongoDB instances to bring up the complete RS. Upon startup, MongoDB replays necessary journal logs and checkpoints them to its data directory, just as though it were recovering from a crash. If there was absence of a stable primary during LWB, MongoDB RS elects a new up-to-date primary from among the secondary nodes, during PP. We take a fresh snapshot of only the primary node in the RS and drop all other previously taken snapshots in LWB. Thus, our solution logically eradicates replica copies in the backup by only retaining primary node's snapshot. Lastly, we update backup metadata to reflect single primary node's snapshot name for each RS.

This phase also helps resolve any error scenarios such as journal corruption of the primary node of a RS due to filesystem or storage faults [26], which was undetected during LWB. As part of post process, MongoDB will be able to identify such corruptions during startup and repair them with help of the secondary nodes, ensuring that restore is repair free. If MongoDB fails to recover from any corruption, which gets detected as part of this phase, we fail the backup and inform user, rather than failing to bring up the cluster during restore. This phase lends the advantages of a) better storage efficiency (~66% saving in 3x replica) by preserving snapshot of only primary's data for each RS, and b) provides a good control point for detecting, correcting and in few unrecoverable cases, alerting about errors during backup. Unlike Cassandra, since MongoDB's PP does not involve reading all the data, and leverages snapshots the backup is incremental.

Restore Workflow

Restore involves mapping the final set of RS snapshots captured as part of PP, to corresponding RS in the restore cluster. The backup metadata helps achieve this mapping. It requires that replica set IDs in backup topology match those in restore cluster. The number of members within each RS may differ across backup and restore, enabling flexibility in restore topology. However, our solution does not allow change in the number of RS across backup and restore, and we plan to consider efficient ways to accomplish this as part of future work. For each shard, we clone the primary's snapshot (taken during PP phase) and map it to as many replicas of the restore cluster. Thus, all nodes in the target cluster start with complete copy of data and do not need to perform an intra-replica-set sync. We reconfigure the restored nodes to reflect their new node IDs, ports and replica set membership, since the cloned snapshot will carry details of the original cluster, using MongoDB APIs.

Limitation: Our solution allows MongoDB recovery to a *fixed-point* in time, based on when snapshot was taken. It does not allow *any* point-in-time restore. We plan to augment the above solution by streaming and storing MongoDB's operation log (Oplog) efficiently, such that it allows replaying operations up to a desired timestamp.

4. Preliminary Evaluation

4.1 Cassandra

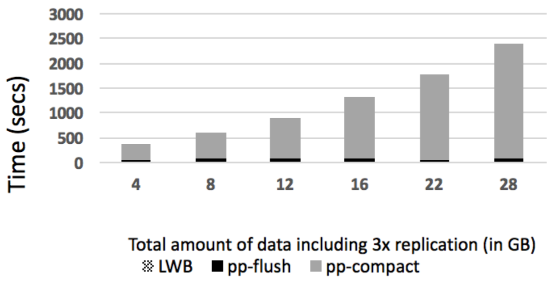

Figure 2: Performance of BARNS backup for Cassandra

We evaluate the total time it takes to backup and restore a Cassandra cluster using BARNS. The production Cassandra cluster consists of 4 nodes, each running Cassandra 4.0, with its SSTables and commit logs on 4 different iSCSI LUNs on shared storage. The post process node is an independent VM with 2 CPUs and 8GB RAM. We ingest data using YCSB [8] into the four node Cassandra cluster and perform backup at different intervals – as the data set increases by 4GB. Figure 2, shows the performance of the different phases of backup with the increase in the data set size. Irrespective of the data set size, LWB and pp-flush phase take constant amount of time, 10-20 secs and 40 secs, respectively. This is because in LWB, we take light weight snapshots of all the data LUNs, which takes constant time. Since the size of the commit log cannot grow beyond a point, pp-flush also takes constant time. The time taken to perform compaction (pp-compact), increases with the amount of data. We observe an increase of around 70% when going from 4GB to 8GB, because 8GB dataset cannot completely reside in memory. Later, we observe a rise in pp-compact time by ~35-40% for every 4GB increase.

The recovery time remains close to **60-80 secs**, irrespective of the data set or cluster size. This is because, there is no repair required during restore as we post process all the data. To compare how much time, it takes to perform repair operation (without post process), we fire *nodetool repair* command on two nodes simultaneously, each containing around 7GB data set. It took ~**456 secs** to complete the repair operation. Thus, we see the benefit of performing post-process during backup.

4.2 MongoDB

We evaluated LWB, PP and restore phases for MongoDB 3.2.7 cluster with 9 nodes consisting of 2 RS with 3 replicas each and a 3-node configuration server RS, with each node hosted over iSCSI LUNs. We observed that LWB takes around 10 seconds, PP takes around 2.5 minutes per RS while restore times are around 2.5 minutes for the entire cluster. These observations are independent of cluster size and dataset size. This is because, MongoDB's master-slave architecture provides us an easy candidate (primary node) for taking space efficient and consistent backup. Even in absence of primary, new leader election only takes only a few seconds. Due to this, we observe negligible increase in recovery time, even if we skip the PP phase. PP though, achieves storage efficient backups, even under error-scenarios and provides opportunity to detect faults in LWB backup.

5. Conclusion and Future Work

In this paper, we present solutions to perform backup and recovery of Cassandra and MongoDB when hosted on shared storage. We leverage database features and data distribution logic to take space efficient, topology oblivious, and cluster consistent backup, to achieve repair-free restore. While our recovery times are constant, we need to improve post processing times for Cassandra. We would also like to extend our work to other masterless and master-slave databases. We plan to explore solutions for multi-site deployment of these databases and integrations with cloud storage.

References

[1] Nadkarni A., Polyglot Persistence: Insights on NoSQL Adoption and the Resulting Impact on Infrastructure. IDC. 2016 Feb.

[2] Russell D., Rhame R., Thomas M., Predicts 2017: Business Continuity Management and IT Service Continuity Management. Gartner. 2016 Nov.

[3] MongoDB Partners with Pure Storage, https://www.mongodb.com/lp/partners/pure-storage

[4] MongoDB Partners with SolidFire, https://www.solidfire.com/press-releases/solidfire-announces-partnership-with-mongodb

[5] Alapati, S., Expert Oracle9i database administration. Apress, 2008.

[6] Chodorow K. MongoDB: the definitive guide. " O'Reilly Media, Inc." 2013 May.

[7] Carvalho N, Kim H, Lu M, Sarkar P, Shekhar R, Thakur T, Zhou P, Arpaci-Dusseau RH, Datos IO. Finding consistency in an inconsistent world: towards deep semantic understanding of scale-out distributed databases. HotStorage, 2016 Jun. USENIX Association.

[8] Cooper BF, Silberstein A, Tam E, Ramakrishnan R, Sears R. Benchmarking cloud serving systems with YCSB. InProceedings of the 1st ACM symposium on Cloud computing 2010 Jun 10 (pp. 143-154). ACM.

[9] Unionfs-fuse man page, http://manpages.ubuntu.com/manpages/precise/man8/unionfs-fuse.8.html

[10] Lakshman A, Malik P. Cassandra: a decentralized structured storage system. ACM SIGOPS Operating Systems Review. 2010 Apr 14;44(2):35-40.

[11] Cassandra Compaction. https://docs.datastax.com/en/cassandra/2.1/cassandra/tools/toolsCompact.html

[12] Mishra V. Cassandra: Administration and Monitoring. Beginning Apache Cassandra Development 2014 (pp. 171-189). Apress.

[13] Zawodny JD, Balling DJ. High Performance MySQL: Optimization, Backups, Replication, Load Balancing & More. " O'Reilly Media, Inc."; 2004 Apr 8.

[14] Preston C. Backup & recovery: inexpensive backup solutions for open systems. " O'Reilly Media, Inc."; 2007.

[15] NetApp Consistency Group, https://library.netapp.com/ecmdocs/ECMP12404965/html/GUID-AA34DCF7-6827-4ACC-AA5E-63B1FEA8EFCE.html

[16] Cassandra Consistency: https://docs.datastax.com/en/cassandra/3.0/cassandra/dml/dmlAboutDataConsistency.html

[17] Mongorestore, https://docs.mongodb.com/manual/reference/program/mongorestore

[18] Datos IO Recover X, White Paper, June 2016

[19] An Inside Look into the Talena Architecture, White Paper, 2016

[20] MongoDB Ops Manager Manual, Release 2.0, MongoDB, Inc., Mar 07, 2017

[21] MongoDB ransomware attacks and lessons learned, http://www.computerworld.com/article/3157766/linux/mongodb-ransomware-attacks-and-lessons-learned.html

[22] Karger D, Lehman E, Leighton T, Panigrahy R, Levine M, Lewin D. Consistent hashing and random trees: Distributed caching protocols for relieving hot spots on the World Wide Web. InProceedings of the twenty-ninth annual ACM symposium on Theory of computing 1997 May 4 (pp. 654-663). ACM.

[23] Mongodump, https://docs.mongodb.com/manual/reference/program/mongodump/

[24] Cassandra Restore, https://docs.datastax.com/en/cassandra/2.1/cassandra/operations/ops_backup_snapshot_restore_t.html

[25] MongoDB Cluster Balancer, https://docs.mongodb.com/manual/tutorial/manage-sharded-cluster-balancer/#disable-balancing-during-backups

[26] Ganesan, Aishwarya, et al. "Redundancy Does Not Imply Fault Tolerance: Analysis of Distributed Storage Reactions to Single Errors and Corruptions."