KVFS: An HDFS Library over NoSQL Databases

Emmanouil Pavlidakis[1], Stelios Mavridis[2], Giorgos Saloustros and Angelos Bilas[2]
Foundation for Research and Technology – Hellas (FORTH), Institute of Computer Science (ICS), 100 N. Plastira Av., Vassilika Vouton, Heraklion, GR-70013, Greece

Keywords: Distributed File Systems, NoSQL Data Stores, Key-value Stores, HBase, HDFS.

Abstract: Recently, NoSQL stores, such as HBase, have gained acceptance and popularity due to their ability to scale-out and perform queries over large amounts of data. NoSQL stores typically arrange data in tables of (key,value) pairs and support few simple operations: get, insert, delete, and scan. Despite its simplicity, this API has proven to be extremely powerful. Nowadays most data analytics frameworks utilize distributed file systems (DFS) for storing and accessing data. HDFS has emerged as the most popular choice due to its scalability. In this paper we explore how popular NoSQL stores, such as HBase, can provide an HDFS scale-out file system abstraction. We show how we can design an HDFS compliant filesystem on top a key-value store. We implement our design as a user-space library (*KVFS*) providing an HDFS filesystem over an HBase key-value store. *KVFS* is designed to run Hadoop style analytics such as MapReduce, Hive, Pig and Mahout over NoSQL stores without the use of HDFS. We perform a preliminary evaluation of *KVFS* against a native HDFS setup using DFSIO with varying number of threads. Our results show that the approach of providing a filesystem API over a key-value store is a promising direction: Read and write throughput of *KVFS* and HDFS, for big and small datasets, is identical. Both HDFS and *KVFS* throughput is limited by the network for small datasets and from the device I/O for bigger datasets.

1 INTRODUCTION

Over the last few years data has been growing at an unprecedented pace. The need to process this data has led to new types of data processing frameworks, such as the Hadoop (White, 2012) and Spark (Zaharia et al., 2010). In these data processing frameworks data storage and access plays a central role. To access data, these systems employ a distributed filesystem, usually HDFS, to allow scaling out to large numbers of data nodes and storage devices while supporting a shared name space.

Although HDFS stores and serves data, it is designed to serve read-mostly workloads that issue large I/O requests. Therefore, applications that require data lookups, use a NoSQL store over HDFS, to sort and access data items. Figure 1 shows the typical layering of a NoSQL store over HDFS in analytics stacks.

With recent advances in analytic frameworks, there has been a lot of effort in designing efficient key-value and NoSQL stores. NoSQL stores typically offer a table-based abstraction over data and support a few simple operations: get, put, scan and delete over a namespace of keys used to index data. Key-value stores, offer similar operations, without however, the table-based abstraction.

Our observation is that the get/put API offered by NoSQL and key-value stores can be used to offer general access to storage. In particular, with the emergence of efficient key-value stores, there is potential to use the key-value store as the lowest layer in the data access path and to provide file-system abstraction over the key-value store.

There are two advantages to such an approach. First, for cases where a key-value store runs directly on top of the storage devices (Kinetic, 2016) without the need for a local or distributed filesystem, there is opportunity to reduce overheads during analytics processing. With increasing data volumes, processing overheads are an important concern for modern infrastructures. Second, there is potential to offer different storage abstractions such as file, object-based, or record-based storage over the same pool of storage. The key-value store can function essentially as a general purpose mechanism for metadata manage-

[1]Also with Department of Computer Science, VU University Amsterdam, Netherlands.
[2]Also with the Department of Computer Science, University of Crete, Greece.

Pavlidakis, E., Mavridis, S., Saloustros, G. and Bilas, A.
KVFS: An HDFS Library over NoSQL Databases.
In *Proceedings of the 6th International Conference on Cloud Computing and Services Science (CLOSER 2016) - Volume 1*, pages 360-367
ISBN: 978-989-758-182-3

ment for all of these abstractions and layers.

In this paper we explore this direction by designing an HDFS abstraction, *KVFS*, over a NoSQL store. We implement *KVFS* as a client-side, user-space library that fully replaces the HDFS client and run unmodified programs that require HDFS, such as Map-Reduce. Although *KVFS* is able to run over any NoSQL store, in our work and initial experiments, we choose HBase because it is widely deployed.

We first discuss the design and implementation of the operations supported by *KVFS* and how these are mapped to the underlying NoSQL abstraction. We then perform some preliminary experiments with DFSIO in a small setup to examine the basic overheads of our approach and to contrast *KVFS* with a native HDFS deployment.

Both *KVFS* and HDFS achieve almost the same read and write throughput, about 900 MBytes/s, for small files. For large datasets *KVFS* has about 20% lower throughputs for reads and about 11% higher throughput for writes, compared to HDFS. Essentially, both systems should be able to achieve the same throughput, given that they operate with large I/O operations.

The rest of this paper is organized as follows. Section 2 presents background for HBase and HDFS. Section 3 discusses our design and implementation of *KVFS*. Section 4 displays our preliminary evaluation. Section 5 presents related work. Finally, Section 6 concludes the paper.

2 BACKGROUND

2.1 HDFS

Hadoop Distributed File System (HDFS) (Borthakur, 2008) is a scale-out distributed filesystem, inspired by Google File System (Ghemawat et al., 2003). HDFS offers simpler semantics than traditional filesystems in two ways: HDFS supports only large blocks, therefore all file metadata fit in memory and it supports only append-to-file operations by a single writer reducing the need for synchronization. In addition, its scale-out properties and robustness makes it a good fit for map-reduce style workloads, offered by frameworks such as Spark and Map-Reduce.

HDFS uses centralized metadata service named ***Namenode***. Namenode is responsible for keeping the catalogue of the filesystem consisting of files and directories. Files are split in fixed-size blocks (typically 64-MBytes). Blocks are replicated, usually between 2-6 replicas per block, and are distributed to ***datanode***. The large block size used by HDFS implies that file metadata fit in memory and that updates to metadata are infrequent. Metadata are synchronously written to persistent storage and are cached in memory. If Namenode crashes a secondary Namenode takes over by reading the persistent metadata.

2.2 HBase

HBase (George, 2015) is a prominent NoSQL store, inspired from Google Big Table (Chang et al., 2008). It offers a CRUD (Create, Read, Update, Delete) API and supports range queries over large amounts of data in a scale-out manner.

HBase supports a lightweight schema for semi-structured data. It consists of a set of associative arrays, named tables. Taqbles comprise of:

- Row: Each table comprises of a set of rows. Each row is identified through a unique row key.
- Column family: The data in a row are grouped by column family. Column families also impact the physical arrangement of data stored in HBase.
- Column qualifier: Data within a column family is addressed via its column qualifier. Column qualifiers are added dynamically in a row and different rows can have different sets of column qualifiers.

The basic quantum of information in HBase is a *cell* addressed by row key, column family and column qualifier. Each cell is associated with a timestamp. In this way HBase is able to keep different versions of HBase cells.

Figure 1 shows the overall architecture of HBase. HBase consists of a master node named HMaster, responsible for keeping the catalogue of the database and also managing Region servers. Region servers are responsible for horizontal partitions of each table called *regions*. Each region contains a row-key range of a table. Clients initially contact HMaster, which directs them to the appropriate Region server(s) currently hosting the targeted region(s). HBase uses the Apache Zookeeper (Hunt et al., 2010) coordination service for maintaining various configuration properties of the system.

HBase uses a log structured storage organization with the LSM-trees (O'Neil et al., 1996) indexing scheme at its core, a good fit for the HDFS append-only file system (Shvachko et al., 2010) it operates on. Records inserted in an LSM-Tree are first pushed into a memory buffer; when that buffer exceeds a certain size, it is sorted and flushed to a disk segment in a log fashion, named *HFile*. Read or range queries examine the memory buffer and then the set of HFiles of the region. For improving indexing performance, the number of HFiles for a region can be reduced

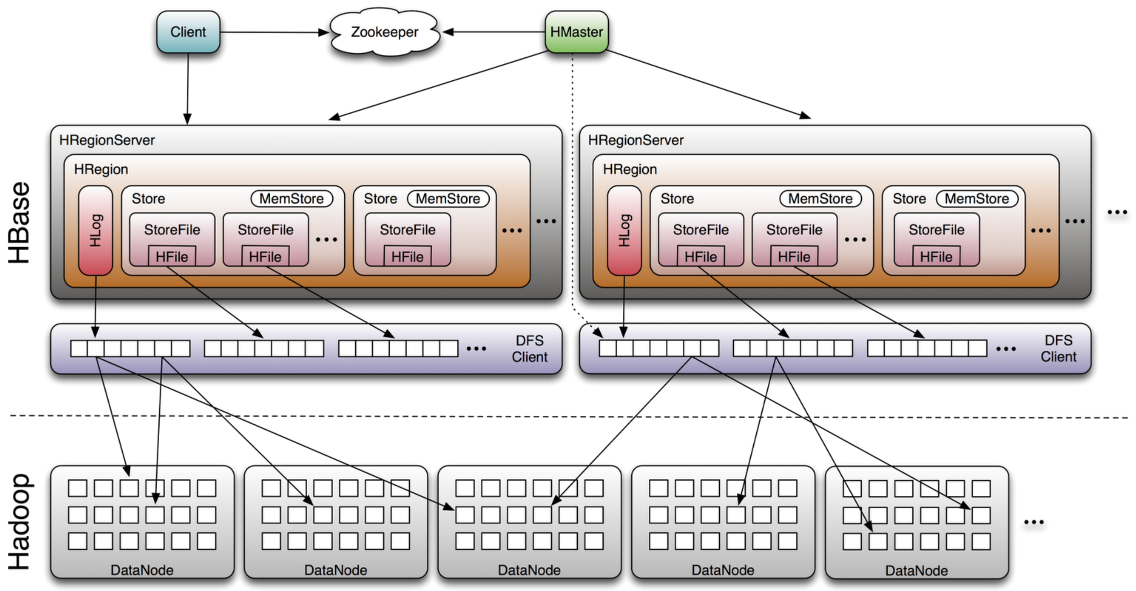


Figure 1: HBase over HDFS architecture (George, 2015).

by periodic merging of HFiles into fewer and larger HFiles through a process called *compaction* (similar to a merge sort). As the data set grows, HBase scales by splitting regions dynamically and eventually moving regions between region servers through move operations. For durability, HBase keeps a write-ahead-log (WAL) in each region server. A WAL is an HDFS file to which all mutations are appended.

3 SYSTEM DESIGN AND IMPLEMENTATION

We modify the existing Hadoop stack as seen in 2(a). We replace HDFS with a shared block device and place *KVFS* above HBase. This results in the stack shown in 2(b). *KVFS* exposes to applications a hierarchical namespace consisting of files and folders which are stored persistently. File operations supported in *KVFS* are the same as in HDFS; append only writes and byte range reads in random offset within a file. In the rest of this section first we explain how the hierarchical namespace of *KVFS* is mapped and stored in HBase. Subsequently we provide information about the basic file operations.

HBase runs on a global file namespace provided by a distributed filesystem, typically HDFS. Essentially, HDFS provides the ability to HBase to access any file from any node, it offers data replication, and space management (via the filesystem allocator and the file abstraction).

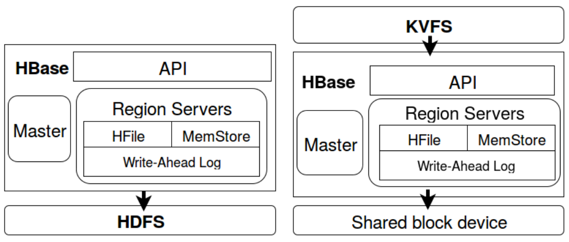
Running HBase on top of a shared block device requires small modifications: Each HTable in HBase is decomposed in regions. Each region can be allocated on a portion of the shared block device, such as the ones provides by a virtual storage area network (vSAN). Similar to HDFS files, such portions of the address space are visible to all nodes for load balancing and fail-over purposes. The information about which region maps to which address portion and which server is responsible for a particular address space is stored as metadata in Apache ZooKeeper, similar to the original HBase.

Additionally, shared block devices can and usually are replicated for reliability purposes. This provides the same level of protection as HDFS, with additional options to use different form of replication or encoding. Finally, shared block devices offered by modern storage systems are typically thin provisioned, which allows them to grow on demand, based on available space, similar to individual files. Therefore, regions that map to device portions can grow independently, without the need to statically reserve space upfront.

KVFS maps files and directories into HBase's row-column schema. We generate row keys for files and directories by hashing the absolute file path (e.g. *hash("/folder/file")*). This eliminates any chance of collision for files with the same names residing in different folders (e.g. *"/test1/file"* and *"/test2/file"*). As shown in Table 1, every HTable row contains all metadata of a single file or directory. Every row contains Column Qualifiers (*CQ*), where every *CQ* contains metadata relevant to a specific file or folder. All rows in an HTable contain *CQ*s that are typical file meta-

Table 1: Mapping of files and directories to HBase in *KVFS*.

Rowkey	**Column Family name**									
	Type	**Permissions**	**Block size**	**Length**	**Timestamp**	**Seed**	**hash ("Dir1")**	**hash ("Dir1/foo2")**	**hash ("Dir1/foo2#0")**	**hash ("Dir1/foo2#1")**
hash(Dir1)	D	Read - Write					NOT Exist	Exist	NOT Exist	NOT Exist
hash(Dir1/foo2)	F	Read - Write					NOT Exist	NOT Exist	Exist	Exist



(a) HBase over HDFS. (b) *KVFS* over HBase

Figure 2: Hadoop architectures.

data such as permissions,file block size, and file size. We also have a *seed* and *timestamp CQ*, both of which are used for our namespace manipulation operations, explained later in this section.

We create a new file/folder by initially calculating the new file row key. Using the key we create a new row and initialize it with the metadata provided by the user and set the *timestamp* with the current time. We also set the *seed CQ* with the result of the hashed concatenation of *(filename,timestamp)*. On successful creation of the new row, we update the parent folder. We add a new empty *CQ* using the hash of the new file as the qualifier key. This bottom-up order of namespace mutations ensures namespace consistency in the presence of failures. Delete operations work in reverse order, first removing the child *CQ* and then removing the child row. For both operations, the worst case scenario is having a 'zombie' row which could be later garbage collected by the system. Finally using *CQ*s for the files metadata ensures good locality.

Locating a file or folder is accomplished by recursively traversing the respective rows of each folder hierarchy (e.g. for "/file/test", "/file" then "/file/test"). Although we could simply access directly the file in question we traverse the path ensuring consistent view of the namespace. Clients send in parallel a window of size N requests which can hit different servers. Continuing, clients traverse the full path in a top-down approach. If a failure is detected we correct this by checking the existence of the row of the next level and updating the parent node.

Another major concern for filesystems besides namespace management is data allocation and placement. In *KVFS*, files are mapped into configurable size blocks, typically of size 64MB. Blocks are represented as rows dedicated to storing file contents able

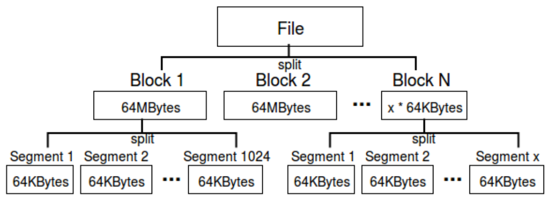


Figure 3: Splitting files to blocks and segments for write operations.

to grow up to block size. During file growth with append operations, client keep track of the file size. Since file data are split into fixed size blocks clients can calculate the next block row key in the following way. The key is calculated by hashing the *seed* value of the files and the block offset (e.g. for the first block of "/file",*hash("/file#0")*). After allocating the block we update the file by updating the file size, ensuring correctness in the presence of a failure. Since block row keys are calculated in a deterministic way we don't need to keep explicitly the blocks into file metadata reducing its size. Seed is used for being able to track file blocks after file rename operations. When a file is renamed, we create a new entry in the namespace discarding the old one. If however we were based solely on file name to produce deterministic the row keys this would impose rename of the block keys. With the use of seed, we create a unique id on file creation which can be used even after rename operations to locate the blocks of the file.

Similar to the dir/file lookup, we could locate a given block directly. In order to ensure consistency with failures, we must first check the file size to ensure the requested block resides inside the file size. We then *get* the requested block from HBase. If we successfully find the requested block although the file size did not 'contain' it then a failure occurred. We recover by adding the missing *CQ* and updating the file size *CQ*.

HDFS supports variable sized reads at random offsets which results in a random read operation at a block. HDFS datanodes, serve this request efficiently, by using the metadata of the local filesystem. In particular, blocks which are mapped in local files by HDFS Datanode use directly the local FS ability to read from a random offset in that file. However, HBase API, where KVFS is built upon does not provide this capability in its API. As we presented above, each block in KVFS maps to row where its value is of size typically 64MB. This is good fit for 64MB reads but imposes significant overhead for small reads. For example, in order for a client to read 64KB from a block, this would impose the region server to send to the client 64MB to pick the corresponding 64KB. To this end we enrich our blocks with CQs metadata named ***segments***, as shown in Figure 3. Each block row consists of a set of segments. Each segment address a configurable size data typically 64KB. With this addition, small reads are served by requesting from the block row only the appropriate segments and then by adding to the request the corresponding segments. The drawback from this approach is that we amplify the metadata but we should optimize this in a later version.

KVFS supports single writer append only writes as original HDFS. For exclusive write access to a file HDFS uses Namenode as the central lock manager. Namenode grants a lease to the client which ensures exclusive write access. Since Namenode is absent in KVFS we redesign the distributed locking mechanism with the use of Zookeeper (Hunt et al., 2010). Zookeeper is a scalable and fault tolerant coordination service which is already part of the HBase stack. It exposes a filesystem like API for implementing a variety of distributed synchronization protocols. It achieve this by using callbacks to clients. In our design, each time a client wants to be granted write access to a file, it tries to create a Zookeeper node (znode) using *hash(filename)* as the znode *id*. If the znode exists, it means that some other client is already writing to the file. If this is the case, client will request for a callback when the znode is deleted. In the opposite case, the client will start appending to the file and at the end it will remove the znode.

MapReduce and other applications tend to produce small write appends (Kambatla and Chen, 2014). For this reason, we use a write buffer to merge multiple small writes into a 64-KByte segment. We batch multiple segments and send them to the region server for network efficiency. In the current version of the prototype, we have tuned the size of the write buffer and the batching factor over the network, thus we are able to overlap computation with network communication.

4 PRELIMINARY EVALUATION

4.1 Experimental Setup

Even though *KVFS* is in an early prototype stage, it is able to run unmodified map-reduce workloads. DFSIO is a read/write benchmark for HDFS. DFSIO measures an HDFS cluster total read and write throughput, simulating IO patterns found in MapReduce workloads. We conduct two experiments, one for reads and one for writes. Both experiments are performed for datasets of 8-GBytes and 96-GBytes respectively with default block size of DFSIO set to 32-MByte. Dataset of 8-GBytes fits in memory whereas the 96-GBytes dataset is IO intensive as it no longer fit in the buffer cache. In the first case we saturate the network and in the second we saturate the device. Moreover the number of mappers, are from 1 till 8 for both datasets. The dataset is split equally across mappers. The number of concurrent tasks is defined for both read and write operations and equals to the number of mappers. The sever uses 4 Intel X25-E SSDs (48-GBytes) in RAID-0 using md Linux driver. Table 2 presents the configuration of the servers we use.

Table 2: Hardware configuration of evaluation setup.

CPU	Xeon(R) CPU E5520
Memory	48-GBytes
NIC	10GBit Myricom NIC
Hard Disk	WD Blue 160-GBytes
OS	CentOS 6.3
HBase	version 0.98
HDFS	version 2.7.1

4.2 Results

Figures 4 and 5, show our results for writes/reads with datasets of 8-GBytes and 96-GBytes respectively. DFSIO reports throughput per mapper. We plot the aggregate throughput of all mappers.

Writes: Write throughput of *KVFS* and HDFS for both 8-GBytes and 96-GBytes datasets is almost equal. Figure 4(a) shows that for the 8-GBytes dataset, *KVFS* achieves 908-MBytes/s and HDFS achieves 938-MBytes/s maximum write throughput. For 96-GBytes dataset, *KVFS* achieves maximum throughput 460-MBytes/s and HDFS 410-MBytes/s.

In more detail, Figures 4(a) and 5(a) show that the write throughput of HDFS and *KVFS* is almost equal, when the number of mappers is 8. For the dataset of 8-GBytes, when the number of mappers ranges from 1 to 4, *KVFS*'s write throughput is 25.4% higher than HDFS's. For the dataset of 96-GBytes, when the number of mappers is 1 *KVFS*'s write throughput is 2.5 times the throughput of HDFS. Whereas, as we increase the number of mappers write throughput of both HDFS and *KVFS* becomes almost identical.

Reads: We note that *KVFS*'s throughput exceeds HDFS's throughput when the number of mappers is small. From Figure 4(b) we observe that when the number of mappers range from 1 to 4, *KVFS*'s read throughput is almost 2 times the throughput of HDFS. This improvement is due to *KVFS*s use of asynchronous I/O using zeromq(Hintjens, 2013) for data requests. As the number of mappers increases, HDFS's and *KVFS*'s throughput become almost equal as the benefits of asynchronous I/O diminish. The maximum throughput achieved by *KVFS*, for the 8-GBytes dataset, is 1170-MBytes/s and for HDFS is 1200-MBytes/s. For the 96-GBytes datasets the maximum throughput for *KVFS* is 390-MBytes/s, whereas for HDFS is 490-MBytes/s with 4 mappers respectively. These difference in performance between are due to the fact that HDFS has a more efficient cache policy. In future versions we are planning to improve the caching policy and we believe that, after tuning, *KVFS* could outperform the native HDFS setup.

Our results for both read and write comply with the throughput of the SSDs for the big dataset (96-GBytes) and with the network speed for the small the dataset (8-GBytes). In both cases we almost achieve the maximum throughput.

5 RELATED WORK

Similar to our approach, the Cassandra File System (CFS) runs on top of Cassandra (Lakshman and Malik, 2010). CFS aims to offer a file abstraction over the Cassandra key-value store as an alternative to HDFS. Cassandra does not require HDFS but runs over a local filesystem. Therefore in a Cassandra installation there is no global file-based abstraction over the data. Our motivation differs, in that we are interested to explore the use of key-value stores as the lowest layer in the storage stack. Although we are currently using HBase, our goal is to eventually replace HBase with a key-value store that runs directly on top of the storage devices, without the use of a local or a distributed filesystem. This motivation is similar to the Kinetic approach (Kinetic, 2016), which however, aims at providing a key-value API at the device level and then build the rest of the storage stack on top of this abstraction.

The last few years there has been a lot of work on both DFSs (Depardon et al., 2013; Thanh et al., 2008) and NoSQL stores (Abramova et al., 2014; Klein et al., 2015) from different originating points of view. Next, we briefly discuss related work in DFSs and NoSQL stores.

DFSs have been an active area over many years due to their importance for scalable data access. Traditionally, DFSs have strived to scale with the number of nodes and storage devices, and to eliminate synchronization, network, and device overheads, without compromising the richness of semantics (ideally offering a POSIX compliant abstraction). Several DFSs are available and in use today, including Lustre, BeeGFS, OrangeFS, GPFS, GlusterFS, and several other commercial and proprietary systems. Providing a scalable file abstraction while maintaining traditional semantics and generality has proven to be a difficult problem. As an alternative, object stores, such as Ceph (Weil et al., 2006), draw a different balance between semantics and scalability.

With recent advances in data processing, the additional interesting and important realization has been that in several domains it suffices to offer simpler APIs and to design systems for restricted operating points. For instance, HDFS uses very large blocks (e.g. 64 MBytes) which simplifies dramatically metadata management. In addition, it does not allow updates, which simplifies synchronization and recovery. Finally, it does not achieve parallelism for a single file, since each large chunk is stored in a single node, simplifying data distribution and recovery. Given these design decisions, HDFS and similar DFSs are efficient for read-mostly, sequential workloads, with large requests.

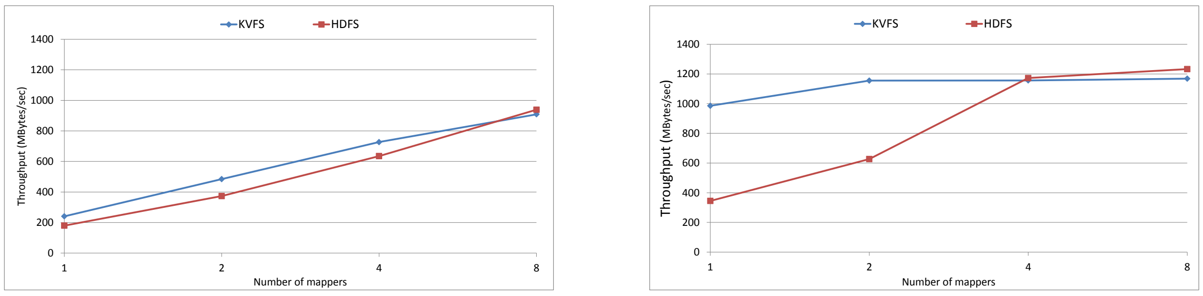
NoSQL stores have been used to fill in the need for fine-grain lookups and the ability to scan data in a sorted manner. NoSQL stores can be categorized in four groups: key-value DBs (Level DB, Rocks DB, Silo), document DBs (Mongo DB), column family stores (HBase, Cassandra), and graph DBs (Neo4j, Sparksee).

Such, data-oriented (rather than device-oriented) approaches to storage and access bare a lot of merit because they draw yet a different balance between semantics and scalability. Until to date these approaches have become popular in data processing frameworks, however, they have little application in more general purpose storage.

We foresee, that as our understanding of key-value stores and their requirements and efficiency improves, they will play an important role in the general-purpose storage stack, beyond data processing frameworks.

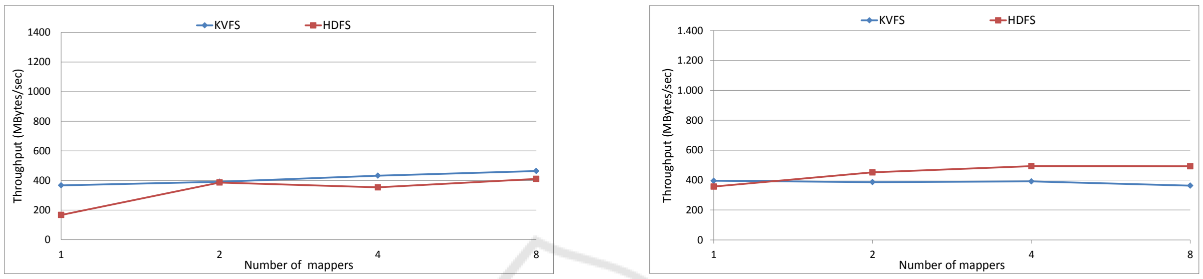
6 CONCLUSIONS

In this paper we explore how to provide an HDFS abstraction over NoSQL stores. We map the file abstraction of HDFS to a table-based schema provided by



(a) Write (b) Read

Figure 4: Write and read throughput for 8-GBytes dataset with increasing number of mappers.



(a) Write (b) Read

Figure 5: Write and read throughput for 96-GBytes dataset with increasing number of mappers.

NoSQL stores. Our approach combines the versatility of the key-value API with the simpler (compared to traditional DFS) semantics of HDFS. We implement a prototype, *KVFS*, which runs on top of HBase, a popular and widely deployed NoSQL store, supports the full HDFS API and is able to run unmodified data analytics applications, such MapReduce tasks. *KVFS* runs as a library that replaces entirely the HDFS client library.

We perform preliminary experiments to contrast the performance of our approach to a native HDFS deployment. We find that *KVFS*'s maximum throughput of reads is 390-MBytes/s and 1100-MBytes/s, for big and small datasets, respectively. These throughputs are slightly smaller than the ones achieved by HDFS. For writes *KVFS*'s maximum throughput is 460-MBytes/s and 910-Mbytes/s, for big and small datasets, respectively. *KVFS*'s write throughput for small datasets is very close to HDFS's throughput, whereas for big datasets it exceeds it.

Overall, our approach shows that the key-value abstraction is a powerful construct and can be used as the lowest layer in the storage stack as a basis for building all other storage abstractions (blocks, files, objects, key-value pairs). We believe that this direction is worth further exploration and that it can lead to more efficient, scale-out data storage and access in modern data centers and infrastructures.

REFERENCES

Abramova, V., Bernardino, J., and Furtado, P. (2014). Experimental evaluation of nosql databases. In *International Journal of Database Management Systems*, pages 01–16.

Borthakur, D. (2008). Hdfs architecture guide. *HADOOP APACHE PROJECT http://hadoop. apache. org/common/docs/current/hdfs design. pdf*.

Chang, F., Dean, J., Ghemawat, S., Hsieh, W. C., Wallach, D. A., Burrows, M., Chandra, T., Fikes, A., and Gruber, R. E. (2008). Bigtable: A distributed storage system for structured data. *ACM Transactions on Computer Systems (TOCS)*, 26(2):4.

Depardon, B., Mahec, G., and Seguin, C. (2013). Analysis of six distributed file systems. In *HAL Technical Report*.

George, L. (2015). *HBase: The Definition Guide*. O'Reilly Media.

Ghemawat, S., Gobioff, H., and Leung, S.-T. (2003). The google file system. *ACM SIGOPS Operating Systems Review - SOSP '03*, 37(5):29–43.

Hintjens, P. (2013). *ZeroMQ: Messaging for Many Applications*. " O'Reilly Media, Inc.".

Hunt, P., Konar, M., Junqueira, F. P., and Reed, B. (2010). Zookeeper: Wait-free coordination for internet-scale systems. In *Proceedings of the 2010 USENIX Conference on USENIX Annual Technical Conference*, USENIXATC'10, pages 11–11, Berkeley, CA, USA. USENIX Association.

Kambatla, K. and Chen, Y. (2014). The truth about mapreduce performance on ssds. In *Proc. USENIX LISA*.

Kinetic (2016). Open storage kinetic project. http://www.openkinetic.org/. Accessed: January 2016.

Klein, J., Gorton, I., Ernst, N., Donohoe, P., Pham, K., and Matser, C. (2015). Performance evaluation of nosql databases: A case study. In *Proceedings of the 1st Workshop on Performance Analysis of Big Data Systems*, PABS '15, pages 5–10, New York, NY, USA. ACM.

Lakshman, A. and Malik, P. (2010). Cassandra: A decentralized structured storage system. *SIGOPS Oper. Syst. Rev.*, 44(2):35–40.

O'Neil, P., Cheng, E., Gawlick, D., and O'Neil, E. (1996). The log-structured merge-tree (lsm-tree). *Acta Inf.*, 33(4):351–385.

Shvachko, K., Kuang, H., Radia, S., and Chansler, R. (2010). The hadoop distributed file system. In *Mass Storage Systems and Technologies (MSST), 2010 IEEE 26th Symposium on*, pages 1–10.

Thanh, T., Mohan, S., Choi, E., Kim, S., and Kim, P. (2008). A taxonomy and survey on distributed file systems. In *Networked Computing and Advanced Information Management, 2008. NCM'08. Fourth International Conference on*, volume 1, pages 144–149.

Weil, S. A., Brandt, S. A., Miller, E. L., Long, D. D. E., and Maltzahn, C. (2006). Ceph: A scalable, high-performance distributed file system. In *Proceedings of the 7th USENIX Symposium on Operating Systems Design and Implementation - Volume 7*, OSDI '06, pages 22–22, Berkeley, CA, USA. USENIX Association.

White, T. (2012). *Hadoop: The definitive guide*. " O'Reilly Media, Inc.".

Zaharia, M., Chowdhury, M., Franklin, M. J., Shenker, S., and Stoica, I. (2010). Spark: Cluster computing with working sets. *HotCloud*, 10:10–10.