The ScoutFS Archiving File System

Zach Brown, Harriet Coverston, Ben McClelland
Versity

ABSTRACT

ScoutFS is a scalable clustered file system designed and implemented for archiving large data sets to low cost external storage resources such as tape, disk, object, and cloud. ScoutFS is implemented in the Linux kernel, and has been released under the GPLv2 open source license. ScoutFS supports full POSIX semantics. A minimalist design and implementation makes the most of the optimized Linux software ecosystem and is built to extract the most value from economical commodity hardware.

While meeting expectations of modern reliable file system design like transactional updates and rich metadata that ensures data integrity, ScoutFS adds additional design elements such as indexing that accelerate the task of using file system metadata to manage external archive resources. The namespace capacity target of 1 trillion files addresses one of the urgent requirements in the file based large scale data archiving market where data growth has pushed the number of files beyond the capacity of legacy file systems.

In this paper we summarize the design elements of the open source ScoutFS file system which solves some of the unique problems associated with large scale file based data archiving.

INTRODUCTION

We have designed and implemented the open source ScoutFS Archival File System to meet the growing need for dramatically enhanced file based archiving tools. Metadata management and namespace capacity limitations have hampered the ability to scale up low cost archival storage systems at a time when exascale sized archives are becoming more common.

We are frequently asked, why did you have to build a whole new file system? The answer is that a modern GPL archiving file system does not exist, and proprietary file systems capable of managing namespaces holding in excess of 10 billion files also either do not exist, are far too complex to deploy and manage, or are cost prohibitive for archival storage solutions.

ScoutFS shares similar design goals with previous block file systems including an emphasis on data integrity, reliability, recovery, performance, and scalability. However, the ScoutFS design has been heavily influenced by direct observations from Versity's large scale archiving deployments. Archive specific workload characteristics led us to a unique set of design goals that does not overlap with existing file system technology. First, both enterprise and HPC users require POSIX for the foreseeable future. Relaxing POSIX solves many scalability problems but it is not compatible with the enormous installed base of applications.

Second, demand for massive namespaces is real and growing. The largest archival storage sites are seeking the ability to manage up to 1 trillion files in a single POSIX compliant namespace without resorting to storing file metadata in a separate non-coherent database.

Third, in addition to the need for a large namespace, there is a requirement for high throughput and high file creation rates. The only way to archive hundreds of millions of files is to spread out metadata handling across a cluster of nodes. By harnessing the power of many nodes, the system is capable of very high file creation rates.

Fourth, finding files in a massive namespace must be efficient. Users and applications frequently need to find files meeting certain criteria. In the archiving use case, lists of both archived and unarchived files must be made available to the userland application in order for it to apply policies and execute work. Scanning a very large namespace is time and resource prohibitive, therefore, a metadata index is needed.

Fifth, a small code base and a radically simple file system design is desired to keep the surface area of the project manageable and focused. The goal was not to produce another general purpose file system.

Finally, an open source GPL archiving file system is an inherently safer and more user friendly long term solution for storing archival data where accessibility over very large time

scales is a key consideration. Placing archival data in proprietary file systems is costly and subjects the owner of the data to many vendor specific risks including discontinuation of the product. As in other technology verticals, it is likely that open source archival file system software will come to dominate the landscape.

DESIGN ELEMENTS

Node Leader

In a ScoutFS cluster a node is chosen to act as the leader. It listens on a TCP socket and sends and receives messages to and from the other nodes. It is responsible for processing requests that reference global state and is the core functional component for maintaining a consistent global POSIX namespace. For example, the leader hands out allocations of large batches of unique inode numbers, transfers extents between nodes and the global extent allocator, and updates the B+Tree index that describes the LSM tree (see Figure 1).

The leader is dynamically elected from amongst all the nodes that currently have the volume mounted. If the current leader crashes or leaves, a new next leader is elected from the remaining nodes.

The requests that are processed by the leader operate in low frequency, high capacity batches. Instead of an inode allocation request resulting in the allocation of a single inode, an allocation request will result in a grant of millions of inodes which can be used to satisfy millions of allocations before it becomes necessary to send the next request to the leader. Rather than a conventional allocation process where a node sends a request to create a file, a ScoutFS node sends a request to index a large region of the block device where the node has already stored the result of thousands of file creations. In this way, the single leader does not become a bottleneck and is only required to process requests at a modest rate in response to very high workloads throughout the cluster.

Shared Block Storage

ScoutFS is deployed on a single node or multiple nodes which all have access to a consistent shared block device. A shared block filesystem architecture was selected in order to obtain the target performance levels with the simplest consistency model. The most basic installation can run on a single node with simple mirrored devices. Typical Versity sites can leverage their experience with enterprise SAN devices that support dozens of nodes, while the most ambitious sites can deploy on large software defined distributed block devices such as Ceph or virtual SAN file systems like ScaleIO, DataCore or others.

B+Tree Index

The heart of a ScoutFS volume is a handful of B+Tree indexes that describe the rest of the persistent structures in the volume. The total size of these indexes is bound by the total device size and can, at most, grow in size to consume a small fraction of the total device capacity.

The two most important B+Tree indexes track global free block extents in the volume and describe the large block segments that store the primary file system metadata.

Updates to the B+Tree index are batched into transactions and are written by the leader node. These transactions are written to free space on the shared device. Previous versions of the indexes may be read and cached concurrently by other nodes while updates are being written. This is crucial to distributing performant concurrent index lookups throughout the cluster.

LSM Index

The primary POSIX file system metadata that makes up a ScoutFS volume is indexed by a Log Structured Merge Tree. Filesystem metadata information is stored as items identified by a multi-field key that positions each item in a sorted global key space. Groups of tens of thousands of adjacent sorted items are packed into large multi-megabyte segments. Sorted groups of segments with non-overlapping keys are then organized into levels with geometrically increasing numbers of segments per level. For example, a common ratio of 10 would result in level 1 containing 10 segments, level 2 containing 100, level 3 containing 1000, and so on (see Figure 2).

Metadata LSM Writes

Each node builds up transactions with the item creations, deletions, and updates which together comprise an atomic update to the POSIX file system metadata. It packs all these items into a new large segment and writes it to free space on the shared device. It then sends a notification of this new segment to the node who is updating the B+Tree index which describes the segments. This

node updates the B+Tree index to reference this new segment in a level 0 that is specifically for storing new segments. Once the atomic B+Tree update is persistent, all other nodes can read the index to discover the segment that contains the new file system metadata.

Metadata LSM Reads

Nodes read POSIX file system metadata by reading items from segments that are described by the index. They ask the current leader that owns the B+Tree for the block that contains the current root of the index. It then traverses the index by directly reading blocks from the shared device. Using the B+Tree index it finds and reads the segments at each level which could contain the item it is seeking. It progresses from smaller newer levels to the older larger levels. The first it finds is the most recent and is used, allowing the reader to avoid reading the rest of the older segments in the index. Segments are never rewritten so they can be read without incurring the overhead of participating in exclusive locking with potential writers of new versions of the segment.

LSM Compaction

As the file system changes, segments accumulate in level 0 of the LSM tree. Periodically these segments are merged with the level 1 segments whose range of keys intersect with the level 0 segments. All of the level 0 and level 1 segments are read, duplicate and deleted items are removed, and the resulting items are written in sorted order to a new group of level 1 segments. This operation is known as compaction and is responsible for both maintaining balance in the LSM tree and reclaiming free space if the tree shrinks. Compactions are performed concurrently throughout the cluster with the leader coordinating the non-overlapping location of compactions in the tree. The leader makes atomic updates to the index that reflect the results of each compaction.

File Data Extents

File contents are directly written to and read from the shared block device. File contents are overwritten to avoid the fragmentation cost of CoW file data updates. Extents of blocks are mapped to files in metadata items that are managed along with the rest of file system metadata in the LSM index. Global free extents are stored in the B+Tree index and are communicated between nodes. Each node has a private key space of metadata items that allows it to allocate and free extents locally without coordination with other nodes. Local free extents are returned for reallocation once a node accumulates sufficient free space or if the node is evicted from the cluster.

Resiliency, Consistency, and Reliability

ScoutFS carefully structures its metadata and performs reads and writes in a way that ensures strong data consistency.

Every change to a ScoutFS volume is written as an atomic transaction. All of the blocks that make up each new transaction are written to free space. This is true for both the B+Tree index blocks and the LSM segment blocks. Once these block writes are stable, a final commit block is written which references all the new blocks. No existing stable metadata blocks are overwritten during an update. If this process is interrupted, the system will ignore any partial writes to free space and will carry on from the previous consistent version of the volume. There is no need to repair interrupted inconsistent images nor replay journals or logs before resuming operation on the volume.

References between metadata blocks in ScoutFS contain robust information which ensures that the correct block is read. Each referenced block contains fields that identify the block as a member of the specific volume, the block location that it was written to, a counter which represents the version of the block, and a strong checksum of the entire block including each of these fields. Each reference to a block specifies both the location and version of the block. By verifying all of these fields, a read is certain that it read exactly the block it was seeking, not a block from another volume, not a previous version of the block, nor a similar version of a block from a wrong location.

Reliable atomic metadata updates are further leveraged to greatly improve the process of recovering from a node failure. ScoutFS uses clustered quorum software to safely determine cluster membership and the role of the leader. If a node leaves the cluster it does not leave behind log fragments that must be replayed. If the leader dies a new leader is elected and it continues operating on the current consistent version of the metadata.

POSIX Coherence Locking

ScoutFS uses a lock manager protocol to ensure safe access to and modification of the multiple dependent items that make up the POSIX file system. Each node can lock a range of keys around an item it is reading or writing. The size of the locked range is chosen to balance locking communication and contention between nodes.

One of the fundamental benefits of the shared LSM index is that it allows ScoutFS to decouple logical metadata item locking from serialized physical structure access.

Nodes obtain locks on the primary items they need to exclusively change given POSIX semantics. The nodes gather up all of these items and write them into a level 0 segment. Nodes can do this concurrently if their logical POSIX operations do not conflict.

ScoutFS uses concurrent writer locks to maintain secondary indexes of primary metadata without creating contention on the secondary index. For example, an exclusive POSIX inode update can also acquire a concurrent writer lock on item keys that make up an index of inodes by a secondary attribute such as the modification time or size. Readers are excluded during the update of the secondary index but all writers can operate concurrently. The secondary index items are included in the level 0 segments and are visible atomically with the primary structures. Readers of the secondary index are relatively infrequent in our design target workload where background archival agents are applying a policy. Concurrent secondary index writing allows ScoutFS to greatly accelerate concurrent updates of the secondary indexes that in turn accelerate archival agents.

Archival Interfaces

ScoutFS uses all of the mechanisms discussed in the previous sections to maintain metadata that presents a coherent POSIX namespace between nodes. In the same way, it simultaneously maintains metadata that enables archiving agents to orchestrate file reading and writing on external archive resources.

First, it maintains a persistent index of files sorted by the order that they were modified. Files most recently updated are found at the end of the index. This allows the archival agent to ensure that archival policies are applied to changed files without having to search through all of the files in the system. The index is persistent and all files are always present so this process applies across service interruptions and can be restarted from arbitrary points in time. The index may be queried using the Accelerated Query Interface (AQI).

Second, it maintains extent records on each file that record whether the file contents are present on the block device or are stored on external archive media. An interface call frees blocks and marks extents offline. Another interface writes contents to existing extents and marks them online. The archival agent marks files offline once they are safely present in the archive and space on the block device needs to be reclaimed. The agent writes file contents when it gets notification that processes are trying to read from offline regions of files. The motion of file contents between the block device and the archive is transparent to users of the filesystem. Whether a file's extents are online or offline can be observed through specific management interface calls.

CONCLUSION

ScoutFS employs a small set of precise tools - concurrent CoW B+Trees, shared LSM Trees, and range locking - to provide a scalable, coherent POSIX file system that enables archiving to cost efficient external storage resources.

Key areas of innovation within the ScoutFS project include increasing the capacity of POSIX namespaces, eliminating the need for file system scans, and harnessing the power of multiple nodes to reach extremely high file creation rates.

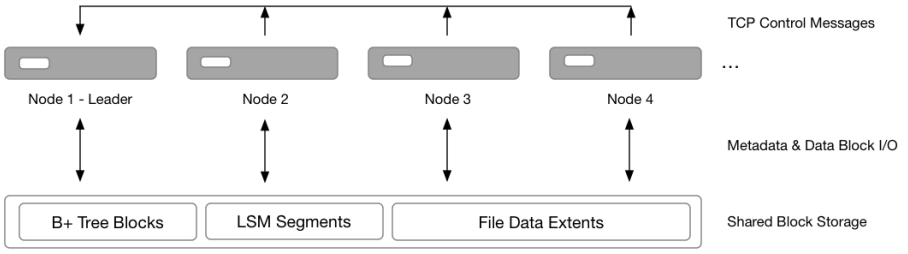
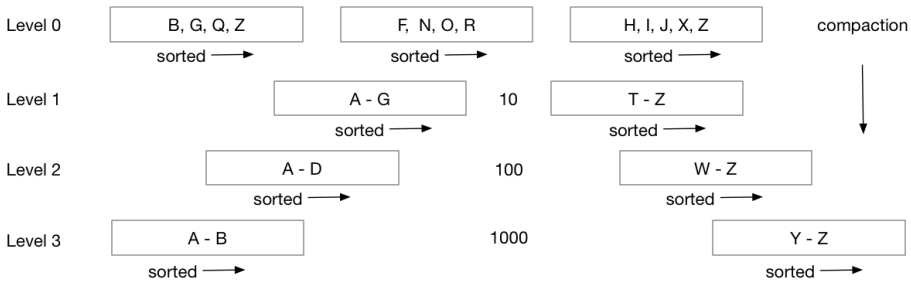


Figure 1: **ScoutFS Cluster**



Fixed size (1MB) segments contain sorted sets of keys
All segments are on-disk, and are immutable once on-disk
Level 0 segments can have overlapping key range
Level 0 has irregular number of segments
Level 1+ segments do not have overlapping key range
10x number of segments in each increasing level

Figure 2: **ScoutFS LSM Structure**