HotROD: Managing Grid Storage with On-Demand Replication

Sriram Rao †1, Benjamin Reed *2, Adam Silberstein #3

† *Microsoft Corp, USA*
1 sriramra@microsoft.com

* *Osmeta Inc, USA*
2 br33d@yahoo.com

# *Trifacta Inc, USA*
3 aesilberstein@yahoo.com

***Abstract*—Enterprises (such as, Yahoo!, LinkedIn, Facebook) operate their own compute/storage infrastructure, which is effectively a "private cloud". The private cloud consists of multiple clusters, each of which is managed independently. With HDFS, whenever data is stored in the cluster, it is replicated within the cluster for availability. Unfortunately, for datasets shared across the enterprise, this leads to the problem of over-replication within the private cloud. An analysis of Yahoo!'s HDFS usage suggests that the disk space consumed due to replication of shared datasets is substantial (viz., to the tune of PB's of storage). New data sets are typically popular and requested by many processing jobs in (different) clusters. This demand is satisfied by copying the dataset to each of the clusters. As data sets age, however, they get used less and become cold. We then have the opposite problem of having data overreplicated across clusters: each cluster has enough replicas to recover from data loss locally, and the sum total of replicas is high.**

We address both the problems of initially replicating data and cross cluster recovery in a private cloud setting using the same technique: *on-demand replication*, which we refer to as Hot Replication On-Demand (HotROD). By making files visible across HDFS clusters, we let a cluster pull in remote replicas as needed, both for initial replication and later recovery. We implemented HotROD as an extension to a standard HDFS installation.

I. Introduction

Over the past few years, it has become increasingly commonplace across organizations ranging from Fortune-500 companies (such as, Yahoo!, LinkedIn, Facebook) to venture-capital funded startups (such as, Quantcast) to build and operate their own compute and storage infrastructure. A typical mode of operation is to begin by constructing a single cluster and then expanding the operations to include multiple clusters, many of which are within the same datacenter. Effectively, these organization operate a *cloud*-based infrastructure some of which is for their own private use.

Data sharing within an enterprise cloud is fairly common. These companies collect massive amounts of data (viz., on the order of Petabytes) which is usually pre-processed and then shared across various business units within the organization. In what follows, based on our experiences from the setting at Yahoo![1], we highlight the issues related to managing shared datasets in a private cloud setting.

Yahoo! has vast amounts of data and a large number of processing jobs, both production and experimental, that continuously churn through this data. All of this data is spread across tens of large clusters, comprising thousands of servers. This scale necessitates we use software such as Hadoop [6], the open source implementation of Map/Reduce [2], and Pig [5] to manage the processing and storage resources of our servers and data.

Hadoop's distributed file system (HDFS) [9] achieves effective utilization of processing and network resources by moving processing to data. Data replication in HDFS provides both failure tolerance and multiple locations for the scheduling of data processing.

For a single Hadoop cluster, Hadoop's basic policies work well. We, however, have multiple clusters across multiple data centers which can be effectively viewed as a private cloud. Multiple clusters bring rise to two specific problems. First, common datasets must be replicated across all of the clusters so that jobs using the data can process it locally. Second, datasets eventually grow cold, and we end up with more replicas across all clusters than are needed for recovery.

Replicating data across clusters can be time consuming and error prone. Tools such as **distcp** run Hadoop jobs to copy data between clusters, but even when using parallelism such data movement is inherently limited by the bandwidth between clusters. Operators must monitor the replication process to make sure it completes successfully. Because of the inherent time it takes to move data across network connections, failures will happen and the transfers must be restarted.

As time passes, we face the opposite problem: overreplication of data. The data becomes old and fewer jobs use it (we refer to this as *cold data*), and we no longer benefit from having multiple replicas of the data. In some cases the data is used so infrequently that we prefer to let the rare job pull the data from a remote cluster.

The issue of overreplication is clearly manifest when we look at the problem of the wasted storage capacity to store more copies of cold data than are needed for recovery. We currently have 13 petabytes of unique, unreplicated data across

[1] Work done at Yahoo! Research

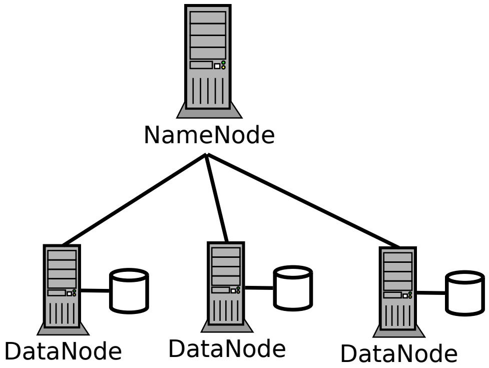

Fig. 1. The basic architecture of HDFS.

our clusters. For recoverability data is usually replicated 2 or 3 times, and so 13PB should expand to 26-39PB. Instead we use 49.7PB of storage. Dropping to 3x replication globally would save us over 13PB of storage. This savings increases as the number of HDFS clusters increase.

In this paper we focus on the problem of replica management for the case of a private cloud. We address both the problems of cross-cluster data replication and overreplication with one simple mechanism: on-demand replication. We employ a *cross-cluster database* that tracks shared datasets and their replication. We implement on-demand replication in HDFS by creating a storage node (ProxyDataNode) that proxies remote data as if were stored locally.

With this approach we achieve almost instant accessibility of cross-cluster data and allow the replication of cold shared data to go to zero on individual clusters.

II. Background

This work is done in the context of HDFS. This distributed file system uses a classic master/worker based architecture, shown in Figure 1, in which a single master server, called the NameNode, manages the file system meta-data, while the workers, the DataNodes, manage the file system data.

DataNodes manage chunks of data called blocks, usually on the order of 128M in size. Blocks are identified using a 64-bit numbers, referred to as BlockIds. DataNodes report to the NameNode which blocks they have.

The NameNode manages both the file system namespace and block replication. The NameNode has an in-memory database that maps filenames to a list of BlockIds. It also manages an in-memory database that maps BlockIds to the DataNodes that store those blocks. If the NameNode detects that a given block is underreplicated, it tells a DataNode that still has a copy of that block to send a copy to another DataNode that does not.

The most common way to access data in HDFS is using the Hadoop Map/Reduce engine. Map/Reduce jobs get the location of the blocks of the files they are processing. They then divide up the processing tasks on block boundaries and try to schedule each task to run on a machines holding the task's block.

At Yahoo! we have multiple HDFS clusters. This is partly for geographic limitations and partly to avoid NameNode scalability limitations, which prevent a cluster from growing past 4000 nodes. Each of our clusters is effectively *owned* by a different business unit and cluster resources are proportionately shared across users/projects within that unit. Finally, each cluster is run completely independently.

There are many common datasets that are copied onto each HDFS cluster, generally under the */data* subdirectory. After the data is copied into one HDFS cluster the **distcp** tool is used to copy data between clusters. This tool runs a Map/Reduce job that splits up file between a set of Map only processes that copy over their assigned portion of a file.

III. Design Requirements

As outlined in Section I, the objective of our work is to enable efficient sharing of common datasets across HDFS clusters that comprise an enterprise's (private) cloud. The design assumptions for a solution are as follows:

- Shared datasets are immutable.
- Shared datasets have controlled creation/deletion policies.
- From a deployment perspective, the software solution should not require any modification to the HDFS NameNode code.

Hadoop in general has immutable datasets. The early releases of HDFS did not have the ability to append to a file. The shared datasets we are targeting with this work are datasets collected from various sources, such as the stream of user events as they visit our sites, which are never changed after they are loaded onto HDFS. Each dataset that is loaded has a timestamp encoded in the name, so a name will always correspond to the same immutable data.

Because these shared datasets are managed by a data group rather than created in an adhoc fashion, the life cycle of the data is different from that of normally generated file. The data is generally loaded onto the clusters and provisioned specifically for sharing. Thus, the data is not randomly created or deleted, but rather replicated across clusters according to a data sharing policy and then removed from all the clusters according to the policy. An example of such a removal policy is legal requirements for retention. For example, data older than 13 months must be deleted [2] to comply with privacy policies.

To get some deployment experience with HotROD, we also want to avoid modifications to the HDFS NameNode. This is a central component of the system that is difficult to upgrade and bugs could take down the cluster. So we restricted our modifications to the HDFS client code that can be distributed with our jobs. We also added new components to the system that transparently fit into existing infrastructure.

In addition to the above requirements, there were two main data management goals we had:

- Near instant access to data;

[2] Laws and policies vary according to location and time, so this is merely an example of a policy, it is not meant to represent Yahoo!'s policy.

- Minimize over-replication of cold data;

As we noted earlier our current method of replicating new shared data to HDFS clusters use **distcp**. This tool has a high latency since a dataset is not available until the copy is completed. Failures during the copy requires manual intervention to restart the copy and additional data movement. Instead we would like the data to be available nearly instantly and actual the data transfer to happen in the backgroud. Any transfer problems should be automatically resolved and restarted from the last sucessful transfer point.

As we noted in Section I we have a large amount of *cold* that is over-replicated according to our replication policy (default of three copies) if we take into account the copies on the different HDFS clusters. We found that the data goes cold rather quickly (viz., typically a week after the data is generated), thus it would be desirable to aggressively reduce the number of copies of a given block to the minimum level required for data availability.

IV. HotROD

HotROD adds a new concept to HDFS called On-demand Replication. We allow file blocks to be "copied" to another cluster by reference rather than value. This allows an HDFS cluster to know about data blocks and know how to retrieve them before actually moving the data across the wire.

The advantage of this approach is that references to blocks in other clusters can be added very quickly leading to extremely low latency file copies. This approach also allows a cluster to have only references to cold blocks that are stored in other cluster and to better use the storage capacity of the cluster.

To make on-demand replication work we need to add additional components to the system. Figure 2 shows how two HDFS clusters are connected by these components. The first component, the meta-data server, maps local BlockIds to cross-cluster references and ensure that we have sufficient copies of blocks across the different replicas to tolerate failures.

We also add a second new system component to HDFS, called the ProxyDataNode, that allows the remote referenced blocks to integrate nicely into the existing data model of HDFS. These ProxyDataNodes do not store data blocks themselves, but instead advertise remote BlockIds that can be retrieved from remote clusters to the NameNode.

There are two processes that are used to administer HotROD. The first, called the *populator*, puts files from an HDFS cluster into a shared pool. The second, called *spoofer*, creates a reference to files from a shared pool in an HDFS cluster.

A. ProxyDataNode

We add a new component to HDFS called a ProxyDataNode which acts like a normal DataNode except that it does not have any data instead it reports to the NameNode that it has blocks that it knows how to get from other clusters. To both the NameNode and the HDFS client the ProxyDataNode looks like a normal DataNode. When the client requests a block for the ProxyDataNode, the ProxyDataNode will retrieve the block from the remote cluster to serve the client request.

TABLE I
THE SCHEMA FOR THE BLOCKS2PATHS TABLE.

Field	Description
BlockId	The proxied BlockId.
Path	The file that the block is part of.
Offset	The offset of the block in the file.
Size	The size of the block.

TABLE II
THE SCHEMA FOR THE PATH2LOCATION TABLE.

Field	Description
Path	The name of the shared file.
Locations	The clusters that have a copy of the file with the number of replicas in the cluster.
Size	The size of the file.

The ProxyDataNode is driven by a meta-data table that maps blocks to paths, called the Blocks2Paths tables. Table I shows the schema for the Blocks2Paths table. When *spoofer* runs, it creates files in the NameNode just as if it were creating files normally: it allocates a name in the namespace and starts requesting BlockIds to add data to the new file; however, unlike a normal file creation, instead of creating the blocks on a DataNode and filling them with data, we add the BlockIds to a table that maps the BlockId to the path in the shared pool with the offset into the file and have the ProxyDataNode report to the NameNode the receipt of the BlockId.

The entire operation involves no movement, so the copy is almost instantaneous. Of course, to process the data it does need to be transferred from the remote location. If the data is cold, this may be acceptable since there is a chance that the data will never be processed again, but if we expect the data to be used, we may want to initiate a background copy to avoid any additional latency when a processing job is run. This is generally the case when we are copying new data onto a cluster. Fortunately, HDFS provides a natural way to do this background copy. When *spoofer* initially creates the files in the NameNode, it uses a replication factor of one, which means that the NameNode will consider the ProxyDataNodes as storing the single replica. If we simply boost the replication factor to something higher than one, the NameNode will instruct other DataNodes to create a replica of the blocks from the ProxyDataNode. This replication will happen automatically and in the background after the *spoofer* has already completed. The spoofed files will be continuously available as the background copy happens, so any clients that try to access the data before the additional copies happen can still access the data through the proxies.

B. Cross-cluster Meta-data

HotROD also has cross-cluster meta-data to track with respect to which clusters have actual replicas of shared data. When *populater* adds content to the shared data pool it also starts tracking the location of that content. The Path2Location table, whose schema is shown in Table II maps a given path

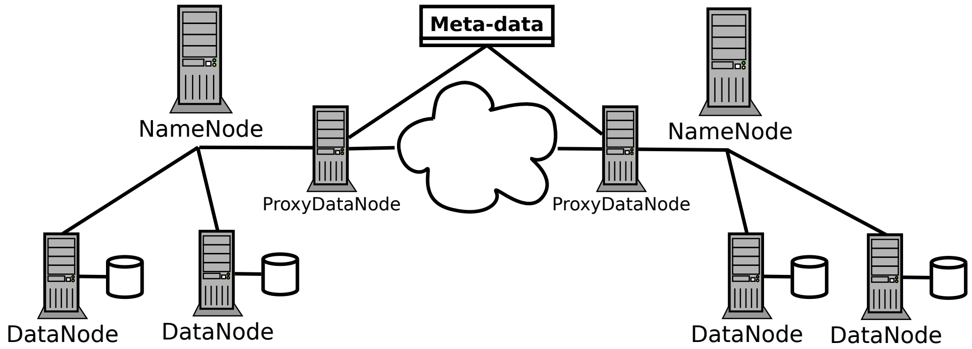

Fig. 2. Two HDFS clusters that share data through ProxyDataNodes.

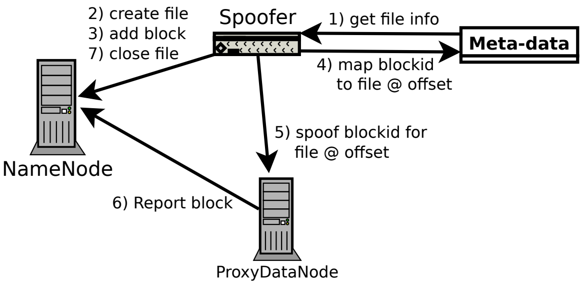

Fig. 3. The steps involved in spoofing a file.

to the clusters that have its data. The ProxyDataNode uses this mapping to figure out which HDFS cluster to request the data from for a given block.

The Path2Location table also helps track the number of replicas in each cluster of a given path. If that data goes cold then the number of replicas in that data can be decreased at each HDFS cluster, sometimes to zero, as long as there are enough replicas across all clusters to satisfy the fault tolerance policy. Our default fault tolerance policy is to have at least three replicas of a given piece of data.

C. Replicating files

The first step of replicating files is to populate the Path2Location table using *populator*. This is a relatively simple step that adds entries to the Path2Location table and thus makes files available for sharing.

The process to spoof a file, using *spoofer*, is more complicated. Figure IV-C shows the process. This process is run on the target HDFS cluster. In steps 1 and 2 *spoofer* gets the name and size of the file and asks the NameNode to create the file.

Steps 3-6 are used to add blocks to the file. *spoofer* asks the NameNode to add blocks to the newly created file and tracks the mapping of the BlockIds to the offsets of the file in the Blocks2Paths table. So these steps are run once for each block of the file. After all the blocks have been added the file is closed. Even though there are multiple steps to the process, each step is light weight and does not move any actual data.

These steps correspond closely to normal HDFS file creation. The main difference is that steps 4 and 5 of a normal HDFS file creation involve transferring data to the DataNode.

D. Handling ProxyDataNode failures

Whenever a ProxyDataNode fails, any of the blocks *spoofed* by that proxy may become temporarily inaccessible. To make the spoofed blocks re-accessible, a new ProxyDataNode is started which then spoofs those blocks. The new ProxyDataNode first uses the Path2Locations table to determine the set of files it should spoof; it then uses the Blocks2Paths table to determine the block id's of the spoofed blocks. Finally, it notifies the HDFS NameNode of the spoofed blocks. The data is now accessible to end-users.

E. Implementation Details

For our prototype we want to use stock HDFS clients and NameNode. We were able to accomplish our goal using the design outlined above.

Since Map/Reduce tasks are not perfectly block aligned, a given block is almost always referenced twice by a Map/Reduce job. It also simplifies our implementation to pull down full blocks and then serve them from a local store. Thus our ProxyDataNodes do use local disk storage for this caching of blocks that are served.

Our ProxyDataNodes report to the NameNode that they are full. This will prevent the NameNode from trying to move blocks to ProxyDataNodes during rebalancing or rereplication of a block.

Our meta-data management is done using HBase [7].

V. Evaluation

We implemented HotROD by modifying the Hadoop 0.20.2 distribution. This involved implementing (1) the ProxyDataNode which satisfies a request for local block by retrieving the appropriate content from the shared file, (2) the populator tool which allows files to be shared, and (3) the spoofer tool which allows shared files to be spoofed by the ProxyDataNode.

To evaluate our implementation of HotROD we emulate a private cloud using two HDFS clusters comprised of 5 nodes each: Cluster A contains shared files; Cluster B runs HDFS datanodes as well as a configurable number of ProxyDataNodes; MapReduce jobs run on Cluster B to access the shared data which then gets downloaded by a ProxyDataNode on-demand. In either cluster, each server has one Xeon dual-core

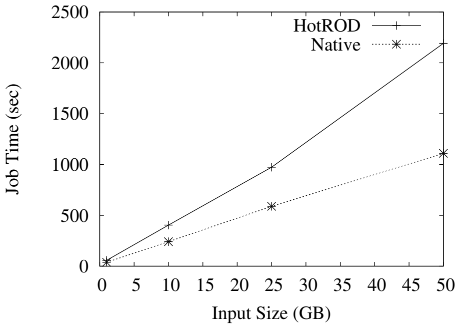

Fig. 4. Change in job execution time as the job's input size changes

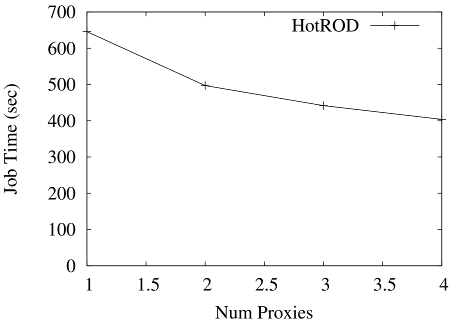

Fig. 5. Varying the number of ProxyDataNodes for a given input size of 10GB.

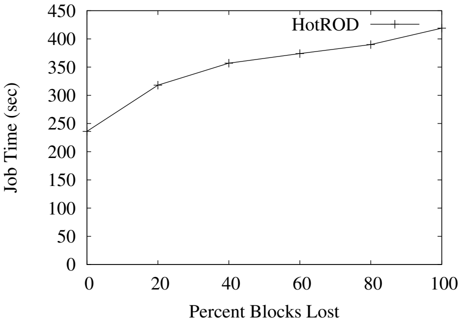

Fig. 6. With 4 ProxyDataNodes and a job input of 10GB, variation in job execution time as the percentage of blocks that must be recovered remotely changes.

2.1GHz processor, 4GB of RAM, gigabit ethernet, and two 1TB disks. The two clusters are within the same datacenter.

For MapReduce jobs, HotROD *only* affects the run-time of map tasks whenever their input has to be downloaded from a remote cluster. Therefore, to study the performance of HotROD for the map-phase of a MapReduce computation, we implemented `MBench`, which is a *map-only* job: Each map task reads its input and writes out the output to local HDFS. Using `MBench`, we perform 3 sets of experiments to measure: (1) with a single ProxyDataNode, change in job running time as the amount of data that has be read from remote cluster varies, (2) change in job running time as we vary the number of ProxyDataNodes, and (3) with multiple ProxyDataNodes, change in job running time as the percentage of blocks read from remote cluster varies.

Varying amount of remote data In this experiment, there are four ProxyDataNodes in Cluster B. As noted earlier, the proxy advertises to the HDFS namenode that the proxy has a copy of all the blocks of all the shared files. Figure 4 shows the results of the experiment. In the graph, we show two sets of results: (1) `Native` when all data for a MapReduce is available on the local HDFS, and (2) `HotROD` when every block of a shared file has to be retrieved from the remote HDFS by the ProxyDataNode. As the job input size increases from 1GB to 50GB, the job run-time with both `Native` and `HotROD` increases linearly as expected. The graph also shows that the rate of increase in job execution time with `HotROD` is much higher than `Native`. This is because of the limitations of a single ProxyDataNode—the rate at which data can be downloaded by the ProxyDataNode is limited by the machine's NIC. Despite this limitation, the `MBench` runs to completion with `HotROD` and is within 1.5 to 2x the run-time of `Native`.

Varying number of ProxyDataNodes In this experiment, we vary the number of ProxyDataNodes from 1 to 4; we fix the input size for `MBench` to 10GB. When multiple proxy datanodes are employed, to avoid the same block from being downloaded multiple times by different proxies, the blocks of a shared file are partitioned amongst the proxy datanodes. The proxy datanode responsible for a block advertises that block to the HDFS namenode. Figure 5 shows the results of our experiments. As expected, having additional proxy datanodes improves job running time since multiple proxy datanodes can concurrently download missing blocks. In our test cluster, as the number of proxy datanodes is increased, the running time of `MBench` decreases from 650 seconds to 400 seconds.

Varying percentage of remote data In this experiment, we fix the number of ProxyDataNodes to 4; we also fix the input size for `MBench` to 10GB. We vary the percentage of blocks that have be read from the remote cluster (from 0% to 100%). Note that, when the percentage of blocks to be read from the remote cluster 0%, it corresponds to the case where all data is available locally; analogously, when percentage of blocks to be read from the remote cluster 100%, it corresponds to the case where all data has to be read from the remote cluster. Figure 6 shows the results of the experiments. For a given input size, as expected, the job run-time increases with the increase in percentage of blocks that have be read from the remote cluster. However the rate of increase is not linear. This is because we have multiple proxy datanodes that can concurrently download the missing data.

VI. Limitations

By ruling out making changes to the HDFS NameNode we found that there were some limitation to what we can do. While the limitations do not prevent us from addressing our use cases, in the future there are changes to the NameNode that we can make to improve performance and functionality.

The approach we use for HotROD really leverages the assumption that the shared data will have the same name across all clusters. For our target scenarios this is a valid assumption and works in practice; however, it would be nice accomodate different names for files as well as dealing with replication at the block level rather than the file level. This would allow us to lower replication of common blocks of different files, and allow us to do block-level deduplication. The problem is that the block garbage collection is based on files that reference a block. To implement this solution well, we would need to indicate to the NameNode that some blocks will be garbage collected and managed by our global metadata manager.

Restricting ourselves to immutable files also limits the applicability of our solution. If we had the block based replication management mentioned above, we observed that we could use copy-on-write [8] to allow shared files to be changed. To implement copy-on-write we would have to involve the NameNode. Making this change was hard to motivate since we did not have production requirements for this feature and would require rather invasive changes to the NameNode.

Another limitation is that the life cycle of the datasets has to be managed in a manner different from non-shared datasets. If a user deletes a shared file from a cluster, it affects the dataset replication of all the clusters. For example, if we have a cold file that is replicated across 7 clusters, there will be three clusters that will actually contain the data. If the files are deleted from those clusters at the same time, the data will be lost. This situation could be avoided if we could indicate to NameNode that those files or blocks are managed by an outside process.

Finally, the NameNode could make better data access decisions if it know that a DataNode was actually a ProxyDataNode. Since the NameNode cannot distinguish a DataNode from a ProxyDataNode we had to lie about space usage to make sure that the NameNode does not try to manage the blocks on the ProxyDataNode, such as move a block from a DataNode to a ProxyDataNode. We also do note want the NameNode to tell the client to access data from a ProxyDataNode if the data is available at a normal DataNode. Adding information about ProxyDataNodes and policies with respect to ProxyDataNodes would be a minor change to the NameNode.

VII. Related work

As we mentioned earlier, the current way to replication files across the grid is using *distcp*. This tool finishes only after all the data is moved, so it has a high latency and will fail if there is a problem with the data transfer. HotROD only sets up the references, so it has very low latency, and the data transfers happen in the background. While data transfer is happening the blocks are still accessible on demand. Any transient transfer problems will eventually get resolved in the background without operator involvement.

DiskReduce [4], used erasure codes with cold data to reduce the space needed to be able to recover data in case of a replica loss. Facebook [10] uses this work in production. This work was done in the context of a single HDFS cluster. It does not take advantage of replicas in other clusters, so it doesn't allow the number of replicas in a file to go to zero. It could be used together with HotROD to reduce even the number of cross-data center replicas needed. For example, one data center could use erasure codes to be able to recover from two failures using a data overhead of less than 100% and the others could have zero replicas.

On-demand migration for databases in cloud settings has been recently studied [1], [3]. Similar in spirit to HotROD, these systems focus on "pull-based" approaches, where a reference to the data being migrated is first created on a new machine; data serving on the new machine can be started before the entire database is migrated and data pages are fetched on-demand. Furthermore, these systems also have to address mutable datasets—the data being migrated is still mutable on the old machine. However, HotROD addresses a simpler problem in that the shared data is immutable and hence, leads to a simpler implementation which is also readily deployable.

VIII. Conclusions

We have found that the concept of On-Demand replication fits nicely into the design of HDFS. We were able to add it with few modifications to the core HDFS code. The resulting system simplifies data sharing and replica management in an enterprise's private cloud. In particular, HotROD allows for instant access to data across clusters and it allows for proper replication of cold data across clusters. HotROD accomplishes both of these tasks using the same simple mechanism of proxying remote references using the ProxyDataNode. It is especially interesting that the same mechanism is used to accomplish both tasks because in some sense the tasks are exact opposites of each other: the first task replicates new (and thus hot data) to all the clusters, and the second tasks removes replicas of cold data from clusters as long as a minimum replication factor is maintain across the enterprise's cloud.

Our work was motivated by the use case from Yahoo!. Any organization with a private cloud comprising of multiple HDFS clusters with shared data can benefit from HotROD. By carefully managing replicated datasets in an automated manner, HotROD enables enterprises to reduce expenses (both, capital expenditure and operating expenditure) for running their private cloud infrastructure.

References

[1] S. Das, S. Nishimura, D. Agrawal, and A. E. Abbadi. Albatross: Lightweight elasticity in shared storage databases for the cloud using live data migration. *Proc. of the VLDB Endowment*, 4(8), 2011.

[2] J. Dean and S. Ghemawat. MapReduce: Simplified Data Processing on Large Clusters. In *Proceedings of Operating Systems Design and Implementation (OSDI)*, 2004.

[3] A. Elmore, S. Das, D. Agrawal, and A. E. Abbadi. Zephyr: Live migration in shared nothing databases for elastic cloud platforms. In *Proceedings of the 2011 International Conference on Management of Data*, SIGMOD. ACM, 2011.

[4] B. Fan, W. Tantisiriroj, L. Xiao, and G. Gibson. Diskreduce: Raid for data-intensive scalable computing. In *Proceedings of the 4th Annual Workshop on Petascale Data Storage*, PDSW '09, pages 6–10, New York, NY, USA, 2009. ACM.

[5] A. F. Gates, O. Natkovich, S. Chopra, P. Kamath, S. M. Narayanamurthy, C. Olston, B. Reed, S. Srinivasan, and U. Srivastava. Building a High-Level Dataflow System on top of Map-Reduce: The Pig Experience. *Proc. of the VLDB Endowment*, 2(2):1414–1425, 2009.

[6] Apache Hadoop. http://hadoop.apache.org/.

[7] Apache Hbase. http://hbase.apache.org/.

[8] R. F. Rashid and G. G. Robertson. Accent: A communication oriented network operating system kernel. *SIGOPS Oper. Syst. Rev.*, 15(5):64–75, Dec. 1981.

[9] K. Shvachko, H. Kuang, S. Radia, and R. Chansler. The hadoop distributed file system. In *Proceedings of the 2010 IEEE 26th Symposium on Mass Storage Systems and Technologies (MSST)*, MSST '10, pages 1–10, Washington, DC, USA, 2010. IEEE Computer Society.

[10] A. Thusoo, Z. Shao, S. Anthony, D. Borthakur, N. Jain, J. Sen Sarma, R. Murthy, and H. Liu. Data warehousing and analytics infrastructure at facebook. In *Proceedings of the 2010 International Conference on Management of data*, SIGMOD '10, pages 1013–1020, New York, NY, USA, 2010. ACM.