DistributedDeletes

As background, recall that a Cassandra cluster defines a ReplicationFactor that determines how many nodes each key and associated columns are written to. In Cassandra (as in Dynamo), the client controls how many replicas to block for on writes, which includes deletions. In particular, the client may, and typically will, specify a ConsistencyLevel of less than the cluster's ReplicationFactor, that is, the coordinating server node should report the write successful even if some replicas are down or otherwise not responsive to the write.

(Thus, the "eventual" in eventual consistency: if a client reads from a replica that did not get the update with a low enough ConsistencyLevel, it will potentially see old data. Cassandra uses HintedHandoff, ReadRepair, and AntiEntropy to reduce the inconsistency window, as well as offering higher consistency levels such as ConstencyLevel.QUORUM, but it's still something we have to be aware of.)

Thus, a delete operation can't just wipe out all traces of the data being removed immediately: if we did, and a replica did not receive the delete operation, when it becomes available again it will treat the replicas that did receive the delete as having missed a write update, and repair them! So, instead of wiping out data on delete, Cassandra replaces it with a special value called a tombstone. The tombstone can then be propagated to replicas that missed the initial remove request.

There's one more piece to the problem: how do we know when it's safe to remove tombstones? In a fully distributed system, we can't. We could add a coordinator like ZooKeeper, but that would pollute the simplicity of the design, as well as complicating ops – then you'd essentially have two systems to monitor, instead of one. (This is not to say ZK is bad software – I believe it is best in class at what it does – only that it solves a problem that we do not wish to add to our system.)

So, Cassandra does what distributed systems designers frequently do when confronted with a problem we don't know how to solve: define some additional constraints that turn it into one that we do. Here, we defined a constant, GCGraceSeconds, and had each node track tombstone age locally. Once it has aged past the constant, it can be GC'd during compaction (see MemtableSSTable). This means that if you have a node down for longer than GCGraceSeconds, you should treat it as a failed node and replace it as described in Operations. The default setting is very conservative, at 10 days; you can reduce that once you have Anti Entropy configured to your satisfaction. And of course if you are only running a single Cassandra node, you can reduce it to zero, and tombstones will be GC'd at the first major compaction. Since 0.6.8, minor compactions also GC tombstones.