RethinkDB — Rethinking Database Storage

Leif Walsh, Vyacheslav Akhmechet, and Mike Glukhovsky
Hexagram 49, Inc.[*]
{leif, slava, mike}@hexagram49.com

July 21, 2009

Abstract

Dropping costs of solid state drives and their physical properties such as low power consumption, durability due to lack of mechanical parts, and lack of rotational latency and arm contention make solid state drives a lucrative alternative to rotational drives for many computing tasks [7, 3]. For decades, core components of commonly used software stacks have been optimized for physical properties of rotational storage. In recent years researches have performed work to optimize file systems [6] and operating systems paging mechanisms [14] for solid state drives. However, little work has been performed to optimize database storage engines for solid state technology. In this paper we introduce RethinkDB, a MySQL storage engine we designed from the ground up for solid state drives. We show that physical properties of solid state drives require a significantly different approach to that of traditional databases, and that designing a system exclusively for solid state drives enables a simple implementation of highly desirable features.

1 Introduction

Solid state drives exhibit two key properties. Firstly, the lack of mechanical parts eliminates rotational latency and arm contention, making their random reads significantly more efficient than those of their rotational counterparts [6]. Secondly, due to the physical design of the storage arrays, random writes require clearing and recopying of pages, making these operations expensive, and decreasing the drive lifetime [13]. The first property makes spacial locality of data less relevant to performance, eliminating (but not prohibiting) the need for such traditional data organization as clustered indexes and data structures optimized for spacial locality such as B-Trees [5].

The second property makes traditional indexing technology less efficient [13] because B-Tree nodes are designed to be modified in place, conflicting with expensive random writes on solid state drives.

In light of these two properties, we take an append-only approach to storing data, pioneered by log-structured file systems [8]. Additionally, we designed an append-only index mechanism [1] to allow efficient performance for indexed and unindexed data alike. The append-only approach is made possible by inexpensive random reads and lack of necessity for clustered indexes, and is made desirable by expensive random writes on solid state hardware.

Efficient append-only storage opens a number of possibilities for lucrative features that cannot be easily and efficiently implemented by traditional storage engines. We dedicate the next section to careful discussion of these features.

2 Consequences

2.1 Data Consistency

The biggest immediate consequence of organizing and modifying data in an append-only fashion is continuous data consistency. While the database modifi-

[*]Copyright Hexagram 49, Inc., 2009. All rights reserved.

cation operations are being performed by appending data to the end of the file, the rest of the file provides an entirely consistent view of the database. Because there are no modifications in the middle of the file, it is not possible to find the database in an inconsistent state, provided knowledge is maintained of the last successful write. This property of RethinkDB allows implementing a number of desirable features that traditionally require complex and inefficient implementations.

2.1.1 Hot Backups

One such feature is backing up the database while it continues to answer queries. Traditionally such a feature requires complex locking, copying, and scheduling [9], in order to carefully ensure the resulting backup contains a consistent view of the data. The RethinkDB engine implements hot backup by simply copying the database file as it is being modified. The only danger is that the resulting backup contains an incomplete transaction at the end of the file, a situation that is easily handled at start-up (Section 2.1.2).

Furthermore, incremental backups can easily be achieved by appending to the backup copy the part of the original database file that is newer than the backup time stamp — an operation easily performed by a freely available utility such as `rsync`.

2.1.2 Instantaneous Recovery

In Section 2.1.1 we mentioned that the only possible data inconsistency may result from an incomplete transaction at the end of the file. This leads us to instantaneous recovery from power failures — another consequence of the append-only approach. Traditionally, on power failure, storage engines either rebuild the indexes entirely by analyzing the entire data set, or replay the transaction log to repair the inconsistent data snapshot that results from an interrupted transaction [11].

RethinkDB does away with both approaches. In an append-only storage engine the database itself acts as the transaction log, so the only action necessary to restore consistency is truncating the database file to the last known consistent transaction — a comparatively inexpensive and trivial to implement operation. This allows the RethinkDB storage engine to become instantaneously available upon restoration of power, a highly desirable feature that comes for free from its append-only structure.

2.1.3 Easy Replication

Another benefit of the append-only storage approach is a more elegant implementation of replication. Traditionally, database storage engines implement replication using a wide variety of approaches. All commonly used replication methods have well known limitations [12]. RethinkDB implements a simpler and more reliable method that shares its implementation with incremental hot backups — the differences in the master and slave database files are simply transferred via a standard utility (such as `rsync`). No other specialized mechanisms are necessary, as standard and well known tools may be used to assist database administrators in this task.

2.2 Lock-Free Concurrency

Efficiently executing concurrent queries has always been a difficult problem in database design. Storage engines take a variety of approaches from full table locks that disable concurrency for the duration of table modifications, row-level locks that allow more granular control of concurrency, and techniques such as multiversion concurrency control that minimize locking but require many short-lived locks and complicated copying algorithms to simultaneously execute reader and writer queries [2].

RethinkDB requires none of these techniques due to the consistency property of append-only systems discussed in Section 2.1. It requires only knowledge of the file offset which marks the last completed transaction, which is trivially maintained with an extremely small lock. No other locks or specialized mechanisms are required — readers may run simultaneously with a queue of writers regardless of the complexity or peculiarity of read and write queries. Because of its append-only storage approach, RethinkDB gets inexpensive and easily implementable concurrency essentially for free.

Dropping costs and increasing availability of inexpensive multicore CPUs [4], and modern application access patterns that include a large number of simultaneous connections [10] make efficient concurrency a necessity in modern database systems. Traditionally, the limitations imposed by data locality requirements have limited database vendors to inefficient and complicated concurrency implementations — a problem we solve using the relaxed requirements of solid state drives.

2.3 Live Schema Changes

With a database that maintains consistency at any given point in time, we efficiently implement long running operations without stopping the database. We already discussed one such operation — hot backup. Another operation that can be implemented efficiently is modification of database schema and table alteration while the database continues to answer queries. Traditionally, customers had to manually administer complex setups, making live schema updates a very desirable feature.

We can continue to answer read queries as the database performs altering operations in an append-only fashion. Handling writers is slightly more complicated, as any data written to the database needs to be handled separately until the alteration is complete, however, this behavior can be implemented in a fairly straight forward way.

2.4 Database Snapshots

Our append-only approach allows customers who are willing to pay for extra disk space to query snapshots of the database the way it looked at any given point during its lifetime. Our indexing strategy enables RethinkDB to answer such queries instantaneously with a single extra index per table [1]. However, as we discuss in the following section, there is a heavy disk space penalty incurred by not garbage collecting such data. It is possible to allow customers to tailor garbage collection to their access needs, trading disk space for snapshot availability.

3 Limitations

While an efficient append-only data storage approach is made possible by the properties of solid-state drives, it does not come without limitations. We have discovered two limitations to date. The first is the large amount of data that quickly becomes obsolete in an environment with a heavy insert or update workload [1]. This workload requires careful collection and reclamation of hard disk space and careful management of memory and machine caches. The second limitation is that eliminating data locality requires a larger number of disk accesses - a significantly cheaper operation on solid state than mechanical drives [7, 3], but still an expensive operation. We discuss these limitations in more depth in the following sections.

3.1 Garbage Collection

In our experimental runs we have seen that our index data structures generate 2GB of data per index, for every 50MB of data — an overhead of nearly 4000% per index [1]. The extra data contains information that allows instantaneous access to the snapshots of the database at any point in its lifetime, however, most users do not require this feature and do not wish to pay the price associated with the extra space required. Furthermore, the extra data places a strain on memory caches and transfer rates to the disk [1] — a situation that can impede the performance of the storage engine.

For these reasons, a carefully tuned garbage collector is required for a production quality append-only storage engine which reclaims unnecessary data both in memory and on disk. Implementation of such a garbage collector is complicated because it is unacceptable for modern databases to have maintenance interruptions, requiring the garbage collector to operate efficiently in parallel with the the live database. A good implementation of such a live garbage collector is an essential component to a production quality append-only storage engine, and we are currently developing it for the RethinkDB engine.

3.2 Data Locality

As mentioned above, good random read performance makes data locality significantly less important on solid state drives, allowing an efficient implementation of an append-only storage engine. However, random read access is still less efficient than sequential read access [6], and the increased number of read transfers imposes significant penalties on pipelining. Additionally, while the performance of a complex I/O scheduler on solid state drives is not optimal, they are prevalent, and this increased number of read transfers poses a more difficult challenge to these schedulers.

We are unable to evaluate this limitation until the garbage collection outlined in the previous section is implemented. However, we have reasons to believe that these limitations will not have a significant impact on the RethinkDB storage engine's performance. Firstly, in a highly concurrent environment, multiple threads can be interleaved to minimize pipelining effects. Secondly, a very large percentage of database customers have sufficiently small databases that they are efficiently maintained in RAM, making disk access a non-issue for a large number of customers.

We will carefully examine the issue of data locality when garbage collection is implemented, and will report our findings.

4 Development Status

Our append-only storage approach allows for efficient and elegant implementation of the features outlined in this paper. However, there is still a considerable effort required to build a production quality storage engine. Currently, RethinkDB is in the alpha stage, with proper support for most of the storage engine API exposed by the MySQL database. It is under heavy development to ensure efficient implementation of the most commonly used database queries.

At this time, hot backups, instantaneous recovery, and lock-free concurrency are fully available. In addition to stabilizing the engine and optimizing performance, the next major feature under heavy development is a concurrent garbage collector as described in Section 3.1. Easy replication and live schema changes are scheduled to be developed after that. We have no plans to support database snapshots until we release a production-ready version of RethinkDB.

5 Future Work

In addition to addressing the limitations discussed above and implementing the features outlined in the development status, work needs to be done to enable transaction support and and multiple simultaneous writers. Supporting transactional access in an append-only storage engine is not a particularly difficult task. However, multiple simultaneous writers require an implementation of transactional memory — a time consuming and complex task. At this stage, we have decided not to implement this feature because most modern businesses have are very few long running transactions [10], which allows us to queue writers and execute a single write transaction at a time without visible latency for most customers.

References

[1] Slava Akhmechet, Leif Walsh, and Michael Glukhovsky. An append-only index tree structure. To appear in fourth quarter of 2009.

[2] Hal Berenson, Phil Bernstein, Jim Gray, Jim Melton, Elizabeth O'Neil, and Patrick O'Neil. A critique of ansi sql isolation levels. *SIGMOD Rec.*, 24(2):1–10, 1995.

[3] Brian Dipert and Lou Hebert. Flash memory goes mainstream. *IEEE Spectr.*, 30(10):48–52, 1993.

[4] David Geer. Industry trends: Chip makers turn to multicore processors. *Computer*, 38(5):11–13, 2005.

[5] Sooyong Kang, Sungmin Park, Hoyoung Jung, Hyoki Shim, and Jaehyuk Cha. Performance trade-offs in using nvram write buffer for flash memory-based storage devices. *IEEE Transactions on Computers*, 58(6):744–758, 2009.

[6] Atsuo Kawaguchi, Shingo Nishioka, and Hiroshi Motoda. A flash-memory based file system. In *In Proceedings of the Winter 1995 USENIX Technical Conference*, pages 155–164, 1995.

[7] George Lawton. Improved flash memory grows in popularity. *Computer*, 39(1):16–18, 2006.

[8] Mendel Rosenblum and John K. Ousterhout. The design and implementation of a log-structured file system. *ACM Trans. Comput. Syst.*, 10(1):26–52, 1992.

[9] Steve Shumway. Issues in on-line backup. In *In Proceedings of the 5th Large Installation System Administration Conference*, 1991.

[10] Michael Stonebraker, Samuel Madden, Daniel J. Abadi, Stavros Harizopoulos, Nabil Hachem, and Pat Helland. The end of an architectural era: (it's time for a complete rewrite). In *VLDB '07: Proceedings of the 33rd international conference on Very large data bases*, pages 1150–1160. VLDB Endowment, 2007.

[11] Joost S. M. Verhofstad. Recovery techniques for database systems. *ACM Comput. Surv.*, 10(2):167–195, 1978.

[12] Matthias Wiesmann, Andr Schiper, Fernando Pedone, Bettina Kemme, and Gustavo Alonso. Database replication techniques: A three parameter classification. *Reliable Distributed Systems, IEEE Symposium on*, 0:206, 2000.

[13] Chin-Hsien Wu, Tei-Wei Kuo, and Li Ping Chang. An efficient b-tree layer implementation for flash-memory storage systems. *ACM Trans. Embed. Comput. Syst.*, 6(3):19, 2007.

[14] Yun-Seok Yoo, Hyejeong Lee, Yeonseung Ryu, and Hyokyung Bahn. Page replacement algorithms for nand flash memory storages. pages 201–212. 2007.