**Detecting Partial Data Writes in Delta Lake on Databricks**

**Methods Summary Table**

|  |  |  |
| --- | --- | --- |
| **Method** | **Detects Partial Writes?** | **Use Case** |
| 1. **Delta DESCRIBE HISTORY** | **✅** | **Audit-based detection** |
| 1. **Write Audit Logging** | **✅** | **Custom ETL pipelines** |
| 1. **Streaming Checkpoints** | **✅** | **Structured Streaming** |
| 1. **Row-level Hash Check** | **✅** | **Critical data integrity** |
| 1. **Row Count Validation** | **✅** | **Batch and streaming** |
| 1. **Delta Constraints** | **⚠️ (limited)** | **Data quality enforcement** |
| 1. **DLT Expectations** | **✅** | **Declarative pipeline QA** |

**Sample.csv file**

|  |
| --- |
| **id, name, age**  **1, Alice, 30**  **2, Bob, 25**  **3, Charlie, 35**  **4, David, 40**  **5, Eva, 28** |

**Step 0: Setup – Load the CSV and Write to Delta Table**

**python**

**CopyEdit**

**# Load sample CSV**

**df = spark.read.csv("/path/sample\_data.csv", header=True, inferSchema=True)**

**# Write to Delta table**

**df.write.format("delta").mode("overwrite").save("/mnt/delta/sample\_table")**

**1. Delta DESCRIBE HISTORY (Audit-based detection)**

**sql**

**CopyEdit**

**-- SQL version (in a notebook cell)**

**DESCRIBE HISTORY delta.`/mnt/delta/sample\_table`;**

**Look at:**

* **operation: should be WRITE**
* **operationMetrics.numOutputRows: should be 5**
* **userMetadata: optionally track job\_id**

**If you see fewer rows than expected, that indicates a partial write.**

**It returns a history of all operations (like WRITE, MERGE, DELETE, etc.) on the Delta table, including:**

**Delta table, including:**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **version** | **timestamp** | **operation** | **operationMetrics** | **userMetadata** |
| **0** | **2025-07-09** | **WRITE** | **{"numOutputRows":"5"}** | **run\_id=...** |
| **1** | **2025-07-09** | **WRITE** | **{"numOutputRows":"2"} ← suspicious** |  |

**2. Write Audit Logging (Custom ETL pipelines)**

**Field Descriptions:**

|  |  |  |
| --- | --- | --- |
| **Field** | **Type** | **Description** |
| **run\_id** | **string** | **Unique ID for the ETL job run (generated with uuid)** |
| **table\_name** | **string** | **Target Delta table name** |
| **expected\_rows** | **integer** | **Number of rows you *expected* to be written** |
| **actual\_rows** | **integer** | **Number of rows actually written (validated after the write)** |
| **write\_success** | **boolean** | **True if the row counts match, else False** |
| **logged\_at** | **string** | **Timestamp of the logging event** |

**Add audit logs for tracking write activity:**

**python**

**CopyEdit**

**from datetime import datetime**

**import uuid**

**run\_id = str(uuid.uuid4())**

**expected\_rows = 5**

**actual\_rows = spark.read.format("delta").load("/mnt/delta/sample\_table").count()**

**log\_df = spark.createDataFrame([{**

**"run\_id": run\_id,**

**"table\_name": "sample\_table",**

**"expected\_rows": expected\_rows,**

**"actual\_rows": actual\_rows,**

**"write\_success": actual\_rows == expected\_rows,**

**"logged\_at": datetime.now().isoformat()**

**}])**

**log\_df.write.mode("append").saveAsTable("monitoring.write\_audit\_log")**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **run\_id** | **table\_name** | **expected\_rows** | **actual\_rows** | **write\_success** | **logged\_at** |
| **1c2b8a8e-1234-4cfd-9b00-abcde1234567** | **sample\_table** | **5** | **3** | **False** | **2025-07-09T18:46:30.123456** |

**# append this above log to a monitoring table**

**#Later, query this audit table to find:**

* **Failed or partial writes**
* **Mismatched row counts**
* **Run-level tracing by run\_id**

**3. Streaming with Checkpoints (Structured Streaming)**

**Step 1: Prepare Directory**

**Place your CSV (e.g., sample\_data.csv) into a directory like:**

**/mnt/stream\_input/**

**Databricks will treat each new file as a "mini-batch" of a stream**

**Step 2: Example Code – Structured Streaming from CSV to Delta**

**from pyspark.sql.functions import \***

**from pyspark.sql.types import StructType, IntegerType, StringType**

**# Define schema explicitly (recommended in streaming)**

**schema = StructType()**

**.add("id", IntegerType())**

**.add("name", StringType())**

**.add("age", IntegerType())**

**# Read as a streaming DataFrame from the directory**

**stream\_df = spark.readStream**

**.schema(schema)**

**.option("maxFilesPerTrigger", 1) #"Only read 1 new file at a time (per trigger interval).**

**.csv("/mnt/stream\_input") # new CSVs added here will be streamed**

**# Write streaming data to Delta table with checkpointing**

**query = stream\_df.writeStream**

**.format("delta")**

**.option("checkpointLocation", "/mnt/checkpoints/sample\_streaming\_table")**

**.outputMode("append")**

**.start("/mnt/delta/sample\_streaming\_table")**

**How This Helps Detect Partial Writes**

**1) Atomicity and Checkpointing**

* **If a micro-batch fails mid-write, it does not commit to the Delta table.**
* **On recovery, the job retries from the last safe checkpoint.**

**2) Partial File Detection**

* **If a malformed CSV is written to /mnt/stream\_input, it will fail parsing.**
* **You can capture those errors in the query listener or logs.**

**Output:**

* **Delta table: /mnt/delta/sample\_streaming\_table**
* **Checkpoints: /mnt/checkpoints/sample\_streaming\_table**
* **Triggered once per file (max 1 file per trigger due to maxFilesPerTrigger)**

**4. Row-level Hash Check (Critical data integrity)**

**python**

**CopyEdit**

**from pyspark.sql.functions import md5, concat\_ws**

**df = df.withColumn("row\_hash", md5(concat\_ws("||", \*df.columns)))**

**# Write with hash**

**df.write.format("delta").mode("overwrite").save("/mnt/delta/hashed\_table")**

**# Validate written data**

**df\_written = spark.read.format("delta").load("/mnt/delta/hashed\_table")**

**# Detect any row differences**

**discrepancy = df.exceptAll(df\_written)**

**if discrepancy.count() > 0:**

**print("Partial write or corruption detected")**

|  |  |  |  |
| --- | --- | --- | --- |
| **id** | **name** | **age** | **row\_hash** |
| **1** | **Alice** | **30** | **0b3c0c8e32b4a4ddfca9c63c7d83b515** |

**Example Result Table Format**

**Let’s say this was your original data (df):**

|  |  |  |  |
| --- | --- | --- | --- |
| **id** | **name** | **age** | **row\_hash** |
| **1** | **Alice** | **30** | **34b7da764b21d298ef307d04d8152dc5** |
| **2** | **Bob** | **25** | **4e07408562bedb8b60ce05c1decfe3ad** |
| **3** | **Charlie** | **35** | **e1671797c52e15f763380b45e841ec32** |

**But df\_written (read back from Delta) only has:**

|  |  |  |  |
| --- | --- | --- | --- |
| **id** | **name** | **age** | **row\_hash** |
| **1** | **Alice** | **30** | **34b7da764b21d298ef307d04d8152dc5** |
| **2** | **Bob** | **25** | **4e07408562bedb8b60ce05c1decfe3ad** |

**Then discrepancy.count() would return 1, and the discrepancy table would be:**

|  |  |  |  |
| --- | --- | --- | --- |
| **id** | **name** | **age** | **row\_hash** |
| **3** | **Charlie** | **35** | **e1671797c52e15f763380b45e841ec32** |

**Summary Table**

|  |  |
| --- | --- |
| **Component** | **Value** |
| **Total rows in df** | **3** |
| **Total rows in df\_written** | **2** |
| **Discrepancy rows** | **1 (missing Charlie)** |
| **.count() > 0 result** | **✅ True** |
| **Message printed** | **"Partial write or corruption detected"** |

**5. Row Count Validation (Batch and streaming)**

**python**

**CopyEdit**

**expected\_count = 5**

**actual\_count = spark.read.format("delta").load("/mnt/delta/sample\_table").count()**

**if actual\_count != expected\_count:**

**raise Exception(f"Partial write detected! Expected {expected\_count}, got {actual\_count}")**

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **Example Scenarios:**  **✅ Case 1: Successful Write**   |  |  | | --- | --- | | **Condition** | **Value** | | **expected\_count** | **5** | | **actual\_count** | **5** | | **Condition Met?** | **❌ (No Exception)** | | **Result** | **✅ No error, write is valid** | | **Case 2: Partial Write Detected**   |  |  | | --- | --- | | **Condition** | **Value** | | **expected\_count** | **5** | | **actual\_count** | **3** | | **Condition Met?** | **✅ (Mismatch)** | | **Result** | **❌ Raises Exception** | |

**Output:**

**Traceback (most recent call last):**

**...**

**Exception: Partial write detected! Expected 5, got 3**

**This clearly tells you that the Delta table is missing rows — only 3 out of 5 were written**

**To avoid hardcoding expected row counts, you can:**

* **Compare with df.count() before the write**
* **Track expected counts in a log or metadata store**

**6. Delta Constraints (Data quality enforcement)**

**Delta constraints help indirectly by catching missing/invalid values:**

**sql**

**CopyEdit**

**-- In SQL notebook cell**

**ALTER TABLE delta.`/mnt/delta/sample\_table`**

**ADD CONSTRAINT id\_not\_null CHECK (id IS NOT NULL);**

**Now if you write rows with null IDs, the write will fail and trigger alerts.**

**Adds a data constraint to the Delta table:**

* **Enforces that every row in the table must have a non-null id.**
* **Prevents writing invalid data into the table.**
* **Works like a guardrail to catch issues at write time.**

**What Happens Internally**

**Step 1: Constraint added**

**The constraint becomes metadata in the Delta transaction log. You can check it using:**

**sql**

**CopyEdit**

**SHOW TBLPROPERTIES delta.`/mnt/delta/sample\_table`**

**Test Case: Try to Write a Bad Row**

**from pyspark.sql import Row**

**# Create a row with a null id**

**bad\_df = spark.createDataFrame([Row(id=None, name="Test", age=33)])**

**# Try to append this invalid row to the constrained table**

**bad\_df.write.format("delta").mode("append").save("/mnt/delta/sample\_table")**

**Expected Output (Error)**

**If you run the write operation, Databricks throws an error like:**

**AnalysisException: CHECK constraint id\_not\_null (id IS NOT NULL) violated by row with values [null, Test, 33]**

|  |  |
| --- | --- |
| **Field** | **Value** |
| **id** | **null** |
| **name** | **Test** |
| **age** | **33** |
| **Result** | **❌ Write Fails** |
| **Why** | **Constraint id IS NOT NULL is violated** |

**When Write Succeeds**

**Summary Table**

|  |  |  |  |
| --- | --- | --- | --- |
| **Scenario** | **Constraint Satisfied?** | **Write Outcome** | **Message** |
| **All rows have valid IDs** | **✅** | **✅ Success** | **No error** |
| **Some rows have id = NULL** | **❌** | **❌ Fail** | **CHECK constraint violated** |

**Best Practice:**

**Use constraints to:**

* **Enforce data quality at write time**
* **Prevent accidental bad writes**
* **Pair with expectations in DLT for auto-monitoring**

**Would you like to see how to list all constraints on a Delt**

**7. Delta Live Tables (DLT) Expectations (Declarative QA)**

**In a DLT pipeline (Python syntax), use expectations:**

**python**

**CopyEdit**

**@dlt.table**

**@dlt.expect("valid\_id", "id IS NOT NULL")**

**def cleaned\_data():**

**return spark.read.format("csv").option("header", True).load("/path/sample\_data.csv")**

**DLT will automatically log failed rows and job status, preventing bad writes.**

**This is a DLT pipeline step that:**

1. **Creates a managed Delta Live Table called cleaned\_data**
2. **Applies a data quality expectation that the id column must not be null**
3. **Automatically tracks, enforces, and logs this expectation in Databricks**

**@dlt.table**

* **Declares this function as a DLT table.**
* **The return value of the function will be materialized as a Delta table.**
* **Equivalent to writing a SQL CREATE TABLE AS SELECT.**

**@dlt.expect("valid\_id", "id IS NOT NULL")**

* **Adds a data quality rule (expectation) named "valid\_id".**
* **Condition: id IS NOT NULL**
* **DLT tracks how many rows pass/fail this rule.**

**✅ Passed rows go into the table  
❌ Failed rows are logged and dropped (or redirected)**

**def cleaned\_data():**

* **Defines the logic for generating the table named cleaned\_data.**

**return spark.read.format("csv")...**

* **Reads a CSV file as a Spark DataFrame and returns it for DLT processing.**

**Behind the Scenes – DLT Monitoring**

**DLT adds built-in observability:**

|  |  |
| --- | --- |
| **Feature** | **What It Does** |
| **Quality dashboard** | **Shows how many records passed/failed each expectation** |
| **Auto logging** | **Logs metadata, errors, row counts, job run ID** |
| **Row dropping** | **Automatically drops rows that fail expectations** |
| **Error redirection** | **(Optional) Redirect bad rows to a quarantine table** |

**Appendix:**

**@step 2:**

**Why It's Useful : maxFilesPerTrigger", 1**

|  |  |
| --- | --- |
| **Benefit** | **Description** |
| **Simulate real-time ingestion** | **Useful in dev/test to mimic streaming from slowly-arriving files** |
| **Avoid processing overload** | **Prevents reading too many files in one batch, which can cause memory issues** |
| **Control ingestion rate** | **Helps apply backpressure and avoid spikes in processing** |

|  |  |
| --- | --- |
| **Example Use Case**  **Assume your input directory /mnt/stream\_input/ has these files:**  **CopyEdit**  **sample\_data\_1.csv**  **sample\_data\_2.csv**  **sample\_data\_3.csv**  **With maxFilesPerTrigger = 1:**   * **Micro-batch 1 → reads sample\_data\_1.csv** * **Micro-batch 2 → reads sample\_data\_2.csv** * **Micro-batch 3 → reads sample\_data\_3.csv**   **Instead of reading all files at once, Spark spaces them out—1 file per micro-batch.** | **Without This Option**  **If you don't set it, Spark may read all available files in the directory in a single micro-batch, which may lead to:**   * **Heavy memory usage** * **Slower processing** * **Poor simulation of real-time data flow**   **You can increase the value for higher throughput:**  **.option("maxFilesPerTrigger", 5) # Process 5 files per trigger**  **Or remove it for unlimited files per batch (default behavior).** |

**@ Step 4:**

**df = df.withColumn("row\_hash", md5(concat\_ws("||", \*df.columns)))**

**Purpose:**

**This line adds a new column called "row\_hash" to your DataFrame (df) that contains a hash value for each row.**

**Explanation:**

|  |  |
| --- | --- |
| **Expression Part** | **Meaning** |
| **\*df.columns** | **Expands the list of column names to pass each column separately as an argument (e.g., "id", "name", "age")** |
| **`** |  |
| **concat\_ws(" md5(...)** | **Computes an MD5 hash of that concatenated string (e.g., `"1** |
| **withColumn("row\_hash", ...)** | **Adds a new column to the DataFrame called row\_hash with the computed hash value** |

**Why This Is Done:**

**To create a unique fingerprint of each row:**

* **Used for row-level integrity checking**
* **Helps detect partial writes, duplicate rows, or data corruption**

**Example:**

**If the input row is:**

| **id** | **name** | **age** |
| --- | --- | --- |
| **1** | **Alice** | **30** |

**The concatenated string will be:**

**"1||Alice||30"**

**Then the MD5 hash might be:**

**"0b3c0c8e32b4a4ddfca9c63c7d83b515"**

**So the final DataFrame becomes:**

|  |  |  |  |
| --- | --- | --- | --- |
| **id** | **name** | **age** | **row\_hash** |
| **1** | **Alice** | **30** | **0b3c0c8e32b4a4ddfca9c63c7d83b515** |

**Use Case:**

**Later, after writing the data to a Delta table, you can re-read the table and recompute the row\_hash again to compare with your source. If any hashes are missing or mismatched → the row was:**

* **Truncated**
* **Partially written**
* **Corrupted**