rethinkdb.com

RethinkDB internals: the caching architecture

Slava Akhmechet January 06, 2011

8-10 minutes

A few days ago Salvatore Sanfilippo wrote about his work on a new Redis diskstore – a reworking of the old VM layer based on a different set of assumptions and tradeoffs. I was very interested in the new approach because it is very similar to the RethinkDB caching architecture in principle. When we designed our caching layer we had a number of constraints we needed to satisfy:

- **Performance** - the system needed to work at memory speeds when the data fits into RAM, as well as when the data far exceeds the amount of available RAM, but the active working dataset mostly fits into RAM. Once the active dataset no longer fits into RAM, the system still needs to work at the speed of the underlying storage layer.
- **Durability** - some clients require every transaction to be durable, others are willing to lose a few seconds worth of data, yet more clients are willing to put up with more significant data loss to sustain high write performance. We needed a system that allows the users fine control of latency, performance, and durability, and still performs well even under the most strict durability

requirements. In order to satisfy this requirement we need fine tuned control of when a given page is flushed to disk (to maximize the possibility to combine multiple changes into a single flush), and when we receive the acknowledgement (so that we can tell the client immediately, or after the data has been persisted, depending on their needs).

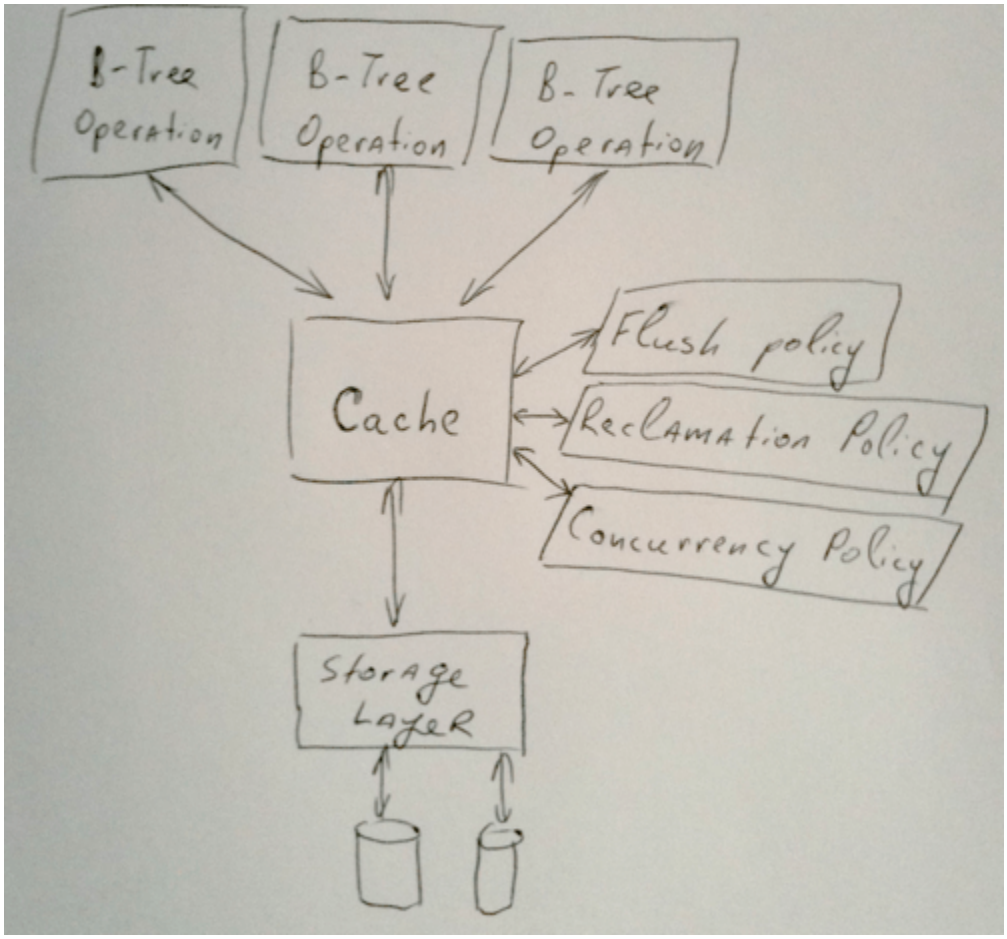
- **Robustness** - there are various edge cases around the size of available memory cache, the distributions of key and value sizes, the number of concurrent clients, network pipelining considerations, etc. We needed a system that works well across all possibilities.
- **Modularity** - the caching subsystem needed to be independent of the data structure layers above it, and the storage layer below it. Currently RethinkDB indexing is implemented using a B-Tree, but we may support other data structures (such as a Hash) in the future. On the other side, the storage subsystem is designed to adapt to the underlying storage device, often generates very different workloads to get good I/O performance, and must be entirely decoupled from the caching layer.
- **Systems considerations** - the underlying operating and storage systems have a number of peculiarities. They support different asynchronous IO APIs, require different sizes of blocks and various location distributions for optimal performance, blocks need to be allocated in RAM on various offsets to work with the DMA controllers, etc. We need to be able to satisfy these considerations in the caching and storage layers.

  The first potential solution to all of the contraints above is to use the page (buffer) cache from the underlying operating system. The

data store file can be memory-mapped into the address space, and the kernel can manage all of the underlying details. However, after prototyping this solution we dismissed it because of the following reasons:

- The page cache does not allow the application fine tuned control over the page reclamation policy. In case of a database, pages can be arranged in a dependency graph where a group of pages cannot be used by the system if one of the pages is flushed (for example, if the root of a B-Tree is flushed out of the page cache, none of the remaining pages can be used for new queries). This can significantly improve the performance of the page reclamation algorithm by empowering it with extra knowledge and by significantly simplifying the necessary bookkeeping.
- The page cache does not allow the application fine tuned control over the flushing policy. Furthermore, it is not possible to implement a storage layer that intelligently writes to certain locations on the disk independently of the memory usage (say, a log-structured system) without implementing a custom file system - a significant development and deployment complication.
- On some kernels (for example, on Linux) the asynchronous disk I/O API (`io_submit` and friends) works reliably only with direct I/O turned on, unless you're using certain specific file systems (currently, XFS). In our testing asynchronous disk APIs performed better than user-space threadpools, and we wanted the system to use it reliably across different file systems and block devices.

  In principle, the high level caching layer architecture is similar to the kernel's buffer cache and is illustrated by the following diagram:



The high level data structures know nothing about the storage layer and only deal with the cache. This gives us the flexibility to implement anything from an unbacked (volatile) cache for testing, to a fallthrough cache that uses no memory and always goes to the storage layer, to a cache that implements different reclamation and flushing policies, backed by different storage layers that handle the specifics of disk performance.

The current (and future) data structures interact with the cache by asking for specific page IDs and access requirements. If the cache contains a given page in memory, it returns it to the layer above. On the other hand, if the page is not in RAM, the page cache asks the underlying storage system to asynchronously schedule an I/O request, and tells the above layer that the page isn't present. This allows the system to move on to handling a different client request

without the application blocking, and when the I/O request is satisfied by the storage system, the cache calls the right layer to tell it to proceed with the operation.

Interestingly, even though the cache knows nothing about the data structure the pages are arranged in, the data structure is maintained implicitly in the cache, and the higher level components are merely functions that transform the data structure in the cache from one state to the next. For example, we may have a B-Tree split operation that splits a page into two pages by adding an extra page to the cache and shuffling around child IDs in other pages, but there is no additional “B-Tree structure” - everything about the B-Tree is automatically in the cache, and therefore on disk. This takes care of persistence automatically, as long as the data in the pages only relies on IDs of other pages and not volatile memory pointers.

In addition to satisfying all of our initial goals, this system gives us a number of additional benefits:

- **Flexibility** - any number of reclamation and flushing algorithms can be implemented, tested, and measured independently of the rest of the code. Furthermore, we can develop and deploy any number of storage algorithms from standard, to log structured, to networked, in user-space and have full control of how we perform I/O.
- **Portability** - our code will work the same way on every operating system and will not depend on different page cache implementations.
- **Concurrency control** - when the cache contains the dependency graph of the pages, concurrency policies can be managed

automatically by the caching layer without interaction with other components. For example, point-in-time copy-on-write for a given B-Tree can be trivially implemented entirely by the cache.

- **Data streaming** - when accessed properly from the higher level structures, very large values (even gigabytes in size) can easily be supported because they can be managed (and therefore, loaded and unloaded by the cache) a few pages at a time.
- **Architectural benefits** - a hierarchy of caches are a good starting point for implementing transactions. For example, when a transaction starts it's trivial to create a new (possibly disk-backed) cache for it that points to the original cache as the parent. At the end of the operation, if the transaction is committed, the cache is merged with the parent, while if the transaction is rolled back, the cache is discarded.

  The caching layer has been one of the most interesting pieces of our infrastructure from both algorithmic, systems, and performance standpoints. The architecture seems simple but there are many details that require careful thinking - I'll try to write more about them after I cover other components.

  Found a typo? Help us improve this page.