

# Revisiting I/O middleware for the Cloud

Karthik Kambatla\*, Naresh Rapolu\*, Jalaja Padma\*, Patrick Eugster, Ananth Grama  
{kkambatl, nrapolu, jpadma, peugster, ayg}@cs.purdue.edu  
Dept. of Computer Science, Purdue University

\* - Student Author  
Submission for both WIP and Poster

Recently, significant research has gone into distributed storage techniques and distributed execution environments to support the storage and analysis of the large amounts of data generated by web-based applications, scientific experiments, business transactions, *etc.*. In storage, specialized distributed file systems like GFS, Ceph; clustered file systems like Lustre, GPFS; key-value storage systems like Bigtable, Dynamo, Cassandra have been proposed to cater to specialized IO workloads.

With the advent of *cloud computing*, all the storage and processing needs are being increasingly shifted to data centers with thousands of nodes and petabytes of storage. Clouds are meant to support a variety of workloads including HPC (Amazon Elastic MapReduce), streaming content delivery (Amazon CloudFront), highly persistent storage/backup of data (Amazon S3). The demands on such underlying storage systems are application specific and involve significant performance trade-offs. For applications with simple storage needs, certain storage platforms prove to be too sophisticated and resource-intensive. The utility computing model of the cloud, where users pay only for the resources they use, encourages lowering resource consumption to reduce costs. Updating the storage system with changing application workloads or hosting multiple storage systems simultaneously is not feasible. For example, we witness memory consumption of about 2GB, in addition to file system usage, while running HBase. Use of Cassandra furthers the resource consumption. The diversity of applications and the lack of prior knowledge of access patterns, workloads and other application-specific requirements pose interesting challenges in designing storage systems. We believe there is a need for a simple, integrated, light-weight, general storage system, accommodating various workloads with different access patterns and consistency requirements. Such a system has to be tunable so that overheads associated with a feature surface only when the feature is used.

## Proposed Idea.

Towards this goal, we have integrated a light-weight key-value store into a distributed file system. Viewing a file as a sequence of blocks/objects (key-value pairs), instead of the traditional byte-offset view, allows (i) efficient key-value and regular accesses, (ii) recording data access patterns, (iii) intelligent placement of file-blocks based on observed access patterns. On these lines, we design a

key-value store atop GFS; intricate issues of security, reliability, and load-balancing handled by GFS. Our design involves a particular file format, KVFile, to store the key-value pairs, and an architecture (closely inter-woven with GFS architecture) to serve file accesses. Our implementation uses HDFS for underlying storage.

The file format, KVFile, incorporates indexing of data for faster access. The file has 3 parts to it — (i) *Data blocks* - which store the actual data as rows of  $\langle key - len, value - len, key, value \rangle$ , (ii) *Data index* - which stores information about the key ranges stored in various data blocks, (iii) *Trailer* - which has information about the file and the pointer to the the data index. Key-value pairs in a data block are sorted. Key-value pair write results in a new data block and a corresponding entry in the data index. In our architecture, the KVFile methods are invoked only for operations on KVFile; otherwise the accesses default to regular HDFS file accesses. To ensure consistent KVFile writes, writes to a file are serialized and are handled by the primary replica (as defined in GFS) alone. The primary replica commits the writes to persistent storage asynchronously. The client library makes a local-copy of the key-value pair before placing a write-request to the primary replica. During reads, the client library checks if a local copy exists; otherwise fetches the data from a replica as in GFS. Similar techniques have been adopted by log-structured storage to handle concurrent accesses in databases. Writes increase the data index size, degrading performance on further accesses. To keep the data index small, multiple partial data blocks are compacted into fewer full data blocks. Writes during compaction are redirected to a new temporary file, and are copied over to the original copy at the end of compaction.

Our evaluation reveals a performance gain of 50% compared to other HDFS file formats on sequential reads/writes, and we support random reads/writes as well. The memory consumption of KVFile storage is lesser compared to HBase, as KVFile accesses are tightly coupled with the underlying storage. Currently, we are comparing KVFile to HBase on different real-world workloads. We are incorporating multiple consistency guarantees for a user to choose from. We are also looking at recording the access patterns on a file so that frequently accessed parts of the file can be replicated and distributed aggressively for higher availability and faster accesses.