Garbage collection algorithm

This document outlines how garbage collection of records happens in our databases. Here, garbage collection is not talking about freeing memory (via RCU, for instance), but rather when our database can release a record, to eventually be freed by the RCU sub-system.

In our system, transaction id numbers have the form $[core, n, epoch]$, where the low bits are on the core bits. n is just some monotonically increasing counter (we ignore wrap-around for now).

The system maintains two global numbers, *global_epoch* and *snapshot_epoch*. The invariant maintained is that either $global\_epoch = snapshot\_epoch$ or $global\_epoch = snapshot\_epoch + 1$. The meaning of *snapshop_epoch* is that the database records are consistent up to (but not including) the *snapshot_epoch*. In other words, $[0, 0, snapshot\_epoch]$ is a consistent transaction id.

Our DB tries to keep at most two versions of a record around- the current version, which updates read, and an older consistent version for snapshots. In fact, this is currently a hard limit. This isn't a correctness problem- if a transaction does a snapshot read and sees that its snapshot tid is lower than any version of the record available, then it simply aborts the transaction. This does mean, however, that there is a small window between when the *global_epoch* is increment and the *snapshot_epoch* has not yet caught up, where popular records will most likely not be able to be consistently read. This is easily fixed by letting the maximum number of old records be two, at the expense of more memory.

There is a small caveat, however, to this dropping of old versions: for a record to be completely removed from the underlying btree, it must ensure that *snapshot_epoch* is greater than its latest version (which indicates removal). This is because absense of a key from the underlying btree can only be interpreted one way, so for the correctness of consistent snapshots, this is a hard requirement.

The GC epoch loop is shown below: Whenever a core begins a transaction, it reads both *global_epoch*

```
$global\_epoch \leftarrow 0$
$snapshot\_epoch \leftarrow 0$
while $forever$ do
    $this\_epoch \leftarrow global\_epoch{+}{+}$
    for $each\ core$ do
        Wait for the core to finish any outstanding transaction
    end
    $snapshot\_epoch \leftarrow global\_epoch$
    for $each\ core$ do
        Wait for the core to finish any outstanding transaction
    end
    Run all tasks scheduled for the end of $this\_epoch$
    Sleep for some fixed duration
end
```

and *snapshot_epoch*, and uses the numbers it read at the beginning throughout the transaction.

Now the remaining question is, what are these tasks to be scheduled at the end of an epoch? The answer is that they are garbage collection tasks. We currently do not have a background tree-walker. Instead, garbage collection happens in response to updates. When a new version of a record is created, we immediately discard the oldest versions in order to keep the total number of versions per record below the maximum. We then schedule a cleaner task to run when the epoch of the latest version has finished. This cleaner simply discards any versions with epochs less than the current *snapshot_epoch* except the latest one (there is no reason to keep around multiple consistent versions around). This allows a record to, in the absense of any more updates, shrink the number of versions back down to one.

If the latest update is a delete, then this cleaner task also looks for the opportunity to remove a record from the underlying btree. See the note above on the restrictions for when this is possible.