**Stateful Transformations**

A **stateful transformation in Spark Streaming involves operations that require maintaining or updating a state across different batches of streaming data**. This means the result of the current batch depends not only on the current input but also on information from previous batches. Checkpointing is often used to ensure fault tolerance for these stateful operations.

Example: Running Word Count

Hi am Remya

Hi 1

Am 1+1=2

Remya 1+2=3

A classic example of a stateful transformation is a running word count in Spark Structured Streaming. This involves continuously counting the occurrences of each word as new data arrives in the stream.

Stream -stream join

What Are Stream-Stream Joins?

Stream-stream joins involve merging two continuous data streams in real-time based on a shared key or join condition, similar to SQL joins on static tables but tailored for the dynamics of streaming data.

How Do They Work?

* Each stream produces unbounded, dynamic data.
* Incoming events from both streams are tracked using a state mechanism (in-memory or on disk).
* The system must buffer events from both streams long enough for a possible matching event to arrive from the other stream.
* Most stream-stream joins implement a time window, such as "join on key within 10 minutes," to ensure the join buffer does not grow indefinitely.
* Watermarks are often used to track event-time progress and discard expired state, handling late-arriving data efficiently.

Use Case Example

A common example is joining an *ad impressions stream* with an *ad clicks stream* to attribute clicks to impressions in real time. When an impression and a click with the same ad ID occur within a defined window (e.g., 2 hours apart), they are joined to determine user engagement

**Windowing Operations**

Windowing operations are fundamental in streaming analytics for aggregating and analyzing real-time events over time-based groups (windows) rather than on a per-event basis. In Apache Spark Structured Streaming (using Scala), windowing enables efficient computation of rolling or fixed-interval metrics (like counts, averages, sums) over event streams.

Types of Window Operations

* Tumbling Windows: Fixed-size, non-overlapping windows. Each event falls into exactly one window (e.g., every 10 minutes).
* Sliding Windows: Fixed-size, potentially overlapping windows, sliding at a user-defined frequency (e.g., 10-minute window sliding every 5 minutes).
* Custom Windows: More advanced windows (e.g., session windows) can be built with custom logic.

Custom Windowing

For special logic (e.g., variable windows, session windows), you can implement custom window boundaries using user-defined functions (UDFs) and more advanced grouping techniques.

Managing State in Streaming

**Managing state in streaming—particularly with Apache Spark Structured Streaming—is crucial for operations where the output at any time depends on past events, such as aggregations, deduplication, window operations, and stream-stream joins.**

What Is State Management in Streaming?

* Stateful Processing means that the streaming application remembers information across events—like running counts, last-seen values, or user sessions. Purely stateless operations (e.g., simple mapping) don’t require any state.
* State can be any intermediate data that must persist across event boundaries and micro-batches.

How Does Spark Handle State?

* State Store: Spark maintains state in special key-value stores called *State Stores*. Each partition has its own state store, which is accessed to read/update the state during processing.
* Where Is State Stored? State is typically persisted to fault-tolerant storage (often **on disk or HDFS**) to provide recoverability and scalability.
* Watermarks and TTLs: Spark cleans up old state using *watermarks* and configurable time-to-live (TTL) policies, preventing memory leaks and optimizing performance.
* **Backends**: Spark uses an internal state store by default, but can also use embedded stores like RocksDB, which increases scalability for large states.

State Management APIs

* **Built-in Operators**: Windowed aggregations, deduplication, stream-stream joins—these use Spark’s internal state management automatically.
* **Custom State Logic:**
  + **mapGroupsWithState and flatMapGroupsWithState:** Let you manage custom per-key state, including **custom timeout handling**.
  + The **new transformWithState API (Spark 4.0+) brings further flexibility—multiple state variables, TTLs, and event/processing time timers.**

Practical Example: Stateful Aggregation in Spark Scala

Complex Streaming Scenarios

Deduplication and aggregations

In streaming with Apache Spark Structured Streaming, deduplication and aggregations are critical operations enabled through stateful processing, often relying on event-time watermarks for efficient state management and late data handling.

Deduplication in Streaming

Deduplication ensures filtering out duplicate records based on unique event identifiers or key columns. In Spark Structured Streaming, you can use the**dropDuplicates** operation for this purpose.

* When deduplicating in streaming, it’s essential to use watermarks on event-time columns to bound the state size and prevent unbounded memory growth.

val deduplicatedStream = inputStream

.withWatermark("eventTime", "10 minutes") // watermark on event-time column

.dropDuplicates("uniqueId", "eventTime") // deduplicate on unique key and event-time

* This approach only retains one record per unique key within the watermark time window, evicting old keys beyond the watermark.
* Note: Dropping duplicates globally without watermarking can cause growing state and potential out-of-memory issues.

**Aggregations in Streaming**

Aggregations compute summaries like count, sum, min, max, average over streaming data, frequently grouped by keys and windowed by time intervals.

* Use window functions with watermarks to define aggregation boundaries over event time.

val aggregatedStream = inputStream

.withWatermark("eventTime", "10 minutes") // watermark for state cleanup

.groupBy(

window($"eventTime", "10 minutes"), // tumbling window on event-time

$"key"

)

.count()

* Aggregation state is also managed based on watermark to prune state associated with old windows.

Late data handling

Late data handling in streaming, particularly in Apache Spark Structured Streaming, is primarily managed via the concept of watermarking. Here's a detailed explanation and how it functions:

What Is Late Data in Streaming?

Late data refers to events that arrive later than expected based on their event time due to network delays, processing delays, retries, or other upstream issues. These events can arrive after the streaming engine has already processed and emitted output results for the time window to which the event belongs.

How Spark Handles Late Data with Watermarks

* Watermark is a threshold on event time that Spark uses to decide how long to wait for late data before closing a window or state.
* When a watermark is defined on an event-time column with a delay (e.g., withWatermark("eventTime", "10 minutes")), Spark tracks the maximum event time seen in the stream and subtracts the delay to establish a watermark.
* Events with event-time earlier than the watermark are considered too late and are ignored or dropped to avoid unbounded state growth.
* Events arriving within the watermark delay window can still update state and affect results, allowing processing of reasonably late data.

Example Scenario

If you set a watermark delay of 10 minutes on event time, Spark will keep the state and allow updates for events that arrive up to 10 minutes late. After this threshold passes, the window is "closed," and late events are dropped.

Watermarks in Multi-Stream Joins

* In stream-stream joins, each stream may have different watermark delays.
* Spark calculates a global watermark, usually the minimum of all watermarks from input streams, to safely discard late data.
* You can also configure this with settings to choose other watermark policies if needed.

Checkpointing and fault tolerance

What Is Checkpointing?

Checkpointing involves periodically saving the state and metadata of a streaming query to a reliable, fault-tolerant storage system such as HDFS, cloud storage (S3, Azure Blob), or DBFS. This persisted checkpoint data allows Spark to resume the query from the last saved state rather than starting from scratch, minimizing data loss and enabling exactly-once processing guarantees.

Types of Checkpoint Data Saved

1. Metadata Checkpointing:
   * Saves configuration, streaming query plans, and offsets of processed input data.
   * Helps recover the streaming query when the driver or the application itself fails.
   * Ensures recovery of incomplete batches and the streaming execution plan.
2. State Checkpointing (Data Checkpointing):
   * Periodically persists the intermediate state of stateful operations like aggregations, joins, and deduplication.
   * Prevents unbounded growth of lineage dependencies, reducing recovery time.
   * Critical for operations that maintain and update state across streaming batches.

val query = streamingDF.writeStream

.format("console")

.outputMode("append")

.option("checkpointLocation", "/path/to/checkpoint/dir")

.start()

Role in Fault Tolerance

* When a failure occurs (driver crash, node failure, or process restart), Spark uses the checkpoint to:
  + Restore the query configuration and execution plan.
  + Recover the last committed offsets and state.
  + Resume processing from where the application left off, avoiding data reprocessing or loss.
* Checkpointing is essential for stateful transformations like windowed aggregations, stream-stream joins, and deduplication.

Stream processing patterns

Stream processing patterns are common design structures and techniques used to build efficient, scalable, and fault-tolerant real-time data processing pipelines using frameworks like Apache Spark Structured Streaming. These patterns help handle unbounded data, manage state, ensure correctness despite out-of-order or late data, and produce incremental results.

Here are some frequently used stream processing patterns, especially relevant for Spark Structured Streaming:

1. Streaming as a Continuously Growing Table

* Streams are treated as unbounded tables that continually append new rows.
* Queries are written like batch SQL queries, but processed incrementally on this ever-growing table.
* This abstraction simplifies reasoning about streaming data and operations like joins and aggregations.

2. Windowed Aggregations

* Stream data is grouped into time windows (e.g., tumbling or sliding) for aggregation—counts, sums, averages over defined intervals.
* Watermarks help handle late data and control the state size by cleaning old window state.
* Typical use case: Counting events per 10-minute intervals or computing metrics over sliding windows.

3. Stream-Stream Joins

* Join two or more streams based on keys and time constraints using event-time windows.
* Requires careful state management and watermarking to handle out-of-order data and limit the buffer size.
* Used in ad attribution, fraud detection, and correlating related real-time events.

4. Deduplication

* Removes duplicate events based on unique keys and event timestamps.
* Uses watermarks to bound state retention and avoid memory overflow.
* Critical in streaming systems where event delivery may be "at least once" leading to duplicates.

5. Stateful Processing and MapWithState

* Managing mutable state per key, such as running totals, sessions, or counters.
* Use Spark’s mapGroupsWithState or flatMapGroupsWithState APIs for custom stateful business logic.
* Supports timeout and state removal based on inactivity.

6. Late Data and Watermarking

* Defines how long to wait for delayed events.
* Watermark thresholds specify when state/windows can be finalized and cleaned up.
* Essential to maintain correctness while bounding resource usage.

7. Output Modes

* Append mode: Output only new rows (used when past results don't change).
* Update mode: Output changed rows only (e.g., aggregation updates).
* Complete mode: Output entire result table (for full aggregations).

8. Checkpointing and Fault Tolerance

* Enables state and progress persistence.
* Restarts queries from last known good state in case of failures, ensuring exactly-once processing.