# Embedded HBase Design

## Motivation

Mainly two points:

1. To see where are we now on hbase core performance
   * As a distributed database, there’re mainly 3 performance related parts for HBase: client, RPC and core
   * Currently we could only see the E2E performance with YCSB or PE, which makes it hard to check the region-based core performance, let alone the comparison with popular databases like rocksdb or ScyllaDB
   * Supporting embedded mode means abstracting the core part and make it possible to test/improve the core performance
2. To be used as Flink state backend
   * Recovery is a big problem for rocksdb-based state backend, since we have to fully copy the backup and restore it into the new rocksdb instance
   * HBase supports to read/write HDFS directly, thus could reduce data copy and accelerate the recovery time a lot

## Design

Currently HBase supports stand-alone mode and suggest to use it for embedded database requirements, which has two main problems:

1. HMaster and HRegionServer are still separated threads, which cost more resource
2. There’s still RPC cost when doing request, which makes a poor performance

And we’re proposing two ways of supporting embedded mode:

* Simple embedded mode
* Only one HMaster thread, which also serves as regionserver, and could serve tables other than meta
* Direct read and no RPC
* Keep using the existing region assignment, procedure and recovery mechanism
* Keep using zookeeper

In this way we could save the RPC cost and test the core performance with the minimum efforts. However, it could not serve as state backend since the startup/shutdown speed is too slow

* Full embedded mode
* Only one region per database, no assignment, no zookeeper
  + Create/open/close region directly against HDFS, no more CreateTableHandler, OpenRegionHandler or CloseRegionHandler
  + Do snapshot/restore directly against HDFS, no coordination for procedure
  + Directly data replay in recovery, no more log split
* Direct read, no RPC
* Keep using existing compaction, flush and WAL rolling mechanism

This way we could better serve as state backend, but there might be bottleneck on write performance

## Main API

Mainly supports below API:

* open
* close
* put
* get
* append
* increment
* delete
* getResultScanner
* snapshot
* restoreSnapshot
* deleteSnapshot
* compact
* favoredNode

Also supports configuration through a Operation instance, just like RocksDB