**Q&A**

**Q1. How to optimize aggregation join operations, is there a way to give additional memory using some settings in topology? or any other recommendations**

**ANS 1:**

To **optimize aggregation and join operations in Kafka Streams**, especially when dealing with **stateful operations** (e.g., join(), aggregate(), reduce()), you can tune **memory**, **topology**, and **operational settings**.

Below is a structured guide to optimize your Kafka Streams application:

**1. Increase Available Memory for State Stores**

**a. RocksDB Settings (default state store backend)**

Kafka Streams uses **RocksDB** for local state. You can tweak its cache size:

props.put("rocksdb.config.setter", "com.your.package.CustomRocksDBConfig");

In your custom class:

public class CustomRocksDBConfig implements RocksDBConfigSetter {

@Override

public void setConfig(final String storeName, final Options options, final Map<String, Object> configs) {

options.setWriteBufferSize(64 \* 1024 \* 1024L); // 64 MB

options.setMaxWriteBufferNumber(3);

options.setMaxBackgroundCompactions(4);

}

}

You can also increase the heap available to your JVM if memory is a bottleneck.

**2. Tune Memory Configurations for Kafka Streams**

Set these configs in your Properties:

| **Property** | **Description** | **Default** | **Suggested Tuning** |
| --- | --- | --- | --- |
| cache.max.bytes.buffering | Cache size before writing to state store | 10 MB | 50MB–200MB |
| buffered.records.per.partition | Max records to buffer per partition | 1000 | 5000+ (test) |
| commit.interval.ms | Commit frequency for state store | 30,000 ms | 1000–5000 ms |

props.put(StreamsConfig.CACHE\_MAX\_BYTES\_BUFFERING\_CONFIG, 100 \* 1024 \* 1024L); // 100 MB

props.put(StreamsConfig.BUFFERED\_RECORDS\_PER\_PARTITION\_CONFIG, 5000);

props.put(StreamsConfig.COMMIT\_INTERVAL\_MS\_CONFIG, 5000);

**3. Optimize Join Strategy**

**a. Use Grace Period with Windowed Joins**

JoinWindows.of(Duration.ofMinutes(5)).grace(Duration.ofMinutes(1));

Grace period helps deal with out-of-order events more efficiently.

**b. Choose Correct Join Type**

* leftJoin vs innerJoin: Use leftJoin if right side may be missing to avoid filtering out useful data.
* Prefer **join() on compacted topics** if the right side is a KTable.

**Q2.**

**As the statefull set operations require memory, how can we ensure we give the correct amount of memory for Kafka streams application so that it doesn't run out of memory? If we store two ktables with 10 million records each with size 5Gb, do we need 10Gb memory just for ktables? Does Kafka streams needs additional memory for its internal processing on top of that?**

**Kafka Streams does require additional memory beyond the raw data size** stored in KTables. Storing two KTables with 10 million records each (5 GB each) **does not mean you only need 10 GB of memory**. Here's why — and how to plan the required memory properly.

**Understanding Memory Usage in Kafka Streams**

Kafka Streams memory consumption comes from multiple areas:

| **Memory Component** | **Description** |
| --- | --- |
| **State store (RocksDB)** | Stores KTable data on disk (not in-heap), uses off-heap/native memory |
| **In-memory caching** | Configured via cache.max.bytes.buffering — used before flush to RocksDB |
| **Record queues / buffering** | Per-task queues hold incoming records |
| **Thread-local JVM heap** | Used for serde, deserialization, and temporary objects |
| **RocksDB native memory** | Uses off-heap memory; you must consider it in overall memory planning |

**Example: Memory Estimation for 2 KTables**

**Scenario:**

* 2 KTables with 10 million records each (~5 GB per KTable)
* Each record: 500 bytes average (key + value)
* JVM heap: 4–8 GB
* cache.max.bytes.buffering: 512 MB
* RocksDB write buffer size: 64 MB \* 3 = 192 MB per store

**Estimated Memory Needs:**

| **Component** | **Memory Need** |
| --- | --- |
| RocksDB (native memory) | 2 KTables × 192 MB ≈ 384 MB |
| Caching | Up to 512 MB |
| JVM Heap (overhead, serde) | 4–6 GB recommended |
| Record queues | 100–500 MB |
| OS + Kafka clients overhead | 500 MB–1 GB |
| **Total memory required** | **6–10 GB (excluding raw disk)** |

**Disk, not RAM, stores the full 5 GB per KTable.** RAM is mainly used for caching, buffering, and operational state.

Q3.

how can we calculate the max number of threads to be configured in a Kafka streams application? If a Kafka stream application is reading from x number of topics and producing to y number of topics, what should be the number of threads configuration given to achieve maximum throughput.

ANS:

To determine the **maximum number of threads** to configure in a Kafka Streams application (num.stream.threads), you must balance **parallelism**, **resource usage**, and **Kafka partitioning**.

**Short Answer**:

The maximum effective number of Kafka Streams threads is typically **equal to the total number of input partitions across all source topics** — assuming you have enough CPU and memory resources.

**Key Principle**

Kafka Streams parallelism is **partition-based**, not topic-based.

* Each thread can **run multiple stream tasks** (1 per partition).
* But **each partition can be processed by only one thread**.
* So, the **maximum useful parallelism = total number of partitions across all source topics**.

**Example:**

| **Topic** | **Partitions** |
| --- | --- |
| topicA | 6 |
| topicB | 4 |

**Total input partitions = 6 + 4 = 10**

You can configure up to num.stream.threads = 10 to fully utilize parallelism.

**Configuration**

props.put(StreamsConfig.NUM\_STREAM\_THREADS\_CONFIG, 10);

But in reality:

* If you only have **4 CPU cores**, setting 10 threads may not improve performance.

Q4.

if we use this latch with 1 as given in lab exercise, will it gracefully handle Kafka stream exits during situations like force kill, out of memory or low cpu? final CountDownLatch latch = new CountDownLatch(1);

Allows for **graceful shutdown** when:

* SIGTERM or SIGINT is sent (Ctrl+C, docker stop, etc.)
* JVM exits normally
* You call .close() manually

Ensures streams.close() is called, triggering:

* Persistent store flush
* Commit of offsets
* Shutdown of internal threads

**What It *Cannot* Handle**

| **Failure Type** | **Handled Gracefully?** | **Why** |
| --- | --- | --- |
| kill -9 (SIGKILL) | No | JVM has no chance to execute shutdown hooks |
| OutOfMemoryError (OOM) | No | JVM may be in unstable state; shutdown hooks may not run |
| CPU starvation or freeze | No | Threads may never get a chance to run |
| Kernel-level resource kill (e.g., OOM killer) | No | Process is terminated abruptly |

**Conclusion**

CountDownLatch(1) is good for clean shutdowns in **normal conditions**, but **not sufficient for hard failure scenarios**.

To build production-grade Kafka Streams apps:

* Pair the latch with proper **error handling**, **resource limits**, **supervision**, and **observability**.

Q5.

Interactive Query in RockDB

To **query RocksDB** in a **Kafka Streams** application, you use **Interactive Queries**, not by accessing the RocksDB database directly. Kafka Streams abstracts the local state store (which is powered by RocksDB) and exposes it via a **Java API** — or optionally via a **REST API** if you build one.

**Two Ways to Query RocksDB in Kafka Streams**

**1. Programmatic Access via Kafka Streams API**

Kafka Streams provides a method to **query local state stores**:

ReadOnlyKeyValueStore<String, Long> store =

streams.store(

StoreQueryParameters.fromNameAndType("word-count-store", QueryableStoreTypes.keyValueStore())

);

Long count = store.get("kafka");

* store(...): Fetches the local state store.
* get(key): Fetches the value for a key from the RocksDB-backed store.
* range(fromKey, toKey): For iterating over a key range.
* all(): Iterate over all key-value pairs.

**2. REST API on Top of Kafka Streams (Interactive Query Service)**

You can expose the above store using an HTTP server:

get("/count/:word", (req, res) -> {

String word = req.params(":word");

Long count = store.get(word);

return count != null ? count : 0;

});

This is what was demonstrated in the lab exercise earlier. It's the **recommended way to interactively query RocksDB** in a running Kafka Streams instance.

**Direct Access to RocksDB? (Not Recommended)**

Technically, you can access RocksDB files under:

/tmp/kafka-streams/<application-id>/<task-id>/rocksdb/

But:

* **Not portable**: RocksDB files are local to each Kafka Streams instance.
* **Not safe**: Accessing these files directly while Kafka Streams is running may lead to corruption.
* **No schema**: You must deserialize keys and values manually using SerDes.

**Avoid accessing RocksDB directly.** Use the **Kafka Streams API**.

**Sample: Iterate All Records in Store**

KeyValueIterator<String, Long> all = store.all();

while (all.hasNext()) {

KeyValue<String, Long> entry = all.next();

System.out.println(entry.key + " => " + entry.value);

}

all.close();

**Use Cases for Querying RocksDB**

| **Use Case** | **Example** |
| --- | --- |
| REST API for dashboards | GET /count/kafka |
| Real-time anomaly detection | Query counters on the fly |
| Interactive analytics tools | Show live word counts |
| Stateful alerting systems | Check thresholds on demand |

A screenshot of a computer

AI-generated content may be incorrect.

ANS

Stateless transformations don’t require a **state store** by default, but you **can materialize the result** (i.e., explicitly write the output to a named state store) to make it **queryable** via **Kafka Streams Interactive Queries**.

**Explanation with Example**

Suppose you're processing a stream of user updates and filtering out inactive users using a **stateless operation**, such as filter.

**Example: Stateless Operation + Materialization**

StreamsBuilder builder = new StreamsBuilder();

// Imagine this is a stream of user records

KTable<String, User> users = builder.table("users-topic");

// Stateless transformation: filter out inactive users

KTable<String, User> activeUsers = users

.filter((userId, user) -> user.isActive(),

Materialized.<String, User, KeyValueStore<Bytes, byte[]>>as("active-users-store") // <--- Materialize here

.withKeySerde(Serdes.String())

.withValueSerde(userSerde)

);

Even though filter is a **stateless transformation**, by passing .as("active-users-store"), you're **materializing** the result into a **state store** named "active-users-store".

**Why Materialize?**

By default, a stateless transformation like filter **does not create a state store** because it doesn’t need one. But when you **materialize**:

1. Kafka Streams writes the filtered results into a local **RocksDB**-backed state store.
2. You can then use **Interactive Queries** to access this store from within your app or through a REST API.

**Interactive Query Access**

ReadOnlyKeyValueStore<String, User> store = streams.store(

StoreQueryParameters.fromNameAndType("active-users-store", QueryableStoreTypes.keyValueStore())

);

User u = store.get("user-1234");

This lets you **query the most recent filtered result** of any given user.

**Why Excessive Joins Are Problematic**

| **Concern** | **Impact** |
| --- | --- |
| **State Storage** | Each join typically requires **internal state stores** (especially for windowed joins), increasing disk and memory usage |
| **Repartitioning** | Joins on non-keyed fields trigger **repartition topics**, increasing network and I/O load |
| **Latency** | Each join adds **processing overhead** and may delay output |
| **Error Handling** | Harder to trace and recover from partial failure across joined streams |
| **Operational Complexity** | Debugging, monitoring, and scaling becomes more difficult with more joins |