The Quantcast File System

Michael Ovsiannikov
Quantcast
movsiannikov@quantcast

Silvius Rus
Quantcast
srus@quantcast.com

Damian Reeves
Google
damianr@google.com

Paul Sutter
Quantcast
psutter@quantcast.com

Sriram Rao
Microsoft
sriramra@microsoft.com

Jim Kelly
Quantcast
jkelly@quantcast.com

ABSTRACT

The Quantcast File System (QFS) is an efficient alternative to the Hadoop Distributed File System (HDFS). QFS is written in C++, is plugin compatible with Hadoop MapReduce, and offers several efficiency improvements relative to HDFS: 50% disk space savings through erasure coding instead of replication, a resulting doubling of write throughput, a faster name node, support for faster sorting and logging through a concurrent append feature, a native command line client much faster than `hadoop fs`, and global feedback-directed I/O device management. As QFS works out of the box with Hadoop, migrating data from HDFS to QFS involves simply executing `hadoop distcp`. QFS is being developed fully open source and is available under an Apache license from `https://github.com/quantcast/qfs`. Multi-petabyte QFS instances have been in heavy production use since 2011.

1. INTRODUCTION

Big data processing is by its nature a large-scale adventure. It can create big opportunities for organizations, and it requires big hardware, which in turn requires big capital and operating investments. A single rack of commodity hardware costs on the order of a quarter million dollars to buy and tens of thousands per year to power, cool and maintain. Designing software for big data is both a technical and an economic challenge—the art of squeezing as much processing out of a hardware investment as possible.

Six years ago, when Apache Hadoop launched, it maximized use of hardware by adopting a principle of data locality. Commodity hardware meant 1 Gbps network links and cluster racks with limited bandwidth to communicate with each other. Since moving data around the cluster was slow, Hadoop strove to leave it where it was and ship processing code to it. To achieve fault tolerance, the Hadoop Distributed File System (HDFS) [13] adopted a sensible 3x replication strategy: store one copy of the data on the machine writing it, another on the same rack, and a third on a distant rack. Thus HDFS is network efficient but not particularly storage efficient, since to store a petabyte of data, it uses three petabytes of raw storage. At today's cost of \$40,000 per PB that means \$120,000 in disk alone, plus additional costs for servers, racks, switches, power, cooling, and so on. Over three years, operational costs bring the cost close to \$1 million. For reference, Amazon currently charges \$2.3 million to store 1 PB for three years.

Hardware evolution since then has opened new optimization possibilities. 10 Gbps networks are now commonplace, and cluster racks can be much chattier. Affordable core network switches can now deliver bandwidth between racks to match disk I/O throughput, so other racks are no longer distant. Server CPUs with vectorized instruction sets are also readily available.

Quantcast leveraged these developments to implement a more efficient file system. We abandoned data locality, relying on faster networks to deliver the data where it is needed, and instead optimized for storage efficiency. The Quantcast File System (QFS) employs Reed-Solomon erasure coding instead of three-way replication. By default it uses a 6+3 configuration, that is, three parity chunks for every six data chunks, which delivers comparable or better fault tolerance vs 3x replication. To store one petabyte of data, QFS uses only 1.5 petabytes of raw storage, 50% less than HDFS, and therefore saves half the associated costs. It doubles the capacity of an existing cluster and saves \$500,000 per petabyte to build a new one. In addition, QFS writes large amounts of data twice as fast, as it needs to write 50% less raw data than HDFS, and brings other performance advantages as well. QFS is plugin compatible with Hadoop and offers several improvements over HDFS:

- 50% disk space savings through erasure coding.
- Corresponding 2x higher write throughput.
- Significantly faster name node.
- Written in C++, easier to connect to system software.
- Support for distributed sorting and logging through a highly scalable concurrent append feature.
- Command line client much faster than `hadoop fs`.
- Global feedback-directed straggler avoidance via centralized device queue size monitoring.
- Deterministic I/O device behavior through direct I/O.

Proceedings of the VLDB Endowment, Vol. 6, No. 11
Copyright 2013 VLDB Endowment 2150-8097/13/09... \$ 10.00.

Table 1: Distribution of MapReduce sort sizes at Quantcast in May and June 2013.

Job sort size	Total sorted bytes
up to 1 TB	7%
1 TB to 10 TB	27%
10 TB to 100 TB	54%
100 TB and over	12%

QFS was developed on the frame of the Kosmos File System [10] an open-source distributed file system architecturally similar to Hadoop's HDFS but implemented in C++ rather than Java and at an experimental level of maturity. What we've built from it over five years is a high-performance file system hardened on our own production workloads, which we have shared with the big data community [2] as a production-ready 1.0.

Throughout this paper we reference the cluster architecture that we developed QFS for. It is a fairly common setup with commodity 1U and 2U servers arranged in racks and connected by a 2-tier Ethernet network, detailed in section 6. While QFS will likely work well on different types of clusters or supercomputers, its performance does depend on the relative ratios of CPU, RAM, disk and network I/O capacity. Section 4 provides information useful to predicting how QFS might perform in a different environment.

2. QFS ARCHITECTURE

QFS has the same basic design goal as HDFS: to provide a file system interface to a distributed, petabyte-scale data store, built on a cluster of commodity servers using conventional hard drives. It is intended for efficient map-reduce-style processing, where files are written once and read multiple times by batch processes, rather than for random access or update operations. The hardware will be heterogeneous, as clusters tend to be built in stages over time, and disk, machine and network failures will be routine.

Architecturally QFS is nearly identical to HDFS, as Figure 1 shows. Data is physically stored in 64 MB chunks, which are accessed via a *chunk server* running on the local machine. A single *metaserver* keeps an in-memory mapping of logical path names to file IDs, file IDs to chunk IDs, and chunk IDs to physical locations. A *client* library, which implements Hadoop's `FileSystem` interface and its equivalent in C++, calls the metaserver to locate the chunks of a file to be read or to allocate a new chunk to be written. Thereafter it reads/writes data directly from/to the chunk server.

2.1 Erasure Coding and Striping

Storing and operating on petabytes of data takes thousands of disks and costs millions. Erasure coding enables QFS not only to reduce the amount of storage but also to accelerate large sequential write patterns common to MapReduce workloads. Moreover, unlike the MapReduce algorithm described by [4], our proprietary MapReduce implementation uses QFS not only for results but also for intermediate sort spill files. As Table 1 shows, almost all our MapReduce workload comes from large jobs. Erasure coding is critical to getting these large jobs to run quickly while tolerating hardware failures without having to re-execute map tasks.

Figure 2 illustrates how a data stream is stored physically using Reed-Solomon 6+3 encoding. The original data is striped over six chunks plus three parity chunks. The QFS client collects data stripes, usually 64 KB each, into six 1 MB buffers. When they fill, it calculates an additional three parity blocks and sends all nine blocks to nine different chunk servers, usually one local and the other eight on different racks. The chunk servers write the 1 MB blocks to disk, until the chunks on disk reach a maximum 64 MB. At this point the client will have written six full data chunks (384 MB total) and three parity chunks. If it has additional data, it will open connections to nine new chunk servers and continue the process.

To read the data back, the client requests the six chunks holding original data. If one or more chunks cannot be retrieved, the client will fetch enough parity data to execute the Reed-Solomon arithmetic and reconstruct the original. Three losses can be tolerated; if any six chunks remain readable the request succeeds. The client will not persist recovered data back to disk but will leave that to the metaserver.

2.2 Failure Groups

To maximize data availability, a cluster must be partitioned into *failure groups*. Each represents machines with shared physical dependencies such as power circuits or rack switches, which are therefore more likely to fail together. When creating a file with Reed-Solomon 6+3 encoding, the metaserver will attempt to assign the nine chunks into nine different failure groups.

Reed-Solomon 6+3 requires at least nine failure groups. With 10 or more one can go offline and still leave a full nine available for writes. With around nine failure groups, they should have about the same raw storage capacity so that one doesn't fill up early. Larger numbers of failure groups will give the chunk placement algorithm more choices, so equal sizing becomes less important. Racks make good failure groups, but for a smaller setup each machine could be its own.

2.3 Metaserver

The QFS metaserver holds all the directory and file structure of the file system, though none of the data. For each file it keeps the list of chunks that store the data and their physical locations on the cluster. It handles client requests, creates and mutates the directory and file structure on their behalf, refers clients to chunk servers, and manages the overall health of the file system.

For performance the metaserver holds all its data in RAM. As clients change files and directories, it records the changes atomically both in memory and to a transaction log. It forks periodically to dump the whole file system image into a checkpoint. In our setup this uses little extra RAM due to the copy-on-write behavior of fork on Linux. On a server crash the metaserver can be restarted from the last checkpoint and the subsequent transaction log.

In the current implementation the metaserver is a single point of failure. This has not been a problem for our batch oriented workload, which can tolerate more downtime than we see. Table 4 shows typical uptimes for our large metaservers. We are planning to implement a high-availability metaserver solution, thus making QFS usable with guaranteed availability in realtime environments as well.

2.3.1 Chunk creation

Spreading load evenly around the cluster is important for

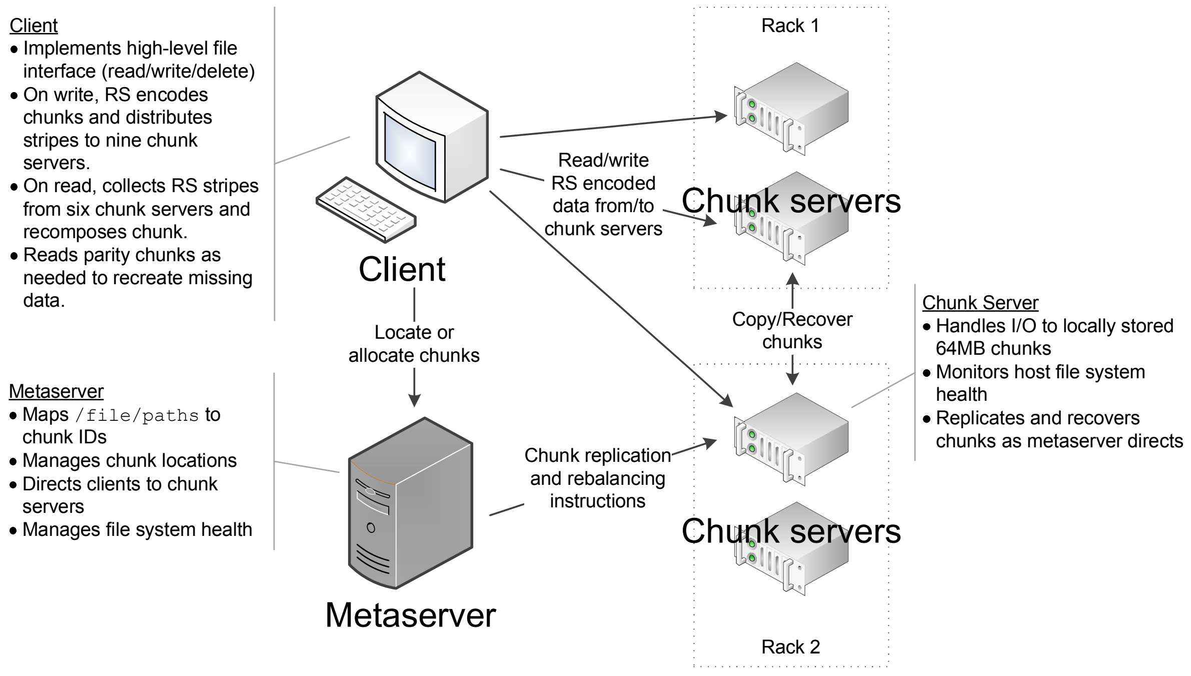

Client
• Implements high-level file interface (read/write/delete)
• On write, RS encodes chunks and distributes stripes to nine chunk servers.
• On read, collects RS stripes from six chunk servers and recomposes chunk.
• Reads parity chunks as needed to recreate missing data.
Client
Locate or allocate chunks
Metaserver
• Maps /file/paths to chunk IDs
• Manages chunk locations
• Directs clients to chunk servers
• Manages file system health
Metaserver
Rack 1
Read/write RS encoded data from/to chunk servers
Chunk servers
Copy/Recover chunks
Chunk Server
• Handles I/O to locally stored 64MB chunks
• Monitors host file system health
• Replicates and recovers chunks as metaserver directs
Chunk replication and rebalancing instructions
Chunk servers
Rack 2

Figure 1: QFS architecture.

scalability. To help do so, chunk servers continuously report to the metaserver the size of I/O queues and available space for each disk they manage. The metaserver dynamically decides where to allocate new chunks so as to keep disks evenly filled and evenly busy. It proactively avoids disks with problems, as they usually have larger I/O queues. This is particularly important in heterogeneous environments, where the throughput ratios of disk I/O to CPU and network vary across machines.

2.3.2 *Space Rebalancing and Re-replication*

QFS rebalances files continuously to maintain a predefined measure of balance across all devices. The rebalance takes place when one or more disks fill up over a ceiling threshold, and moves chunks to devices with space utilization below a floor threshold. The rebalance algorithm is aware of failure groups and will keep each encoding tuplet spread over nine different failure groups.

The rebalance process has a throttle, a maximum number of replications in flight, to avoid taking over the I/O entirely during a large event and to enforce an implicit recovery delay. Most outages resolve within minutes, so replication gets stopped before it uses I/O unnecessarily.

Similar to rebalance, the metaserver also performs asynchronous replication at a client's request.

2.3.3 *Maintaining Redundancy*

In a large cluster, components are failing constantly. Due to such failures, replication target changes, or other causes, the file system can be caught with less redundancy than it should have. Data loss will occur given sufficient simultaneous failures, with a probability dependent on how long it takes the system to restore a missing replica or parity chunk. So it's important to restore missing data promptly.

The metaserver continuously monitors redundancy and recreates missing data. For a replicated chunk this means copying a healthy replica. For an erasure-coded chunk it means reading six of the corresponding chunks and running either the Reed-Solomon encoding (for a lost parity chunk) or decoding (for a lost original data chunk). RS recovery thus involves 250% more I/O than a copy, and some CPU as well. Using vector instructions on an Intel X5670 Nehalem core, encoding runs at about 2.1 GB/s and decoding at 1.2 GB/s. Without vector instructions decoding speed drops significantly, to about 0.7 GB/s.

A terabyte of data represents thousands of chunks, so redundant information for a drive will generally be spread throughout the cluster, and the whole cluster will participate in restoring it. Our cluster recovers a lost drive in a matter of minutes.

2.3.4 *Evicting Chunks*

Eviction is a request to recreate a chunk server's data elsewhere so that its machine can be safely taken down, for example for repairs or upgrades. Simply copying the data off would use the least bandwidth but a good bit of time: for a 12-disk machine with 21.6 TB of data and a 4 Gbps network interface, a copy takes at least 12 hours. The metaserver speeds eviction with RS recovery, allowing quick machine power-downs at the expense of greater network load.

In our setup, draining a rack with 520 TB of data takes about 8 to 10 hours using 500 to 600 Gbps of core network bandwidth. We have evicted full racks while MapReduce jobs were running at full speed without increasing the number of task failures significantly.

2.3.5 *Hibernation*

For quick maintenance such as an operating system kernel

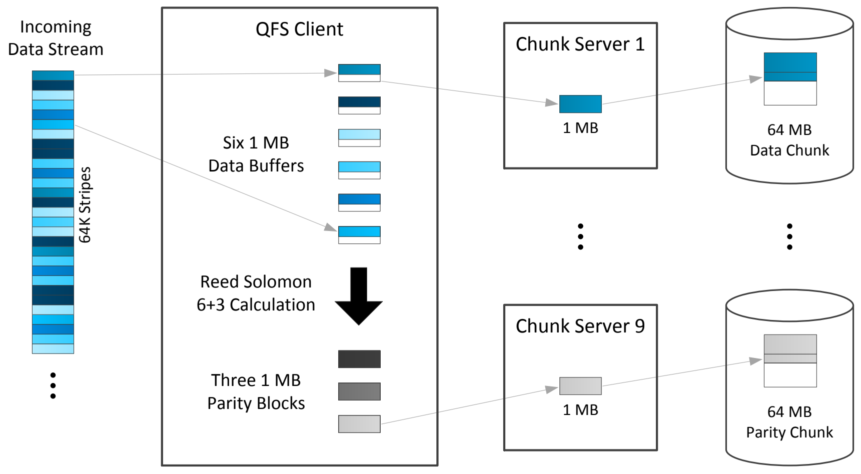


Figure 2: Reed-Solomon encoding for six-way stripes with three parity blocks.

upgrade, chunks are not evicted. Instead, the metaserver is told that chunk server directories are being hibernated. This will set a 30-minute window during which the metaserver will not attempt to replicate or recover the data on the servers being upgraded. Reads from data on the hibernated chunk servers will be serviced through temporary recovery for Reed-Solomon encoded files. Several nodes can be hibernated at once but, to ensure data availability, only within the same failure group.

2.4 Chunk server

Each chunk server stores chunks as files on the local file system. The chunk server accepts connections from clients to write and read data. It verifies data integrity on reads and initiates recovery on permanent I/O errors or checksum mismatches. For replicated files, the chunk server also coordinates synchronous writes to daisy-chained replica chunk servers. It forwards data to the next chunk server at the same time it writes it locally, and reports a successful write only after the rest of the chain has written successfully.

2.5 Interoperability

QFS is plugin compatible with Hadoop, so running in a Hadoop environment involves setting up appropriate bindings to the client library. Migrating data from HDFS to QFS is a matter of running a trivial MapReduce job, for example Hadoop's `distcp` utility. QFS does not depend on Hadoop, though, and can be used in other contexts. The open-source distribution includes FUSE bindings, command-line tools, and C++/Java APIs.

3. QFS IMPLEMENTATION

In this section, we discuss various implementation decisions and optimization specifics in QFS.

3.1 Direct I/O for MapReduce Workloads

By default QFS uses direct I/O rather than the system buffer cache, for several reasons. First, we wanted to ensure that data is indeed written contiguously in large blocks. The kernel would otherwise choose the block size based on a number of local factors that might be globally suboptimal. The kernel does still manage the disk layout, however QFS uses the space reservation feature of the underlying host file system to make it highly likely that data gets written contiguously. Our production file systems use XFS [3] as an underlying file system. However, QFS can run on other host file systems, such as ext4.

Second, we wanted RAM usage to be predictable. Our distributed nodes run principally chunk servers, sorter processes, and MapReduce JVMs. If their memory footprints can be kept fixed, we can configure them to make maximal use of available RAM without the risk of paging, which would create a sudden performance bottleneck and degrade cluster throughput. Using direct I/O prevents variable RAM usage due to the buffer cache.

Third, the QFS metaserver makes chunk allocation decisions based on global knowledge of the queue sizes of all the I/O devices it manages, as discussed in Section 2.3.1. Using direct I/O shortens the feedback loop and keeps its size deterministic and feedback-driven decisions timely. If the buffer cache were turned on, by the time the metaserver learned that the I/O on a particular machine was slow, a large amount of data would have already been passed to the buffer cache, prolonging a performance problem.

3.2 Scalable Concurrent Append

QFS implements a *concurrent append* operation similar to the one described in [7], which scales up to tens of thousands of concurrent clients writing to the same file at once. This feature has been used by Quantsort [12] and Sailfish [11].

The client declares the maximum size of the block it will write, and the metaserver directs it to a chunk server also being written by other clients. The metaserver creates new chunks as needed to keep the chunk size and the number of simultaneous writers constrained and takes locality into account. The chunks thus created will have gaps, since clients' request sizes do not add up exactly to 64 MB, but the client code knows how to read these logical files sequentially. Currently concurrent append is implemented only for replicated (not erasure-coded) files.

3.3 Metaserver Optimization

The metaserver is the most performance-critical part of the file system and has been carefully optimized for fast, deterministic response at large scale.

The metaserver represents the file system metadata in a B+ tree to minimize random memory access, for similar reasons that conventional file systems use such trees to minimize random disk access. The tree has four types of nodes: internal, file or directory (i-node) attribute, directory entry, and chunk info. All keys in the tree are 16-byte integers consisting of a 4-bit node type, a 64-bit key that holds the i-node number (directory id), and a 60-bit subkey holding either a chunk position within a file or a hash of a directory name. See statistics in Table 4.

The keys are constructed such that directory entry nodes immediately follow a directory attribute node to optimize directory listings, and chunk info nodes immediately follow a file attribute node to optimize opening and reading files. The directory entry name hash is used to avoid a linear search with i-node number lookups, where the search key is a parent directory ID and an i-node name. Use of the name hash permits an arbitrarily large number of directory entries within the same directory with no performance penalty, keeping access cost logarithmic.

The metaserver uses its own pool allocators for metadata nodes, to avoid the performance and space overhead of many small OS allocations. It allocates new pools with the OS as necessary, doubling the size of each new allocation up to a limit, and never frees the memory. So the process footprint can grow with the file system size but will not shrink.

A sorted linear hash table allows looking up chunk info by chunk ID, which chunk servers need to do. The hash table implementation delivers constant insertion, lookup, and removal time from few entries to a half billion, while keeping space, RAM access, and CPU overhead to a minimum.

3.4 Client

The QFS client library is designed to allow concurrent I/O access to multiple files from a single client. The library consists of non-blocking, run-until-completion protocol state machines for handling a variety of tasks. One manages a generic QFS protocol for communicating with chunk servers and metaservers, and specialized state machines handle basic reading and writing, write-append, and RS-encoded reading and writing. The latter is implemented as plugins to the basic reading and writing state machines, simplifying the implementation of other encoding schemes in the future.

The state machines can be used directly to create highly scalable applications. For example the Quantsort [12] radix sorter is implemented by using the write append state machine directly. The file read and Reed-Solomon read state machines are used to implement chunk recovery in the chunk server.

The QFS library API is implemented by running the protocol state machines in a dedicated protocol worker thread. All file I/O processing including network I/O, checksumming, and recovery information calculation are performed within this thread. The invoking thread enqueues the requests into the protocol worker thread queue and waits for responses.

The file system meta information manipulations such as *move*, *rename*, *delete*, *stat*, or *list* require communication only with the metaserver. These operations are serialized by the QFS client library and block the caller thread until the metaserver responds. Meta operations don't use the protocol worker thread, which is used only for actual file I/O, which means only a single meta operation can be in flight at at a time on one client instance. Since a client can communicate with only one metaserver, parallel meta operations wouldn't execute faster assuming a low latency network and could create undesirably high contention for the metaserver. On the other hand, if required, it is always possible to create more than one QFS client instance per application or OS task process.

The QFS client library implements file and directory attribute caching to reduce the metaserver load from redundant attribute lookup operations. The main source of such operations is file path traversal, *stat*, and *glob* (name pattern search) calls. The cached file attribute entries can be accessed by directory id (i-node number) and file or directory name, or by full path name. Using full path names avoids path traversal and reduces CPU utilization. The Hadoop Java shim always uses full path names.

The use of *read ahead* and *write behind* keeps disk and network I/O at a reasonable size. The default read-ahead and write-behind parameters are based on the number of data stripes to keep disk I/O requests around 1 MB. The buffer size can be changed per file through the QFS client API. The buffering implementation leverages scatter-gather optimization in the kernel to keep buffer copying to a minimum. The protocols are detailed in the Appendix.

4. PERFORMANCE IMPLICATIONS

A high-performing cluster requires a complex, dynamic balance between resources such as storage, CPU, and network, under different types of load. This section presents a somewhat theoretical exploration of how QFS's erasure coding shifts the balance relative to 3x replication. Note that QFS supports either on a per-file basis, so what follows is a comparison not so much of file systems as of different optimization approaches.

Our hypothetical cluster contains nine or more racks of hardware and a two-tier network, in which a top-of-rack switch handles each rack's internal traffic, and racks communicate with each other via a layer of core switches. The servers contain only spinning disks, gigabit or faster Ethernet, and relatively modern CPUs—Intel Core, Nehalem and Sandy Bridge. We further assume default QFS allocation policy: one chunk is written to the client's host and the other eight to hosts on eight different racks. For three-way replication we assume the default HDFS policy, which places one replica on the client's host, another in the same rack and the third in a different rack.

Table 2 shows a high-level summary of how erasure coding affects resource usage. The space required is cut in half, three host failures are tolerated rather than two, and three rack failures rather than one. You can take any rack down for maintenance and the system is still more resilient to failures than a mint HDFS installation. A different replication policy that kept all replicas on different racks would improve resilience at a cost of doubling core network consumption.

It is important to note that erasure coded files can actually use more space than replicated files. When the file size is smaller than the stripe block size, the effective space use factor will be 4x with Reed-Solomon 6+3 because each

Table 2: Fundamental tradeoffs when storing 384 MB with erasure coding vs. replication.

	RS 6+3	3-way repl
Disk space used	576 MB	1,152 MB
Write Disk I/O	576 MB	1,152 MB
Write Host NIC I/O	512 MB	768 MB
Write Intra-rack I/O	0 MB	384 MB
Write Core Switch I/O	512 MB	384 MB
Read Disk I/O	384 MB	384 MB
Read Host NIC I/O	384 MB	0–384 MB
Read Intra-rack I/O	0–64 MB	0–384 MB
Read Core Switch I/O	320–384 MB	0–384 MB
Host failures tolerated	3	2
Rack failures tolerated	3	1

parity stripe is the same size as the original. Across our production file systems the effective space use factor is between 1.5x and 1.53x, as Table 4 shows. We use a block size of 64 KB.

4.1 Sequential Write Performance

A 6+3 erasure-coded cluster can theoretically write twice as fast as a 3x replicated one, since it has only half as much physical data to write, assuming disk I/O is the bottleneck. Erasure coding will similarly reduce overall network load for writing about 33%, since it will transmit 1.3x the original data versus 2x for replication. It will also increase CPU load on clients somewhat to calculate parity information.

From the point of view of a single client, a replicated file can be written only as fast as a single disk, on the order of 50 MB/s. An erasure-coded file can be written six times faster since it is striped in parallel across multiple disks, assuming other resources permit. A 1 Gbps network link would throttle writes near 94 MB/s, since parity data will need an additional 47 MB/s, and 1/9 of the total will be written locally. This highlights the benefit of 10 Gbps NICs. And those parallel spindles must be free: although the client can benefit from having six drives to write to, every drive will serve six times as many clients. This parallelism will speed writes only to the extent the cluster has idle drives that can be put to work. Due to load variations and imperfect scheduling, even a busy cluster often will.

Note that strictly speaking the parallelism boost comes from striping, not from erasure coding, and will vary with the striping parameter. A similar effect could be achieved by using replication with a very small chunk size and writing to multiple chunks at once. For example, a chunk size of 1 MB would come close, though it would increase memory use on the name node, which would have to keep track of 64 times as many chunks.

4.2 Sequential Read Performance

Peak system read throughput will be comparable between erasure-coded and replicated clusters. Both schemes must read the same amount of physical data, and both will bottleneck on disk I/O throughput, again assuming adequate core network bandwidth.

For a single client, reading from erasure-coded (or simply striped) files is up to six times faster, network and spindles permitting, for the same reasons discussed above.

4.3 Random Read Performance

Regardless of the redundancy scheme, random read performance is generally poor on spinning disks, which cannot generally perform more than 50 operations per second (IOPS). To maximize performance, block size on disk must be larger than requested read size so the request can be done in a single operation. Erasure coding will not affect performance one way or the other as long as the stripe size is significantly larger than the average read. When stripes are smaller a read will have to merge data from multiple chunk servers. Although this wouldn't slow down a single client, it increases the system IOPS load by up to 6x, potentially creating a system bottleneck.

The stripe block size is a file attribute which can be set according to each file's access pattern. We use 1 MB for files intended for random reads and 64 KB otherwise.

4.4 Broadcast Performance

Another common access pattern in distributed data processing is broadcasting data via the file system. For example, thousands of workers may need to receive some common configuration data before they can begin, so Hadoop writes the data to the file system and they all read it from there. The reads can create local network bottlenecks, since although the configuration data is replicated across several hosts, each has too many clients to serve it to.

Although a specialized peer-to-peer system would solve the problem well, broadcasts via the file system have the advantage of simplicity, and striping allows them to scale well. A large striping factor (we use 128) and small replication count (we use 2) distributes the data and therefore the read load much more widely around the cluster, avoiding network bottlenecks. Even without striping, a small chunk size will have a similar effect, though at the cost of additional RAM pressure on the name node.

4.5 MapReduce Performance on QFS

MapReduce processing involves primarily large sequential reads and writes, so the discussion above applies, but a specific MapReduce pattern merits focus. A key determinant of MapReduce job run time, which is to say how long users must wait for results, is the long tail of "straggling" workers. [4] Inevitably a few of the hundreds or thousands of devices a job reads from will be suffering problems or will host data in high demand from other jobs, and will run slowly as a result. The slowest 1% of workers can account for 50% of the elapsed job time.

Timely job output therefore depends on good straggler management. With replication, a strategy for handling stragglers is to start redundant speculative tasks and assign them to replicas in order to avoid whatever problem affects the slow device. It's a gamble that pays off when the source of the slowness is, for example, CPU contention. In that case, a redundant worker is likely to be able to read the data quickly and finish it quickly. The payoff is even bigger using erasure coding, since the worker can read from six devices in parallel.

But when the slowness was due to a disk I/O bottleneck, speculative re-execution can make a straggler even slower. A large cluster will often have three slow nodes, so often some unlucky chunks will have all three replicas bottlenecked. The redundant tasks will then pile additional load on those nodes, slowing them even further. Pile-on is a

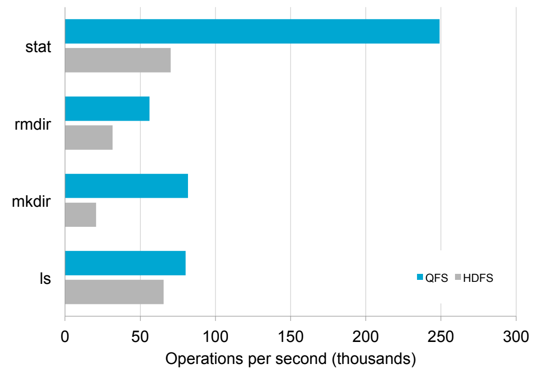


Figure 3: Metaserver operation performance.

certainty using erasure coding, since there is only one copy of the original data. But QFS has an alternate recovery mechanism. If a chunk server takes more than 20 seconds to respond, the client will read parity blocks and reconstruct the data it needs. This adds a delay but avoids rereading the full data, and the three parity stripes reduce the odds of stragglers delaying the job. Replication can tolerate two slow nodes, 6+3 RS can tolerate three. In our environment this timeout approach appears to work better than speculative re-execution plus replication.

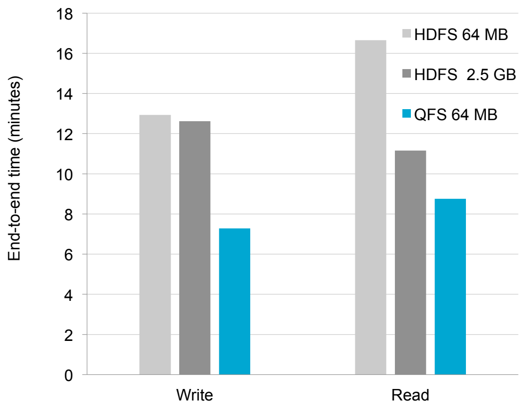
Writing to the distributed file system could also stall due to a slow device, which can mean a whole stalled MapReduce job. Reed-Solomon 6+3 is theoretically more vulnerable to slow drives, since a write involves three times as many drives. On the other hand, QFS's chunk allocation avoids drives with long I/O queues.

5. BENCHMARKS

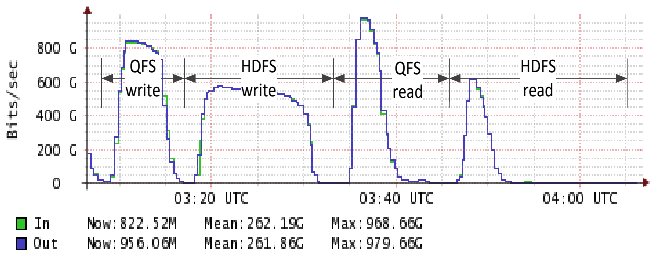
We conducted two tests of QFS relative to HDFS at scale, designed to compare performance under controlled conditions that approximate real-life load patterns. Of course, a cluster is a multi-user system, and our real-life MapReduce jobs run concurrently with scores of other jobs, so "controlled" and "real-life" are conflicting goals. We nevertheless offer these as our results and invite others to conduct their own tests. The metaserver benchmark in particular is fairly environment independent, and we've shared its test scripts on Github [2].

5.1 Metaserver/Name Node Performance

Our first test compared the performance of the QFS metaserver vs. the HDFS name node. The test setup involved one QFS metaserver and one HDFS name node running (not at the same time) on a dual Xeon E5-2670 machine with 64 GB RAM, loaded by 512 client processes running across 16 machines. Each client process built, queried, or dismantled a balanced directory structure of 384 directories each with 384 subdirectories. In other words the tests executed 75.7 million operations on a file system with 75.7 million directories at its peak. We focused on directories rather than files because although creating a file in QFS involves only the



(a) 20 TB MapReduce job performance.



(b) Network usage across whole cluster.

Figure 4: MapReduce performance, QFS vs. HDFS.

metaserver, in HDFS it involves extra overhead to communicate with a data node, which seemed an unfair comparison. We used as a reference HDFS version 2.0.0-cdh4.0.0.

As Figure 3 shows, QFS's lean native implementation and optimizations pay off consistently, from a 30% edge for a simple directory listing to nearly 4x improvement in directory creation.

5.2 MapReduce Read/Write Job Throughput

To test the performance of the file system as a whole, we ran simple MapReduce jobs that either wrote or read 20 TB of data on our production cluster, described in section 6. We used the `teragen` program shipped with the Hadoop 1.0.2 distribution, configured it to use 8,000 MapReduce workers, and ran it on an otherwise idle cluster. We compared its performance using QFS 1.0 as the data store and using the Apache HDFS 1.0.2 distribution. We tried two block sizes for HDFS, 64 MB and 2.5 GB, and 64 MB for QFS. We much prefer to keep the chunk size small so that MapReduce tasks can be kept short, thus easier to schedule.

Figure 4(a) shows the end-to-end job execution times. The write times show the expected speed-up due to 50% less I/O volume. Figure 4(b) shows the network use measured as sum of bandwidth used at host network interface across all hosts while these benchmarks were run. The QFS write volume (the area under the first peak) was clearly smaller than the HDFS volume (the area under the second). HDFS used more disk I/O than the graph suggests, since it writes a third of its data locally.

Neither file system had a data volume advantage for the

Table 3: Quantcast MapReduce cluster setup.

			Network per host	
CPU	RAM	Disks	in rack	to core
1 Intel x32xx	8 GB	4	1 Gbps	0.9 Gbps
2 Intel x56xx	48 GB	12	4 Gbps	3.3 Gbps
2 Intel E5-26xx	64 GB	12	10 Gbps	6 Gbps

read test: both had to read the same 20 TB of physical data. The third and fourth peaks of Figure 4(b) show that QFS loaded the network significantly more, while HDFS was able to read more data locally, but both took about five minutes of active time to read most of the data. However on QFS the job ran for another five minutes, and on HDFS for another 10, waiting for straggling workers to finish. The cluster was mostly idle in the straggler phase, and QFS's read parallelism delivered a significant advantage. While straggling tasks using replication were bound by single-disk read speed around 50 MB/s, QFS's striping allowed them to read up to six times faster. Figure 4 shows QFS finished significantly faster when both systems used the same 64 MB block size, though using 2.5 GB blocks for HDFS brought it closer to QFS performance. The read experiment was run with default Hadoop settings, including speculative tasks. A similar number of speculative tasks was run both with QFS and HDFS.

6. QFS USE IN PRODUCTION

Quantcast has run QFS in production contexts throughout its development since 2008, and exclusively since 2011. We rely on several QFS instances, as Table 4 describes. One is used to store several petabytes of log data. Another, also holding several petabytes of data, is used as our MapReduce input/output store. Other instances are used to store sort spills or backup data.

These different file system instances run across the whole cluster, and every machine runs a chunk server for each. The file systems are configured with different fill and rebalance thresholds, which allows us to provision space evenly for the sort spill file system by setting its fill threshold higher than the other file systems. They may run different versions of QFS, which allows us to perform gradual upgrades, starting with the least-sensitive-to-loss sort spill file system and ending with the most sensitive backup file system.

Table 3 shows the cluster composition, which totals about 7,000 disks, 9,000 CPU cores, 25 TB of RAM, and 10 PB of disk. On a typical day we run about 3,000 MapReduce jobs, and QFS handles about 5 to 10 PB of compressed data I/O. We're satisfied with QFS uptime; though individual nodes fail constantly, the system as a whole runs reliably for months at a time between maintenance activity.

Shortly after QFS's open-source launch in September 2012, Neustar [9] reported adopting it for a 3.6 PB store, and that it allowed them to double their data retention with no additional hardware cost.

7. RELATED WORK

Comparison to HDFS. Much of the paper was about this. To sum it up, QFS offers 50% storage savings by employing erasure coding rather than replication as a fault tolerance mechanism. Note that HDFS offers a RAID implementation [5] layered on top of an HDFS file system, which makes it harder to use. At the time of this writing there was no mention of HDFS RAID being used for anything other than large backups. Beyond the hardware savings, QFS offers significant speed improvements out of the box, simply by using QFS instead of HDFS. The QFS MapReduce implementation, Quantsort [12], was developed in tandem with QFS and makes use of the QFS scalable concurrent append feature to speed up MapReduce significantly and make it easier to operate. Also, sort spills are written only to QFS, never to the local file system. This makes it easier to work around hardware failures, as write workload can be diverted to another host. It also makes it easier to tolerate CPU / I/O imbalance in heterogeneous clusters.

Comparison to GFS. Both HDFS and QFS follow the GFS [7] architecture. However, the original GFS design suffered from the same 3x storage expansion ratio as HDFS. We found limited information online [6] on an upgrade to GFS that provided distributed metadata as well as small block size and Reed-Solomon encoding. GFS is proprietary, thus we could not compare its performance head-to-head.

Table 5 presents a brief comparison of features of different large-scale distributed file systems. QFS, HDFS, and GFS are fully fault tolerant, while GPFS generally relies on the hosts not to fail. However, GPFS tolerates drive failures by using single-machine RAID. GPFS is the only one supporting POSIX semantics. Only QFS and HDFS are open source and support Hadoop MapReduce.

DFS [8] is a high performance distributed blob store. It uses striping and works well on clusters with balanced disk and network throughput. Unlike QFS, it replicates not only chunk data but metadata as well. It has a single point of failure, does not offer a file system interface, and requires space comparable to HDFS as it employs 3x or higher replication. DFS is currently at an advanced experimental maturity level.

7.1 Limitations and Future Work

A notable QFS limitation is the single point of failure in the metaserver. In many MapReduce environments, including our own, a highly available central node would be nice to have but isn't critical, as temporary outages could be tolerated if they occurred. As table 4 suggests, outages tend to be rare and deliberate. High availability is on our roadmap but may have more business impact in other environments.

Additional improvements on our roadmap include federation capability and secure authentication, improving on QFS's existing POSIX-like user and group structure.

8. CONCLUSIONS

It is the nature of big data that it tends to become bigger, with hardware and operating costs scaling to match, quickly into the millions for many organizations. By reducing disk space requirements by 50%, QFS offers data-bound organizations dramatically better economics. It allows them to squeeze twice as much data into their existing hardware, to put off their next data center expansion for a year or two, and to spend half as much when they finally do expand.

QFS integrates easily with Hadoop but does not require it and may provide a benefit in environments that read and write large, sequential blocks of data through other means. As an open-source project, it's free of license fees that could undermine its economics and is positioned to benefit from community scrutiny and collaboration. It has also been

Table 4: Some of the QFS instances currently in production at Quantcast. The third column shows the effective replication factor.

					Metaserver			
Purpose	Size	Repl.	Files	Blocks	Clients	RAM	Uptime	Last Restart
MapReduce input & output	3.6 PB	1.506	$31 * 10^6$	$240 * 10^6$	40,000	45 GB	197 days	HW upgrade
Logs	3.1 PB	1.500	$46 * 10^6$	$398 * 10^6$	10,000	74 GB	111 days	Increase RAM
Backups	406 TB	1.530	$8 * 10^6$	$45 * 10^6$	400	9 GB	156 days	Can't remember
Column store	21 TB	1.509	$2 * 10^6$	$11 * 10^6$	115,000	10 GB	197 days	Can't remember
Sort spill file system	0-400 TB	N/A	$0.5 * 10^6$	$2.7 * 10^6$	25,000	6 GB	50 days	SW Optimization

Table 5: Comparison of a selection of distributed file systems.

Feature	QFS	HDFS [13]	GFS [7]	GPFS [1]	DFS [8]
Raw space needs	+50%	+200%	+200%	+100% (flexible)	+200%
Tolerate disk failures	yes	yes	yes	yes	yes
Tolerate host failures	yes	yes	yes	no	yes
Scale	PBs	PBs	PBs	PBs	100s of TB
Open source	Apache License	Apache License	no	no	no
Hadoop compatible	yes	yes	no	no	no
Federated namespace	no	yes	yes	yes	yes
High availability name node	no	yes	yes	no	no
Small blocks	stripes	no	yes	no	possible
POSIX compliant	no	no	no	no	no
Implementation language	C++	Java	C++	unknown	unknown

severely tested over five years in Quantcast's production environment.

QFS will not meet everyone's needs but has met ours, and we believe it will add a great deal of value for other organizations working with big data. We're pleased to be able to offer it back to the open-source community and welcome collaboration on its continued development.

9. ADDITIONAL AUTHORS

- Chris Zimmerman (Quantcast, zim@quantcast.com)
- Dan Adkins (Google, dadkins@google.com)
- Thilee Subramaniam (Quantcast, thilee@quantcast.com)
- Jeremy Fishman (Quantcast, jfishman@quantcast.com)

10. REFERENCES

[1] GPFS. http://en.wikipedia.org/wiki/GPFS.

[2] QFS Repository. http://quantcast.github.com/qfs.

[3] XFS. http://en.wikipedia.org/wiki/XFS.

[4] J. Dean and S. Ghemawat. Mapreduce: Simplified data processing on large clusters. In *OSDI*, pages 137–150. USENIX Association, 2004.

[5] H. K. et al. HDFS RAID. http://wiki.apache.org/hadoop/HDFS-RAID, 2010.

[6] A. Fikes. Storage architecture and challenges (google). http://tinyurl.com/6vbhgzn.

[7] S. Ghemawat, H. Gobioff, and S.-T. Leung. The google file system. In M. L. Scott and L. L. Peterson, editors, *SOSP*, pages 29–43. ACM, 2003.

[8] E. B. Nightingale, J. Elson, J. Fan, O. Hofmann, J. Howell, and Y. Suzue. Flat datacenter storage. In *Proceedings of the 10th USENIX conference on Operating Systems Design and Implementation*, OSDI'12, pages 1–15, Berkeley, CA, USA, 2012. USENIX Association.

[9] M. Peterson. Using hadoop to expand data warehousing (neustar). http://tinyurl.com/cpjc7ko, 2013.

[10] S. Rao et al. The kosmos file system. https://code.google.com/p/kosmosfs, 2010.

[11] S. Rao, R. Ramakrishnan, A. Silberstein, M. Ovsiannikov, and D. Reeves. Sailfish: A framework for large scale data processing. In *ACM Symposium on Cloud Computing*, 2012.

[12] S. Rus, M. Ovsiannikov, and J. Kelly. Quantsort: Revolution in map-reduce performance and operation. http://tinyurl.com/c4hkftm, 2011.

[13] K. Shvachko, H. Kuang, S. Radia, and R. Chansler. The hadoop distributed file system. In M. G. Khatib, X. He, and M. Factor, editors, *MSST*, pages 1–10. IEEE Computer Society, 2010.

APPENDIX

A. QFS CLIENT PROTOCOLS

Write protocol for replicated files

1. The client requests metaserver to allocate a chunk by specifying a file ID and chunk position within the file.
2. The metaserver creates a chunk write lease, if no corresponding read leases exist.
3. When the metaserver creates a new chunk it chooses chunk servers to host replicas, based on rack, host, and load constraints, and forwards an allocation request to the chosen chunk severs.
4. Once all chunk servers respond the metaserver returns a list of chosen chunk servers to the client.

5. The client requests the chunk servers hosting replicas to allocate a write ID.
6. The client computes the data checksum and sends the data along with the checksum to the first chunk server in the chosen chunk server list (the write master).
7. The write master receives the data, validates the data checksum, write ID, and write lease. If successful, it issues the corresponding chunk write, forwards the data to the chunk server hosting the next replica (if any), and waits for it to return a write status. This process repeats on all chunk servers hosting chunk replicas.
8. When the write master receives a write status from the last chunk server in the synchronous replication chain, it forwards its status to the client.
9. If successful, the client returns the status to the caller, otherwise it retries the write starting from step 1.

Read protocol for replicated files

1. The client gets a list of the chunk servers hosting chunk replicas for a given file ID at a specified position within the file.
2. The client acquires a chunk read lease.
3. The client chooses one of the chunk servers hosting replicas and issues a read request to this server.
4. The chunk server fetches data from disk, verifies the checksum, and returns the response to the client.
5. The client verifies the data checksum and if successful returns the response to the caller. Otherwise it retries the read with a different chunk server if available.

Write protocol for striped files (with and without erasure coding)

1. The client computes data and recovery chunk data. Every chunk corresponds to a single stripe.
2. For each chunk the replicated write protocol described the above is used. All chunks writes are issued in parallel. The client can invalidate a chunk if all replicas (single replica with replication 1) of the chunks are lost during the write. The metaserver will schedule chunk recovery once the client relinquishes the write lease.

Read protocol for striped files (with and without erasure coding)

1. The client issues reads of the appropriate chunks and stripes using the replicated file write protocol in parallel.
2. If the reads are successful, the client assembles the response and returns the data to the caller. If one of the reads fails the client issues a read of one of the parity stripes, and if successful recomputes the missing data.

Reliable concurrent write append protocol. The write append protocol at the high level is similar to replicated files write protocol. The main difference is that with multiple writers appending to a chunk, clients do not know the final chunk that the data will end up in. This decision is ultimately deferred to the metaserver. Under normal circumstances the first chunk in the synchronous replication chain (write master) makes this decision and notifies the metaserver. Should the write master become unavailable, the metaserver makes this decision. With write append each chunk server keeps in memory the status of the last request for every client / write ID. In case of a timeout (chunk server failure or communication outage) the client can recover the state of the last append operation by querying the (remaining) chunk servers in the synchronous replication chain.