Thu, Jan 24, 2019 by Manish Rai Jain

Why we choose Badger over RocksDB in Dgraph

At Dgraph, we're building **the most advanced graph database** in the world. It does distributed transactions, low-latency arbitrary depth joins, traversals, provides synchronous replication and horizontal scalability — with a simple GraphQL-like API. *Does it sounds like a tough problem?* That's because it is.

This blog post is about Badger, the key-value database that makes it all happen under the hood, housing all Dgraph data, including Raft logs. Badger uses an LSM tree structure for holding keys. Badger is different from other LSM tree based DBs in that it colocates only small values or pointers along with these keys, storing the values separately in a value log.

Many Go developers have to make this exact decision when they need to choose a key-value database. Badger, RocksDB, or the various other KV DBs around, which one they should base their projects on. As I write this blog post, Badger is crossing 5000 GitHub stars proving its popularity in the Go community. So, this is a perfect time for a post about our experience using Badger and what makes it unique.

The Motivation

Recently, our friends at CockroachDB wrote about why they use RocksDB. The various APIs they rely upon internally and how they use them. They also mentioned the things they're doing to get around the Cgo barrier.

That reminded me of the alternate path we embarked upon almost two years ago, where we decided to cut out RocksDB and Cgo entirely from our code base and write a fast key-value database purely in Go (see original post for a deeper look into the whys).

The Go language prides itself on the speed of compilation and its amazing debugging tools. In fact, the debug tools built by Go are directly informed by the tooling that existed at Google. The pprof charts that are common place now in the Go community had only been available to a select few who worked at Google backends. All of these benefits of the Go langauge are negated when Cgo is used. That was the motivation which led us to write Badger.

We started at a disadvantage. *Replace a DB started at Google and pioneered at Facebook? Go vs C++? New vs Established?* So, I knew we had to do things differently.

When starting the project, I remember telling my team that if we can be within 75% performance range of RocksDB, it would be a good outcome. But as I started spending more and more time thinking and reading about the problem, it became clear that not only can we get the performance of RocksDB, **we can actually do better.** There was research out there which improved upon LSM trees, in particular, a paper I came across called WiscKey. The beauty of this paper compared to others was in its logical simplicity: **Separate key-values, reduce write amplification, and reduce read amplification.**

Was it the right move?

Now that it's been 1.5 years since launch, and almost 2 years since the initial commit, we have had time to reflect on our decision to build Badger. And I can tell you, *there's not been a single day of regret over writing Badger from scratch, and we'd do it all over again.* The time and effort required to build and maintain Badger *dwarfs* in comparison to the advantages we have had using it in Dgraph. Over time, we've pushed more and more things from Dgraph to Badger, which has made Dgraph simpler and Badger even more enticing to a broader community of Go developers.

Because Badger acted as a base for Dgraph first and foremost, we built some unique things into it. **These features are unique enough that it would be prohibitively hard for any other KV DB to replace Badger in Dgraph.** In this blog post, I'd like to talk about those unique features, that make Badger different from other storage engines, *including RocksDB,* and provide wheels for the *engineering wagon* at Dgraph Labs.

Performance Benefits

Badger provides better performance than RocksDB, both in terms of reads **and** writes. Writes because each table in LSM tree can hold a lot more keys, so there are fewer compactions. Reads because the LSM tree in Badger is shallower, so fewer levels have to be queried.

Many developers automatically assume that RocksDB is faster than Badger and dismiss performance of Badger over RocksDB as being due to lack of Cgo[1]. Also, comparisons between Badger and RocksDB typically assume a zero-sum game. *If Badger is faster, then it must be less consistent, or must have fewer features.* That might be the case if the design was exactly the same. But that's not the case here.

Badger's design is inherently more complex than RocksDB. Not only does Badger need to deal with an LSM tree, but it also needs to maintain a value log. There's the added complexity of maintaining a value log, doing live value log garbage collection, and moving valid key-value pointers around—all while keeping the LSM tree constraints valid (newer versions at the top, older below). This is a different design and therefore there are bigger wins in performance, beyond just the cost of Go to Cgo calls.[2] The implementation of this wasn't easy, and the paper had left out exactly these complex and important topics, worthy of another paper on its own.

Imitation is the best form of flattery. Our performance numbers and design decisions have triggered a fork of RocksDB by authors of TiDB, where they intend to implement separation of keys from values.

Overall, we got a *Strongly Exceeds Expectations* on performance with Badger. Today Badger is the most actively maintained KV DB in Go language and is being adopted by more and more Go projects. It has crossed 5000 GitHub stars. In addition to Dgraph, Badger is being used by IPFS, Usenet Express (Usenet Express is serving Petabytes of data via Badger), various blockchain companies like Decred, Insolar, IoTex and many others. Godoc shows there are 194 Go packages relying on Badger.

Badger features

I'll start by mentioning all the staple features that Badger has. Badger stores keys in a lexicographically sorted manner. It exports Transaction API, which can Set, Get and Delete keys. You can iterate over the keys in both forward and reverse order, using Iterators. At Dgraph, we use iterations in both directions to provide various inequality operators.

Beyond these basics, Badger has a lot of unique features that are hard to find in other storage engines, built to help with the specific demands of a distributed graph database. In fact, some of these features were built in Dgraph first, before getting ported over to Badger (like Stream framework), while others were built specifically to avoid rebuilding the same in Dgraph (like Managed mode).

In general, our philosophy towards development in Dgraph has been that **any change which is generic enough and can be used by a wide-array of users, should lie in Badger.**

So, let's have a look at these unique features.

ACID SSI Transactions

Dgraph relies heavily on the transactional nature of Badger. Badger provides multiple version concurrency control, snapshots and strong ACID transactional guarantees. It provides serializable snapshot isolation (SSI) guarantees, while also running transactions concurrently to achieve great performance.

You can see these in action, by running `badger bank test`, which runs a Jepsen-style bank test, (by default) running 16 concurrent transactions over 10K accounts, moving money from one to another, constantly checking if the total money in the bank has changed at all (it shouldn't). With this running nightly for 8hrs, we get the confidence that Badger is capable of supporting distributed ACID transactions in Dgraph.

Key-Only Iteration

Badger can push a lot of keys into a single SSTable, whatever the size of values might be, because only small values or value pointers are stored in LSM tree. Hence, our LSM tree is much smaller than the total amount of data, so we can easily load the tables in RAM (via `options.LoadToRAM`, or `options.MemoryMap`).

This quick RAM access to the LSM tree allows Badger to provide an option to iterate over the keys very quickly (`PrefetchValues=false`), and get a bunch of unique information about the values without fetching them: value identification bits, size of the value, expiry time and and so on.

In Dgraph, we use `Item.EstimatedSize()` to calculate the size of certain ranges of data without ever touching the values. And use the identification bits to support the has function. We also use this in the Stream framework, described later.

3D Access (Key-Value-Versions)

Badger is probably the only key-value database which exposes the versions of the values to the user. When opening a Badger DB, you can set the `options.NumVersionsToKeep`. We set this by default to 1 in Badger, but in Dgraph, we set this to infinity [3].

A typical `Txn.Get` in Badger would only retrieve the latest version back. But, `Txn.Iterator` can be instructed to retrieve all the available versions, via `IteratorOptions.AllVersions`. Badger stores versions in reverse order. That is, if you are iterating in the forward direction, Badger would return the latest version first, then the older versions in descending order (opposite in reverse iteration).

In Dgraph, we use this feature very heavily. All the writes to Dgraph, are stored as *deltas* in Badger, using value identification bits. Periodically, we *roll-up* the deltas into a state, using the Stream framework (described below). Which keys need to be rollup can be quickly determined using the very fast key-only iteration.

Single-Key Iteration

Just the idea of this is beyond the scope of other storage engines. Because Badger supports 3D access (described above), it can optimize iterations where the user only wants to iterate over versions of the same key (as explained in the Fast scans section of CockroachDB's blog post). It can do that by using table boundaries and bloom filters to eliminate tables from LSM tree in advance, picking only those which might have the key (multiple versions of the key could lie across multiple levels in the LSM tree). This is more efficient than typical iteration.

In Dgraph, whenever we need to read a key, we use this iterator.

Value Identification Bits

Badger allows users to specify an optional byte when storing keys and values, exposed via `Txn.SetWithMeta`. This byte can be used to uniquely identify the type of the value. In Dgraph, we use this byte in all our writes, to identify whether the value is a delta, a state, a schema, or empty; before even retrieving the value (see code here). This knowledge allows our access to be very efficient.

Separation, but only as much as needed

While Badger is designed around key-value separation, it provides a knob to the user to specify a `ValueThreshold`. Any values below this threshold are colocated with the keys in the LSM tree. For values above this threshold, only their pointers are stored in the LSM tree. By default, the threshold is set to 32 bytes.

Dgraph uses this fact to fine tune performance based on access patterns. Dgraph opens up two instances of Badger: one for data, and one for Raft logs. We set these knobs differently for these two. For data, we set the `ValueThreshold` to 1KB. For Raft logs, we use `badger.LSMOnlyOptions`, which sets the threshold to the maximum allowed value of `uint16` (minus a few bytes to avoid edge cases), around 64KB.

And surely, the resulting behavior is quite different. Raft logs are constantly being written and deleted. LSM tree compactions do a good job of quickly reclaiming the space while maintaining the valid small sized data, which we keep in memory via `options.LoadtoRAM`.

The data instance keeps long-lived keys, can have large values, and can grow unbounded to TBs in size. But we know our users are not all on SSDs, using a range of different devices (or network drives) with varying capabilities of IOPS. So the `ValueThreshold` of 1KB gives us a sweet spot where our writes are still fast, while our reads require less random reads from the value log.

Managed Mode

Badger was written to support building another DB on top of it. That knowledge is part of Badger's DNA. It provides a unique mode, called *managed mode* — which allows a user to specify the version a key-value gets written at. Similarly, it allows specifying a read timestamp to use to get a consistent snapshot of the data. This avoids having to build another versioning system on top of Badger (which is already versioned), something we rely upon in Dgraph. In other words, **we can directly use Dgraph transaction timestamps to write data in Badger.**

Badger provides a set of APIs to that extent: `db.OpenManaged`, `db.NewTransactionAt`, `db.NewStreamAt`, etc. to support this use case.

In *normal mode*, Badger relies on an internal `Oracle` to hand out monotonically increasing timestamps. This is what most users of Badger use.

Stream Framework

A key-value database's job is not just to store data, but also to allow very fast ways to access that data. Typical ways of accessing data provided by most DBs are `Txn.Get`, or `Txn.Iterator`, etc. But Dgraph deals with a lot of data movement and doing a serial iteration over the entire DB or even certain prefixes isn't efficient. Synchronous replication, for example, requires that in case of machine failures resulting in data loss, or in case a server falls behind its peers, a data snapshot must be sent to the follower to bring it up to date.

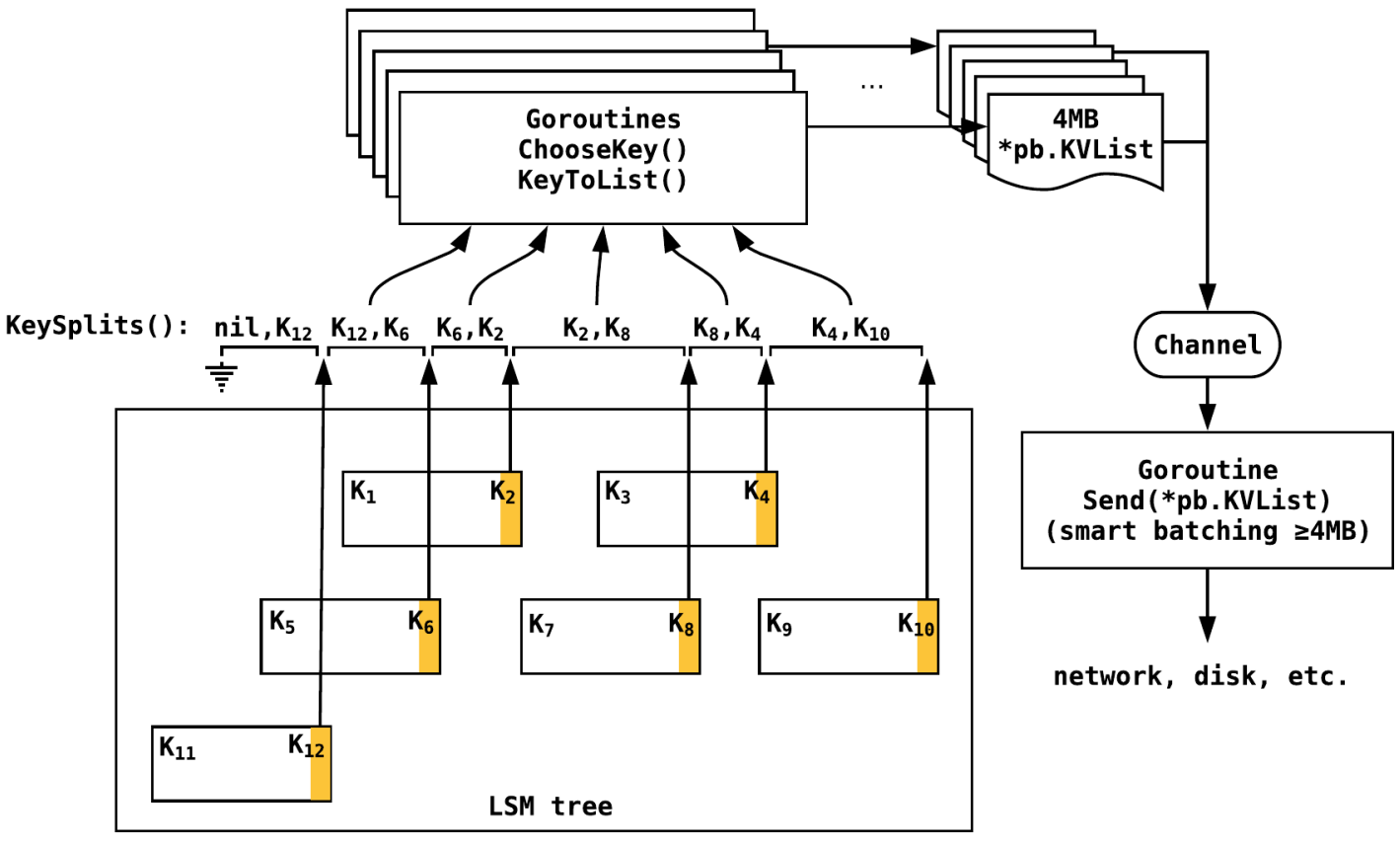
We realized we **needed to quickly grab a lot of data from Badger from certain ranges and move that over the wire.** Not only that, but we also realized that sending a lot of small bits over Go channels or gRPC is very slow. For example, sending every key-value individually over any of these mediums is almost *disastrous* for the performance that we need in Dgraph. So, within Dgraph, we developed the `Stream` framework, which is highly optimized for concurrent iteration and data transmission while avoiding contention.

Usage of the `Stream` framework, while initially designed only to move data between servers, quickly spread to other modules and before we knew it, data exports, data backups, rolling up deltas into states, all were using it. At some point, I determined that the framework itself was generic enough to lie in Badger, and I ported it over.

What `Stream` framework does is that, for any prefix (or even for the entire DB), it splits up the entire data into key ranges (exposed via `db.Tables` and `db.KeySplits`). These key ranges are then split among a bunch of goroutine workers which iterate over these ranges, while providing a few methods for the users to overwrite.

```
ChooseKey func(item *Item) bool
KeyToList func(key []byte, itr *Iterator) (*pb.KVList, error)
Send func(*pb.KVList) error
```

In addition to specifying a key `Prefix` to iterate over, a user can optionally choose if they want to pick a certain key or not. For example, when doing roll-ups (as described in 3D access), we use the `item.UserMeta` in `ChooseKey(item)` to determine if the latest key version is already a state or a delta, so we can skip the keys whose latest value is already a state.



The framework then provides a way for a user to create a list of key-values from a single key in `KeyToList` (default implementation is provided). The framework picks up multiple `KVList`, batches them up into 4MB chunks, and sends them out to a channel. Most users would return a single KV for a single key input, but in Dgraph, we use versioning heavily, and so can return multiple KVs for a single key. Note that both `ChooseKey` and `KeyToList` are run concurrently via goroutines to speed the data retrieval process.

Finally, a goroutine picks up data from this channel, further utilizing the power of smart batching to sequentially call `Send`. In smart batching, data gets buffered until the current Send call returns. Thus, there's always one pending call to `Send`. **This framework can move data as fast as `Send` would allow it.**

In Dgraph, we use this function to plug into gRPC and send data over the wire (send snapshots or predicates out, or take backups over the network), or even back to Badger (roll up deltas into states). In fact, we're using it in 5 different places in Dgraph. Badger itself uses it to provide fast full and incremental backups.

DropAll: A clean slate

When a new Snapshot is received, Dgraph needs a way to quickly drop all the data present in Badger and get ready to receive the new data. Initially, we did this using key-only iteration and setting delete markers at the latest version of their values. However, we found that this caused issues when the new data's version was below the delete marker.

So, we built a way in Badger to drop all the data and start from scratch, without changing the DB pointer held by the process. Badger achieves this complex feat in a very methodical way by stopping the writes, then the compactors, deleting LSM tree, value logs, and finally restarting the compactors and writes. During this process, concurrent reads and writes calls can still be made, they'd return errors without causing panics. In Dgraph, this avoids having to stop all the reads from users while `DropAll` is running, which saves us from code complexity.

We have plans to introduce dropping a prefix of the keys, which is the other functionality that we use in Dgraph.

Tooling

Over time, we have built some very interesting tools around Badger. Most users have concerns about data correctness and integrity. We ourselves wanted to ensure that Badger provides the highest level of ACID transaction guarantees (SSI). So, we baked a tool, called `badger bank`, as described above.

We built `badger fill` to quickly fill up Badger with random data for testing purposes. Similarly, we built `badger flatten` to flatten up the LSM tree into a single level. This uses the `db.Flatten` API, which can also be run live.

Badger, of course, has `badger backup` and `badger restore`, both using internal `db.Backup` and `db.Load` APIs.

Finally, a user can use `badger info` to check the state of the instance, as seen by MANIFEST: the levels where each table is stored, the size of each table, its SHA-256 checksum and the size of each level, value logs and a summary of any abnormalities observed by the tool.

We use these tools on a regular basis to help debug Dgraph instances.

There are others which are not mentioned here, like TTL for keys and setting `Txn.SetWithDiscard`, etc. But, I'll take a break from features here and talk about the challenges of building Badger and the unique design choices we made.

What WiscKey did not mention: Complexity

The WiscKey paper made a good case for the novel idea of key-value separation. During the implementation of Badger, we realized that the LSM tree part, i.e. reading, writing and running compactions, was the easier portion of the code. The harder portion was to garbage collect data from value logs. The paper briefly mentioned doing value log GC offline with a technique which doesn't work well for production systems. In particular, punching holes in files isn't supported by Windows or Mac and does not play well with the various things you need to do to ensure data consistency across file system failures.

The paper missed out two very important questions:

1. How to run value log garbage collection live?
2. How to do that, while maintaining the 3D key-value-version access?

Getting those details right has been an iterative process for us, causing us a bunch of headaches. The solution that we finally put in place is a good one and warrants a blog post on its own (or a paper).

Unique Design Choices

Badger does not have any cache or do any compression [4]. Badger's LSM tree is typically small enough that it can fit in RAM, or at least be memory mapped efficiently. We use prefix-diffing over keys to achieve key compression, but because we can already fit millions of keys in a single table, there's no need to pay for the higher cost of block or table compression.

RocksDB has a lot bigger LSM tree, lot more tables, lot more blocks. To achieve good read performance, they do both compression and caching. In fact, it is spread across two levels, one for compressed blocks and one for uncompressed blocks. All this is unnecessary in Badger. If values are large, they would lie in the value log and be retrieved via a single lookup (LSM tree does the hard work of finding the key). A user can optionally compress the value upfront, before passing it to Badger, and use the value identification bits to know when a value is compressed if needed.

Future Work

Recently, we removed the LRU cache from Dgraph because of a high level of contention and gained a lot in query performance—we saw Badger was perfectly capable of providing us the fast access that we needed to run Dgraph queries. But we think, we can get more performance out of it. So, we plan to introduce an LRU cache in Badger instead.

We also have plans to add encryption at rest in Badger, which might require us to introduce a table level or block level cache (read from disk, decrypt, cache).

And with over 5000 GitHub stars and almost 200 projects using Badger, within the span of 1.5 years, we have a very active community whom we like to hear from. *So, tell us what you'd like to see happen in Badger this year!*

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

Top image: Caption the image and identify the location of two easter eggs in it. The first three developers to find them and reply back to us will receive a Dgraph T-shirt.

---

1. We have already tackled the crash resilience of Badger in our previous blog post. In addition, we have an open bounty on proving that Badger loses data. ↩

2. One could argue that if that was the only differentiator, then Go being a slower language than C++ in a single-threaded execution, Badger would always be slower than RocksDB. ↩

3. This is similar to how Google's Bigtable works, which exposes the z-axis, i.e. the timestamp/version axis, and allows users to set the version when writing a key-value. One could also set the GC policy on the versions, based on the number (`NumVersionsToKeep` in Badger), age (`SetWithTTL`), or other factors. And the way we're using the versions in Dgraph is exactly like how I wrote a real-time graph indexing system back in 2012 at Google to unify all structured data in the company under one system. ↩

4. This might change later with the introduction of new features, as described in Future work. ↩