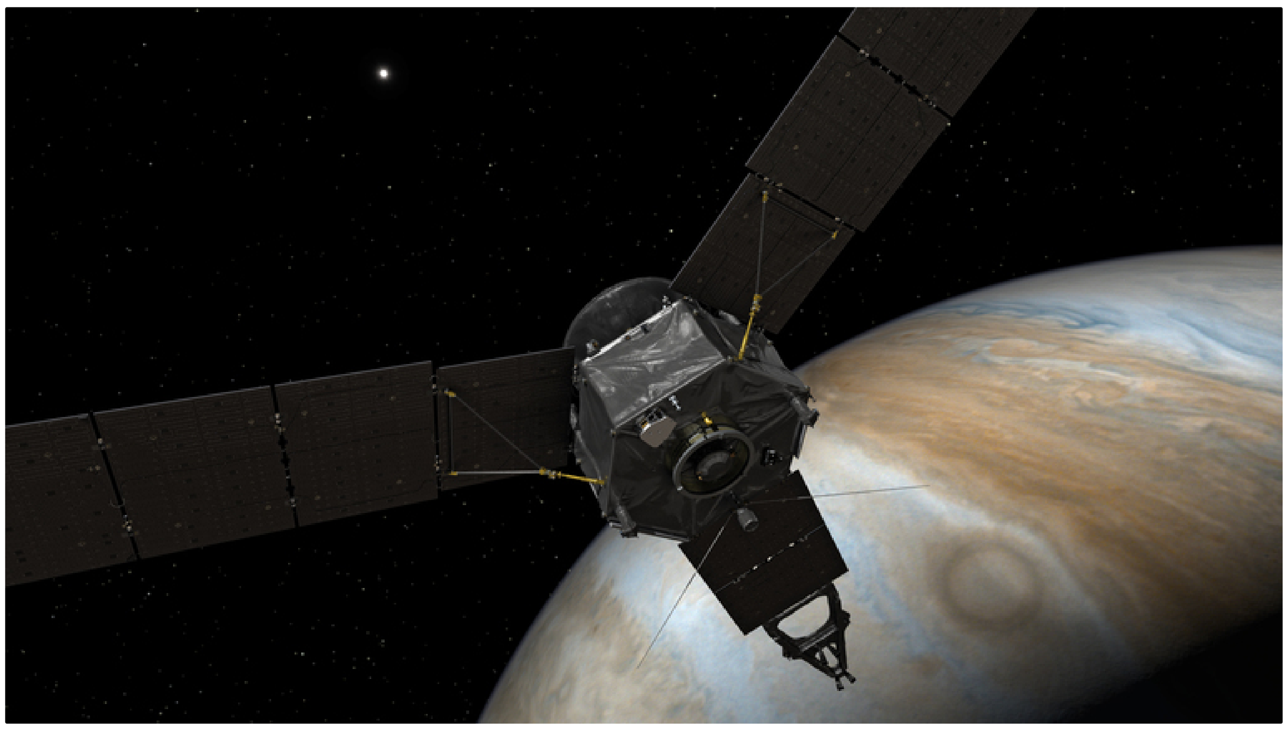
Sun, May 14, 2017 by Manish Rai Jain

Introducing Badger: A fast key-value store written purely in Go



We have built an efficient and persistent log structured merge (LSM) tree based key-value store, purely in Go language. It is based upon WiscKey paper included in USENIX FAST 2016. This design is highly SSD-optimized and separates keys from values to minimize I/O amplification; leveraging both the sequential and the random performance of SSDs.

We call it Badger. Based on benchmarks, Badger is at least **3.5x faster than RocksDB** when doing random reads. For value sizes between 128B to 16KB, data loading is 0.86x - 14x faster compared to RocksDB, with Badger gaining significant ground as value size increases. On the flip side, Badger is currently slower for range key-value iteration, but that has a lot of room for optimization.

Background and Motivation

Word about RocksDB

RocksDB is the most popular and probably the most efficient key-value store in the market. It originated in Google as SSTable which formed the basis for Bigtable, then got released as LevelDB. Facebook then improved LevelDB to add concurrency and optimizations for SSDs and released that as RocksDB. Work on RocksDB has been continuously going on for many years now, and it's used in production at Facebook and many other companies.

So naturally, if you need a key-value store, you'd gravitate towards RocksDB. It's a solid piece of technology, and it works. The biggest issue with using RocksDB is that it is written in `C++`; requiring the use of Cgo to be called via Go.

Cgo: The necessary evil

At Dgraph, we have been using RocksDB via Cgo since we started. And we've faced many issues over time due to this dependency. Cgo is not Go, but when there are better libraries in C++ than Go, Cgo is a necessary evil.

The problem is, Go CPU profiler doesn't see beyond Cgo calls. Go memory profiler takes it one step further. Forget about giving you memory usage breakdown in Cgo space, Go memory profiler fails to even notice the presence of Cgo code. Any memory used by Cgo would not even make it to the memory profiler. Other tools like Go race detector, don't work either.

Cgo has caused us `pthread_create` issues in Go1.4, and then again in Go1.5, due to a bug regression. Lightweight goroutines become expensive pthreads when Cgo is involved, and we had to modify how we were writing data to RocksDB to avoid assigning too many goroutines.

Cgo has caused us memory leaks. Who owns and manages memory when making calls is just not clear. Go, and C are at the opposite spectrums. **One doesn't let you free memory, the other requires it.** So, you make a Go call, but then forget to `Free()`, and nothing breaks. *Except much later.*

Cgo has given us a unmaintainable code. Cgo makes code ugly. The Cgo layer between RocksDB was the one piece of code no one in the team wanted to touch.

Surely, we fixed the memory leaks in our API usage over time. In fact, I *think* we have fixed them all by now, but I can't be sure. Go memory profiler would never tell you. And every time someone complains about Dgraph taking up more memory or crashing due to OOM, it makes me nervous that this is a memory leak issue.

Huge undertaking

Everyone I told about our woes with Cgo, told me that we should just work on fixing those issues. Writing a key-value store which can provide the same performance as RocksDB is a huge undertaking, not worth our effort. Even my team wasn't sure. I had my doubts as well.

I have great respect for any piece of technology which has been iterated upon by the smartest engineers on the face of the planet for years. RocksDB is that. And if I was writing Dgraph in C++, I'd happily use it.

But, I just hate ugly code.

And I hate recurring bugs. No amount of effort would have ensured that we would no longer have any more issues with using RocksDB via Cgo. I wanted a clean slate, and my profiler tools back. Building a key-value store in Go from scratch was the only way to achieve it.

I looked around. The existing key-value stores written in Go didn't even come close to RocksDB's performance. And that's a deal breaker. **You don't trade performance for cleanliness. You demand both.**

So, I decided we will replace our dependency on RocksDB, but given this isn't a priority for Dgraph, none of the team members should work on it. This would be a side project that only I will undertake. I started reading up about B+ and LSM trees, recent improvements to their design, and came across WiscKey paper. It had great promising ideas. I decided to spend a month away from core Dgraph, building Badger.

That's not how it went. I couldn't spend a month away from Dgraph. Between all the founder duties, I couldn't fully dedicate time to coding either. Badger developed during my spurts of coding activity, and one of the team members' part-time contributions. Work started end January, and now I think it's in a good state to be trialed by the Go community.

LSM trees

Before we delve into Badger, let's understand key-value store designs. They play an important role in data-intensive applications including databases. Key-value stores allow efficient updates, point lookups and range queries.

There are two popular types of implementations: Log-structured merge (LSM) tree based, and B+ tree based. The main advantage LSM trees have is that all the foreground writes happen in memory, and all background writes maintain sequential access patterns. Thus they achieve a very high write throughput. On the other hand, small updates on B+-trees involve repeated random disk writes, and hence are unable to maintain high throughput write workload[1].

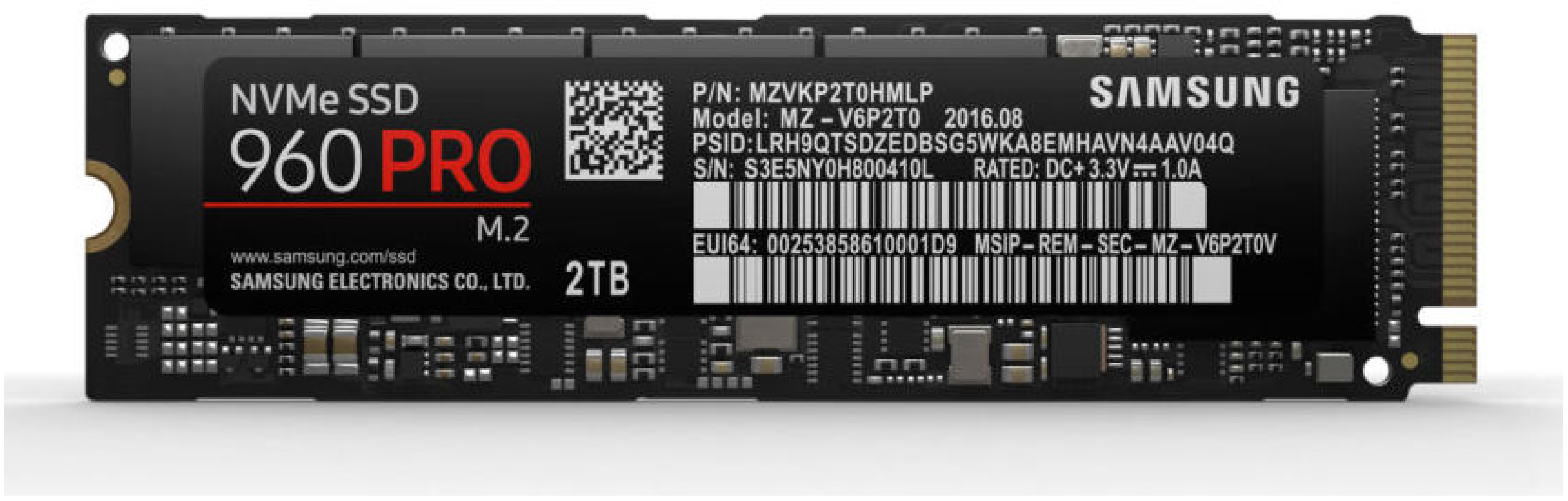
To deliver high write performance, LSM-trees batch key-value pairs and write them sequentially. Then, to enable efficient lookups, LSM-trees continuously read, sort and write key-value pairs in the background. This is known as a `compaction`. LSM-trees do this over many levels, each level holding a factor more data than the previous, typically `size of Li+1 = 10 x size of Li`.

Within a single level, the key-values get written into files of fixed size, in a sorted order. Except level zero, all other levels have zero overlaps between keys stored in files at the same level.

Each level has a maximum capacity. As a level `Li` fills up, its data gets merged with data from lower level `Li+1` and files in `Li` deleted to make space for more incoming data. As data flows from level zero to level one, two, and so on, the same data is re-written multiple times throughout its lifetime. Each key update causes many writes until data eventually settles. This constitutes *write amplification*. For a 7 level LSM tree, with 10x size increase factor, this can be 60; 10 for each transition from L1->L2, L2->L3, and so on, ignoring L0 due to special handling.

Conversely, to read a key from LSM tree, all the levels need to be checked. If present in multiple levels, the version of key at level closer to zero is picked (this version is more up to date). Thus, a single key lookup causes many reads over files, this constitutes *read amplification*. WiscKey paper estimates this to be 336 for a 1-KB key-value pair.

LSMs were designed around hard drives. In HDDs, random I/Os are over 100x slower than sequential ones. Thus, running compactions to continually sort keys and enable efficient lookups is an excellent trade-off.


NVMe SSD
960 PRO
M.2
www.samsung.com/ssd
SAMSUNG ELECTRONICS CO., LTD.
2TB
SAMSUNG
P/N: MZVKP2T0HMLP
Model: MZ-V6P2T0 2016.08
PSID:LRH9QTSDZEDBSG5WKA8EMHAVN4AAV04Q
S/N: S3E5NY0H800410L RATED: DC+3.3V 1.0A
EUI64: 00253858610001D9 MSIP-REM-SEC-MZ-V6P2T0V

However, SSDs are fundamentally different from HDDs. The difference between their sequential and random reads are not nearly as large as HDDs. In fact, top of the line SSDs like Samsung 960 Pro can provide 440K random read operations per second, with 4KB block size. Thus, an LSM-tree that performs a large number of sequential writes to reduce later random reads is wasting bandwidth needlessly.

Badger

Badger is a simple, efficient, and persistent key-value store. Inspired by the simplicity of LevelDB, it provides `Get`, `Set`, `Delete`, and `Iterate` functions. On top of it, it adds `CompareAndSet` and `CompareAndDelete` atomic operations (see GoDoc). It does not aim to be a database and hence does not provide transactions, versioning or snapshots. Those things can be easily built on top of Badger.

Badger separates keys from values. The keys are stored in LSM tree, while the values are stored in a write-ahead log called the *value log*. Keys tend to be smaller than values. Thus this set up produces much smaller LSM trees. When required, the values are directly read from the log stored on SSD, utilizing its vastly superior random read performance.

Guiding principles

These are the guiding principles that decide the design, what goes in and what doesn't in Badger.

- Write it purely in Go language.
- Use the latest research to build the fastest key-value store.
- Keep it simple, stupid.
- SSD-centric design.

Key-Value separation

The major performance cost of LSM-trees is the compaction process. During compactions, multiple files are read into memory, sorted, and written back. Sorting is essential for efficient retrieval, for both key lookups and range iterations. With sorting, the key lookups would only require accessing at most one file per level (excluding level zero, where we'd need to check all the files). Iterations would result in sequential access to multiple files.

Each file is of fixed size, to enhance caching. Values tend to be larger than keys. When you store values along with the keys, the amount of data that needs to be compacted grows significantly.

In Badger, only a pointer to the value in the value log is stored alongside the key. Badger employs *delta encoding* for keys to reduce the effective size even further. Assuming 16 bytes per key and 16 bytes per value pointer, **a single 64MB file can store two million key-value pairs.**

Write Amplification

Thus, the LSM tree generated by Badger is much smaller than that of RocksDB. This smaller LSM-tree reduces the number of levels, and hence number of compactions required to achieve stability. Also, values are not moved along with keys, because they're elsewhere in value log. Assuming 1KB value and 16 byte keys, the effective write amplification per level is `(10*16 + 1024)/(16 + 1024) ~ 1.14`, a much smaller fraction.

You can see the performance gains of this approach compared to RocksDB as the value size increases; where loading data to Badger takes factors less time (see Benchmarks below).

Read Amplification

As mentioned above, the size of LSM tree generated by Badger is much smaller. Each file at each level stores lots more keys than typical LSM trees. Thus, for the same amount of data, fewer levels get filled up. A typical key lookup requires reading all files in level zero, and one file per level from level one and onwards. With Badger, filling fewer levels means, fewer files need to be read to lookup a key. Once key (along with value pointer) is fetched, the value can be fetched by doing random read in value log stored on SSD.

Furthermore, during benchmarking, we found that Badger's LSM tree is so small, it can easily fit in RAM. For 1KB values and 75 million 22 byte keys, the raw size of the entire dataset is 72 GB. Badger's LSM tree size for this setup is a mere 1.7G, which can easily fit into RAM. This is what causes Badger's random key lookup performance to be at least 3.5x faster, and Badger's key-only iteration to be blazingly faster than RocksDB.

Crash resilience

LSM trees write all the updates in memory first in memtables. Once they fill up, memtables get swapped over to immutable memtables, which eventually get written out to files in level zero on disk.

In the case of a crash, all the recent updates still in memory tables would be lost. Key-value stores deal with this issue, by first writing all the updates in a write-ahead log. Badger has a write-ahead log, it's called value log.

Just like a typical write-ahead log, before any update is applied to LSM tree, it gets written to value log first. In the case of a crash, Badger would iterate over the recent updates in value log, and apply them back to the LSM tree.

Instead of iterating over the entire value log, Badger puts a pointer to the latest value in each memtable. Effectively, the latest memtable which made its way to disk would have a value pointer, before which all the updates have already made their way to disk. Thus, we can replay from this pointer onwards, and reapply all the updates to LSM tree to get all our updates back.

Overall size

RocksDB applies block compression to reduce the size of LSM tree. Badger's LSM tree is much smaller in comparison and can be stored in RAM entirely, so it doesn't need to do any compression on the tree. However, the size of value log can grow quite quickly. Each update is a new entry in the value log, and therefore multiple updates for the same key take up space multiple times.

To deal with this, Badger does two things. It allows compressing values in value log. Instead of compressing multiple key-values together, we only compress each key-value individually. This provides the best possible random read performance. The client can set it so compression is only done if the key-value size is over an adjustable threshold, set by default to 1KB.

Secondly, Badger runs value garbage collection. This runs periodically and samples a 100MB size of a randomly selected value log file. It checks if at least a significant chunk of it should be discarded, due to newer updates in later logs. If so, the valid key-value pairs would be appended to the log, the older file discarded, and the value pointers updated in the LSM tree. The downside is, this adds more work for LSM tree; so shouldn't be run when loading a huge data set. More work is required to only trigger this garbage collection to run during periods of little client activity.

Hardware Costs

But, given the fact that SSDs are getting cheaper and cheaper, using extra space in SSD is almost nothing compared to having to store and serve a major chunk of LSM tree from memory. Consider this:

For 1KB values, 75 million 16 byte keys, RocksDB's LSM tree is 50GB in size. Badger's value log is 74GB (without value compression), and LSM tree is 1.7GB. Extrapolating it three times, we get 225 million keys, RocksDB size of 150GB and Badger size of 222GB value log, and 5.1GB LSM tree.

Using Amazon AWS US East (Ohio) datacenter:

- To achieve a random read performance equivalent of Badger (at least 3.5x faster), RocksDB would need to be run on an `r3.4xlarge` instance, which provides 122 GB of RAM for `$1.33` per hour; so most of its LSM tree can fit into memory.
- Badger can be run on the cheapest storage optimized instance `i3.large`, which provides 475GB NVMe SSD (`fio` test: 100K IOPS for 4KB block size), with 15.25GB RAM for `$0.156` per hour.
- The cost of running Badger is thus, **8.5x cheaper** than running RocksDB on EC2, on-demand.
- Going 1-year term all upfront payment, this is $6182 for RocksDB v/s $870 for Badger, still 7.1x cheaper. **That's a whopping 86% saving.**

Benchmarks

Setup

We rented a storage optimized i3.large instance from Amazon AWS, which provides 450GB NVMe SSD storage, 2 virtual cores along with 15.25GB RAM. This instance provides local SSD, which we tested via `fio` to sustain close to 100K random read IOPS for 4KB block sizes.

The data sets were chosen to generate sizes too big to fit entirely in RAM, in either RocksDB or Badger.

VALUE SIZE	NUMBER OF KEYS (EACH KEY = 22B)	RAW DATA SIZE
128B	250M	35GB
1024B	75M	73GB
16KB	5M	76GB

We then loaded data one by one, first in RocksDB then in Badger, never running the loaders concurrently. This gave us the data loading times and output sizes. For random `Get` and `Iterate`, we used Go benchmark tests and ran them for 3 minutes, going down to 1 minute for 16KB values.

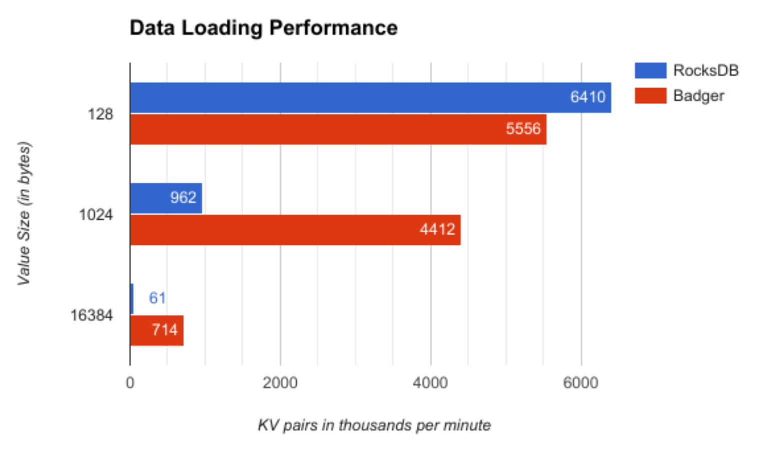
All the code for benchmarking is available in this repo. All the commands ran and their measurements recorded are available in this log file. The charts and their data is viewable here.

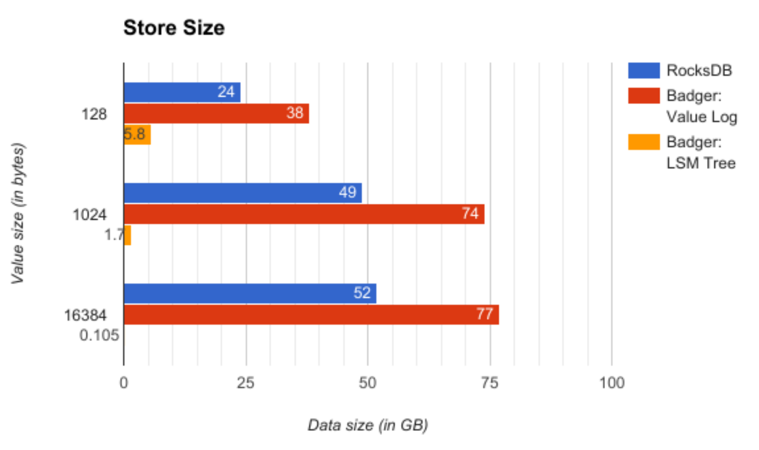
Results

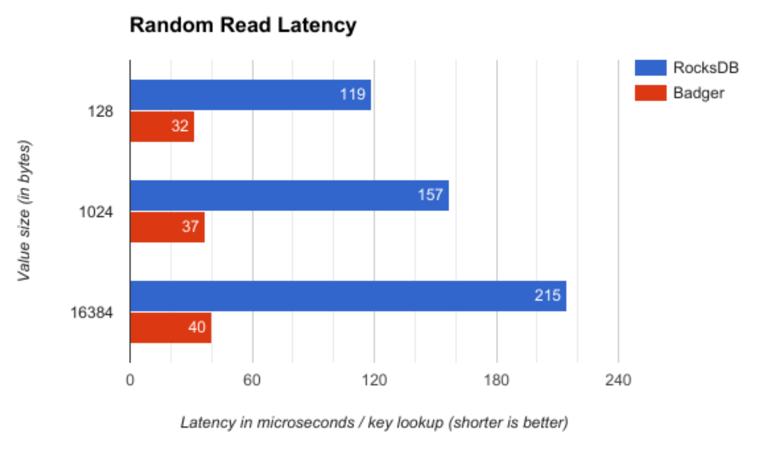
In the following benchmarks, we measured 4 things:

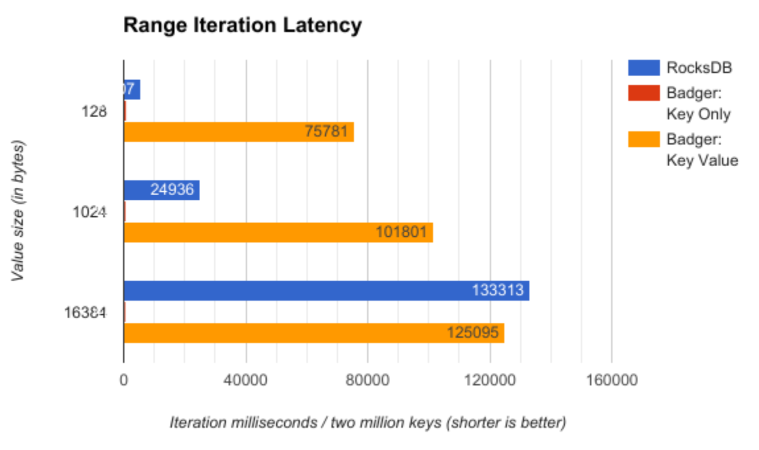
- Data loading performance
- Output size
- Random key lookup performance (`Get`)
- Sorted range iteration performance (`Iterate`)

All the 4 measurements are visualized in the following charts.









Data loading performance: Badger's key-value separation design shows huge performance gains as value sizes increase. For value sizes of 1KB and 16KB, Badger achieves 4.5x and 11.7x more throughput than RocksDB. For smaller values, like 16 bytes not shown here, Badger can be 2-3x slower, due to slower compactions (see further work).

Store size: Badger generates much smaller LSM tree, but a larger value size log. The size of Badger's LSM tree is proportional only to the number of keys, not values. Thus, Badger's LSM tree decreases in size as we progress from 128B to 16KB. In all three scenarios, Badger produced an LSM tree which could fit entirely in RAM of the target server.

Random read latency: Badger's `Get` latency is only 18% to 27% of RocksDB's `Get` latency. **In our opinion, this is the biggest win of this design.** This happens because Badger's entire LSM tree can fit into RAM, significantly decreasing the amount of time it takes to find the right tables, check their bloom filters, pick the right blocks and retrieve the key. Value retrieval is then a single SSD `file.pread` away.

In contrast, RocksDB can't fit the entire tree in memory. Even assuming it can keep the table index and bloom filters in memory, it would need to fetch the entire blocks from disk, decompress them, then do key-value retrieval (Badger's smaller LSM tree avoids the need for compression). This obviously takes longer, and given lack of data access locality, caching isn't as effective.

Range iteration latency: Badger's range iteration is significantly slower than RocksDB's range iteration, when values are also retrieved from SSD. *We didn't expect this, and still don't quite understand it.* We expected some slowdown due to the need to do IOPS on SSD, while RocksDB does purely serial reads. But, given the 100K IOPS `i3.large` instance is capable of, we didn't even come close to using that bandwidth, despite pre-fetching. This needs further work and investigation.

On the other end of the spectrum, Badger's key-only iteration is blazingly faster than RocksDB or key-value iteration (latency is shown by the almost invisible red bar). This is quite useful in certain use cases we have at Dgraph, where we iterate over the keys, run filters and only retrieve values for a much smaller subset of keys.

Further work

Speed of range iteration

While Badger can do key-only iteration blazingly fast, things slow down when it also needs to do value lookups. Theoretically, this shouldn't be the case. Amazon's i3.large disk optimized instance can do 100,000 4KB block random reads per second. Based on this, we should be able to iterate 100K key-value pairs per second, in other terms six million key-value pairs per minute.

However, Badger's current implementation doesn't produce SSD random read requests even close to this limit, and the key-value iteration suffers as a result. There's a lot of room for optimization in this space.

Speed of compactions

Badger is currently slower when it comes to running compactions compared to RocksDB. Due to this, for a dataset purely containing smaller values, it is slower to load data to Badger. This needs more optimization.

LSM tree compression

Again in a dataset purely containing smaller values, the size of LSM tree would be significantly larger than RocksDB because Badger doesn't run compression on LSM tree. This should be easy to add on if needed, and would make a great first-time contributor project.

B+ tree approach

[1] Recent improvements to SSDs might make B+-trees a viable option. Since WiscKey paper was written, SSDs have made huge gains in random write performance. A new interesting direction would be to combine the value log approach, and keep only keys and value pointers in the B+-tree. This would trade LSM tree read-sort-merge sequential write compactions with many random writes per key update and might achieve the same write throughput as LSM for a much simpler design.

Conclusion

We have built an efficient key-value store, which can compete in performance against top of the line key-value stores in market. It is currently rough around the edges, but provides a solid platform for any industrial application, be it data storage or building another database.

We will be replacing Dgraph's dependency on RocksDB soon with Badger; making our builds easier, faster, making Dgraph cross-platform and paving the way for embeddable Dgraph. The biggest win of using Badger is **a performant Go native key-value store.** The nice side-effects are **~4 times faster `Get` and a potential 86% reduction in AWS bills,** due to less reliance on RAM and more reliance on ever faster and cheaper SSDs.

So try out Badger in your project, and let us know your experience.

P.S. Special thanks to Sanjay Ghemawat and Lanyue Lu for responding to my questions about design choices.

We are building a fast, transactional and distributed graph database. If you like what you read above, come join the team!

Get started with Dgraph	https://docs.dgraph.io
Star us on GitHub	https://github.com/dgraph-io/dgraph
Ask us questions	https://discuss.dgraph.io

Fortune 500 companies are using Dgraph in production. If you need production support, talk to us.

Top image: Juno spacecraft is the fastest moving human made object, traveling at a speed of 265,00 kmph relative to Earth.