**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 stateful 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.

**Q. Session Window Example**

**Input Topic (clicks) Example**

| **Key** | **Value** | **Timestamp** |
| --- | --- | --- |
| user1 | click1 | 2024-06-22T10:00:00 |
| user1 | click2 | 2024-06-22T10:01:00 |
| user1 | click3 | 2024-06-22T10:06:00 |

**Behavior**:

* click1 and click2 will form **one session** (within 2 mins)
* click3 will start a **new session** (gap > 2 mins)

**Output Example**

User=user1, Start=2024-06-22T10:00:00, End=2024-06-22T10:01:00, Count=2

User=user1, Start=2024-06-22T10:06:00, End=2024-06-22T10:06:00, Count=1

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 |

Determining the **optimal number of Kafka Streams instances, threads, and tasks** is critical for performance, scalability, and fault tolerance. Here's a structured, practical guide to help you **decide the right configuration**.



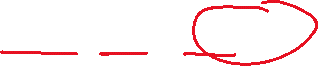
**1. Understand the Core Units of Kafka Streams**



| **Component** | **Description** |
| --- | --- |
| **Partition** | Kafka’s fundamental unit of parallelism; each partition maps to **1 task** |
| **Task** | A unit of work assigned to a **thread** that consumes one or more partitions |
| **Thread** | Executes multiple tasks in parallel within one instance |
| **Instance** | A running copy of your Kafka Streams app (process or container) |

**2. Rule of Thumb for Configuration**

**🔹 Total Number of Tasks = Number of Partitions in Input Topics**



Kafka Streams creates **1 task per input partition (or partition group in joins)**.

**If input topic has 12 partitions ⇒ 12 tasks will be created**



**🔹 Minimum Threads per Instance = (Tasks per Instance)**



Each thread can process **multiple tasks**, but threads do **not** scale beyond partition count.



Having more threads than tasks is wasteful; more tasks than threads is okay — they get queued.



**3. Key Formula**



Max Parallelism = Number of Input Partitions



Therefore:

* **Total Tasks = Total Input Partitions**



* **Total Threads (across all instances) ≤ Total Tasks**



* **Total Instances × Threads per Instance = Total Threads**



**4. Guidelines to Decide**



| **Metric** | **Recommendation** |
| --- | --- |
| **#Partitions** | Plan based on expected throughput and parallelism needs (more = better scaling) |
| **#Tasks** | Equal to #Partitions (or fewer if repartitioning happens) |
| **#Threads/Instance** | Start with 2–4 per instance; monitor CPU & throughput |
| **#Instances** | Scale horizontally to match or exceed tasks count (for HA and throughput) |

**Example Scenario**

* Topic: orders



* Partitions: 12



* You want high availability and performance



**Suggested Deployment**

| **Config** | **Value** |
| --- | --- |
| Total Tasks | 12 (1 per partition) |
| Threads per Instance | 2 |
| Number of Instances | 3 to 6 (to scale and allow HA) |
| Total Threads | 3 instances × 2 threads = 6 threads (each thread handles 2 tasks) |

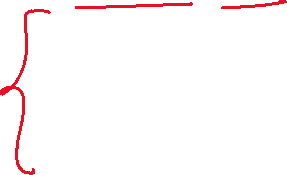
This means 12 tasks are distributed across 6 threads → 2 tasks per thread.



**5. Monitoring and Tuning**

Use these **Kafka Streams metrics** for tuning:

* active-task-count
* standby-task-count
* record-consumption-rate



* process-rate
* commit-latency-avg
* JVM CPU and memory usage

**Scale out if:**

* Threads are CPU-bound
* Task lag is high
* Throughput is bottlenecked

**6. Recommendations**



| **Objective** | **Strategy** |
| --- | --- |
| **Throughput** | Increase partitions and instances to parallelize tasks |
| **Low latency** | Use fewer partitions per thread, balance CPU/thread usage |
| **Fault tolerance** | Use 2+ instances for standby tasks and automatic failover |
| **Cost efficiency** | Match instance/thread count to actual data volume and partition count |

**Extended Scenario**

**Kafka Topic: orders**

* **Partitions**: 6 (orders-0 to orders-5)
* Each partition will be assigned to one **task**  
  → So, **6 tasks** in total.

**Kafka Streams Configuration:**

| **Parameter** | **Value** |
| --- | --- |
| Application Instances | 2 (on separate machines) |
| Stream Threads per Instance | 2 threads |

So you have:

* 2 instances × 2 threads = **4 threads total**
* 6 tasks to distribute among these 4 threads

**How Tasks are Distributed**

Kafka Streams will automatically assign tasks (based on partitions) across instances and threads during rebalance.

**Example Assignment:**

| **Machine** | **Instance** | **Thread** | **Tasks (Partition → Task)** |
| --- | --- | --- | --- |
| M1 | App-Instance-1 | Thread-1.1 | Task-0 (P0), Task-1 (P1) |
|  |  | Thread-1.2 | Task-2 (P2) |
| M2 | App-Instance-2 | Thread-2.1 | Task-3 (P3), Task-4 (P4) |
|  |  | Thread-2.2 | Task-5 (P5) |

**Q. what is best practice for effective topology**

Designing an **effective Kafka Streams topology** is crucial for building scalable, maintainable, and performant stream processing applications. A well-structured topology improves code clarity, reduces unnecessary data movement, and ensures efficient resource usage.

Here are the **best practices for designing an effective Kafka Streams topology**:



**1. Keep the Topology Modular and Composable**



* Break down the topology into **logical modules**, each handling a specific responsibility (e.g., filtering, enrichment, aggregation).



* Use helper methods or classes to encapsulate parts of the topology:

KStream<String, Order> filtered = filterInvalidOrders(stream);

KStream<String, EnrichedOrder> enriched = enrichOrders(filtered);

* This makes the topology easier to test, maintain, and extend.

**2. Preserve Keys Wherever Possible**



* Many operations (groupByKey, join, aggregate) depend on a correct key.



* Avoid operations like map() that may lose key semantics unless intentional.
* Use mapValues() when you don’t need to change the key:

stream.mapValues(value -> transform(value))

**3. Minimize Repartitioning**



* Repartitioning is expensive: it creates intermediate topics and adds latency.
* Avoid unnecessary groupBy() or selectKey() unless the key change is absolutely needed.
* If repartitioning is required, name the intermediate topics for visibility:

stream.selectKey((key, value) -> value.customerId)

.through("repartitioned-orders-topic")

**4. Use Materialized Views Wisely**

* When aggregating or counting, materialize the state with a named store:

groupedStream.count(Materialized.as("order-count-store"));



* Helps with debugging and enables interactive queries (optional).

**5. Use branch() for Conditional Logic**



* Instead of multiple filter() chains, split the stream cleanly using branch():

KStream<String, Order>[] branches = stream.branch(

(key, value) -> value.isHighPriority(),

(key, value) -> true

);

**6. Design for Scalability**

* Align topology design with **Kafka partitioning**:
  + Use consistent keys so that partitions are evenly distributed.
  + Avoid using static or low-cardinality keys (like country = "IN" for all records).
* Use **stateless operations** (filter, map) early in the pipeline to reduce volume before aggregation.

**7. Join Operations: Co-Partitioned and Time-Windowed**

* Make sure topics in joins are **co-partitioned** and keyed appropriately.
* Define **suitable window sizes and grace periods** for time-based joins:

streamA.join(streamB, ..., JoinWindows.ofTimeDifferenceWithNoGrace(Duration.ofMinutes(5)))

**8. Name Nodes in Topology for Debugging**

* Use .name() to assign explicit names to critical nodes for monitoring/debugging:

stream.filter(...).name("filter-valid-orders")

This helps identify bottlenecks when inspecting topology with Kafka Streams' Topology.describe() or logs.

**9. Topology Versioning and Evolvability**

* Design your topology so that **schema evolution** or **field additions** won’t break the logic.
* If significant changes are made:
  + Use a new topic
  + Or change the application.id (with caution) to avoid state incompatibility

**Q. Ktable and GlobalKTable**

**Key Difference: KTable vs GlobalKTable**

| **Feature** | **KTable** | **GlobalKTable** |
| --- | --- | --- |
| **Partitioning** | Partitioned like a Kafka topic | Fully replicated on **every instance** |
| **Join Support** | Can be joined with KStream **on matching partition key** | Used for **foreign-key joins** |
| **State Storage** | Only stores **assigned partitions** | Stores **entire table** on each app instance |
| **Scalability** | Scales horizontally (partitioned state) | Limited scalability (full replication) |
| **Use Case** | When both streams are partitioned the same | When you need to join on a key not part of KStream’s key |

**Example of KTable**

**Scenario:**

You want to join **orders** with **customer status** where both are keyed by customerId.

**Kafka Topics:**

* orders → key: customerId
* customers → key: customerId

**Code:**

KStream<String, Order> orders = builder.stream("orders");

KTable<String, Customer> customers = builder.table("customers");

KStream<String, EnrichedOrder> enriched = orders.join(customers,

(order, customer) -> new EnrichedOrder(order, customer));

**Why KTable?**

* Join is based on the **same key (customerId)**
* Topics are **co-partitioned**
* Efficient and scalable

**Example of GlobalKTable**

**Scenario:**

You want to join **orders** keyed by orderId with **products** keyed by productCode.

The join key is **not the stream key** (productCode is inside the Order object).

**Kafka Topics:**

* orders → key: orderId, value contains productCode
* products → key: productCode

**Code:**

KStream<String, Order> orders = builder.stream("orders");

GlobalKTable<String, Product> products = builder.globalTable("products");

KStream<String, EnrichedOrder> enriched = orders.join(products,

(orderKey, orderValue) -> orderValue.getProductCode(), // foreign key extractor

(order, product) -> new EnrichedOrder(order, product));

**Why GlobalKTable?**

* You can **join on a different field (productCode)**
* No need for key repartitioning
* Each instance has access to **all products**

**Summary**

| **Use KTable When...** | **Use GlobalKTable When...** |
| --- | --- |
| Keys match and topics are co-partitioned | You need to join on a **foreign key** not in the stream key |
| Scalable, distributed joins are needed | Table size is small enough to **replicate everywhere** |

Q. **Initializer, Adder in Aggregation**



**Key Aggregation Components**



| **Operation** | **When It Runs** | **What It Does** | **Applies to Stream?** |
| --- | --- | --- | --- |
| **Initializer** | When a **new key** (bucket) is seen | It defines the **starting value** for that key | Yes |
| **Adder** | When a **new record** is added | It **updates the aggregate** based on the new input | Yes |

**🔹 Initializer**



* Called **only once per key** (per session/window/group)



* Sets the **starting point** for aggregation



* Example: For counting, () -> 0



() -> 0 // Every new key starts with count 0



**🔹 Adder**

* Called **every time a new record** for that key arrives



* Updates the current state (e.g., adds to the count or modifies the aggregate)



(key, value, aggregate) -> aggregate + 1 // For count



**Example: Count Page Views per User**

stream.groupByKey()

.aggregate(



() -> 0, // Initializer



(key, value, aggregate) -> aggregate + 1 // Adder



);

* First record from user alice → initializer sets alice → 0



* Each new click from alice → adder increments alice’s count

**Summary**



| **Component** | **Role** | **Required For Streams Aggregation?** |
| --- | --- | --- |
| Initializer | Sets the **starting state** | Yes |
| Adder | Defines **how to update the state** | Yes |

**Q. Would you please talk about the order in which subtractor and adder are called and if there is any guarantee of the order?**



In **Kafka Streams**, especially with **KTable aggregations** and **windowed operations**, the **order in which the subtractor and adder are called** matters and depends on the event type (update or delete) and the **source record ordering**.



Let’s break it down clearly:

**When Are adder and subtractor Called?**



In **KTable aggregations** (e.g., aggregate(), count() with a subtractor):

* **If a key’s value is updated** (e.g., value1 → value2):



* + The **subtractor is called first** to remove value1 from the aggregation
  + Then the **adder is called** to add value2



**Order of Operations**



| **Situation** | **Called in this order** |
| --- | --- |
| Value is **updated** | subtractor → adder |
| Value is **deleted** (tombstone) | subtractor only |
| Value is **newly inserted** | adder only (no subtraction) |

**Example**

Assume an aggregation is tracking total sales per productId. You use:

.aggregate(

() -> 0,



(key, newValue, aggregate) -> aggregate + newValue, // adder



(key, oldValue, aggregate) -> aggregate - oldValue // subtractor



)

And this sequence of records arrives for key = "product123":

| **Timestamp** | **Value** |
| --- | --- |
| t1 | 100 |
| t2 | 150 |
| t3 | null (delete/tombstone) |

**Processing:**

1. At t1:



* + No previous value → Only **adder(100)** → Total = 100



1. At t2:



* + Previous = 100



* + **subtractor(100)** → Total = 0



* + **adder(150)** → Total = 150



1. At t3:



* + Value is null → Only **subtractor(150)** → Total = 0



**Is the Order Guaranteed?**



**Yes — If the following hold:**



1. **Records come in order** from the **same partition**



1. You’re using **KTable semantics**, where updates are handled transactionally



Kafka ensures that:



* Updates (oldValue → newValue) for the same key **arrive in order per partition**



* KTable’s changelog provides **oldValue → newValue** pair for each update



**No Order Guarantee If:**



* Events for a key are spread across **multiple partitions**



* You use **KStream** (no subtractor, no old value tracking)



**Q. When would we use 'reduce' vs 'aggregate'?**



**Core Difference: reduce() vs aggregate()**



| **Feature** | **reduce()** | **aggregate()** |
| --- | --- | --- |
| **Output Type** | Must be **same** as input value | Can be **different** from input |
| **Logic Complexity** | Simple: combine 2 values into 1 | Complex: can build custom structures or combine logic |
| **Use Case** | retain best value | Compute totals, averages, counts, or build objects |

**Example : Using reduce() — Keep Latest or Maximum Value**



**Use Case:**

You want to keep the **maximum transaction amount** per customer.



KGroupedStream<String, Integer> grouped = stream.groupByKey();



KTable<String, Integer> maxTransaction = grouped.reduce(



(value1, value2) -> Math.max(value1, value2) // Reducer



);

* Input and output types are the same (Integer)



* No need to initialize the value



* Very concise for simple use cases like max, min, latest



**Summary Comparison**

| **Use When You Want To...** | **Use** |
| --- | --- |
| Keep max/min/latest/simplified result | reduce() |
| Maintain running total + count / custom object | aggregate() |
| Same input/output type | reduce() |
| Output type differs from input, or complex logic | aggregate() |
| Use subtractor in sliding/session windows | aggregate() |