How to configure your RocksDB state store

Samza provides build-in support for persistent state store, backed by RocksDB on the disk and changelog in Kafka. There are many possible usage scenarios that users want to have various different features to be enabled. This guide attempts to give a general suggestion to various configuration of RocksDB state store in different application scenarios.

Supported Persistent KeyValue Store Use Cases

changelog	TTL	Host-affinity	Description
No	No	No	Not recoverable local state stores, will lose all data when container restarted
Yes	No	No	Recoverable local state stores, keeps old data, re-bootstrap when container restarts
No	Yes	No	Not recoverable local state stores, expiring old data, will lose all data when container restarted
Yes	Yes	No	Recoverable local state stores, expiring old data, re-bootstrap when container restarts (**see note for TTL**)
No	No	Yes	(UNSUPPORTED) Not recoverable local state stores, re-use local state with best-effort when host-affinity succeeds, otherwise losing all data
Yes	No	Yes	Recoverable local state stores, keeps old data, quick-recovery when host-affinity succeeds
No	Yes	Yes	(UNSUPPORTED) Not recoverable local state stores, expiring old data, re-use local state with best-effort when host-affinity succeeds, otherwise losing all data
Yes	Yes	Yes	Recoverable local state stores, expiring old data, quick-recovery when host-affinity succeeds (**see note for TTL**)

Note: host-affinity feature applies to all stores used in a Samza job, while changelog and TTL can be configured per store.

Using RocksDB TTL

1. RocksDB TTL is set for the local instance of RocksDB table on disk. When the records are expired from the table on local disk, they are not immediately deleted from the changelog.
2. If changelog topics are not created apriori, Samza will create changelog topics w/ default configuration, which is to use logcompact cleanup policy, not time-retention policy.
3. Records that are inserted into RocksDB are also written to changelog topics. Records that are expired via TTL are not deleted from the changelog topics immediately.
4. If you manually configure your changelog topic to be time-retention based, records in the changelog will be deleted from Kafka changelog according to the time-retention policy in changelog topic. If the time-retention in changelog topic is shorter than the TTL set for the local RocksDB, you will run into the risk of losing data when the container restarts.
5. Setting the changelog topic to have a logcompact cleanup policy or a time-retention policy with TTL bigger than RocksDB TTL may lead to some expired records re-appearing when the container restarts and re-seeds the state store from the changelog.
6. If there is a changelog topic configured for the store and it is using logcompact policy together with host-affinity, a record that was deleted for long time may be revived: If the revived local RocksDB store is older than delete.retention.ms for the changelog topic, we may miss the deletion of the old record in recovery and may revive some records that were deleted before delete.retention.ms.

Hence, it is recommended that if you use RocksDB TTL feature, do not design your application to be strictly rely on the TTL for correctness (i.e. a record from the state store w/ expired timestamp can re-appear when container restarts). Use it only for opportunistic purging of old records by setting the changelog cleanup policy to either logcompact or time-retention w/ bigger TTL than RocksDB TTL.

Tuning the Memory needed for RocksDB

Samza allows users to configure the memory size used by RocksDB **per store per container**, for cache and for write buffer:

stores.store-name.container.cache.size.bytes	1048576000	The size of RocksDB's block cache in bytes, per container. If there are several task instances within one container, each is given a proportional share of this cache. Note that this is an off-heap memory allocation, so the container's total memory use is the maximum JVM heap size *plus* the size of this cache.
stores.store-name.container.write.buffer.size.bytes	33554432	The amount of memory (in bytes) that RocksDB uses for buffering writes before they are written to disk, per container. If there are several task instances within one container, each is given a proportional share of this buffer. This setting also determines the size of RocksDB's segment files.

Since the above configuration is **per store per container**, you should calculate the total native memory used by your RocksDB stores **per container** using the following formula:

```
numStores * (${stores.store-name.container.cache.size.bytes} + ${stores.store-name.container.write.buffer.size.
bytes})
```

Deleting the whole DB (A.K.A. resetting the state)

There are various cases when you might want to remove all data in RocksDB and restart (e.g. incompatible schema upgrade, restarting with a clean slate). Currently, the recommended solution for that is to rename your RocksDB store.

Let's say the job is using a RocksDB store *my_rocks_store* and now we want to reset the whole DB. You should:

1. Reconfigure the job to use a new store name, e.g. *my_rocks_store_v2.*
2. Re-deploy your job to start using *my_rocks_store_v2.*