ScoutFS Whitepaper

Table of Contents

Abstract 2

Introduction 2

Design Elements 4

Conclusion 8

About Versity 8

Abstract

The Scale Out File System (ScoutFS) is a scalable shared block clustered file system designed and developed to support the unique requirements of modern mass storage workloads. 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 make the most of the optimized Linux software ecosystem and is built to deliver the most value from economical commodity hardware devices.

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 data storage resources. The namespace capacity target of 1 trillion files addresses one of the urgent requirements in the file-based mass storage market where unstructured 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 filesystem which resolves the major performance bottlenecks associated with mass storage systems.

Introduction

Versity designed and implemented the open source ScoutFS File System to meet the growing need for dramatically enhanced scalability of mass storage platforms. Metadata management and namespace capacity limitations have hampered the ability to scale up cost efficient storage systems at a time when exabyte-scale archives are becoming more common.

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

ScoutFS shares similar design goals with previous shared 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 data storage deployments. Mass storage and archive specific workload characteristics led us to a unique set of design goals that does not overlap with existing file system solutions. 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 extremely high file creation rates. The only way to manage hundreds of millions of files efficiently 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 ScoutFS design goal was not to produce another general-purpose file system.

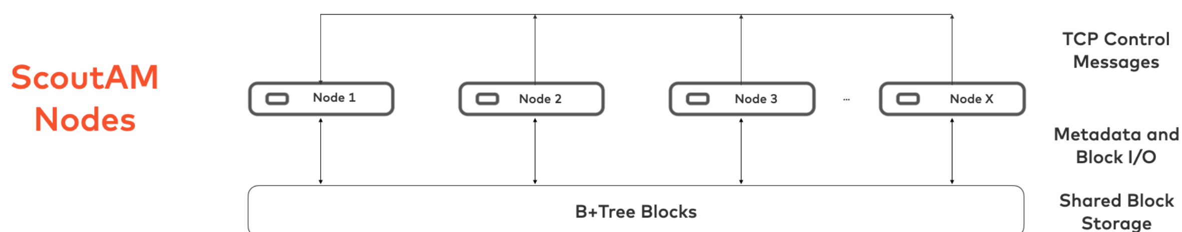
Finally, an open source GPL mass storage file system is an inherently safer and more user friendly long term solution for storing archival data where accessibility over very large time scales is a key consideration. Placing archival data in proprietary file systems is costly and subjects the owner of the data to many vendor specific risks including discontinuation of the product. As in other technology verticals, it is likely that open-source mass storage file system software will come to dominate the landscape.

Design Elements

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 storage devices. Larger sites can utilize enterprise SAN or HPC SAN devices that support dozens of nodes.

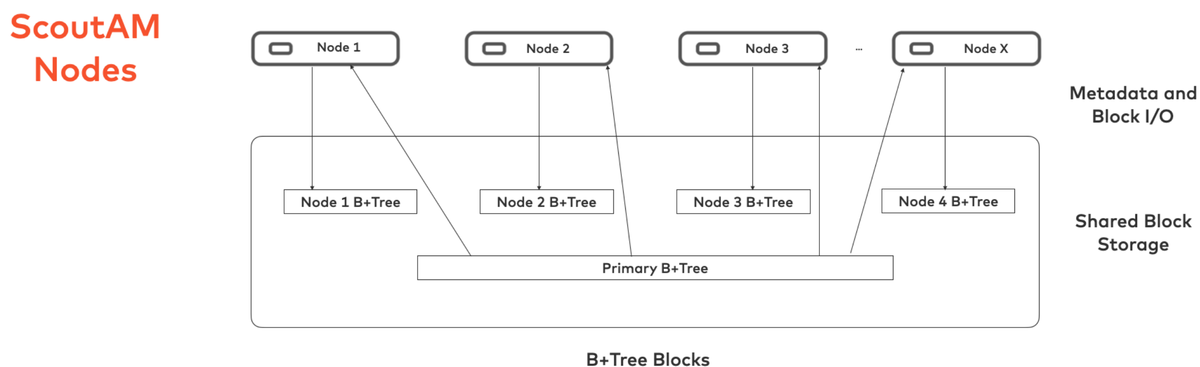
ScoutAM Nodes



B+tree Instances and Structure

ScoutFS metadata is stored in a mix of primary and individual B+trees.

ScoutAM Nodes



B+Tree Blocks

The heart of a ScoutFS volume is a primary B+tree that describes persistent data 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 the total metadata device capacity.

Each node within the ScoutFS cluster has a unique B+tree, which enables fast, local, updates. These structures can contain information about updates to data from individual nodes. When certain conditions are met, such as the size of the tree, these individual B+trees will be merged back into the primary tree. The node will then start a new B+tree and the cycle will begin again. There are individual B+trees per mount, so there can be more than one per ScoutFS node.

Metadata 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. For each write there is an update to the local B+tree structure.

Metadata Reads

Nodes read POSIX file system metadata from the primary B+tree first. 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 tree by directly reading blocks from the shared device.

To ensure that the information is current, the individual node B+trees must be checked for updated information as well. An item cache is used for fast lookups.

Bloom filters are implemented for each tree on each node so that the full tree does not have to be traversed, which makes the overall check for whether there is relevant information on the node very fast. The bloom filter is based on the hash of the metadata and is set/updated at write time.

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 and transfers extents between nodes and the global extent allocator.

The leader is dynamically elected from amongst all the nodes that currently have the volume mounted. If the leader crashes or leaves the cluster, a new 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 for example, an allocation request will result in a bulk grant 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.

POSIX Coherence Locking

ScoutFS uses an optimized lock manager protocol to ensure safe access 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.

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 data structures. Readers of the secondary index are relatively infrequent in our design target workload where background orchestration agents are applying a policy. Concurrent secondary index writing allows ScoutFS to greatly accelerate concurrent updates of the secondary indexes that in turn accelerate orchestration agents.

Mass Storage 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 orchestration agents within the mass storage platform to orchestrate file reading and writing on external storage resources such as on-premises cloud devices, public cloud services, and tape libraries.

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 orchestration agent to ensure that 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 storage media. An interface call frees blocks and marks extents offline. Another interface writes contents to existing extents and marks them online. The orchestration agent marks files offline once they are safely present in the external storage pool so that space on the block device can be reclaimed. The agent writes file contents when it receives notification that processes are trying to read from offline regions of files. The motion of file contents between the block device and the mass storage devices is transparent to users of the filesystem. Whether a file's extents are online or offline can be observed through specific management interface calls.

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. 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. 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.

VERSITY.COM | © 2022 Versity Software Inc.

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, 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 accessed exactly the block it was seeking, not a block from another volume, or a previous version of the block, or a similar version of the block from a different location.

Reliable atomic metadata updates are further leveraged to greatly improve the process of recovering from a node failure. ScoutFS uses cluster majority 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.

Conclusion

ScoutFS employs a small set of precise tools - concurrent CoW B+trees and range locking - to provide a coherent POSIX file system that enables a scalable mass storage system.

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.

For more information about ScoutFS or to arrange a briefing, please contact us at info@versity.com

About Versity

Versity is an independent, software-defined mass storage company focused on rapid innovation and long-term growth. We build scalable, modular and efficient exabyte-scale data storage solutions and aggressively invest in new technology. Our customer-friendly business model, exclusive focus on mass storage and highly rated customer support set us apart from the legacy storage vendors.