**Title: Study Guide - Advanced Structured Streaming in Apache Spark**

**1. Micro-Batch vs Continuous Processing**

**Micro-Batch Mode:**

* Default mode in Spark Structured Streaming.
* Processes data in small, discrete intervals (e.g., every 1 second).
* Offers strong fault tolerance, support for complex aggregations, joins, and windowing.
* Achieves near-real-time latency (1-2 seconds).

**Continuous Mode:**

* Experimental mode in Spark.
* Data is processed as soon as it arrives, without waiting for a trigger interval.
* Lower latency (<1 ms) but limited to simple operations (no aggregations, no joins).
* Use for ultra-low-latency applications.

**Comparison Table:**

| **Feature** | **Micro-Batch** | **Continuous** |
| --- | --- | --- |
| Latency | Seconds | Sub-millisecond |
| Fault Tolerance | Yes | Limited |
| Aggregations/Joins | Supported | Not Supported |
| Use Cases | ETL, Aggregations | Real-time Dashboards |

**2. Watermarks & Handling Late Data**

**Why Watermarks?**

* Allows Spark to handle late-arriving event-time data in aggregations.
* Prevents unbounded state accumulation by bounding how late data can arrive.

**Mathematical Interpretation:**

* Watermark = max(event\_time seen) - threshold
* Any data with event\_time < watermark is considered late and may be dropped.

**Code Example:**

val stream = df.withWatermark("eventTime", "10 minutes")

.groupBy(window(col("eventTime"), "5 minutes"))

.count()

**Practical Note:**

* Must be used with append mode for accurate aggregations.

**3. Output Modes**

**Append Mode:**

* Outputs only new rows that were added since the last trigger.
* Best used with watermarks.

**Complete Mode:**

* Outputs the entire result table every trigger.
* Useful for queries with complete aggregations.

**Update Mode:**

* Outputs only rows that changed.
* Default for most aggregations without watermarks.

| **Output Mode** | **Use Case** | **Performance** |
| --- | --- | --- |
| Append | Late-data safe, lightweight | High |
| Complete | Full-table aggregations | Expensive |
| Update | Incremental updates | Moderate |

**4. Checkpoint Internals**

**Purpose:**

* Maintain fault tolerance and state recovery.
* Enable exactly-once processing.

**Components:**

* **Offsets Log:** Keeps track of how much data has been read from sources.
* **Commit Log:** Tracks what's been written to the sink.
* **State Store:** Holds intermediate state (e.g., window aggregations).

**Directory Layout:**

/checkpoint/

├── offsets/

├── commits/

├── state/

**Best Practice:** Always enable checkpointing when writing to sinks.

**5. Auto Loader & Schema Inference**

**What is Auto Loader?**

* High-performance file ingestion utility in Databricks.
* Supports schema inference and evolution.
* Uses directory listing (or cloud notifications) to detect new files.

**Key Options:**

* cloudFiles.format: file format (e.g., csv, json)
* cloudFiles.schemaLocation: location to store schema
* cloudFiles.inferColumnTypes: infer data types

**Example:**

val df = spark.readStream.format("cloudFiles")

.option("cloudFiles.format", "csv")

.option("cloudFiles.inferColumnTypes", "true")

.load("/mnt/data")

**6. Stream to Delta Lake**

**Why Delta Lake?**

* Adds ACID transactions, schema enforcement, time travel to Parquet.
* Ideal sink for streaming queries.

**Code Example:**

df.writeStream

.format("delta")

.option("checkpointLocation", "/mnt/chkpt")

.start("/mnt/delta/streamed\_data")

**Features Supported:**

* Append and merge
* Schema evolution
* Scalable ingestion

**7. Trigger Types in Structured Streaming**

**Types of Triggers:**

* **ProcessingTime("interval")**: runs every defined interval
* **Once**: runs one micro-batch and stops
* **AvailableNow**: processes all available data and exits

**Example:**

.writeStream.trigger(Trigger.ProcessingTime("10 seconds")).start()

.writeStream.trigger(Trigger.Once()).start()

.writeStream.trigger(Trigger.AvailableNow()).start()

**When to Use:**

* Use ProcessingTime for real-time pipelines.
* Use Once for backfills.
* Use AvailableNow for batch-like processing of streamed data.

**8. Graceful Restart**

**Purpose:**

* Restart Spark jobs without data loss.

**How It Works:**

* On restart, Spark reads checkpoint to resume progress.

**Steps:**

1. Configure checkpoint path.
2. Stop the job.
3. Restart with same checkpoint location.
4. Spark will recover offsets and state.

**9. Kafka Stream with Exactly-Once Semantics**

**Goal:**

* Consume from Kafka and write to Delta without data loss or duplication.

**Kafka Source Options:**

.option("kafka.bootstrap.servers", "localhost:9092")

.option("subscribe", "events")

.option("startingOffsets", "earliest")

**Delta Sink with Checkpointing:**

.writeStream

.format("delta")

.option("checkpointLocation", "/mnt/chkpt/kafka")

.start("/mnt/delta/kafka\_events")

**Validation:**

* Stop job and re-run.
* Confirm no duplicated rows using Delta commit logs.

**Summary**

* Micro-batch and continuous modes serve different needs.
* Watermarks control event-time lateness.
* Output modes define result delivery.
* Checkpoints ensure exactly-once semantics.
* Auto Loader simplifies incremental ingestion.
* Delta Lake enables ACID guarantees.
* Triggers control execution frequency.
* Graceful restarts and Kafka integrations make for resilient streaming pipelines.

**References**

* <https://spark.apache.org/docs/latest/structured-streaming-programming-guide.html>
* <https://docs.delta.io/latest/delta-streaming.html>
* <https://spark.apache.org/docs/latest/structured-streaming-kafka-integration.html>