Adwaith Venkataraman
adwaithv

Cloud File System Hybrid Storage

Introduction

This project involves the design of a hybrid file system spanning the local solid-state drive (SSD) and the cloud storage. The cloud storage used was an emulation of the Amazon S3 cloud service following a "pay as you go" concept. Due to this, the ultimate aim of the file system was to minimize the number of requests/accesses to the cloud. However, the complexity of the code or program should not increase considerably in compensation of a few additional requests. Therefore, the tradeoff between cloud cost and system complexity must be balanced.

The overall file system consists of three layers namely, a base file system layer, followed by a deduplication and compression layer, and finally a caching layer. Among all, the major decisions that were made were the caching policy and maintaining a persistent cache folder. The details of these will be discussed in detail under cache design.

The report consists of goals, design, evaluation and evidence of success of the base layer, deduplication & compression layer, cache layer.

Base Layer

This forms the lowest layer of the file system, with the following goals:

- Size based placement of files on the SSD and cloud;
- Attribute replication to avoid performing small IOs on the cloud.

With respect to the above goals, the following decisions were made:

- Keep files larger than a given threshold on the cloud and those smaller on the SSD;
- Create a proxy (a file with the same name but with a '.' at the beginning to hide it) on the SSD for every file stored on the cloud, as attempting to access the metadata from the cloud also would incur an operation cost. The properties of the proxy would hold the properties of the file on the cloud, but would be of zero size. Therefore, no additional space is utilized.
- Although accessing larger files from the cloud would increase the amount of bytes transferred between the SSD and the cloud, it was implemented in such a way with a view that along with deduplication, it would reduce the cost considerably.
- Secondly, greater number of accesses to smaller files would cumulatively have a greater number of byte transfer as against lesser number of accesses to larger files on the cloud. This conclusion is with a view that when deduplication, compression and caching are implemented, the number of bytes transferred would be comparable.
- When the number of bytes transferred in the two possibilities is comparable, the number of accesses made to the cloud would make the difference. For smaller files, there would be more number of accesses as against lesser number of accesses to larger files.
- The above justification is evident when say, there are 10 files of 10 MB each (case 1) as against 2 files of 50 MB each (case 2). The cost comparisons are shown below:

Case	Cloud Capacity Cost	Operation Cost	Data Transfer Cost	Total Cost
1	(10*10 MB)*$0.095 = **$9.50**	10*$0.01 = **$0.10**	100*$0.12 = **$12.00**	**$21.60**
2	(2*50 MB)*$0.095 = **$9.50**	2*$0.01 = **$0.02**	100*$0.12 = **$12.00**	**$21.52**

- From the above two cases when similar quantities of data are exchanged, a difference of 8 cents are observed.
- When suppose the net quantity of data becomes of the order of gigabytes, the cost difference is magnified and results to about $0.80 for transferring 1 GB; and $8.00 for transferring 10 GB.
- One could argue that with deduplication, compression and caching, the size of data fetched will be smaller than the actual size of the file. But with smaller files, the size of segments will be very small and not feasible to record so many segments as that would increase the size of the hash table considerably as well as increase the number of gets and puts commands to and from the cloud.
- Now, to distinguish between locally stored files and those stored on the cloud, the extended attributes were used. In this particular implementation, every file (including the proxy files) will have an extended attribute set i.e., the file size. But, the files that entirely reside on the SSD would have the attribute value as zero, and the proxy would have the actual size of the file. The actual size of the file is read before transferring the file to the cloud from the lstat return value. The reason for having the file size as the extended attribute is because the lstat

for the proxy would return a zero size i.e., it would indicate the size of the proxy file and not of the actual file that resides on the cloud.
- Hence, it was implemented such that the files that grew beyond a threshold would be sent to the cloud, but in place of which there would be a substitute (proxy) file made in the same name, and existing on the SSD, but would hold attributes corresponding to that in the cloud.

Deduplication and Compression Layer

From the previous section we observed that when large files are to be read off the cloud, the whole file is being fetched to the SSD. However, there can arise a situation where the entire file may not fit on the SSD. Another situation that can arise is when the same content is repeated over several files, there would be duplication of data. In order to make up for this, deduplication is implemented as a layer above the base layer. In other words, all files that become larger than the threshold get broken into several segments before being sent to the cloud. This process is implemented using the Rabin Fingerprinting algorithm. The files are split into segments based on the content. Thereby, when multiple files have similar content, the previously generated segment that represents that same block of content can be used. Hence, duplication among files is prevented. Also, to improve the utilization of the space available on the cloud, the segments are compressed before being sent to the cloud. Therefore, the goals for this layer of the file system are

- Implement deduplication of files;
- Compression of segments.

The following changes were made with respect to deduplication from the base layer of the file system:

- Whenever a file grows larger than the threshold, it will be split into segments using the rabin fingerprinting algorithm. A hash map is used to maintain a reference to these segments generated.
- Since the segments are formed based on content, the md5 hash values generated are unique and can therefore be used as the key. Along with the md5 hash value, the reference count i.e., number of files that have the same segment occurring in them is maintained.
- In order to keep a track of the order in which these segments appear in the original file, the hash value and its length are recorded in the proxy file. The order in which the segments are logged into the proxy file is the order in which the blocks of data appear in the original file.
- Since the files that remain on the SSD are small i.e., in comparison to the threshold, splitting them into segments would not be required, as the whole idea of segmenting the file is to retrieve only a portion of a file from the cloud and thereby reduce the byte transfer cost. But if the entire file exists on the SSD, there is no question of incurring cost.
- In this particular case, the size of the cache considered is the entire SSD i.e., considerably larger than a conventional cache. Therefore any segment read or fetched from the cloud is kept in the SSD and not unlinked.
- When a file that is on the cloud has to be read, the proxy file is checked for the segment hash values and their corresponding segment length. Based on the overall number of bytes required to be read and the offset within the file, the starting segment and the local offset within that segment is identified. Now, beginning from that segment, until the number of bytes yet to be read becomes zero, segments are either read straight away from the SSD (if cached previously) or fetched from the cloud.
- When unlinking a file, the proxy file is checked for the hash entries and their reference count in the hash map is decremented by 1 since now there will 1 less file that will be referencing to it. If the reference count becomes zero, simply remove that entry from the hash map and delete from cloud, as well as remove from the SSD (if cached previously).
- In order to maintain persistency of the hash map of segments, it is saved in a file called '.hash' on the mount point. So, whenever the file system is rebooted, it retrieves the hash table from the SSD from the above-mentioned file.

The changes incorporated for implementing compression were as follows:

- When a file that previously existed in the cloud, or has now grown beyond the threshold size, when segmenting the new contents, the segments are compressed and transferred to the cloud.
- This enables a cumulatively lesser cloud usage in terms of number of bytes stored.
- When retrieving a segment from the cloud, it is read into a temporary buffer, decompressed and read into the proxy file.

Considering there are 300 files, each 10 MB in size. If the threshold were set to 5 MB, the files would have to be sent to the cloud. It is known that files can be compressed from 0.8 to 0.3 times their actual size. Also, when deduplication is applied, assuming that we use only 50% of the overall space, let us consider the difference in costs incurred when implementing deduplication and compression, when trying to read two of those files. For the

scenario, let us consider that we require two files from the cloud which we were initially able to compress to only 80% of the original size and the amount of similar content between the two files are about 1 MB (for ease of calculation). Using a window size of 8192 bytes, we obtain the following costs:

	Cloud Capacity Cost	Operation Cost	Data Transfer Cost	Total Cost
No Deduplication or Compression	(300*10 MB)*$0.095 = **$285.00**	2*$0.01 = **$0.02**	(2*10 MB)*$0.12 = **$2.40**	**$287.42**
Deduplified but no Compression	(150*10 MB)*$0.095 = **$142.50**	(19*128)*$0.01 = **$24.32**	(19 MB)*$0.12 = **$2.28**	**$169.10**
Deduplication and Compression	(150*10 MB)*0.8*$0.095 = **$114.00**	(19*128)*$0.01 = **$24.32**	(19 MB)*0.8*$0.12 = **$1.824**	**$140.144**

From the above results, we can observe that when the number of files is more, the total cost incurred is considerably high when neither deduplication nor compression is applied i.e., with just the base layer file system. However, when deduplication is implemented, we observe that only about half the originally used capacity is being used. Also, the data transfer cost is lesser due to presence of common segments. Therefore, we are able to see that deduplication helps in improving the cost. When compression is present along with deduplication, the capacity used further reduces since every segment stored in the cloud takes up lesser space than originally needed. Also, the transfer cost is lesser as it is based on number of bytes transferred. Therefore, a combination of deduplication as well as compression gives the lowest cost possible.

Cache Layer

The purpose of a cache layer is to improve the performance of the system on the whole by storing the segments fetched from the cloud in the cache, and not evicting one until space is required for another incoming cache. The policy followed for selecting the segment to evict can vary from system to system. With caching, the system is expected to make lesser accesses to the cloud as some of the segments may reside in the local cache. This would result in an improvement in the latency as an SSD access is any day faster than accessing the cloud. Therefor, the overall goals for caching were

- Improve performance;
- Minimize cloud operation costs

But in order to incorporate the concept of caching, the following changes were made in the design:

- Instead of treating the entire SSD as a cache to store the segments, a specific, and persistent hidden folder of restricted size called '.cache' was created and the segments fetched were stored there.
- However, data structures to maintain a count of which segments were cached and which weren't were done.
- The possible means of implementing the count is to either add an additional parameter to the existing hash table or maintain a completely new hash map just for the segments stored in the cache. The final solution implemented was the former.
- Consider a scenario where the ratio of cache size to SSD size is large. In such a case, there would a lot of segments being stored in the cache hash table. Now, when trying to sort that hash table, and subsequently access it to find the segment to evict from the cache would cause an increase in the latency. Also, with an increased cache size, duplication of data i.e., the hash map in this case occurs. But with the addition of a flag to the existing hash structure, one can iterate through it and find out which segments need to be removed. This prevents the latency of sorting the hash map. By this way, at comparable overall access rates, usage of the extra space for the separate hash table was prevented.
- As per the design followed of using a flag to indicate if the segment is cached or not, whenever a new segment was added to the hash map, the 'cached' flag was set to zero before sending to the cloud. And when the segment is pulled to the cache on the SSD from the cloud, the 'cached' flag would be set to 1.
- The focus of this layer was the eviction policy used. When the cache does not have adequate space to accommodate a new incoming segment from the cloud, the file system must be able to evict an already cached segment. In this case, the cached segment with the least reference count was picked. In other words, the segments that are used by the least number of files are removed. Such a policy was followed because the probability that such segments would be called for again would be the least among all other segments. Now, there can arise a possibility that multiple segments can be the least referenced segments i.e., having the same reference count.

- In order to decide between such conflicting segments, the LRU concept i.e., the least recently used policy was used to decide between the two segments. There was a decision that had to be made between the MRU (most recently used) and the LRU. Considering the target usage of this file system where the user has a large number of items that are referenced randomly, or say some files accessed more often than the others, or even accessed in batches i.e., accessed frequently over a short period of time and then accessing it rarely, the LRU would be much more efficient than the MRU.
- Secondly, the LRU never suffers from Belady's anomaly.
- If the end user was say, a student, who would work on a particular project for a while and probably access that after a few weeks or even months, then this would certainly result in better efficiency.
- Now, if there is a conflict even in this i.e., two segments have the least and same reference count, have the same access times (might occur only in a multi user system), then the segment to be evicted is decided based on its size. The larger segment gets evicted off the cache.
- Until the required space is available on the cache, segments are evicted from the cache based on the above policy mentioned.
- Now returning back to compression, another design decision that was performed was whether to compress the segments when keeping it on the cache. This resulted in a tradeoff between access times and cloud access cost. If the segments were compressed on local too, one can store a greater number of segments on the local and thereby improving the cache hit rate and reducing the cloud accesses. But, whenever the segments are to be read from the cache, they would have to be decompressed. This would cause an increased latency when accessing data on the SSD.
- Another major decision that was taken was with respect to the persistency of the cache. The cache with the existing segments will always be maintained on the SSD in order to improve the performance of the file system. The segments will be removed only when another segment is being fetched from the cloud (during eviction). This was done because when the files in the system are removed, the reference count is altered accordingly. But there would still be a few files, which would be referencing to some segments in the cache. So in order to maintain cache persistency and prevent fetching of those segments from the cloud again the next time, the segments are not deleted. Hence, the '.cache' folder will stay undeleted.

Evaluation of the File System

The file system, when tested with different caching policies on the part3 test scripts, namely LRU and MRU, the following results were obtained.

Test Case	Requests to Cloud	Bytes read from Cloud	Capacity usage in Cloud	Reads to SSD	Writes to SSD	Cloud Cost
Least Recently Used + Least Reference Count						
small_test	2603	15961868	4293606	135	311	$28.24
big_test	2213	13794018	4408011	138	351	$24.10
large_test	2400	15011295	4350413	138	318	$26.10
Most Recently Used + Least Reference Count						
small_test	3037	16935120	4440350	21	382	$32.71
big_test	2355	13796812	4335510	133	372	$25.52
large_test	2787	15173723	4161155	137	400	$29.98

From the above test results, we observe that LRU with least reference count gives a more economical result. Also, the number of requests made to the cloud is much lesser and the amount of data read from the cloud is lesser.

The file system was tested for performance with the provided scripts for part 2, with separate layers and also combined. From the results below, we can see that the base layer alone has the highest cost. This is because when neither deduplication nor compression nor caching is implemented, number of bytes read from the cloud and capacity usage are very high. When deduplication and compression is implemented, the bytes read and capacity utilized reduces drastically and so does the cost. However, when caching is implemented, the cache size is restricted as against usage of the entire SSD, so the cloud capacity usage with deduplication for first 2 layers is lesser than when all 3 layers are implemented.

Adwaith Venkataraman
adwaithv

Test Case	Requests to Cloud	Bytes read from Cloud	Capacity usage in Cloud	Reads to SSD	Writes to SSD	Cloud Capacity Usage		Cloud Cost
						Without dedup	With dedup	
Base Layer Test								
small_test	0	0	0	18	7	3194884	3194884	$0.00
big_test	16	524324	1048652	17	10	3194884	3194884	$0.31
large_test	18	4767782	9535568	18	10	3194884	3194884	$1.59
Base Layer + Deduplication and Compression Layer								
small_test	0	0	0	17	7	3194884	1634669	$0.00
big_test	16	342	346	27	9	3194884	1615444	$0.16
large_test	17	352	356	30	10	3194884	1660496	$0.17
Base Layer + Deduplication and Compression Layer + Cache Layer								
small_test	0	0	0	18	7	3194884	1647032	$0.00
big_test	16	342	346	40	14	3194884	1655523	$0.16
large_test	17	352	356	43	15	3194884	1660991	$0.17

The final file system was tested with various combinations of segment size and rabin window size. The variations in the results of the random workload when tested with part3 scripts are shown below. It can be seen that with a larger segment size, the cost reduces. And, with a larger rabin window size, the cost reduces. However, these results are with respect to this file system implementation.

Test Case	Requests to Cloud	Bytes read from Cloud	Capacity usage in Cloud	Reads to SSD	Writes to SSD	Average Segment Size	Rabin Window Size	Cloud Cost
small_test	3037	16935120	4440350	21	382	4096	48	$32.71
big_test	2355	13796812	4335510	133	372			$25.52
large_test	2787	15173723	4161155	137	400			$29.98
small_test	2133	13961585	4579209	130	332	8192	48	$23.34
big_test	2411	15957186	4815774	131	73			$26.37
large_test	2143	14289468	4883931	131	320			$23.50
small_test	2463	14067669	4557314	137	398	4096	96	$26.65
big_test	2596	14772809	4341709	21	405			$28.04
large_test	2433	14071730	4581099	20	387			$26.35
small_test	1932	12976528	4716899	130	311	8192	96	$21.23
big_test	2260	15030286	4844027	132	310			$24.75
large_test	2334	15627616	4777742	129	75			$25.56

After running the file system against all possible combinations of parameters, the following costs were obtained when tested with part 3 scripts. The caching policy used was LRU with least referenced segment.

Test Case	Requests to Cloud	Bytes read from Cloud	Capacity usage in Cloud	Reads to SSD	Writes to SSD	Average Segment Size	Rabin Window Size	Cloud Cost
small_test	2089	14802450	4661392	17	257	8192	96	$23.00
big_test	1947	13603017	4717846	132	273			$21.45
large_test	2266	15853234	4729258	18	266			$24.90

Hence, based on the design of the system and other tradeoff considerations, the most economical figures obtained for the tests based on the part 3 scripts are $23.00 for small test, $21.45 for big test and $24.90 for large test.