Venti FS: The Re-Implementation

Daniel Diemer '18 and Dalton Varney '18
The Claremont Colleges

Abstract

The Venti file system was designed as an archival storage system to optimize data storage with hash-based indexing and de-duplication techniques. Using Fuse, we implemented a "lite" version of the Venti file system to experiment with SHA-1 hash-based file indexing. For the original paper see "Venti: a new approach to archival storage"[1].

1 Introduction

The Venti file system interested us due to its use of the SHA-1 hash function to locate, store, and identify data. Compared to the FAT file system, which requires searching through directory entries to find the proper location of files, Venti allows very fast access using unique fingerprints (hashes). Upon a write, the data gets hashed with the SHA-1 hash function and gets mapped to a specific block or blocks through an index table. Upon retrieval, the data can be found by using the same hash and retrieved by searching the index table as opposed to searching through the file system directory entries. In addition, Venti offered many attractive features such as arenas, which allow for multiple data blocks to be allocated together and sealed once they are fully written since Venti is an archival file system. Our implementation uses certain aspects of Venti, while others were not implemented due to time constraints or lack of relevance for our final project.

2 Our Design

We adjusted the Venti design slightly in our implementation to make certain features easier to develop. For instance, we used a free list similar to the FAT file system where each block we allocate is marked in a table to know it has been used. Since Venti is an append-only file system[1], when a block is allocated and data is written, the block can never be written to again. While this append-only feature wasn't necessary to our design, we implemented it because the advantages gained outweighed the disadvantages. Block allocation is much easier without insertion due to the use of the unique fingerprints. Due to the nature of our file system, we decided it would be better to hash the path of a specific file rather than all the contents. We made this design decision because we didn't have much archival data to place in our file system so hashing a specific path allowed us to test the hash and retrieval functionality. Despite this difference in design we decided to follow the same principle as Venti, which doesn't allow insertions into data blocks nor the deletion of data blocks. Insertion would allow fingerprints to become disassociated with a data block and the data could be lost, if a file name or directory were changed.

While implementing our version of a hash-function file system, we came to the realization that certain features would be stretch goals and we would do our best to make a functioning file system rather than try to implement a complete version of Venti. A crucial part of developing a file system is knowing the limits of the implementation so that we could make a working file system rather than a broken one with plenty of features in development.

The focus of our project was on the SHA-1 hash functionality. Since the most crucial aspect of Venti was the use of the SHA-1 function to hash the data contents in order to get impressive data reduction, we made sure the hash performed as expected. The difference in hashing the specific file path versus the data contents was for the purpose of testing the file system. In a future implementation, hashing the data block contents would be a priority in order to achieve the coalesced data storage described in the Venti Paper[1]. Since Venti hashed data block contents, no data block could have the same data as another, otherwise SHA-1 would record the same fingerprint. This is part of how Venti managed to get such im-

Figure 1: The block layout of our hash based file system

pressive data reduction and a key feature we would like to implement given more time. In addition, our design features a free list similar to that of the FAT file system where allocated blocks are recorded as used in a table immediately after the superblock.

The layout of our disk (shown in Figure 1) consists of 4K blocks with a superblock, block table, index-fingerprint map, and then the data log (which consists of blocks and headers).

To properly allocate data blocks without hash collisions, we used the free list as a table similar to that of the FAT file system. Un-allocated blocks have a integer value of -1, while allocated blocks have an integer value of 0. When calling `mkdir`, `mknod`, or `create` the file system searches for the earliest un-allocated block and returns that to the file creation function. In section 4.3 we discuss the advantages of using a FAT-based table for extending files to use multiple data blocks. While we haven't yet implemented this feature completely, we have set up our design to allow for files and directories to use as many data blocks as necessary. Since a crucial aspect of file systems is to store files greater than one block size, we plan to complete this feature, however we prefer a functioning file system with a file size upper bound rather than an entirely non-functioning file system.

3 Implementation

3.1 FAT File System

Our Hash Function File System is built off the code base of the FAT (File Allocation Table) File System. This choice was made because it was much easier to build on top of an existing and functioning code base than to start entirely from scratch. With certain adaptations, we carried over a few principles from the FAT File System, like the superblock and the free list, into our Hash-Function file system. Structurally, the biggest difference from the FAT file system was the addition of the index table. While this led to rewriting a lot of the code we used in our FAT file system, in the end it made our code much cleaner and easier to read. We also hope that these changes will lead to performance improvements in addition to the readability.

3.2 Hash Function

Similar to the Venti file system, we decided to use the SHA-1 hash function. SHA-1 takes an input and creates a 20 byte (160-bit) hash value that we use as a unique identifier called a fingerprint[1]. While there exists other hash-function implementations that can output hash values of 256, 512, and 1600 bits, we went with the 160-bit option because that is the size outputted by SHA-1.

3.3 File Creation

When calling `mkdir`, `mknod`, or `create` (which in turn calls mknod), a series of hashing, index table lookups, and disk seeks occur. The process of making a new directory starts by passing the full path into the SHA-1 hash function and then searching the index table to see if that specific hash already exists. If no results are found, we search the free list for an un-allocated block of data and store that path in the index table with the corresponding fingerprint. When making a directory, we create a list of entries to place in the data block we just allocated. The preliminary entries are "." and "..". If we are making a new file, we simply place the file's contents into the data block allocated and write it to disk. To properly display the file or directory's attributes in the parent directory, we also write a directory entry to the parent directory's data block. This entry will contain the name and the first block location.

3.4 Finding a file

In order to find a file, the path is hashed with SHA-1 and the resulting fingerprint is compared to the entries in the index table. We do a linear lookup by combing through the index table until a fingerprint match is found. Once the match is found, we seek to the file block location and return either the directory entries in the data block through `readdir`, or the actual file contents through a read buffer.

3.5 Challenges

3.5.1 Hash Function

As mentioned in this paper, the hash function is one of the most important parts of the file system, and of course, this is where we encountered our first problem. When trying to access the OpenSSL library on Wilkes, which allows us to use the built-in SHA-1 function, we continually ran into, "library not found" errors. Taking it back to the sandbox of our own machines, we tried to implement a SHA-1 hash function completely separate from our file system. Even on our own machines we still ran into problems. Eventually we discovered that, "Apple has deprecated use of OpenSSL in favor of its own TLS and crypto libraries", which explains the library linking errors we got on our computers.

We decided to implement our own hash function until we could figure out a solution. To keep consistent with the SHA-1 function, our own hash function would output a 20 byte string. We implemented multiple versions of our own hash function; these ranged from picking the first letter of our path name and placing it semi-randomly into a 20-character string to ditching the 20 characters completely and just scrambling the full path of a file to create the fingerprint.

Once we completed the necessary functionality to use the fingerprint index system, we worked on implementing the proper SHA-1 function. This time we stuck with Wilkes and debugged from there. We were stuck on memory allocation errors for awhile that were fixed by correctly linking the "crypto" library from our GCC command.

3.5.2 Data Log & Block Allocation

We encountered issues when designing the relationship between our index table, free list, and our data log. To avoid hash collisions we decided to allocate our blocks consecutively in the data log. While this makes the relationship between our index Table, free list, and our data log quite simple with a linear mapping, it limited us to only being able to write files as one block. While this was not a problem for our prototype, it could theoretically be a problem when other people put our file system into practice. This also means that once a directory is full, there is no way to expand the number of entries in a directory except by removing other files. The solution is to extend the free list table to also include the data block of the next section of a file. This is covered in more detail in section 4.3.

4 Future Work

Due to our limited time frame, we figured certain aspects of Venti would serve as stretch goals if our original goals of implementing the SHA-1 function proved easier than anticipated. Our primary stretch goals in prioritized order:

- Data deduplication with hash function
- Variable block sizes
- Data block combinations for large files
- Index and block cache
- Benchmarking

4.1 Data Deduplication with a Hash Function

Venti creates the fingerprint of a data block based off of the contents of the block. This supports Venti's most useful attribute: coalesced writes. Since the fingerprint is based of the contents of a block, Venti can easily detect the addition of duplicated data based off the fingerprint. If the user tries to write two files with the same data, they will have the same fingerprint and Venti will coalesce this write. This can help increase the capacity of a storage system since it will never have duplicates of the same data. Since we only take the hash of the path name of a file, our file system does not have this data deduplication ability. For this to be implemented we would have to use the contents of a data block available to us during the first steps of a `mkdir, mknod, or readdir` command. Currently, the only thing passed into our hash function is the path name of a file.

4.2 Variable Block Sizes

To follow a "to the books" implementation of Venti, variable sized data blocks are necessary. These types of data blocks allow content in a range of sizes to be stored with one fingerprint as opposed to splitting the data into fixed sized blocks and assigning a fingerprint to each one. Since Venti is an archival storage system, fixed blocks would suffice for the purposes of deduplication. In fact, using smaller, fixed-size data blocks would allow for more chunks of data to be coalesced using the fingerprint system. However, the benefit of variable data blocks would most likely outweigh the rare additional coalesced writes from smaller data blocks. Given more time, our first step would be to look into implementing variable sized data blocks based on the size of the files being written.

4.3 Data Block Combinations

Venti uses "arenas" to achieve data block combinations[1]. Each arena contains a header, a certain amount of data blocks, and a trailer. These arenas are used to cluster related data, whether it's part of one file or a collection of related files. Our implementation includes the header feature, yet does not allow multiple data blocks to be combined into one consecutive section. We considered implementing a feature similar to that of the FAT file system, which records files that extend multiple data blocks in the FAT table. However, since Venti does not use this technique, datablock combinations are the best solution. Arenas keep sections of archival data together with one header and trailer. Once all of the data in a section is written, an

arena can be sealed permanently, never to be added to again.

4.4 Index and Block Cache

A crucial aspect of Venti that resulted in fast access times independent of block location was the index cache. The index cache allowed frequently or recently used fingerprints and their respective block locations to be saved for rapid access. Since we use linear lookup for data blocks in the index table, a index cache could serve to speed up access times dramatically. A hash table would've been a better implementation for the lookup. If the block location were saved for frequently used files and directories, a reduced lookup time could be achieved.

4.5 Benchmarking

Since we built our file system based off of the structure of FAT, we would like to see if our hash function and index table show any performance increase from the basic FAT system. We tried a simple benchmark using epoch time in order to find the speed of `write`, `read`, `mkdir`, and `readdir`. Since epoch time is measured in seconds, we concluded that all of our functions ran in less than a second. We then measured the functions with `gettimeofday` and it recorded 0.0000 as the runtime for `mkdir` and `readdir`. We would expect to see increased speeds compared to FAT because FAT has to search through the file system directory entries while our file system just needs to do a linear search of the index table to locate a block. However, we were not able to compare the function speeds of FAT and our Venti implementation.

5 What We Would Do Differently

5.1 Mistakes

While developing our adjusted version of Venti we made multiple mistakes along the way that allowed us to learn some important lessons. There are certain computer science practices that we were reminded of in our development such as the avoidance of magic numbers, proper memory allocation, and descriptive variable names. The biggest mistake we made was not properly investigating the linking of the openssl library "ssl" as well as the library "crypto". This lead to our greatest mistake of searching for a memory leak when there in fact was none; our libraries were not linked properly and thus produced memory errors.

In addition, we were reminded about the importance of avoiding magic numbers by our intensely confused selves while trying to figure out the reason we set certain file name character arrays to "56".

In CS70, great importance was placed on descriptive variable names and we took that to heart in this project by making our variable names as helpful as possible to allow the code to be its own version of documentation, thus needing few comments throughout.

One mistake that we made during our FAT file system implementation was doing a lot of copying and pasting of code. We realized this during our FAT implementation and set up helper functions, but we still reused code in lots of places. Since our Venti implementation was based off our FAT file system, certain changes were more challenging to make because of this legacy code. We would have to comb through our large code base to check every time we reused certain elements. As this became tiresome and tedious, we quickly replaced much of this reuse with helper functions.

5.2 What We Have Learned

This class has taught us plenty of useful computer science development tools, especially when the libraries used lack extensive documentation. Fuse turned out to be an incredible tool which allowed us extensive access to manipulate file system storage methods, but came with little documentation. Professor Kuenning's CS137 documentation on developing with Fuse was the most useful tool we found online. In addition, we learned a lot using GDB debugging methods and stepping through functions to find the root cause of bugs that we otherwise had no hope of fixing. Designing file systems is hard due to the importance of reducing data-critical bugs. We found that the best approach is to move slowly and make sure each line of code is necessary and fulfills its duty in the simplest way possible. Overly complex code often has more bugs and is harder to understand and fix.

5.3 Greatest Surprises

After struggling through implementing our FAT file system, one of the greatest surprises of this project was how far we had come as programmers. Part of what gave us trouble during the FAT assignment is that apart from the brief lecture in class and reading an awful Wikipedia page describing how FAT works, we had almost no experience in coding a file system. What makes this even harder is that, very few people have experience writing file systems. This meant that the computer scientist's best friend, Google, could only be so useful. This meant that most of our problem solving had to come straight from our own heads. Using GDB and countless print statements, we were able to track down most of our problems on our own. GDB was not a tool that either of us

were very familiar with but by the end of this project we had come to love, and sometimes hate, this tool.

5.4 Favorite Aspects

The reason we chose the Venti file system to investigate and try to implement was due to our interest in hash functions. While other file systems papers sparked our interest, Venti showed an impressive ability to reduce data through the use of hash functions that extended passed the typical use of hash tables. We decided to try our hand at implementing Venti because of the innovative idea of de-duplication being done through a 160-bit hash fingerprint. Using the SHA-1 hash function to avoid collisions Venti can place data throughout a file system while using the content itself as the address. This is different than most other file systems which address the data with no correlation to the content being stored.

The fingerprint system serves multiple purposes that show the power of hash functions. The fingerprint can be used to validate the data's integrity. By hashing the data returned from the file system, the content can be verified if both the original fingerprint and the return data's fingerprint match. In addition, the fingerprint can also be used to coalesce writes. Since data blocks are addressed by their fingerprint, if another data block has identical content its fingerprint will match and therefore it will not be replicated in the file system. Instead, only one version of the data will be stored while the file system can *show* multiple copies of the data. This technique lead to impressive data reduction and showed the power of using a hash function to address data blocks.

6 Acknowledgements

We would like to thank Professor Kuenning for his help in this class, as well as Sean Dorward and Sean Quinlan for their development of the coolest file system we have read about. We would also like to applaud them for finding a file systems teammate with the same name. Daniel and Dalton have not yet decided which one will change their name to match the other and it remains a highly debated topic. One potential option is a combination of the two names, for example replacing the first letter of one name with the first letter of the other to make: Dalton and Daniel.

References

[1] QUINLAN, S., AND DORWARD, S. Venti: a new approach to archival storage. *FAST Conference on File and Storage Technologies* (2002).