**Handle Schema Drift with Schema Evolution ( Except Data Type Changes)**

**Schema drift** and **schema evolution** handle **VARIANT data types**, what causes drift in Databricks Delta Lake.

This is especially important if you’re working with **semi-structured data** (JSON, Avro, Parquet), which is where schema drift shows up the most.

|  |  |
| --- | --- |
| **Schema Drift:** Unexpected changes in the incoming data structure over time, such as new columns, missing fields, or altered types. | **Schema Evolution:** The capability of a system to adapt and incorporate schema changes automatically without failing data ingestion. |

**Topics:**

* What VARIANT data is
* Why VARIANT increases schema drift
* How schema evolution applies to VARIANT
* Practical examples step by step
* What best practices help you manage it

Contents

[**What Is VARIANT Data? 2**](#_Toc202762051)

[**How schema drift arises with VARIANT / nested structures: 2**](#_Toc202762052)

[**What Causes Drift With VARIANT? 2**](#_Toc202762053)

[**How Schema Evolution Works With Nested Data 3**](#_Toc202762054)

[**Detailed Example Step by Step 3**](#_Toc202762055)

[**# 1 – Define Initial Schema 3**](#_Toc202762056)

[**# 2 – Load Data Without Drift 4**](#_Toc202762057)

[**# 3 – New Nested Field Arrives (Schema Drift) 4**](#_Toc202762058)

[**# 4 – Append Data With Schema Evolution 4**](#_Toc202762059)

[**# 5 – Data Type Conflict Appears (Schema Drift) 5**](#_Toc202762060)

[**# 6. How To Handle Type Conflicts 6**](#_Toc202762061)

[**Option 1 – Cast Before Write 6**](#_Toc202762062)

[**Option 2 – Store Entire Row as String JSON 6**](#_Toc202762063)

[**Save Silver Table 7**](#_Toc202762064)

[**Option 3 – Use Auto Loader with Permissive Mode 8**](#_Toc202762065)

[**Best Practices Table 8**](#_Toc202762066)

[**Recap: How Schema Drift vs Schema Evolution Apply to VARIANT/Nested Data 8**](#_Toc202762067)

**What Is VARIANT Data?**

**VARIANT** is a flexible column type used in systems like **Snowflake** (in Databricks you typically work with MapType or StructType instead) to store **semi-structured** or **nested data**.

For example:

json

CopyEdit

{

"user\_id": "U123",

"device": {

"type": "mobile",

"os": "iOS"

},

"event\_properties": {

"video\_length": 120,

"playback\_speed": 1.0

}

}

In Delta Lake, you’d store this using:

* StructType
* MapType
* or a string column with raw JSON

# How schema drift arises with VARIANT / nested structures:

* New fields appear under device or event\_properties.
* Data types of fields change (e.g., string to integer).
* Entire sub-objects are added or removed.

**What Causes Drift With VARIANT?**

**Examples of schema drift:**

|  |  |
| --- | --- |
| **Snapshot in time** | **Sample JSON** |
| **Day 1** | json { "user\_id": "U123", "device": { "type": "mobile" } } |
| **Day 2** | json { "user\_id": "U123", "device": { "type": "mobile", "os": "iOS" } } |
| **Day 3** | json { "user\_id": "U123", "device": { "type": 100 } } |
| **Day 4** | json { "user\_id": "U123", "device": "mobile" } |

These changes are **schema drift** because:

* New nested fields (os) appeared.
* Data types changed (type string → integer).
* Entire object changed shape (device object → string).

**How Schema Evolution Works With Nested Data**

In Delta Lake:

* **Schema evolution** (mergeSchema) can **add new fields** to nested structures automatically.
* It **does NOT automatically reconcile data type conflicts**.

|  |  |
| --- | --- |
| What Delta handles: | What Delta **does NOT** handle automatically: |
| * Adding new fields to structs * Adding new top-level columns | * Type conflicts within nested structures * Renaming nested fields * Removing fields |

**Detailed Example Step by Step**

Let’s do a **concrete example** in Databricks.

**# 1 – Define Initial Schema**

python

CopyEdit

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

schema = StructType([

StructField("user\_id", StringType()),

StructField("device", StructType([

StructField("type", StringType())

]))

])

**Initial Data:**

json

CopyEdit

{

"user\_id": "U123",

"device": {

"type": "mobile"

}

}

**# 2 – Load Data Without Drift**

python

CopyEdit

df = spark.read.schema(schema).json("/mnt/raw/day1/")

df.write.format("delta").mode("overwrite").save("/mnt/delta/user\_events")

**Schema after Day 1:**

go

CopyEdit

root

|-- user\_id: string

|-- device: struct

|-- type: string

**# 3 – New Nested Field Arrives (Schema Drift)**

**Day 2 Data:**

json

CopyEdit

{

"user\_id": "U456",

"device": {

"type": "tablet",

**"os": "Android"**

}

}

**# 4 – Append Data With Schema Evolution**

This **adds the new nested field** automatically:

python

CopyEdit

df\_new = spark.read.json("/mnt/raw/day2/")

df\_new.write \

.format("delta") \

.option("**mergeSchema**", "true") \

.mode("append") \

.save("/mnt/delta/user\_events")

**Schema after Day 2:**

go

CopyEdit

root

|-- user\_id: string

|-- device: struct

|-- type: string

|-- os: string

**This is schema evolution at work.**

# **5 – Data Type Conflict Appears (Schema Drift)**

**Day 3 Data:**

json

CopyEdit

{

"user\_id": "U789",

"device": {

**"type": 100**

}

}

**Attempting to append this fails:**

python

CopyEdit

df\_bad = spark.read.json("/mnt/raw/day3/")

df\_bad.write \

.format("delta") \

.option("mergeSchema", "true") \

.mode("append") \

.save("/mnt/delta/user\_events")

❌ **Error:**

go

CopyEdit

Failed to write: Found incompatible data types in column **device.type**:

- existing type: string

**- new type: long**

S**chema evolution does NOT automatically fix data type conflicts.**

# # 6. How To Handle Type Conflicts

## Option 1 – Cast Before Write

python

CopyEdit

df\_clean = df\_bad.withColumn("device",

expr("named\_struct('type', **cast(device.type as string)**)")

)

df\_clean.write \

.format("delta") \

.mode("append") \

.save("/mnt/delta/user\_events")

## Option 2 – Store Entire Row as String JSON

* Instead of parsing nested columns, keep a raw JSON string column.
* Process in Silver layer.

df\_raw = (

spark.readStream

.format("cloudFiles")

.option("cloudFiles.format", "**text**") # **Read as raw lines**

.load("/mnt/raw/user\_events/")

)

root

|-- value: string (the raw JSON line)

**Write Raw Text into Bronze Delta Table**

(

df\_raw.writeStream

.format("delta")

.option("checkpointLocation", "/mnt/checkpoints/bronze\_user\_events")

.start("/mnt/delta/bronze\_user\_events")

)

**Inspect Bronze Table**

spark.read.format("delta").load("/mnt/delta/bronze\_user\_events").show(truncate=False)

value

{"user\_id":"U789","device":{"type":100,"os":"Android"},"event\_timestamp":"..."}

**Success:** You preserved **all original JSON** as a string.

**Process JSON in Silver Table**

Now, in Silver, you **parse the JSON safely** using from\_json().

**Define the target schema you want to extract:**

python

CopyEdit

from pyspark.sql.functions import from\_json, col

from pyspark.sql.types import StructType, StructField, StringType, TimestampType, StructType

parsed\_schema = StructType([

StructField("user\_id", StringType()),

StructField("device", StructType([

StructField("type", StringType()),

StructField("os", StringType())

])),

StructField("event\_timestamp", TimestampType())

])

**Read Bronze table and parse JSON:**

python

CopyEdit

df\_bronze = spark.read.format("delta").load("/mnt/delta/bronze\_user\_events")

df\_silver = (

df\_bronze

.withColumn("jsonData", from\_json(col("value"), parsed\_schema))

.select(

col("jsonData.user\_id").alias("user\_id"),

col("jsonData.device.type").alias("device\_type"),

col("jsonData.device.os").alias("device\_os"),

col("jsonData.event\_timestamp").alias("event\_timestamp")

)

)

**Save Silver Table**

python

CopyEdit

df\_silver.write.format("delta").mode("overwrite").save("/mnt/delta/silver\_user\_events")

**Silver table schema:**

CopyEdit

root

|-- user\_id: string

|-- device\_type: string

|-- device\_os: string

|-- event\_timestamp: timestamp

**Logic**

|  |  |
| --- | --- |
| **Layer** | **What You Store** |
| **Bronze** | **Raw JSON string** (value column) |
| **Silver** | Parsed structured columns using from\_json() |
| **Gold** | Aggregations, business-ready outputs |

Advantages of This Pattern

**Future-proof:**  
New fields added upstream don’t break ingestion—your raw JSON is preserved.

**Flexible:**  
You can re-parse with a new schema if business requirements change.

**Reliable:**  
No load failures due to unexpected schema drift.

## Option 3 – Use Auto Loader with Permissive Mode

* Auto Loader can load unknown columns and nulls, but still won’t auto-fix conflicting types.

**Best Practices Table**

|  |  |
| --- | --- |
| **Best Practice** | **Why** |
| Use explicit StructType schemas | Avoids unpredictable type inference |
| Enable mergeSchema=true for adding new fields | Lets you evolve schema without recreating tables |
| Validate incoming types | Catch conflicts before write |
| Separate raw (Bronze) and clean (Silver) data | Keep drift isolated |
| Store raw JSON if schema changes frequently | Avoid frequent schema churn in Delta tables |
| Use time travel and schema history | Recover previous states if needed |

**Recap: How Schema Drift vs Schema Evolution Apply to VARIANT/Nested Data**

|  |  |
| --- | --- |
| **Concept** | **Description** |
| **Schema Drift** | New fields or types appear in nested structures |
| **Schema Evolution** | Allows new fields to be merged automatically (mergeSchema=true) |
| **NOT handled automatically** | Data type conflicts or field renames |

**Summary:**

* **Schema drift happens** whenever new keys or types arrive in your nested data.
* **Schema evolution helps** automatically add new fields to the Delta table.
* **Type conflicts still require you to clean or cast data manually.**

**Appendix**

..

✅ **In short:**  
parsed\_schema is the **schema definition** you want Spark to use to **parse** your raw JSON string into structured columns.

When you have JSON text in a column (value), Spark has no idea what fields are inside unless you explicitly tell it.

So you define parsed\_schema using StructType to **describe** what you expect in the JSON.

## Example Recap: Where parsed\_schema Comes From

Below is the complete snippet again:

python

CopyEdit

from pyspark.sql.functions import from\_json, col

from pyspark.sql.types import StructType, StructField, StringType, TimestampType

# Here you define parsed\_schema:

parsed\_schema = StructType([

StructField("user\_id", StringType()),

StructField("device", StructType([

StructField("type", StringType()),

StructField("os", StringType())

])),

StructField("event\_timestamp", TimestampType())

])

This schema says:

* The JSON has:
  + a user\_id field (string)
  + a nested device object with:
    - type (string)
    - os (string)
  + event\_timestamp (timestamp)

**So when you do:**

python

CopyEdit

.withColumn("jsonData", from\_json(col("value"), parsed\_schema))

you are telling Spark:

“Parse the JSON text in value using this parsed\_schema. Put the resulting structured data into a new column called jsonData.”

## What the output looks like

After parsing, jsonData is a **Struct** column with this nested structure:

go

CopyEdit

jsonData

├── user\_id: string

├── device: struct

│ ├── type: string

│ └── os: string

└── event\_timestamp: timestamp

## Next step: Extract structured fields

You then unpack it like this:

python

CopyEdit

.select(

col("jsonData.user\_id").alias("user\_id"),

col("jsonData.device.type").alias("device\_type"),

col("jsonData.device.os").alias("device\_os"),

col("jsonData.event\_timestamp").alias("event\_timestamp")

)

Resulting table:

|  |  |  |  |
| --- | --- | --- | --- |
| **user\_id** | **device\_type** | **device\_os** | **event\_timestamp** |
| U789 | 100 | Android | 2023-07-15 12:00:00.000 |

## Tip: Inspect Raw JSON First

Before you write parsed\_schema, always **look at a sample** of your JSON to see:

* What fields are present
* What types they have
* What nested structure exists

**Summary**

* parsed\_schema is the blueprint describing **what your JSON looks like**.
* from\_json() uses it to convert raw strings into structured columns.
* You then **select or explode** nested fields as needed.