Implementing an Append-Only Interface for Semiconductor Storage

Colin W. Reid, Philip A. Bernstein
Microsoft Corporation
Redmond, WA, 98052-6399
{philbe, colinre}@microsoft.com

Abstract

Solid-state disks are currently based on NAND flash and expose a standard disk interface. To accommodate limitations of the medium, solid-state disk implementations avoid rewriting data in place, instead exposing a logical remapping of the physical storage. We present an alternative way to use flash storage, where an "append"İ interface is exposed directly to software. We motivate how an append interface could be used by a database system. We describe bit rot and space reclamation in direct-attached log-structured storage and give details of how to implement the interface in a custom controller. We then show how to make append operations idempotent, including how to create a fuzzy pointer to a page that has not yet been appended (and therefore whose address is not yet known), how to detect holes in the append sequence, and how to scale out read operations.

1 The Case for a Log-Structured Database

Most database systems store data persistently in two representations, a database and a log. These representations are based in large part on the behavior of hard disk drives, namely, that they are much faster at reading and writing pages sequentially than reading and writing them randomly. The database stores related pages contiguously so they can be read quickly to process a query. Writes are written sequentially to the log in the order they execute to enable high transaction rates.

Inexpensive, high-performance, non-volatile semiconductor storage, notably NAND flash, has very different behavior than hard disks. This makes it worthwhile to consider other storage interfaces than that of hard disks and other data representations than the combination of a database and log. Unlike hard disks, semiconductor storage is not much faster at sequential operations than at random operations, so there is little performance benefit from laying data out on contiguous pages. However, semiconductor storage offers enormously more random read and write operations per second per gigabyte (GB) than hard drives. For example, a single 4GB flash chip can perform on the order of 10,000 random 4KB reads per second or 5,000 random writes per second. So, in a shared storage server cluster with adequate I/O bandwidth, 1TB of flash can support several million random reads per second and could process those requests in parallel across physical chips. This compares to about 200 random reads for a terabyte of hard drive capacity. However, per gigabyte, the raw flash chips are more than ten times the price of hard drives.

Bulletin of the IEEE Computer Society Technical Committee on Data Engineering

Given the large number of random reads that are available with flash storage, the main benefit of storing data in two representations disappears. However, if data is to be stored one way, which representation is better, a database that is updated in place or a log? We advocate a log, for several reasons.

Flash has two properties that encourage writing sequentially. The first property is that a flash page cannot be updated in-place. It must be erased before it can be over-written (or "programmed" in flash terminology). In typical NAND flash chips the erase operation wipes dozens or hundreds of pages at a time, and blocks other operations on a large portion of the chip for about 2 ms. Therefore, it is best to erase pages within storage regions that will not be needed for other latency-sensitive operations, such as a random read to service a database cache miss. Second, erase operations cause flash to wear out. Flash can be erased a limited number of times, roughly 10,000 to 100,000 times depending on the technology being used. Thus, flash needs "wear-leveling"İ to ensure all blocks are erased at about the same rate.

The problems of slow erase cycles and wear leveling are solved by the use of log-structured storage. In this design, an update is appended after the last written page of storage. To recycle storage capacity, reachable data on the first non-empty chip C is collected and appended to the updatable end of storage so that C can be erased. This design minimizes the impact of slow erase cycles by ensuring that a chip is erased only when it has no data that is worth reading from it. It also solves the problem of wear leveling because all chips are erased the same number of times.

Another benefit of log-structured storage arises when parallel processors need to synchronize their random updates to storage. The overhead of this synchronization is relatively expensive when the update itself is cheap, as it is with flash memory. Log-structured storage can offer a self-synchronizing, atomic append operation that allows many processors to independently store data at will, with no need for another synchronization process. One problem with log-structured disk-based storage is that it is hard to maintain physical contiguity of related data. This is important because sequential reads are faster than random reads. But this property does not hold for flash. In fact, when using many flash chips as components of a storage system, optimal response times are achieved when related data is spread across multiple chips and retrieved in parallel.

2 The Hyder System

Given these benefits of log-structured storage, we have designed a new database system, called Hyder, which uses an append-only log as its one and only storage medium. It is a data-sharing system, where many servers read from and write to shared storage. Its main feature is that it scales out without partitioning the database or transactions. We briefly sketch Hyder's use of log-structured storage here, as motivation for the mechanisms that support it in the rest of this paper. The architecture is described in detail in [1].

In Hyder, each transaction T executes in the normal way on some server S, reading and writing shared data items. Hyder uses multiversion storage, so T executes against the latest committed database state D known to S, which is defined by the last committed transaction in the log that contributed to D. When T writes a data item, its update is stored in a cache that is private to T. When T has finished computing, S gathers the after-images of T's updates in a log record, L. L also contains a pointer P to the earlier log record that identifies D (the database state that T read). S appends L to the log and broadcasts it to all other servers. The shared log defines a total order of all log records. Thus, all servers have a copy of the log consisting of the same sequence of log records.

When a server S′ receives L, it needs to determine whether T committed. Conceptually speaking, it does this by checking whether there is a log record in between P and L that contains a committed transaction's update that conflicts with T, where the definition of conflict depends on T's isolation level. (This is only conceptual; the actual algorithm works differently and is much more efficient.) If there is no such conflicting committed update, then T has committed, so S′ incorporates T's updates into its cached partial-copy of the last-committed database state. Otherwise, T has aborted, so S′ ignores T's updates . If S′ = S (i.e., S′ is where T originally executed) then it informs T's caller of T's outcome.

Since all servers see the same log, for every transaction T they all make the same decision whether to commit or abort T, without any server-to-server communication. The only point of arbitration between servers is the operation to append a record to the log, which occurs only once per multi-step transaction.

Notice that a server must continually roll forward the log in order to have a cached copy of a recent database state against which it can run newly-submitted transactions. Since log records describe both committed and aborted transactions, it is only through the roll forward process that a server knows which are which and hence knows the content of the last-committed state.

Hyder was partly inspired by work of Dave Patterson and Jim Gray that latency lags bandwidth [4] and bandwidth lags capacity [2]. While physical laws limit latency to light speed, bandwidth can continue to grow, and raw storage capacity is increasingly cheap. So eventually, the capacity cost of append-only storage will decrease to the point of irrelevance, the bandwidth cost of sending information about all transactions to all servers will be less significant, and the latency benefit of a single synchronization per transaction will become more and more obvious. These trends are as relevant to multicore handheld devices as to large clusters, and they can be harnessed at both extremes with a simple abstraction for log-structured storage.

3 A Log-structured Storage Abstraction for Reliable Interconnects

Direct-attached solid-state storage, as in a phone or standalone server, is typically very reliable and can be accessed without message loss. A log-structured storage abstraction for such devices requires only a few trivial operations. The Append operation takes a page of data as an argument, durably appends it to storage, and returns the address at which it was stored. Other operations are GetLastPage (which returns the last page written by an Append), GetSpecificPage (which takes a page-frame address as a parameter), and Erase (which erases all of the page frames within a given storage unit, such as a flash chip).

A small device with direct-attached storage and a reliable interconnect is likely to have only a single storage controller. The controller can track a small amount of state, such as the address of the last appended page, in volatile memory. On power failure, the in-memory state can be consistently flushed to non-volatile storage before the device stops functioning. This greatly simplifies the implementation of the GetLastPage operation, avoiding a more expensive search for the last page during restart.

3.1 Bit Rot

The volatility of semiconductor storage is a continuum, not a binary distinction. “Non-volatile" memories such as NAND flash and phase-change memory do actually degrade over time, and “volatile" memories like DRAM do actually store data without power between their refresh cycles. In all these memories, data needs to be refreshed or rewritten occasionally to prevent it from degrading.

In high availability systems with large amounts of long-lived data, it is usually necessary to scrub storage periodically in order to detect partial failures before they become unrecoverable. In other words, modern semiconductor storage devices periodically read data, and then rewrite it, even if it is static and only rarely accessed by an application. This can be done blindly, by continually scrubbing and refreshing all storage capacity regardless of whether it contains interesting information, or more intelligently, by processing only the storage containing data that is actually useful.

The latter option requires some interaction with higher layers of the system in order to determine which data is still useful. It provides an important opportunity to consolidate, defragment, compress and otherwise optimize the stored representation of important data. It is also an opportunity to free up storage capacity that is no longer useful.

3.2 Space Reclamation

The major downside of append-only storage is that it fills up. Since semiconductor storage capacity is still a fairly valuable resource, a useful system must be able to reclaim space and reuse it to store new information.

The task is very much like garbage collection in memory-managed programming environments like .NET and Java. There are essentially two types of garbage collectors in those types of environments: mark-sweep collectors and copying collectors [3].

In a sense, block storage systems that target disks today are using a form of mark-sweep collection. Applications allocate and release blocks of storage, and a file system or database storage manager marks pages and attempts to allocate blocks of storage on demand from the free pool.

Instead, it is possible to reclaim storage using a copying collector. In this scheme, which has well-understood benefits and tradeoffs in the managed-memory world, reachable data from a sparsely populated region of memory is simply copied densely to an empty region. Append-only storage is well-suited to this approach, particularly if the erase-block size of the device is relatively large, as it is in NAND flash. Copying collection frees large blocks of storage that can be efficiently erased. It also has the advantage of automatically scrubbing and refreshing storage to mitigate bit rot. Reachable information from an old segment of the log is simply re-appended by the application to a new one, and the old segment is then erased and recycled. This requires a small logical-to-physical remapping table of log segments, but is otherwise very straightforward. Such practices are common in log-structured file systems today [5].

This approach can result in unnecessary copying if very large contiguous sections of the log contain long-lived static information. To address that, applications can maintain two logs: one for data that has been copied once, and one for new data. This is similar to the generational garbage collection that is commonly used in managed-memory runtimes.

4 Log-structured Shared Storage over Unreliable Interconnects

The same four Append, GetLastPage, GetSpecificPage, and Erase operations can support log-structured storage in locally distributed compute clusters. As long as the storage devices are able to continue serving their stored pages reliably, transient errors such as lost messages can be accommodated by slightly augmenting Append.

4.1 Making Append Idempotent

The Append operation stores a page of data and returns the address at which it was stored. By not specifying the address at which to write the page, servers do not have to coordinate the regions of storage that they intend to modify. This "just-in-time synchronization" is useful in any parallel environment, regardless of the interconnect protocol's reliability, and the storage unit thus provides a dense log of append operations.

A problem arises if a caller (i.e., a server) sends an Append operation to a controller and does not receive a reply, either because the operation did not execute or because the reply message got lost. In the former case, the caller should re-execute the Append. In the latter case, the caller needs to get another copy of the reply. The problem is that it does not know which to do. It needs help from the storage controller to take the appropriate action, such as to get another copy of the reply.

To solve this problem, we design Append to be idempotent, which means that executing an Append operation multiple times has the same effect as executing it just once. Each page that is sent to the controller has a unique key. After the controller processes an Append and before it returns the appended page's address to the caller, it stores the page's key and storage address in a table that is local to the controller. We call it the Append Result table. The controller uses the Append Result table to cope with duplicate Append operations. When the controller receives an Append operation, it first checks to see whether the key of the page to be appended is in its Append Result table. If so, it replies with the address in the Append Result table and does not re-execute

the Append. If not, then it executes the Append operation, which is safe to do since it has not executed the operation previously. Therefore, if a server does not receive a reply to an Append, it is safe for it to resend the same Append operation to the controller.

The key must be generated in such a way that there is negligible probability that two different pages that are received by the controller within a certain time period have the same key. The time period is the amount of time it takes to execute N Append operations, where N is the number of entries in the Append Result table. This ensures that the controller will not be fooled into thinking that two different pages are the same because they have the same key. For example, the key could be a 128-byte globally-unique identifier, commonly known as a GUID, a cryptographic hash of the appended state, or a sequence number. Or, if the storage controller caches payloads of recently appended pages anyway, the key could even be the entire content of the page.

Since Append is idempotent, a server can increase the likelihood that the operation will execute successfully by sending it multiple times. For example, if there are two communication links that connect the server to the storage controller, then the server can eagerly send the Append on both links. This has less latency than retrying after a timeout, since the server can proceed based on the reply from whichever operation executes first. It also decreases the chances that the server will fail to receive a reply, since both Append operations would have to malfunction for that to occur.

A problem with the above technique for making Append idempotent is that the Append Result table has a fixed size. New entries in the Append Result table need to overwrite the oldest ones. Therefore, if a server resends an Append a long time after sending its initial Append, then it is possible that the result of the first Append was overwritten in the Append Result table before the storage controller receives the second Append. In this case, the controller would incorrectly conclude that the second Append did not previously execute and it would execute the operation again.

4.2 Bounding the Storage Address

We solve this problem by including a “concurrency bound" parameter to the Append operation. The concurrency bound defines a range of pages where the page must be appended. If the first page in the range is not in the Append Result table, then the operation returns an exception, because it cannot tell whether the Append is a duplicate. The operation also returns an exception if there are no erased pages in this range, which means that all of the requested page frames are full and cannot be written by the Append. Since storage is append-only, the controller can implement this simply by comparing the address of the last page frame it wrote (which is in the Append Result table) to the top of the concurrency bound.

A good way express the concurrency bound is as a page-frame address and an extent. The page-frame address is the next page frame address beyond the last written page frame seen by the caller (e.g., in the reply to a previous Append). The extent is the number of page frames in the concurrency bound.

A concurrency bound can be used to store a soft pointer to a page whose address is not yet known. For example, consider a more reliable distributed storage architecture where a multi-page stripe $\{P_1, , P_n\}$ needs to be stored atomically on several log-structured storage devices. One way to do this is, after the pages have been appended, to store a commit recordİ C that contains pointers to those pages. For recovery purposes, it is helpful to have a pointer from each P_i to C to make it easy to determine whether the set of pages that included P_i was committed and which other pages are part of its atomically-written set. However, since C's location can only be known after it is appended to the store, its location cannot be known before P_i is written. Instead, a concurrency bound for C can be stored in each P_i, which limits the search required to find C.

4.3 Broadcast

In some applications of log-structured storage, it is useful to have Append operations and their results be made known to all servers. This is important in Hyder, where all servers need to know the content of the tail of the

log. It may be useful in other applications that need to maintain coherent server caches.

A server can easily make its Appends known to other servers by broadcasting the Append to other servers at the same time it sends the Append to the storage controller. The other servers also need to know the storage address where the appended page is applied. That information is only known after the controller executes the Append and returns the address to the caller. The caller could broadcast that result to the same servers to which it sent the Append. Although this works, it is inefficient because it introduces extra delay, since the server must wait until it receives the result from the controller and then send the address to other servers.

A better solution is to have the controller broadcast the result of the Append to all servers. This avoids the extra message delay of first sending the result to the server that issued the Append and then having the server broadcast it to other servers. It is especially efficient if the network that connects the servers to the shared storage controller has a built-in broadcast capability.

Broadcast is realistic over common connectionless protocols like UDP, because a 4KB flash page fits comfortably in an Ethernet jumbo frame. (However, this is not true across public networks where the maximum transfer unit is typically 1.4KB.) Since pages are totally ordered by their storage addresses, we do not need TCP to help maintain an ordered session to storage. We can therefore take advantage of the network's broadcast capabilities.

4.4 Filtering Broadcasts

It is frequently unnecessary for all storage operations to be broadcast to all servers. In this case, it is beneficial to include a parameter to the Append that tells the controller whether or not to broadcast the result to all servers. The server performing the Append determines whether a page is worth broadcasting, and if so it indicates this in the broadcast parameter and uses the broadcast facility to transmit the message. Otherwise the page is sent to storage via a unicast mechanism, and the result is unicast back to only the originating server. In the Hyder system, the broadcast parameter is used to differentiate large transaction-specific values from the transaction metadata. Alternatively, the response to a unicast append can still be broadcast, in order to make it easier for other servers to monitor the state of the log.

4.5 Detecting Holes

Replaying the log occurs in two ways. At startup, a server uses the GetLastPage operation to retrieve a page from the log. Every page can contain application information about checkpoints or other interesting log positions from which to replay, and the server can use that information to start requesting each page in sequence beginning at some checkpoint. As discussed below, this process can be heavily optimized.

At some point during replay the server will reach the end of the stored log. However, if the rest of the system is still appending, the log will continue to grow. During replay, when the server encounters its first end-of-log response to a GetPage operation, it begins monitoring for broadcasts from other servers and from the log itself.

As with any other broadcast communication, this introduces the possibility of missing a logged page. A server that has caught upİ and is replaying the log from the live broadcasts will normally need to detect these holes in the log. This is trivial when all pages are broadcast to all servers, because there will be a gap in the sequence of page numbers. The replaying server can simply request the missed page from storage.

However, broadcast filtering makes hole-detection more difficult. The replaying server needs to determine whether the gap in the page number sequence is due to a missed message or simply because the page was never intended to be broadcast.

One way to tell the difference is to have the storage controller track the highest stored page that was flagged for broadcast. It can then include that value in its response to the next Append that is flagged for broadcast, thus producing a linked list of broadcast messages. Whenever a server receives a broadcast from the controller, it

checks this value to ensure that all broadcasts have been received. If it detects a hole, it must replay part of the log to fill in the hole.

4.6 Speed Reading

Reading from a log can be scaled out arbitrarily, because the content of an append-only log is by definition immutable, and therefore the content can be shared, copied and cached without concern for cache coherence. A page number is a unique identifier for the content of a particular page, and the page is permanently identified by that number. So any number of servers can capture a mapping of page numbers to pages, in any form that they find useful, and can keep their cache relatively up-to-date by monitoring broadcasts. Even application-specific functions over the data in the cached pages can in turn be cached and invalidated efficiently.

One convenient way to cache the log is to index the pages that were marked for broadcast, and to keep the payloads of recent such pages in memory. In this way it becomes possible for a server to cheaply enumerate the broadcast pages in some recent region of the log. Such an index provides an excellent way to speed up log replay: when a server encounters a hole in the log, it simply asks any neighboring server to provide the missing information. This avoids having to request each page from the storage controller. The neighboring server can normally respond to such requests instantly, with a dense representation of the missing pages that had been originally intended for broadcast. This same facility can be used when replaying the log after catching up, or when recovering after a restart.

Likewise, it is trivial to implement a distributed hash table that caches frequently accessed log pages. When a server needs to fetch a page to perform a computation, it first requests it from one or two of the neighbors that are responsible for caching pages in the appropriate hash bucket. This allows a large system based on appendable storage to scale out indefinitely for read-only operations.

4.7 Conclusion

By briefly summarizing the kinds of mechanisms and data structures designed for Hyder, our goal was to show that interesting systems can be built over an append-only interface for semiconductor storage. In fact, such systems can run much more efficiently on modern storage devices if the devices stop trying to pretend they are disks. They fundamentally are not, and the impact on the storage stack is as significant as that last major storage hardware shift, from tape to disk.

References

[1] Bernstein, Philip A., Colin W. Reid, and Sudipto Das: Hyder – A Transactional Record Manager for Shared Flash. 5th Conf. on Innovative Database Systems (CIDR 2011), to appear.

[2] Gray, Jim: Long Term Storage Trends and You, research.microsoft.com/en-us/um/people/gray/talks/io_talk_2006.ppt

[3] Jones, Richard and Rafael Lins. Garbage Collection: Algorithms for Automatic Dynamic Memory Management. John Wiley and Sons, July 1996.

[4] Patterson, David A.: Latency Lags Bandwith. Commun. ACM 47(10): 71-75 (2004)

[5] Rosenblum, Mendel and John K. Ousterhout: The Design and Implementation of a Log-Structured File System. ACM Trans. Computer Systems 10(1), 26-52 (1992).