Online NamedQueue framework with optional persistence layer (HBASE-24528, HBASE-22978, HBASE-24718)

This document provides an overview of how HBase master and regionserver daemons can serve some critical usecases to clients by using the NamedQueue framework.

Purpose:

The purpose of the *NamedQueue framework* is to provide a generic and scalable in-memory *ring-buffer queue* to emphasis on providing only the recent data from the ring buffer, and the framework also provides persistence support by permanently persisting all the records in HBase system tables in case maintaining the entire history of data is also the priority of the client in addition to serving the recent data from the ring-buffer queue.

Overview and potential usecases:

A *ring-buffer* is an array which can be used as a queue. The ring-buffer has a read position and a write position which marks the next position to read from and write to the ring-buffer. When the write position reaches the end of the array, the write position is set back to 0. The same is true for the read position as well. Setting the read and write position back to zero when they reach the end of the array is also sometimes referred to as "*wrapping around*". When the read and write position reaches the end of the array, they continue from the beginning of the array, just as if the array was a ring. Hence the name *ring-buffer*. It is also known as *circular queue* data structure. The ring-buffer is expected to begin overwriting entries after the wrap-around, this is how they re-use the oldest entries for storing the newest ones.

The primary usecase of the ring-buffer data structure is to persist recent history of data upto a certain limit. For HBase, we have several usecases where very recent history of events can be really helpful to analyze and improve the usecase or identify any critical fixes. For instance, maintaining the recent history of RPC requests that are judged to be too slow or too large than usual at regionservers can be really helpful for operators to analyze any pattern of requests requiring special attention to fix or identify any user triggered requests or requests coming from specific IP addresses requiring our servers to be really busy enough to impact the overall system performance. Similarly, a large scale cluster trying to keep up with the increasing number of regions might also face some troubles with balancing of the regions at the cluster level. Several executions of balancer in the active master might be overall very slow at keeping the cluster balanced. Cautious determination of balancer factors to achieve the faster balancing of the cluster is very crucial and in order to analyze the balancer decision factors, one of the best ways is to maintain the recent history of balancer decisions/rejections in the ring-buffer. Another usecase of the ring-buffer could be to maintain the recent region assignment history. If a region is stuck in transition, it is really important to figure out the root cause and fix the code bug if any or make the workflow robust to avoid the stuck region-in-transition case. If we could maintain a significant history of region movements or administrative region assignments in the ring-buffer data structure at master and regionservers, it would be quite helpful to quickly lookup the history of region transitions and help operators/developers identify the root cause based on the pattern of the history. We could have a variety of such usecases to maintain the recent history: region flush, compaction, merge, server crash performing WAL split and so on. Some of these usecases might also prefer to persist the entire history of records. Keeping all this in mind, the Online *NamedQueue framework* is designed to provide the in-memory ring-buffer per usecase and also provide an optional built-in persistence layer.

Admin API:

```
List<LogEntry> getLogEntries(Set<ServerName> serverNames, String logType,
  ServerType serverType, int limit, Map<String, Object> filterParams)
  throws IOException;
```

getLogEntries API is used by client to retrieve online (recent history) records from the ring-buffer queue maintained at master or regionservers. This API is quite *generic* and used by all usecases that are served by the NamedQueue framework.

Params:

1. **logType:** This is the parameter to determine which usecase we are referring to. Based on the log type we provide, a particular ring-buffer queue is used to retrieve data. Log type is also referred to as the type of the usecase client is interested in. So far, we support four log types (usecases):
    a. **LARGE_LOG** - recent too large RPC requests based on the size of the payload
    b. **SLOW_LOG** - recent too slow RPC requests based on the processing time taken by regionserver
    c. **BALANCER_DECISION** - recent decisions taken by the balancer in the active master
    d. **BALANCER_REJECTION** - recent decision factors (including cost factors) used by the balancer to reject the need for balancing
2. **serverNames:** This parameter is used to provide target regionservers to retrieve ring-buffer records from. If the usecase retrieves records from the ring-buffer deployed at active master, this parameter is not used as we have only one active master to get the data. On the other hand, the usecase served by ring-buffers running at regionservers, we should provide selective set of servers to get the records. This is particularly useful when analyzing some recent events that might have more impact on the particular set of servers based on the factors like network bandwidth, rack distribution, hardware or operating system issues.
3. **serverType:** The usecase is either served by active master or regionservers. Hence this parameters takes enum values: MASTER and REGION_SERVER for Admin APIs to make RPC calls to either master or regionservers.
4. **limit:** Provide a limit to the number of records that server should send in response instead of returning all records available in the queue.
5. **filterParams:** Additional filter params to be used at server side. While retrieving the data from the ring-buffer queue, client might be interested in specific filter only, in which case, server can return the filtered data only.

The API returns list of LogEntry objects, where each LogEntry represents online records retrieved from servers for the given usecase.

Server side design:

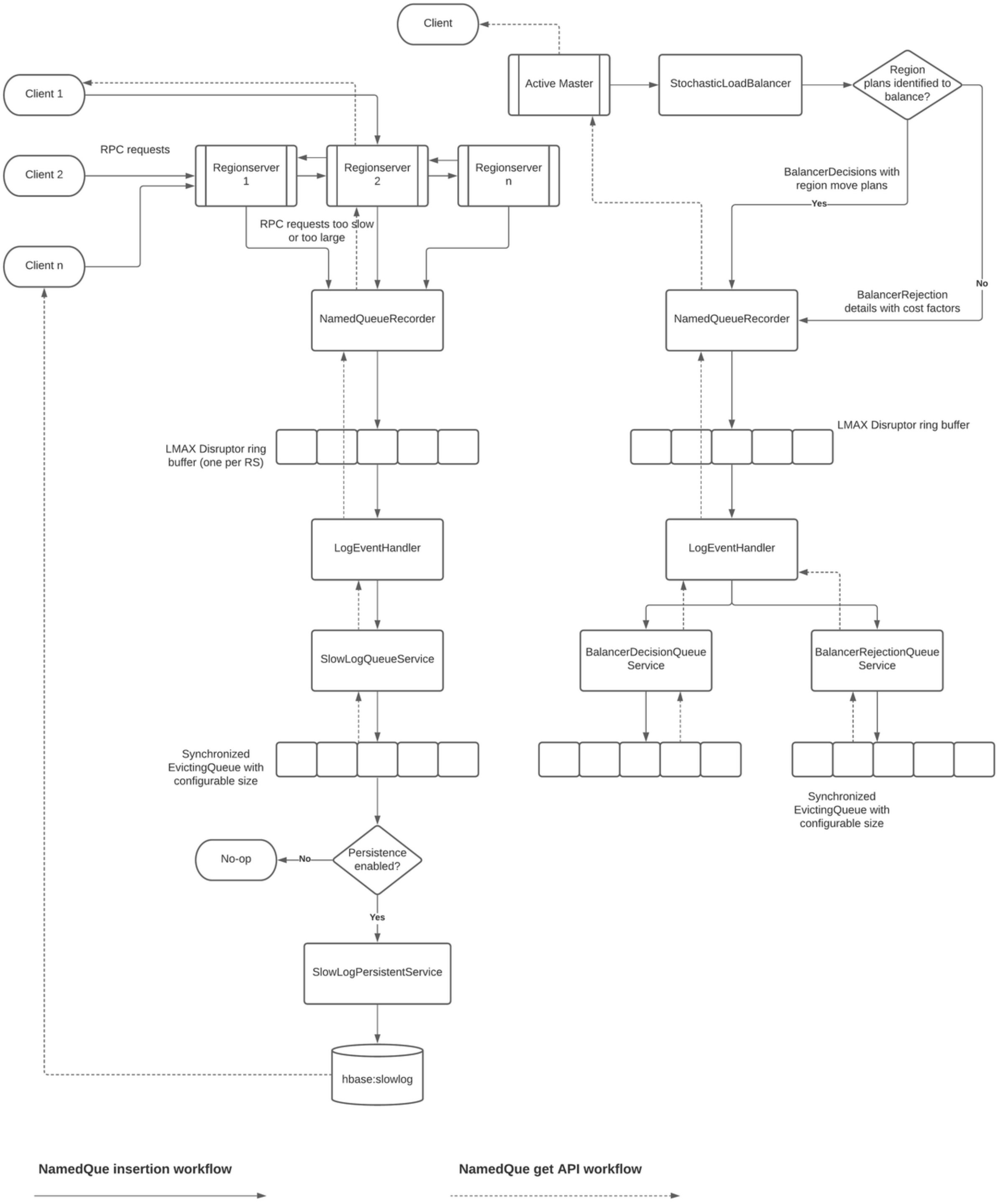
If the NamedQueue usecases are enabled using the relevant configurations, active master and/or all regionservers depending on the usecase type enabled, will create a singleton instance of *NamedQueueRecorder* as part of the server's initialization process. NamedQueueRecorder provides the generic APIs for active master and all regionservers to add/delete/retrieve ring-buffer records for all usecases. For instance, whether active master is trying to add balancer decision or rejection records or if regionservers are inserting new records for recently observed too slow or too large RPC requests, the same API *NamedQueueRecorder#addRecord* is used.

NamedQueueRecorder initializes an LMAX Disruptor ring-buffer instance. LMAX Disruptor is high performance alternative to bounded queues for exchanging data between concurrent threads (more details here). It is a framework which has "mechanical sympathy" for the hardware it's running on, and that's lock-free. We already use LMAX Disruptor in the write-ahead-log (WAL) implementations (FSHLog and AsyncFSWal) to publish append and sync events.

LogEventHandler is the event handler for the records published to LMAX Disruptor using NamedQueueRecorder. It is LogEventHandler that determines the type of the event being consumed and redirects to the consumer of the event based on the type of the event (i.e. usecase). The final consumer of the event should be an implementor of *NamedQueueService*. Each usecase of the NamedQueue framework has it's own implementor of the NamedQueueService. NamedQueueService usually has a synchronized Evicting queue with configurable size to save the incoming records in-memory. This is the final queue from where the client can read the elements from.

If the optional persistence layer is required, the NamedQueueService implementation can use an additional service layer to provide the usecase specific permanent storage and add the records to it after they are inserted in the ring-buffer queue. Slow/LargeLog usecase prefers system table to persist all entries, the system table access is provided by *SlowLogPersistentService* (HBASE-23938). The similar implementation can be extended for other potential usecases. Persistence service layer is responsible for permanent storage of records unlike the in-memory ring-buffer queue where the queue is used to access the recent records only and hence, the persistence service can use any storage - HDFS FileSystem, S3 like object stores, HBase system tables, Zookeeper, Bookkeeper etc. Hence, this service layer is flexible and new usecase should provide their own persistence layer with the custom logic. SlowLogPersistentService use hbase:slowlog system table. The benefit in the case of using system table is that we can reuse hbase-client Table APIs for CRUD operations and operators can scan the table using shell commands.

Here is the in-depth data flow diagram for Slow/LargeLog as well as Balancer decision/rejection usecases:


Client
Active Master
StochasticLoadBalancer
Region plans identified to balance?
Client 1
RPC requests
Client 2
Regionserver 1
Regionserver 2
Regionserver n
Client n
RPC requests too slow or too large
BalancerDecisions with region move plans
Yes
No
BalancerRejection details with cost factors
NamedQueueRecorder
NamedQueueRecorder
LMAX Disruptor ring buffer (one per RS)
LMAX Disruptor ring buffer
LogEventHandler
LogEventHandler
SlowLogQueueService
BalancerDecisionQueue Service
BalancerRejectionQueue Service
Synchronized EvictingQueue with configurable size
Synchronized EvictingQueue with configurable size
No-op
No
Persistence enabled?
Yes
SlowLogPersistentService
hbase:slowlog
NamedQue insertion workflow
NamedQue get API workflow

HBase shell commands:

Shell commands are provided per usecase that utilize the same Admin API getLogEntries() as mentioned above.

1. get_slowlog_responses:

```
hbase> get_slowlog_responses '*'

(get slowlog responses from all RS)


hbase> get_slowlog_responses '*', {'LIMIT' => 50}

(get slowlog responses from all RS with 50 records limit,
default limit: 10)


hbase> get_slowlog_responses ['SERVER_NAME1', 'SERVER_NAME2']

(get slowlog responses from SERVER_NAME1, SERVER_NAME2)


hbase> get_slowlog_responses '*', {'REGION_NAME' => 'hbase:meta,,1'}

(get slowlog responses only related to meta region)


hbase> get_slowlog_responses '*', {'TABLE_NAME' => 't1'}

(get slowlog responses only related to t1 table)


hbase> get_slowlog_responses '*', {'CLIENT_IP' => '192.162.1.40:60225',
 'LIMIT' => 100}

 (get slowlog responses with given client IP address and get 100 records
  limit, default limit: 10)


hbase> get_slowlog_responses '*', {'REGION_NAME' => 'hbase:meta,,1',
 'TABLE_NAME' => 't1'}

 (get slowlog responses with given region name or table name)


hbase> get_slowlog_responses '*', {'USER' => 'user_name',
 'CLIENT_IP' => '192.162.1.40:60225'}

 (get slowlog responses that match either provided client IP address
  or user name)
```

2. get_largelog_responses:

```
hbase> get_largelog_responses '*'

[get largelog responses from all RS]


hbase> get_largelog_responses '*', {'LIMIT' => 50}
```

```
[get largelog responses from all RS with 50 records limit
 (default limit: 10)]


hbase> get_largelog_responses ['SERVER_NAME1', 'SERVER_NAME2']

[get largelog responses from SERVER_NAME1, SERVER_NAME2]


hbase> get_largelog_responses '*', {'REGION_NAME' => 'hbase:meta,,1'}

[get largelog responses only related to meta region]


hbase> get_largelog_responses '*', {'TABLE_NAME' => 't1'}

[get largelog responses only related to t1 table]


hbase> get_largelog_responses '*', {'CLIENT_IP' => '192.162.1.40:60225',
 'LIMIT' => 100}

 [get largelog responses with given client IP address and get
 100 records limit (default limit: 10)]


hbase> get_largelog_responses '*', {'REGION_NAME' => 'hbase:meta,,1',
 'TABLE_NAME' => 't1'}

 [get largelog responses with given region name or table name]


hbase> get_largelog_responses '*', {'USER' => 'user_name',
 'CLIENT_IP' => '192.162.1.40:60225'}

 [get largelog responses that match either provided client IP address
  or user name]
```

3. get_balancer_decisions:

```
hbase> get_balancer_decisions
[Retrieve recent balancer decisions with region plans]


hbase> get_balancer_decisions LIMIT => 10
[Retrieve 10 most recent balancer decisions with region plans]
```

4. get_balancer_rejections:

```
hbase> get_balancer_rejections
[Retrieve recent balancer rejections with region plans]
```

```
hbase> get_balancer_rejections LIMIT => 10
[Retrieve 10 most recent balancer rejections with region plans]
```

Configurations:

Let’s take a look at the configurations available for the existing usecases:

```
Key: hbase.regionserver.slowlog.buffer.enabled
Default value: false
Description:
Indicates whether RegionServers have ring buffer running for storing
Online Slow logs in FIFO manner with limited entries. The size of
the ring buffer is indicated by config: hbase.regionserver.slowlog.ringbuffer.size
The default value is false, turn this on and get latest slowlog
responses with complete data.
```

```
Key: hbase.regionserver.slowlog.ringbuffer.size
Default value: 256
Description:
The size of ringbuffer to be maintained by each RegionServer in order
to store online slowlog responses. This is an in-memory ring buffer of
requests that were judged to be too slow in addition to the responseTooSlow
logging. The in-memory representation would be complete.
```

```
Key: hbase.regionserver.slowlog.systable.enabled
Default value: false
Description:
Should be enabled only if hbase.regionserver.slowlog.buffer.enabled is enabled.
If enabled (true), all slow/large RPC logs would be persisted to system table
hbase:slowlog (in addition to in-memory ring buffer at each RegionServer).
The records are stored in increasing order of time.
Operators can scan the table with various combination of ColumnValueFilter.
```

```
Key: hbase.master.balancer.decision.buffer.enabled
Default value: false
Description:
Indicates whether active HMaster has ring buffer running for storing
balancer decisions in FIFO manner with limited entries. The size of
the ring buffer is indicated by config: hbase.master.balancer.decision.queue.size
```

```
Key: hbase.master.balancer.rejection.buffer.enabled
Default value: false
Description:
Indicates whether active HMaster has ring buffer running for storing
balancer rejection in FIFO manner with limited entries. The size of
the ring buffer is indicated by config: hbase.master.balancer.rejection.queue.size
```

Jiras:

Slow/Large RPC Log and Balancer decision details getting persisted in the in-memory ring-buffer with optional persistence layer has been implemented as part of these Jiras:

1. HBASE-22978 Online slow response log
    a. HBASE-23936 Thrift support for get and clear slow_log APIs
    b. HBASE-23937 Retrieve online large RPC logs
    c. HBASE-23938 Replicate slow/large RPC calls to HDFS
    d. HBASE-23941 get_slowlog_responses filters with AND/OR operator support
2. HBASE-24528 Improve balancer decision observability
3. HBASE-24718 Generic NamedQueue framework for recent in-memory history
4. HBASE-25790 NamedQueue 'BalancerRejection' for recent history of balancer skipping