Design and Implementation of a Storage Repository Using Commonality Factoring

Dr. Jim Hamilton
Avamar Technologies
jhamilton@avamar.com

Eric W. Olsen
Avamar Technologies
ewo@avamar.com

Abstract

In this paper, we discuss the design of a data normalization system that we term commonality factoring. A real-world implementation of a storage system based upon data normalization requires design of the data normalization itself, of the storage repository for the data, and of the protocols to be used between applications performing data normalization and the server software of the repository. Each of these areas is discussed and potential applications are presented. Building on research begun in 1999, Avamar Technologies has implemented an initial application of this technology to provide a nearline, disk-based system for backup of primary storage.

1. Introduction

The twin problems of rapidly increasing demands for storage capacity and the management of the resultant storage repositories have been extensively discussed. In addition to these well-known issues, a sharp growth profile has been noted in the demand for on-line or near-line storage of static digital content. This demand is fueled by the conversion of vast amounts of data from an analog to a digital format, including document archives and multimedia, as well as by huge quantities of static digital data from geophysical sources, medical and genome research, sensor data, etc.

Despite the dramatic improvements in primary storage technology, bit-for-bit storage of all the desired digital data onto primary storage is not cost effective. Even if economically feasible, such an approach would stretch our current abilities to manage such vast data stores. Accordingly, hierarchical techniques are used to supplement primary storage, most commonly the archive of data and backups to relatively less expensive media such as tape. Such solutions, however, are themselves expensive and error-prone, and in addition add to the complexity of storage management. There is a major loss-of-opportunity cost associated with these solutions, since they limit the user access to data. For example, if larger amounts of oil exploration or earth sensor data were immediately available to researchers, data analysis could be significantly more complete.

There are at least four primary strategies for increasing the amount of data online. One can simply ride the crest of ever more dense storage, with the reducing cost/bit, and attempt to meet the demand with very large primary storage. To be realistic, such an approach must consider cost and storage management issues. In addition, backup of the primary store becomes a major consideration. The cost of the backup solution may rival the cost of the primary store once administrative costs are considered. For many enterprises, the network bandwidth needed for backup of large primary store repositories has led to the need for separate backup networks.

A second approach to increasing online content is to store the data in compressed form, maximizing the information density of the primary store. Compression is a valuable tool in the arsenal of IT systems designers and application developers. While compression involves a trade between computation/latency and storage usage, relatively lightweight "real-time" compression algorithms have had great success for specific data types. However, there are factors working against compression as a universal panacea for storage. Compression algorithms are typically closely associated with data type. While there are exceptions, compression is typically dealt with at the application level, rather than being offered transparently as a storage service. High compression ratios are achieved at the expense of significant computation (and corresponding latency). Many compression algorithms, in order to have reasonable performance, use in-memory techniques that do not work well for very large datasets. Lossy compression algorithms are not always acceptable. Finally, the data reduction achievable from compression is typically limited to an order of magnitude or so, even for the data to which it is applicable.

A third approach to increasing online content is to create a hierarchical storage architecture where the successively lower cost/higher latency storage tiers are virtualized, presenting a seamless global data repository to the user. To date, hierarchical storage has not lived up to its promise, primarily because of complexity, management, and usability issues. However, a number of new approaches are coming to market.

A fourth approach for online or nearline storage is some form of data normalization. By data normalization we mean finding subsets of the data that are replicated (across time or within and across datasets), and reducing or eliminated the redundancy. Approaches to data normalization and applications of the technique can be quite diverse. In addition to database normalization, we are all familiar with systems that store differences across time, such as change management systems and the familiar "full vs. incremental" approach to tape backup. Vendors now offer solutions that store single instances of files. Filesystem snapshot technologies and backup systems that factor out redundant fixed data blocks are other examples of data normalization.

Unlike compression algorithms, data normalization techniques need not be data type specific. While the level of data reduction achievable is obviously data dependent, the upper bound for this value is not limited mathematically, as is the case with compression techniques. If every employee of a large corporation has a copy of the employee stock plan, storing pointers instead of replicated data could have an effective data reduction ration of 100,000 to 1.

In achieving benefit from data normalization, many issues must be addressed. Some sort of indexing system or pointer scheme is required. The indexing system itself is subject to concerns regarding scalability, performance, availability, and fault tolerance. The algorithms for identifying common data, factoring commonality, and re-integration of data must exhibit acceptable performance and reliability. With the elimination of redundancy, fault tolerance for the normalized data representation becomes particularly important. If the application involves storage over time, it may be necessary to provide some form of deletion and storage reclamation. Since data elements are shared by potentially unrelated users or applications, reliable and correct deletion is becomes a significant design consideration.

2. Elements of Commonality Factoring Design

Design of a data normalization system may be partitioned into at least two subject areas. First, the basic units of data that are candidates for redundancy comparisons must be defined. Once the approach to redundancy is defined, companion mechanisms for indexing and storage are needed. This section discusses the first of these topics.

We define the term "Commonality Factoring System" (CFS) to mean a system that defines and computes atomic units of data, providing a mechanism for data normalization. The atomic units of data themselves will be simply termed atomics.

Design of a commonality factoring system involves a number of trade offs:

- Algorithms can be tuned for specific data type(s), taking advantage of patterns unique to that data. The advantages gained must be traded against the increased complexity of multiple algorithms and the loss of potential commonality across data types
- The level of data abstraction to which commonality factoring is to be applied. For example, file vs. data (block or bit) layer. Whether to design commonality factoring at a single abstraction level, or at multiple layers
- For data layer algorithms, trades must be made for fixed vs. variable size atomics. Fixed size atomics simplify atomic identification and indexing/storage, but often with significantly reduced commonality when compared to variable size approaches
- Atomic sizes and (in the case of variable size atomics) atomic size distribution have their own considerations. Smaller atomics increase commonality (a one-bit atomic has wonderful commonality…), but at added processing expense and increased demand upon the indexing and storage system. For variable size atomics, the algorithm needs to produce a well-behaved distribution of atomic sizes
- Atomics must have an associated identifier. For large repositories, this is a non-trivial issue. Identifiers must be small compared the data itself, yet unique to a very high probability (on the order of 1-10**-20 or better for most applications). At the same time, the computational burden and latency in calculating identifiers and in comparing identifiers must be acceptable

Avamar has designed and implemented a CFS supporting both fixed and variable sized atomics. Typically, fixed size atomics are used for applications such as databases where the application is implemented around a concept of fixed blocksize. For general filesystems or data sets, a variable atomic size algorithm is employed.

To partition a data stream into variable sized atomics, we have designed and implemented an algorithm that, for a given input stream, consistently factors that input stream into the same sequence of atomics (see Figure 1. below). When the algorithm is presented with a slightly different input stream (as when a file is modified), it will identify the same atomics up to a point of difference and then resynchronize very quickly following the difference. This client-side algorithm is central to the overall implementation of our CFS.

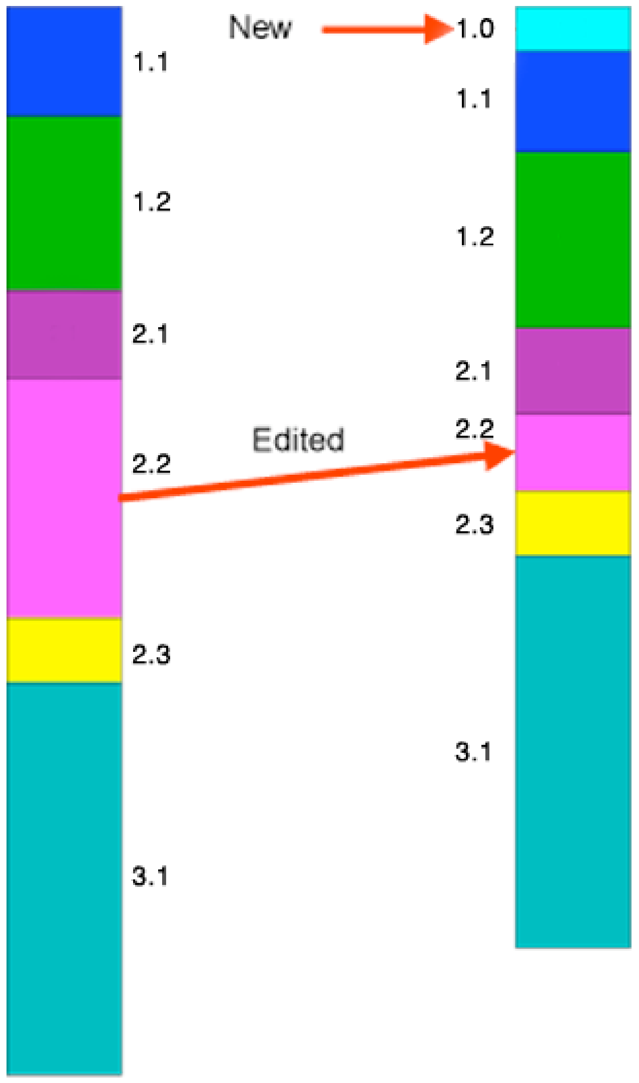

Figure 1. Partitioning a Data Stream

A CFS needs identifiers for its atomics. Ultimately, much of the activity of a CFS will be in comparison of atomics. Except for specialized applications where custom hardware may be used, it is generally infeasible to perform direct bit comparisons of atomics. Identifiers also enable queries and comparisons to be done by reference, with large savings in computation, communication and storage. It is no understatement to say that the properties and utility of a CFS are determined as much by the approach to identifiers as by the selection of atomics.

The CFS design is not complete until mechanisms are defined for creation of composite structures and storage of metadata. These topics are coupled with both the (client-side) CFS implementation and the (server-side) storage repository for atomics. Data structures are needed to provide the "recipe" for the reconstruction of data from atomics, and this information must be stored with the data. In the case of a file-level CFS, data structures must

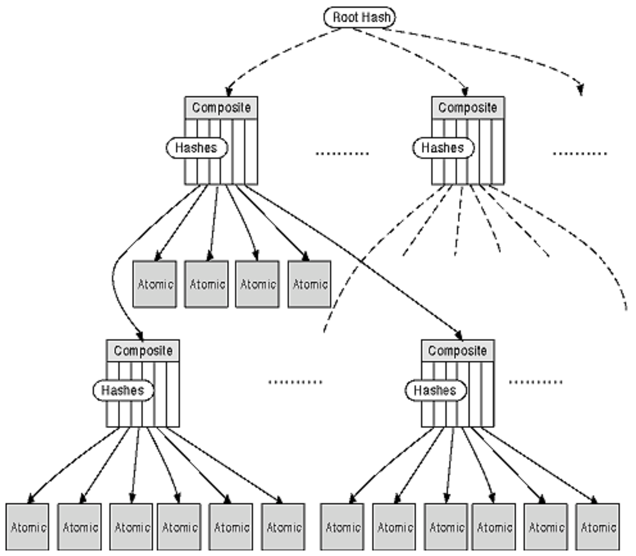

Figure 2. CFS Object Tree Structure

accommodate reconstruction of files from atomics, storage of directory structures, and reconstruction of filesystems. In a similar manner to the familiar block-level inode structures used in filesystems, Avamar has designed the data structures for its CFS to accommodate both atomic and composite data in a hierarchical structure we term the "hash file system (HFS)". This tree-like structure (see Figure 2. above) allows the atomic representation of an entire filesystem to be "rolled up" into a single identifier, termed the root hash. In a recursive fashion, interpretation of the root hash permits reconstruction of arbitrarily large filesystems. In our initial application of CFS to filesystem backup, the 160-bit root hash for a backed up filesystem image may represent a full backup image of a multi-terabyte filesystem.

Similarly, metadata must be stored and reconstructed. For filesystem applications, metadata is filesystem dependent and typically consists of ownerships, permissions, access controls, and other information such as encryption or compression.

To complement its CFS design, Avamar has developed a unique storage architecture. The first application of the resultant system provides a solution for disk-based backup and for nearline storage of static content data.

3. Design of a Storage Architecture Utilizing a CFS

Avamar's CFS design incorporates the needed elements for modern filesystems, including composite structures for storage of large files and hierarchical directory structures, and mechanisms for storage and reconstruction of metadata. A storage repository for the CFS data was needed. Avamar's design goals for the storage repository included the following:

- A single repository should be scalable in size to at least 100 exabytes
- The repository should be distributed
- It should be possible to start with a modest sized repository and to scale gracefully to very large data stores without issues of data migration or data management
- The repository should present a single system image and be managed as such
- The repository should have reliability, availability, maintainability, and fault tolerance features appropriate for nearline store of mission critical data
- Read, write, and query performance should meet or exceed nearline storage application needs
- Performance should be independent of repository size
- To be cost effective, the repository overhead associated with data storage must be low, without impacting performance

To meet these stringent goals, Avamar has designed and implemented a distributed storage repository (see Figure 3. below). The repository is implemented as a server with a scalable number of computational nodes: there are no practical limits to the number of nodes that

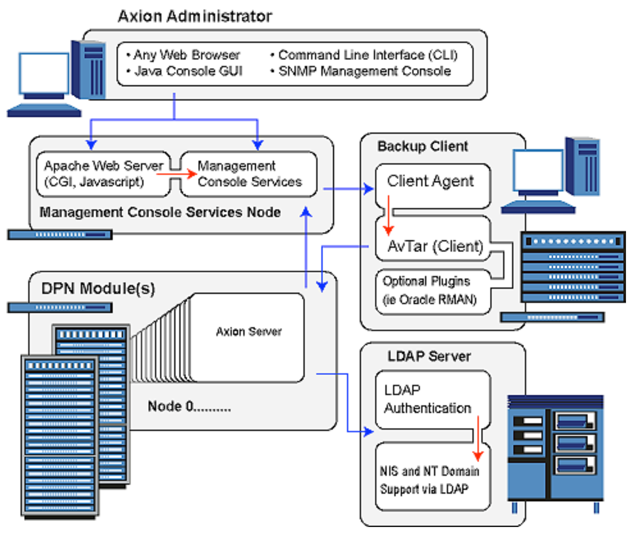
Axion Administrator
• Any Web Browser
• Java Console GUI
• Command Line Interface (CLI)
• SNMP Management Console
Apache Web Server (CGI, Javascript)
Management Console Services
Management Console Services Node
Backup Client
Client Agent
AvTar (Client)
Optional Plugins (ie Oracle RMAN)
DPN Module(s)
Axion Server
Node 0..........
LDAP Server
LDAP Authentication
NIS and NT Domain Support via LDAP

Figure 3. Axion System Architecture

may be configured, and additional nodes may be added dynamically. Each node is running identical software – there is no central controller with the associated fault tolerance and performance issues. Nodes themselves have a uniform and consistent view of the repository as a whole. This is achieved with a loosely synchronized distributed architecture based on message passing and a few small global data structures. Two major benefits of this approach are that clients may contact any server node with requests, and the system is managed as a single image. The Avamar distributed server provides a single global pool of "data stripes". Stripes are logical fault-tolerant storage units which are mapped to a set of distributed files similar to a RAID volume. A stripe ID and offset are used to determine the location of any data element.

The Avamar storage repository has the following features:

- Use of 160 bit hashes [1] as identifiers for both atomics and composites
- Content addressability using these hashes
- An innovative addressing scheme with dynamically variable stripe/offset masking
- Virtualized global stripe pool across a potentially very large distributed server farm

The combination of these elements results in a storage architecture that scales to petabytes while maintaining performance, with bit-level error probabilities below those of hardware raid controllers. When combined with CFS, the result is a scalable repository with excellent commonality factoring, making large-scale, long-term nearline storage of data on hard disk useful and cost effective.

Ability to delete data from a shared repository is desirable both to support data retention policies and to reclaim disk space. In a large-scale shared repository with a CFS implementation, atomic data may be shared by applications knowing nothing of one another. These clients of the repository may have differing retention policies. It is up to the server to ensure that a variety of data retention policies may be supported, without requiring application coordination. Storage repository architectures that leave knowledge of data on the client side increase application complexity and are unable to fully manage the repository. Avamar has solved this difficult issue by providing an automated "garbage collection" capability which is able to identify and remove data which is no longer referenced. Applications may independently manage their retention policies with confidence that data integrity will be maintained.

Repositories for mission-critical data must have robust fault-tolerance. Avamar's approach to fault-tolerance is three-fold:

- A generalized approach to stripe parity and recovery from physical media failures
- A repository checking utility similar to familiar file system integrity checking
- Automated checkpointing with rollback capabilities.

4. Applications of the Technology

In this section we discuss the applications of a CFS-based storage repository. Avamar's initial product offering based upon this technology is Axion. Axion provides a solution for disk-based backup and for nearline storage. Development of Axion has involved supplementing the core technology with clients for specific platforms and applications, integrating an account management system, and providing a management console for backup. The management console provides a single point of administration for the system, enabling backup policies and schedules to be defined and clients to be managed.

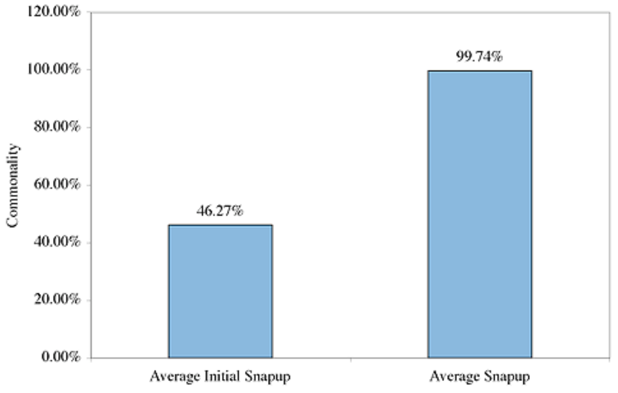

Figure 4. Typical System Commonality

Initial experience with Axion has been positive. For backup applications, Axion's underlying technology means that each system backup appears as a full system image, while consuming less bandwidth and storage than traditional incrementals: to distinguish the Avamar approach, these images are termed "snapups". The resultant system simplifies traditional backup administration in several ways:

- No need to define and manage a hierarchy of incremental and full backups
- No need for tape management and for managing "offline" vs. "online" backups
- Single system interface and single repository image for all enterprise backups
- Online access to all backup data

In addition, CFS eliminates redundancy in data, greatly reducing the cost of disk storage.

Referring to Figure 4. above, it is apparent that both initial and day-to-day commonality is dramatic in the Microsoft Windows environment. (Unix commonality is similarly dramatic.) These results, which are typical for nearly every site tested thus far, lead to striking reductions in bandwidth and space required for snapups. Those sites that have not achieved this degree of commonality are (predictably) those that have a significant influx of unique data such as satellite imagery or other physical phenomena. However, even these sites have excellent day-to-day commonality for existing data.

Having archived data available on disk has other benefits. Much larger amounts of data may be kept in near-line storage. This makes such data accessible to users and applications without the inefficiencies and latency inherent in tape systems. Unlike other approaches to fixed content store, Axion provides a global index, and manages data aging and deletion policies without the need to build logic into the data applications.

Other areas such as data communications can benefit from application of these technologies.

5. Related work

Others have discussed storage systems with some similar characteristics, including Venti [2], SFSRO [3], Centera [4], and OceanStore [5].

References

[1] FIPS 180-1. Secure Hash Standard. U.S. Department of Commerce/NIST, National Technical Information Service, Springfield, VA, April 1995.

[2] Sean Quinlan and Sean Dorward, "Venti: a new approach to archival storage," USENIX File and Storage Techniques (FAST), 2002.

[3] Kevin Fu, M. Frans Kaashoek, and David Mazières, "Fast and secure read-only file system," 4th USENIX Symposium on Operating Systems Design and Implementation (OSDI), 2000.

[4] EMC Centera Product Description Guide, EMC Corporation, 2002.

[5] John Kubiatowicz, David Bindel, Yan Chen, Steven Czerwinski, Patrick Eaton, Dennis Geels, Ramakrishna Gummadi, Sean Rhea, Hakim Weatherspoon, Westley Weimer, Chris Wells, and Ben Zhao. "OceanStore: An Architecture for Global-Scale Persistent Storage," Proceedings of the Ninth international Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS 2000), November 2000.