Gingko Documentation

Gingko works like Antidote in the way that requests are sent to different nodes using the riak core vnodes to manage the load. The backend store is based on mnesia (the checkpoint store will be able to use alternative backends, but the initial version will also use mnesia). This document will describe the main gingko modules and how they interact with each other. At the end, some important details on the mnesia store are described and the operations are explained in the context of the gingko modules.

Modules

gingko

This module will be the main access point for the transaction manager to interact with gingko. This module delegates the calls from the transaction manager to gingko_server/gingko_vnode. This module will contain all public methods of gingko. The current operations are read, read_multiple, update, update_multiple, begin_txn, prepare_txn, commit_txn, abort_txn, and checkpoint. The operations are described after all the modules have been introduced because they interact with almost every module.

gingko_server/gingko_vnode

These two modules are have the same functionality. While the gingko_server is a single gen_server running the gingko_vnode is a riak vnode. Instead of keys the gingko_vnode is partitioned by the transaction ids (riak core consistent hashing) which allows should in theory better distribute the load than a single gen_server. However currently both approaches are considered and there is little overhead because the gingko_server uses the same methods as the gingko_vnode.

The reason why this gen_server/vnode is needed is to track transactions and separate that logic from the log and the cache. These modules can be compared to the Antidote clocksi_vnode (but some parts are implemented in the gingko_log_vnode).

gingko_log_vnode

This module is responsible for handling the journal operations and the checkpointing mechanism. The partitioning is done with keys using riak core consistent hashing. Each vnode has its own mnesia table for its journal. There is some complexity in the creation and loading of these mnesia tables with multiple nodes and this will be explained in more detail at the end of this document. For the scope of this document it is assumed that each vnode has its journal mnesia table and it will move it correctly if the vnode is moved between nodes during a riak core handoff. Compared to the antidote implementation (logging_vnode) the gingko_log_vnode is always called as part of a transaction and it interacts with the gingko_cache described later. Each call from the gingko_server/gingko_vnode adds a journal entry to the mnesia table and then performs the called operation if there is anything else to be done.

gingko_cache_vnode

This module handles the cache of gingko and it currently uses dicts to store snapshots. The cache does not store operations only snapshots. The current implementation of the cache might change because it differs quite significantly from the given specification. Most importantly the cache is designed to use only a single function 'get' and all the other functions of the specification are integrated into the cache and not called by other modules. The specification describes many invariants on the different functions however most of them are implicit and also caused by concurrency problems that don't exist here since only a single process has access to the memory. Concurrency is achieved by riak core but a single vnode will only access its cache in a sequential non-concurrent way. The present bit was removed because it is simpler to just remove the cache entry. (This seems more intuitive however it can be changed again) The used bit was extended to a record that contains more information on the usage of the cache entry for sophisticated caching strategies like LFU or LRU. The original rather simple caching strategy of the specification is kept in the implementation.

The eviction process can triggered via a time interval or only if a certain threshold is reached. While it sounds complex the implementation is very simple, not optimized, and only a proof on concept.

The idea of the cache is to use the 'get' function and it takes care of finding and creating (and caching) the requested snapshot using the journal and the checkpoint store.

Mnesia store

The mnesia store for the journal is a ram-copies store meaning the journal entries are stored only in main memory and only written to disk when an operation requires it. The default checkpoint store is also a mnesia store that is using disk-copies meaning it is stored both on disk and in main memory (this can later be changed). Mnesia can do replication but we only use that for the default checkpoint store. This means the checkpoints will be automatically replicated to all nodes in a gingko cluster. For the journal, the mnesia table is only stored on the node where the vnode is running. During a handoff the mnesia table of the journal must be moved to the new node for which the final implementation is not decided. Either the mnesia replication is used to move the table or the contents are copied entry by entry. During the start-up process all nodes of the riak cluster must be added to the mnesia system meaning their schemas must be in sync. This is quite complex especially if existing clusters are started with new nodes and even more if two clusters are merged. This is a one time implementation effort.

Operations

All operations are started at the gingko module and directly sent to the gingko_server/gingko_vnode. There the actual processing happens.

begin_txn

Asynchronous

The begin_txn operation is sent at the start of transaction and is broadcast to all gingko_log_vnodes. This operation does not do anything except start a new transaction. It requires a dependency vectorclock.

prepare_txn

Synchronous

The prepare_txn operation is sent when the client wants to commit a transaction.

commit_txn

abort_txn

read

Synchronous (multiple reads are maybe asynchronous)

A read starts from the gingko module then is sent to the gingko_server/gingko_vnode (currently reads will force a prepare on the affected partition) which then sends it to the log

update

checkpoint