

---

## STATUS

Twitter is no longer maintaining this project or responding to issues or PRs.

### **Gizzard: a library for creating distributed datastores**

Check out [Using gizzard](#) for details on requirements, how to build gizzard, and a demo app.

Also check out the [gizzard mailing list](#).

### **An introduction to sharding**

Many modern web sites need fast access to an amount of information so large that it cannot be efficiently stored on a single computer. A good way to deal with this problem is to “shard” that information; that is, store it across multiple computers instead of on just one.

Sharding strategies often involve two techniques: partitioning and replication. With *partitioning*, the data is divided into small chunks and stored across many computers. Each of these chunks is small enough that the computer that stores it can efficiently manipulate and query the data. With the other technique of *replication*, multiple copies of the data are stored across several machines. Since each copy runs on its own machine and can respond to queries, the system can efficiently respond to tons of queries for the same data by adding more copies. Replication also makes the system resilient to failure because if any one copy is broken or corrupt, the system can use another copy for the same task.

The problem is: sharding is difficult. Determining smart partitioning schemes for particular kinds of data requires a lot of thought. And even more difficult is ensuring that all of the copies of the data are *consistent* despite unreliable communication and occasional computer failures. Recently, a lot of open-source distributed databases have emerged to help solve this problem. Unfortunately, as of the time of writing, most of the available open-source projects are either too immature or too limited to deal with the variety of problems that exist on the web. These new databases are hugely promising but for now it is sometimes more practical to build a custom solution.

### **What is a sharding framework?**

Twitter has built several custom distributed data-stores. Many of these solutions have a lot in common, prompting us to extract the commonalities so that they would be more easily maintainable and reusable. Thus, we have extracted Gizzard, a Scala framework that makes it easy to create custom fault-tolerant, distributed databases.

---

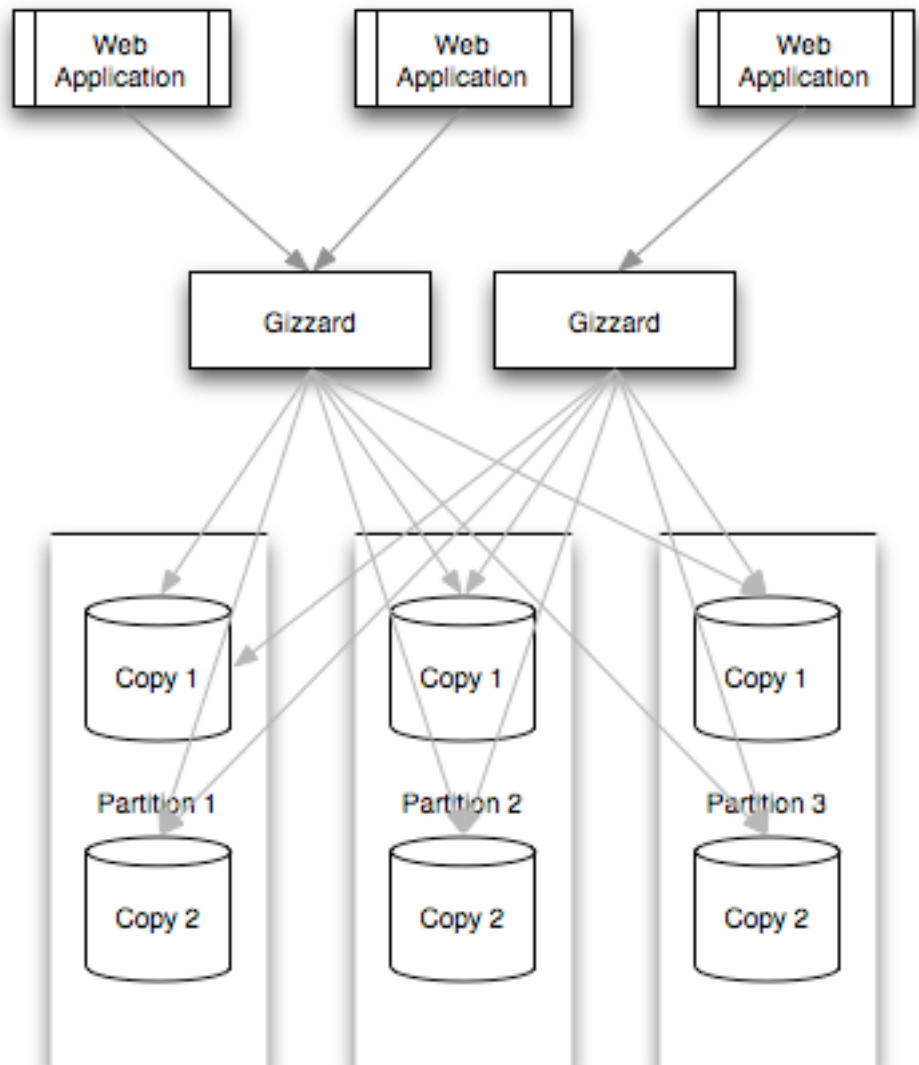
Gizzard is a *framework* in that it offers a basic template for solving a certain class of problem. This template is not perfect for everyone's needs but is useful for a wide variety of data storage problems. At a high level, Gizzard is a middleware networking service that manages partitioning data across arbitrary backend datastores (e.g., SQL databases, Lucene, etc.). The partitioning rules are stored in a forwarding table that maps key ranges to partitions. Each partition manages its own replication through a declarative replication tree. Gizzard supports "migrations" (for example, elastically adding machines to the cluster) and gracefully handles failures. The system is made *eventually consistent* by requiring that all write-operations are idempotent *and* commutative and as operations fail (because of, e.g., a network partition) they are retried at a later time.

A very simple sample use of Gizzard is Rowz, a distributed key-value store. To get up-and-running with Gizzard quickly, clone Rowz and start customizing!

But first, let's examine how Gizzard works in more detail.

---

## How does it work?



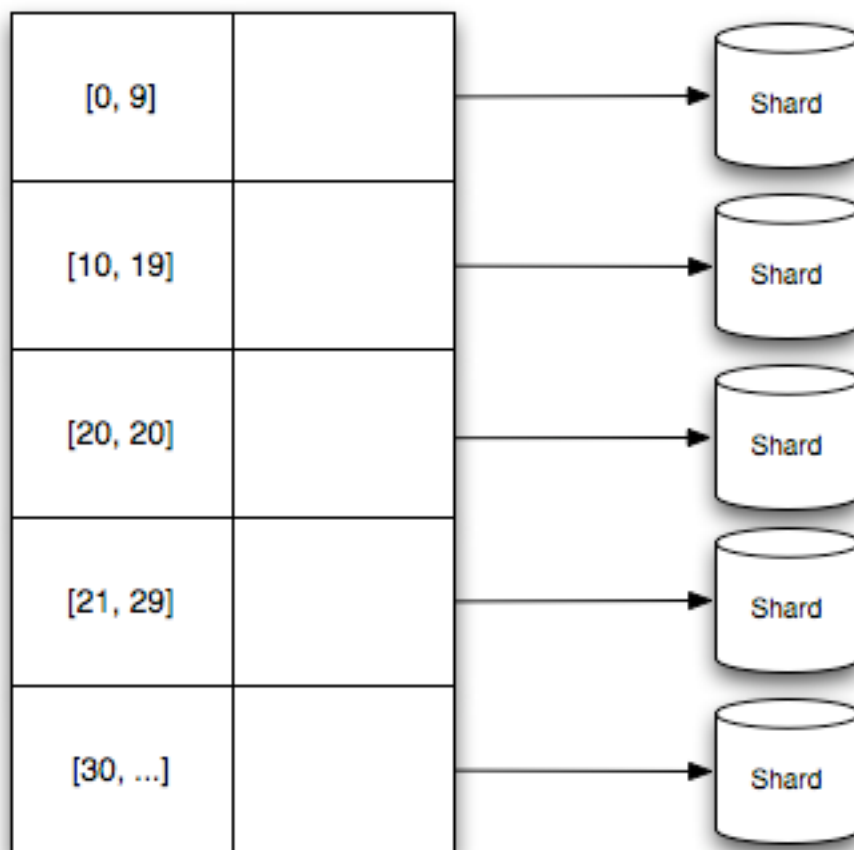
### Gizzard is middleware

Gizzard operates as a *middleware* networking service. It sits “in the middle” between clients (typically, web front-ends like PHP and Ruby on Rails applications) and the many partitions and replicas of data. Sitting in the middle, all data querying and manipulation flow through Gizzard. Gizzard instances are stateless so run as many gizzards as are necessary to sustain throughput or manage TCP connection limits. Gizzard, in part because it runs on the JVM, is quite efficient. One of Twitter’s Gizzard applications (FlockDB, our distributed graph database) can serve 10,000 queries per second per commodity machine. But your mileage may vary.

---

**Gizzard supports any datastorage backend** Gizzard is designed to replicate data across any network-available data storage service. This could be a relational database, Lucene, Redis, or anything you can imagine. As a general rule, Gizzard requires that all write operations be idempotent *and* commutative (see the section on Fault Tolerance and Migrations), so this places some constraints on how you may use the back-end store. In particular, Gizzard does not guarantee that write operations are applied in order. It is therefore imperative that the system is designed to reach a consistent state regardless of the order in which writes are applied.

**Gizzard handles partitioning through a forwarding table** Gizzard handles partitioning (i.e., dividing exclusive ranges of data across many hosts) by mappings *ranges* of data to particular shards. These mappings are stored in a forwarding table that specifies lower-bound of a numerical range and what shard that data in that range belongs to.



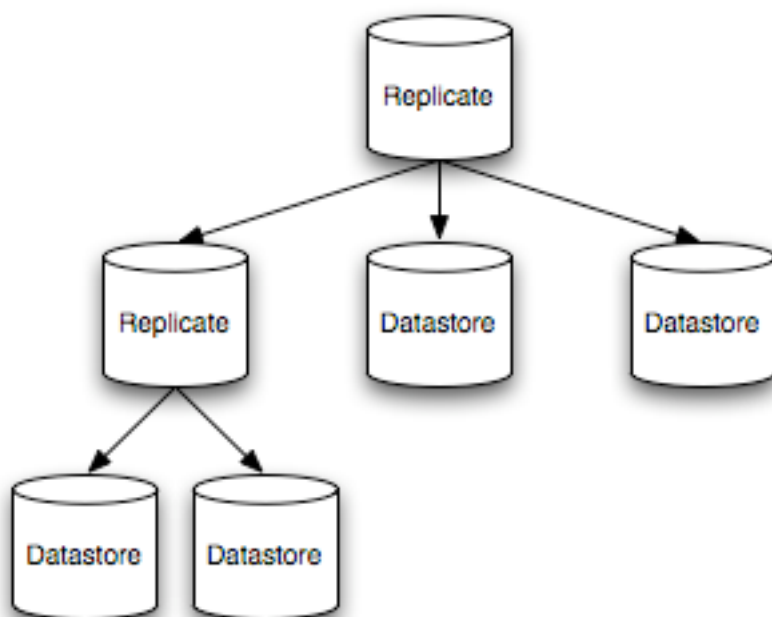
To be precise, you provide Gizzard a hash function that, given a key for your data (and this key can be application specific), produces a number that belongs to one of the ranges in the forwarding table. These functions are programmable so you can optimize for locality or balance depending on your

---

needs.

This range-based approach differs from the “consistent hashing” technique used in many other distributed data-stores. This allows for heterogeneously sized partitions so that you easily manage *hotspots*, segments of data that are extremely popular. In fact, Gizzard does allow you to implement completely custom forwarding strategies like consistent hashing, but this isn’t the recommended approach. For some more detail on partitioning schemes, read wikipedia:

**Gizzard handles replication through a replication tree** Each shard referenced in the forwarding table can be either a physical shard or a logical shard. A physical shard is a reference to a particular data storage back-end, such as a SQL database. In contrast, A *logical shard* is just a tree of other shards, where each *branch* in the tree represents some logical transformation on the data, and each *node* is a data-storage back-end. These logical transformations at the branches are usually rules about how to propagate read and write operations to the children of that branch. For example, here is a two-level replication tree. Note that this represents just ONE partition (as referenced in the forwarding table):



The “Replicate” branches in the figure are simple strategies to repeat write operations to all children and to balance reads across the children according to health and a weighting function. You can create custom branching/logical shards for your particular data storage needs, such as to add additional transaction/coordination primitives or quorum strategies. But Gizzard ships with a few standard strategies of broad utility such as Replicating, Write-Only, Read-Only, and Blocked (allowing neither reads nor writes). The utility of some of the more obscure shard types is discussed in the section on

---

## Migrations.

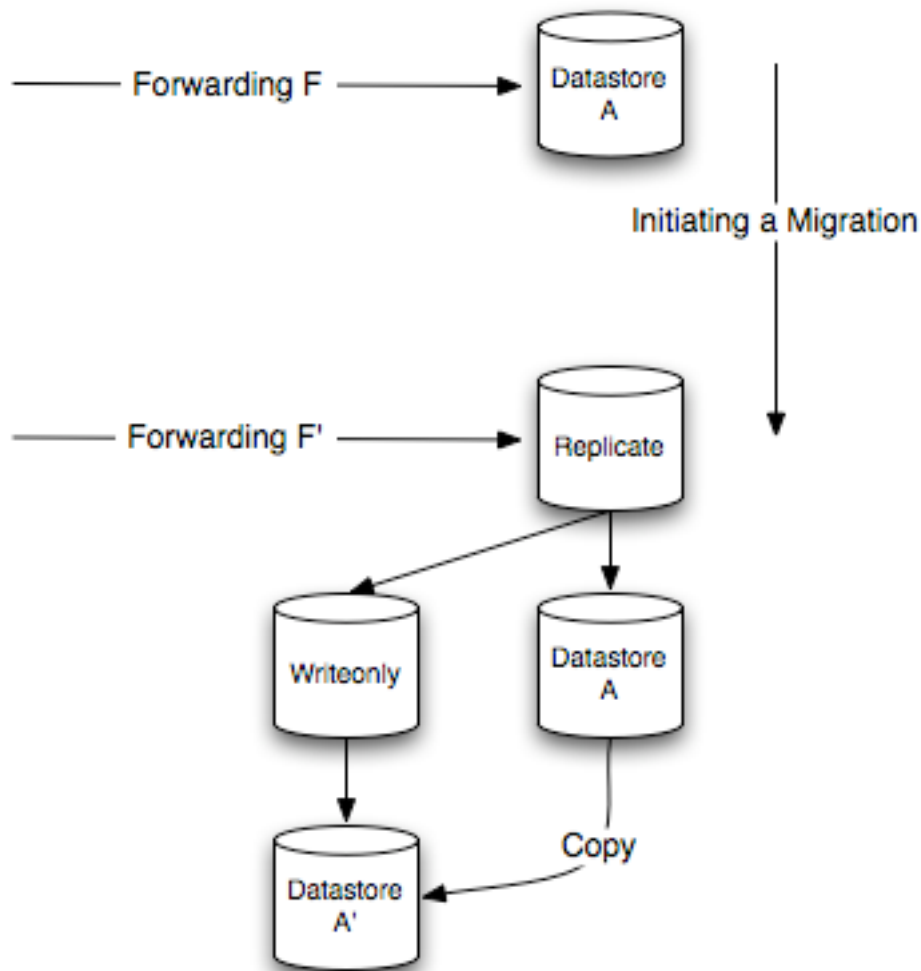
The exact nature of the replication topologies can vary per partition. This means you can have a higher replication level for a “hotter” partition and a lower replication level for a “cooler” one. This makes the system highly configurable. For instance, you can specify that the back-ends mirror one another in a primary-secondary-tertiary-etc. configuration for simplicity. Alternatively, for better fault tolerance (but higher complexity) you can “stripe” partitions across machines so that no machine is a mirror of any other.

**Gizzard is fault-tolerant** Fault-tolerance is one of the biggest concerns of distributed systems. Because such systems involve many computers, there is some likelihood that one (or many) are malfunctioning at any moment. Gizzard is designed to avoid any single points of failure. If a certain replica in a partition has crashed, Gizzard routes requests to the remaining healthy replicas, bearing in mind the weighting function. If all replicas of in a partition are unavailable, Gizzard will be unable to serve read requests to that shard, but all other shards will be unaffected. Writes to an unavailable shard are buffered until the shard again becomes available.

In fact, if any number of replicas in a shard are unavailable, Gizzard will try to write to all healthy replicas as quickly as possible and buffer the writes to the unavailable shard, to try again later when the unhealthy shard returns to life. The basic strategy is that all writes are materialized to a durable, transactional journal. Writes are then performed asynchronously (but with manageably low latency) to all replicas in a shard. If a shard is unavailable, the write operation goes into an error queue and is retried later.

In order to achieve “eventual consistency”, this “retry later” strategy requires that your write operations are idempotent *and* commutative. This is because a retry later strategy can apply operations out-of-order (as, for instance, when newer jobs are applied before older failed jobs are retried). In most cases this is an easy requirement. A demonstration of commutative, idempotent writes is given in the Gizzard demo app, Rowz.

**Winged migrations** It’s sometimes convenient to copy or move data from shards from one computer to another. You might do this to balance load across more or fewer machines, or to deal with hardware failures. It’s interesting to explain some aspect of how migrations work just to illustrate some of the more obscure logical shard types. When migrating from [Datastore A](#) to [Datastore A'](#), a Replicating shard is set up between them but a WriteOnly shard is placed in front of [Datastore A'](#). Then data is copied from the old shard to the new shard. The WriteOnly shard ensures that while the new Shard is bootstrapping, no data is read from it (because it has an incomplete picture of the corpus).



Because writes will happen out of order (new writes occur before older ones and some writes may happen twice), all writes must be idempotent and commutative to ensure data consistency.

**How does Gizzard handle write conflicts?** Write conflicts are when two manipulations to the same record try to change the record in differing ways. Because Gizzard does not guarantee that operations will apply in order, it is important to think about write conflicts when modeling your data. As described elsewhere, write operations must be both idempotent and commutative in order to avoid conflicts. This is actually an *easy* requirement in many cases, way easier than trying to guarantee ordered delivery of messages with bounded latency and high availability. As mentioned above, Rowz illustrates a technique of using time-stamps to only apply operations that are “newer”. More documentation on this will be forthcoming.

---

## **Contributors**

- Robey Pointer
- Nick Kallen
- Ed Ceaser
- John Kalucki
- Matt Freels
- Kyle Maxwell