

Streams Runtime Architecture with New Threading

- [Goals and Key Ideas](#)
- [Modular Design Breakdowns](#)
- [Table of Threads](#)
- [Core Procedures](#)
 - [New Task Creation](#)
 - [Existing Task Shutdown](#)
 - [Active Task Scheduling](#)
 - [State Management for Active Processing Tasks](#)
 - [Committing Active Tasks](#)
- [Implementation Milestones](#)

This page summarizes the detailed design of the new Kafka Streams runtime with the refactored threading model.

Goals and Key Ideas

Just a quick recap on the motivations of this new design

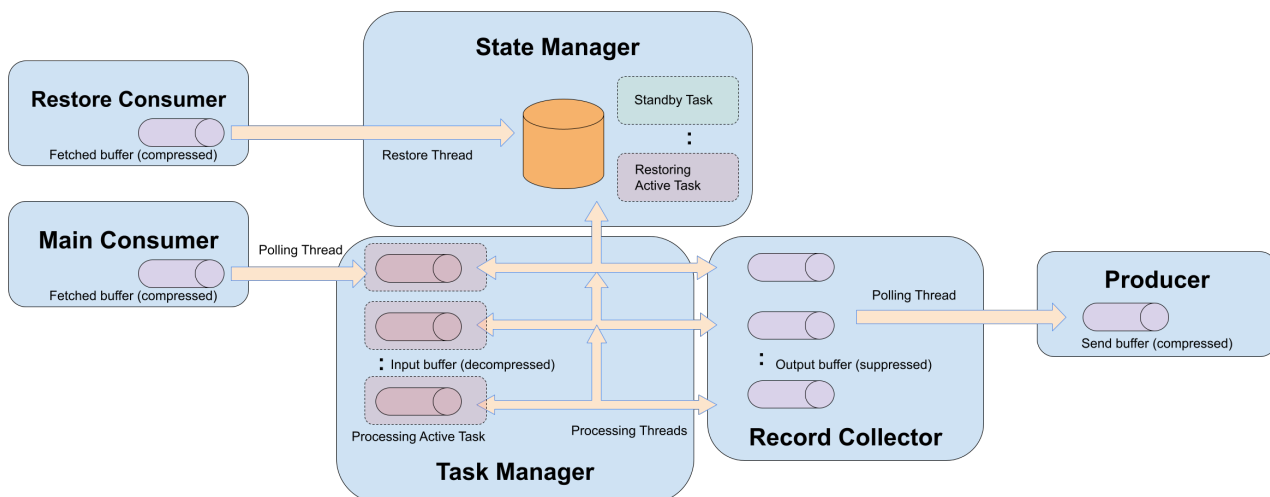
1. Improve cost-effectiveness with high utilization of allocated pod resources (CPU and memory primarily). E.g. we should be able to efficiently saturate the resources of the given pods before considering to scale and add more pods.
2. Optimize operational knobs that users need to learn about to make the runtime performant and stable. E.g. users should not need to worry about the embedded client's configuration and stay with the default values most of the time.
3. Enable scheduling prioritization and isolation between query workloads within a cluster / app.
4. A better framework to integrate with cloud-native architectures, E.g. remote state management.

And the key ideas behind the design to tackle on the above goals:

1. Move from thread-dedicated Kafka clients to **shared clients** within an instance, with reduced num.connections and improved batching => goal 1).
2. **Dedicated thread** to interact with Kafka clients for blocking calls and misc workloads => goal 1), 2).
3. Dynamic task assignment with (optional) prioritized scheduling among processing threads => goal 1), 3).
4. **Move state restoration workload** from processing threads to a centralized state manager with dedicated threads => goal 2).
5. **Move caching / state management** into the centralized state managers as well, which may interact with a remote state service => goal 2), 4).

With the above key ideas in mind, below is the diagram of a single runtime instance (please compare it with the [status-quo architecture](#) of a single KS runtime), where:

- Blue-boxes are modules that interact with each other and accessed by threads,
- Yellow arrows indicate threads doing CPU work and move data between modules,
- Dotted rectangles represent the unit of workload i.e. tasks,
- Horizontal logs represent in-memory buffers, and
- Vertical orange logs represent physical storage engines.



Modular Design Breakdowns

Each Kafka Streams runtime instance would contain the following modules:

1. One **main consumer**, used to fetch input partition records for assigned active tasks from the Kafka brokers. It has a fetched records buffer which contains the compressed record batches fetched by the consumer's background thread. It is also used to participate in the rebalance protocol to assign tasks among instances.
2. One **restore consumer**, used to fetch changelog records for assigned standby tasks as well as restoring active tasks from the Kafka brokers. Similar as the main consumer it also has a buffer of compressed record batches that is fetched by the consumer's background thread.
3. A **task manager**, which:
 - a. Maintains the list of assigned active tasks that are ready to be processed (i.e. have completed initialization and restoration).
 - b. For each of the maintained active tasks, keeps a decompressed input buffer for its fetched records polled from the main consumer. These records are polled and decompressed by the polling thread (see below).
 - c. Has a pool of processing threads along with a scheduler that determines dynamically which threads will process which assigned active tasks within the task manager. At any given time, each processing thread will process at most a single task, interact with the state manager to update the task's states, and send result records as well as changelog records to the record collector. These threads will switch between active tasks periodically (see below).
4. A **state manager**, which:
 - a. Manages the state stores of all the assigned tasks. Logical states across multiple tasks can be maintained as consolidated physical stores.
 - b. Has a single restore thread that polls compressed record batches from the restore consumer and apply to the restoring standby and active tasks.
 - c. Interacts with the remote state management service to checkpoint local states to remote stores, and download state from remote storages upon restoration (see [this doc](#) for details.).
5. A **record collector**, which:
 - a. Maintains the output buffer per outgoing partition (both changelog as well as sink).
 - b. Suppress changelog records when possible before sending to the producer.
 - c. The same polling thread would drain from the output buffer and send to the producer.
6. A single **producer** that gets records from the polling thread, accumulates and compresses in its own send buffer, and finally send the compressed record batches to destination brokers.

Table of Threads

The following table summarizes all the threads in the new runtime. Within the **Work** column, the major resource consumption workload are highlighted.

Thread Type	Number	Interacting Modules	Work
Processing thread	N (configurable)	Task Manager State Manager Record Collector	Runs in iterations until being shutdown. Periodically: <ul style="list-style-type: none">• Takes the assigned task from the task manager's scheduler. Grab the lock on the task and release locks of the previously processing task. Within an iteration: <ul style="list-style-type: none">• Read record from task's buffer, process the record, update the state of the task with the state manager (CPU and state update IO).• Put the changelog and result records into the record collector's buffer.• Check if the processing is paused by the pooling thread in order to commit the task.

Polling thread	1 (not configurable);	Main Consumer Task Manager Producer Record Collector	<p>Runs in iterations until being shutdown.</p> <p>Within an iteration:</p> <ul style="list-style-type: none"> Fetch from main consumer's compressed record batch buffer, put the polled record into corresponding task's input buffers (CPU decompressing). Peek into the record collector's output buffer, drain the records when necessary, and send to producer's accumulator. Check the exception queues from other threads and handle accordingly (see the other doc for details). <p>During the main consumer's poll call, it may:</p> <ul style="list-style-type: none"> Trigger rebalance callbacks, in which it will: <ul style="list-style-type: none"> Create/recycle and initialize newly assigned tasks, and sent the newly created tasks into the state manager if restoration is necessary; otherwise send them to the task manager. Close those revoked tasks from the task managers. Commit tasks upon requests or on timely manners, in which it will: <ul style="list-style-type: none"> Notify all the stream threads to stop processing their current assigned tasks. For those tasks that have stopped processing, flush their state with the state manager (state IO). Drain the record collector and flush the producer to make sure all outgoing records are sent and acked. Execute the commit procedure (either EOS or ALOS, which would rely on different client's APIs, CPU on blocking calls) Notify all the stream threads to resume processing their tasks.
Consumer's background thread	2; one from the main consumer and one from the restore consumer	Main Consumer Restore Consumer	<ul style="list-style-type: none"> Doing IO on network socket to receive fetch responses (network IO) Put parsed record batch from read response into consumer's buffer Group rebalance related computations
Producer's background thread	1; from the producer	Producer	<ul style="list-style-type: none"> Doing IO on network socket to send produce requests (CPU compressing and network IO) Transaction related computations
Admin's background thread	1; from the admin	Admin	<ul style="list-style-type: none"> Doing IO on network socket to send various admin requests (network IO)
Restoration thread	1 (not configurable)	Restore Consumer State Manger	<ul style="list-style-type: none"> Get notified when new tasks are added / remove for restoration. Fetch from restore consumer's record batch buffer (CPU decompressing) May need to translate the record batch into state format if direct byte copy-paste cannot be done (CPU deserialization and reserialization) Doing IO to apply the translated state write-batch to the state stores (disk IO work)
Cleanup thread	1 (not configurable)	State Manger	<ul style="list-style-type: none"> Periodically check local state dir, and doing IO work to clean up states (disk IO)
RocksDB Metrics Triggering thread	1 (not configurable)	State Manger	<ul style="list-style-type: none"> Periodically read the stats object of the RocksDB instances and update the metrics registry

Core Procedures

In this section we describe the procedures of certain core events.

New Task Creation

Polling thread gets the newly assigned tasks from the rebalance callback.

- If the task does not exist at all, it will create and initialize the task. For standby tasks and active tasks that needs restoration, it will send them to the state manager.
 - During the initialization of the task, the local metadata checkpoint would be loaded first; if the checkpoint is not found, treat it as the metadata pointing at the beginning.
 - If the local state supports transactions, call the state to revert to the snapshot indicated by the loaded metadata.

- If the task exist as an active task and now assigned as a standby task, try to recycle it as standby: if the task was not in task manager (which means it's in the state manager), send a notification to the state manager to let it stop restoring the state and send it back to the task manager, and then after recycling it into a standby task, send it back to the state manager; otherwise directly recycle it inside the task manager and then send the task to the state manager.
- If the task exist as a standby task and now assigned as an active task, try to recycle it as active: the task has to be in the state manager for now, so send a notification event to the state manager to let it stop restoring the state and send it back to the task manager. The task manager would recycle it to an active task which would still be in the restoring state. The task would then be sent back to state manager to complete the restoration before it would be returned back to the task manager.

From the task's point of view, it should be either at the task manager, accessed by the polling thread / processing threads; or at the state manager, accessed by the restoration thread, but never at both.

Existing Task Shutdown

Polling thread gets the revoked tasks from the rebalance callback.

- If the task was not in the task manager (which means it's in the state manager, either a standby, or an active task which's still restoring), send a CLOSE event to the state manager. The tasks closure inside the state manager does not require committing the task, but only updating the state checkpoint.
- Otherwise, close the processing active task inside the task manager directly after committing them.

Active Task Scheduling

Active tasks inside the task manager are ready to be processed, and there is a stream thread pool that grabs those tasks dynamically and process them. Each task inside the task manager has an exclusive lock that can be grabbed by at most one thread at a time.

The scheduling algorithm can be extensible with new requirements such as the new to assign priorities among queries (and hence their corresponding tasks). For example, A simple scheduling algorithm would work in the following way:

- Each stream thread would periodically look into the task manager looking for tasks to process. The tasks would be sorted by the number of buffered input records, so that tasks with more buffered records would be picked first.
- The thread would grab the lock on the task so that no other threads would be process this selected task. It would then move on to process the task's records continuously, until 1) a pre-defined period of time has elapsed, or 2) the buffered input records have been exhausted. After that it will release the lock of the task and try to pick the next "high-priority" tasks.

State Management for Active Processing Tasks

The state management procedure would be "extracted" out of the tasks themselves and be handled within a consolidated module, a.k.a. the State Manager. Each task's processing still interacts with the "state store" APIs to read from/write into, and flush the states. But the actual implementations are provided by this State Manager which could optionally maintain multiple logical state stores (even from different tasks) into the same physical store engines.

The state store APIs are still layere but the layering would be slightly changed as "metered change-logged cached" where the caching layer is managed by the state manager. More specifically:

1. Stream thread would touch on the metered layer to trigger the recording of the state store metrics.
2. Stream thread would touch on the change-logged layer to send the changelog record into the record collector (note it would not call on the producer, as it's done by the polling thread).
3. Stream thread would touch on the state manager to finally update the state, and it's abstracted away from the thread whether it only reaches the cache or get to the persistent layer at all. Nevertheless, reads from the state should be able to return uncommitted states.

Committing Active Tasks

Any task's progress are tracked by the running metadata including "position" and "time", which is also used to identify the state snapshot of the task.



For standby tasks and restoring active tasks that are maintained in the state manager, they do not need to execute a full committing process. Instead, the state manager only need to periodically persist the advanced task metadata locally (when we have remote state management services, we may still need to persist the task's metadata which would guide the state local cache warmup process).


As for those active processing tasks within the task manager, since we only have a single producer/consumer pair to execute the committing process, we need to always commit all the tasks at once. More specifically, we need to execute the following steps when we want to commit:

- Flush the state of all active tasks within the task manager. If the state supports transactional updates, let the flush return a token which would be written as part of the task metadata to be persisted.
- Write the changelog record of the metadata into the record collector.
- Drain all the records inside the record collector and flush the producer.
- Trigger the corresponding producer / consumer API depending on the processing mode (EOS or ALOS) to complete the commit.
- Persist the advanced task metadata locally. **NOTE** this step can be done outside the EOS committing process and not be atomic, since even if we failed before this last step, we can still either 1) revert the local state to the last checkpoint and complete the restoration of the latest snapshot, or 2) bootstrap from the beginning to the latest snapshot.

Implementation Milestones

Here's a proposal for achieving the above architecture in a step-by-step manner, where each step still leaves the architecture as a workable state.

Step	Scope	Benefits	
Move restoration out of processing thread	<ol style="list-style-type: none">1. Add a state updater module, which would be a smaller scope of the final state manager.2. Create a restoration thread inside the state updater that fetches from the restore consumer and apply the batches to restore states.3. Maintain the state manager within each processing thread, so that we have a total of N state updater / restore thread, where N == number of stream threads.	Restoration would not impact processing thread from being kicked out of the group and triggering rebalance	<div> Unable to render Jira issues macro, execution on error.</div>
Introduce the processing threads	<ol style="list-style-type: none">1. Augment the record collector with the output buffer. Let the changelog records to be buffered in the record collector2. Add a total number of N processing threads, N == configured stream threads. The original stream thread would still be fetching from the consumer and put records into the input buffer, and draining records from the record collector and send to the producer.3. The stream thread still executes the committing procedure, and the stream assignor does not need to be modified.	Reduce the likelihood of long blocking due to record processing	<div> Unable to render Jira issues macro, execution on error.</div>

Complete KIP-588	<ol style="list-style-type: none"> 1. Complete KIP-588 on the broker side and add an internal config in producer to handle ProducerFenced as TransactionTimedOut 	Let the Streams to be more resilient with EOS error handling	 Unable to render Jira issues macro, execution on error.
Consolidate the polling threads into a single thread NOTE: this is a major upgrade barrier and would be better included in a major release.	<ol style="list-style-type: none"> 1. Only maintain one consumer/producer client per instance instead of per thread. 2. Reduce the number of restoration thread and the polling thread to one. 3. Simplify the task assignor to only do the per-client assignment 4. Within the client, let processing thread dynamically choose tasks to process 5. Poll thread would synchronize with processing threads to commit tasks. 	Reduce the number of embedded clients as stated above	
Refactor the task committing procedure and exception handling logic	<ol style="list-style-type: none"> 1. Refactor the shared record collector module to accumulate and suppress sending records. 2. Let the poll thread to take the records and send to the shared producer. 3. Decouple the state store caching from emitting, to always emit downwards. 4. Also include completing KIP-691. 	Make the runtime more resilient to errors.	
Move state stores into the state manager	<ol style="list-style-type: none"> 1. Augment the state updater module added in the "Move restoration out of processing thread" step, to manage all physical state stores of all tasks (including both restoring and active processing). 2. Integrate with the proposed metadata management inside the state manager. 	Efficient IO and less footprint in physical state stores. Integration with proposed cloud state storage.	