Spark 4.0

ANSI Compliance in Spark

- **ANSI (American National Standards Institute)** Compliance refers to Spark's ability to follow the ANSI SQL standard for database query language behavior. This includes how it handles errors, type coercion, and other SQL-related operations.

- **spark.sql.ansi.enabled**: This is a configuration in Spark that, when set to true, ensures that Spark follows the ANSI SQL standard strictly. By default, this is set to false, meaning Spark may not strictly follow the standard in certain cases.

Why Enable ANSI Mode?

- When ANSI mode is enabled, Spark will throw runtime exceptions for invalid operations like:
  - Integer overflows
  - String parsing errors
  - Invalid type casting

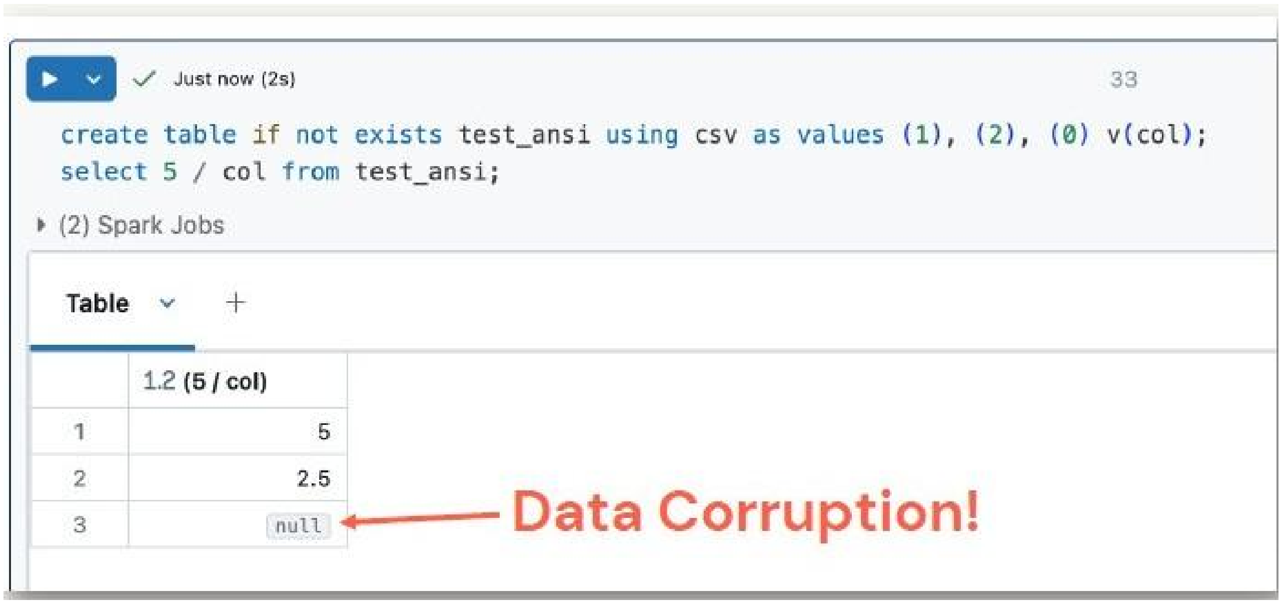
- Without ANSI mode (spark.sql.ansi.enabled = false), Spark may return NULL or allow certain automatic type conversions that could lead to silent errors or data corruption.

Benefits of Enabling ANSI Mode:

- More predictable and standard behavior: You avoid surprises during query execution, as Spark will behave more consistently with ANSI SQL rules.
- Error Handling: It will surface runtime errors instead of ignoring them, allowing you to fix issues early and avoid potential data corruption.
- Type Conversion Control: When writing data to a column of a different type, Spark will follow ANSI rules for type coercion, preventing invalid type conversions.

Explanation

Scenario 1



```
create table if not exists test_ansi using csv as values (1), (2), (0) v(col);
select 5 / col from test_ansi;
```

(2) Spark Jobs

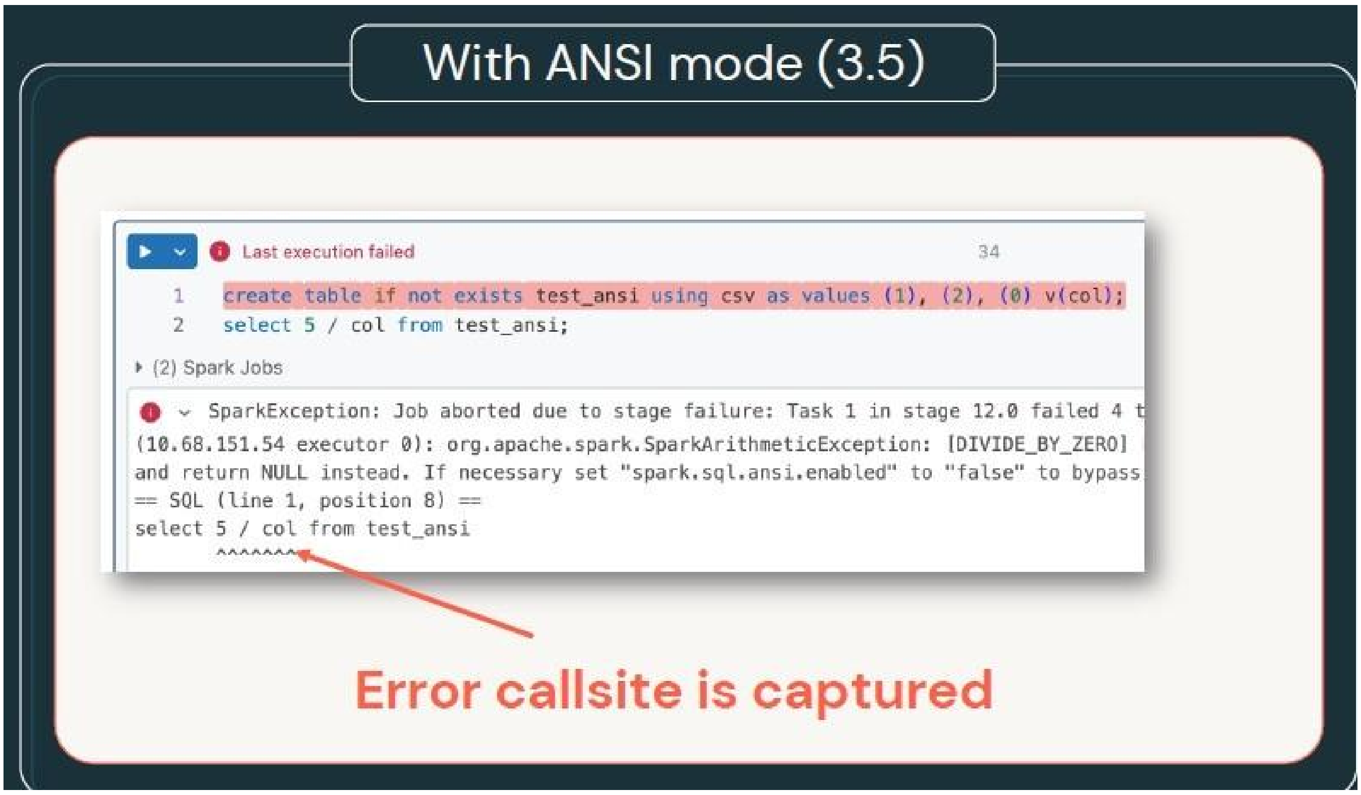
	1.2 (5 / col)
1	5
2	2.5
3	null

1. Without ANSI Mode (Scenario 1)
   - **Query**: select 5 / col from test_ansi;
   - **Behavior**: When you try to divide 5 by a column (col) that contains values, including a a zero, Spark does

not throw an error. Instead, it returns NULL for the division by zero.

- **Issue**: This can lead to silent data corruption, as you might not realize there's an issue with the data, and Spark allows the query to proceed without surfacing the problem.

Scenario 2



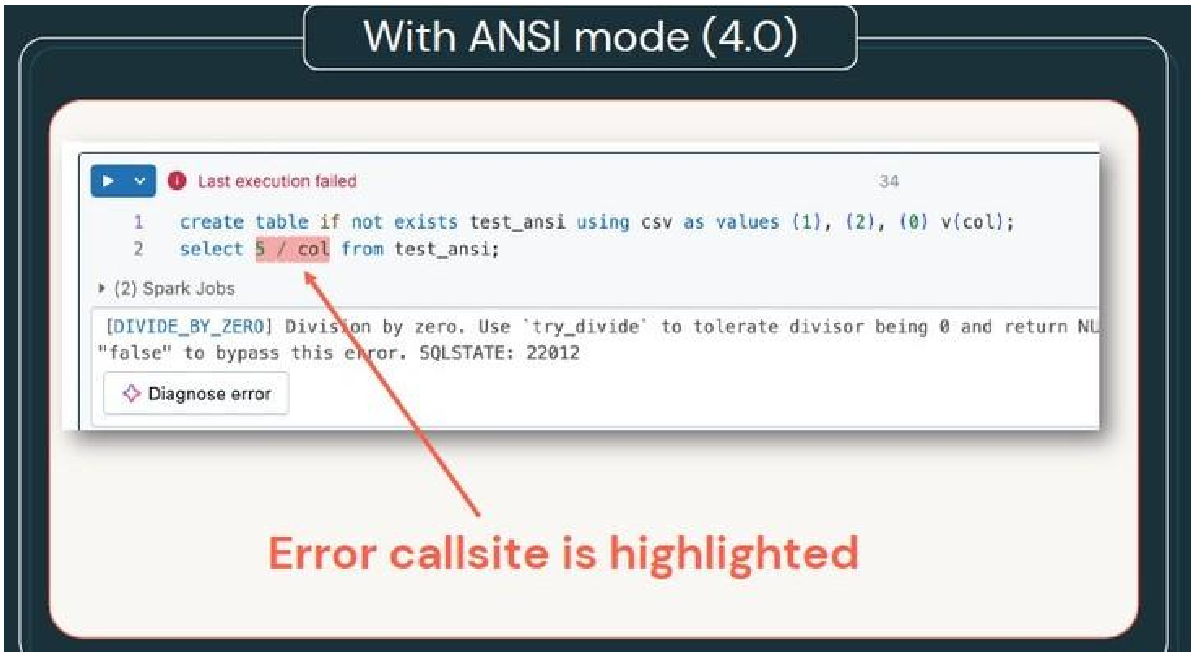
```
create table if not exists test_ansi using csv as values (1), (2), (0) v(col);
select 5 / col from test_ansi;
```

```
SparkException: Job aborted due to stage failure: Task 1 in stage 12.0 failed 4 t
(10.68.151.54 executor 0): org.apache.spark.SparkArithmeticException: [DIVIDE_BY_ZERO]
and return NULL instead. If necessary set "spark.sql.ansi.enabled" to "false" to bypass
== SQL (line 1, position 8) ==
select 5 / col from test_ansi
       ^^^^^^^
```

2. With ANSI Mode (Scenario 2,Spark 3.5)
   - **Query**: Same query (select 5 / col from test_ansi;).
   - **Behavior**: In this case, ANSI mode is turned on (spark.sql.ansi.enabled = true), so when a division by zero occurs, Spark throws a SparkArithmeticException, halting the job.
   - **Result**: The error message is clear, indicating a DIVIDE_BY_ZERO error, and it provides the specific

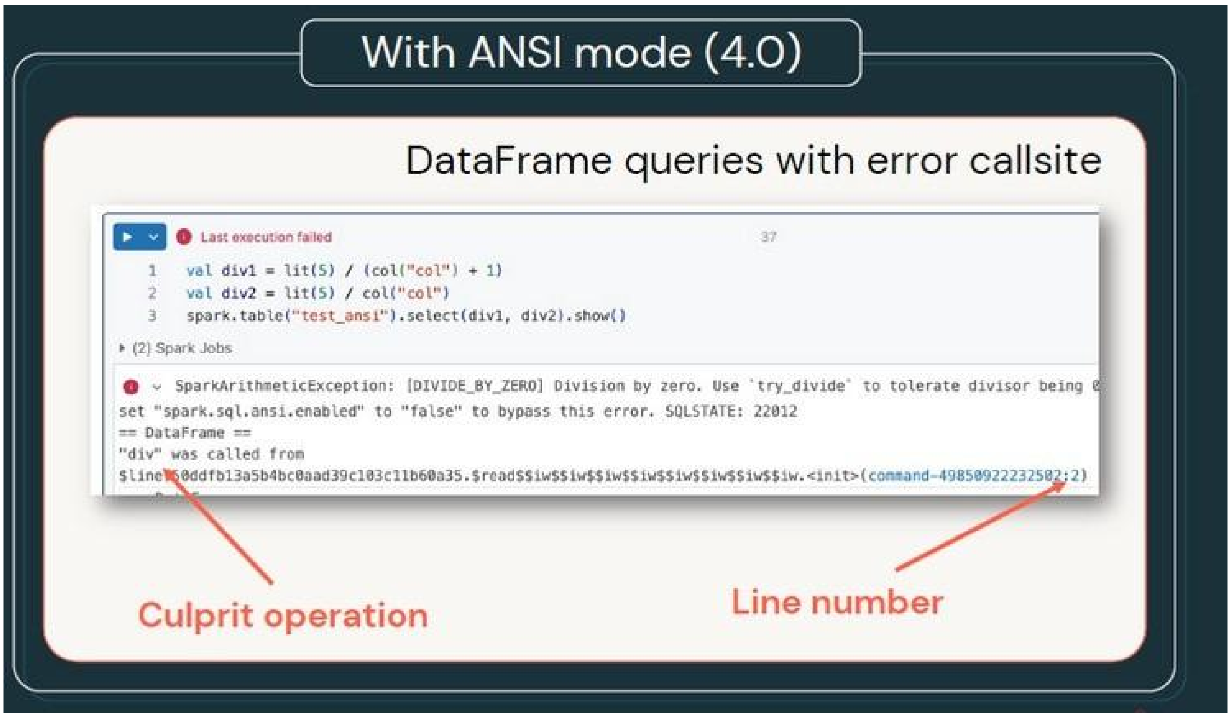
location of the problem (SQL query). This prevents silent errors from going unnoticed

Scenario 3(Whats new with Spark 4.0)



```
create table if not exists test_ansi using csv as values (1), (2), (0) v(col);
select 5 / col from test_ansi;
```

```
[DIVIDE_BY_ZERO] Division by zero. Use `try_divide` to tolerate divisor being 0 and return NU
"false" to bypass this error. SQLSTATE: 22012
```



```
val div1 = lit(5) / (col("col") + 1)
val div2 = lit(5) / col("col")
spark.table("test_ansi").select(div1, div2).show()
```

```
SparkArithmeticException: [DIVIDE_BY_ZERO] Division by zero. Use `try_divide` to tolerate divisor being 0
set "spark.sql.ansi.enabled" to "false" to bypass this error. SQLSTATE: 22012
== DataFrame ==
"div" was called from
$line60ddfb13a5b4bc0aad39c103c11b60a35.$read$$iw$$iw$$iw$$iw$$iw$$iw$$iw$$iw.<init>(command-4985092223258 2:2)
```

3. With ANSI Mode (Scenario 3,Spark 4.0)

Query: Same query.
Behavior: In Spark 4.0, ANSI mode provides enhanced error messages with more detailed information.
Improvement: The error call site is highlighted in the error message. This means the exact point of failure (where the error occurred) is more explicitly shown, making debugging **much easier.**

DataFrame Queries with Error Call Site:

- **DataFrame Example**: Here, two divisions are performed with DataFrame operations (lit(5) / col("col")).

- **Behavior**: When ANSI mode is enabled and a division by zero occurs, Spark provides a detailed error message showing not only the operation that caused the error but also the line number in the code where it happened.

- **Result**: This is extremely useful for debugging in complex Spark jobs. It shows which DataFrame operation caused the issue and where it is located in the code, streamlining the troubleshooting process.

Variant Data Type for Semi-Structured Data

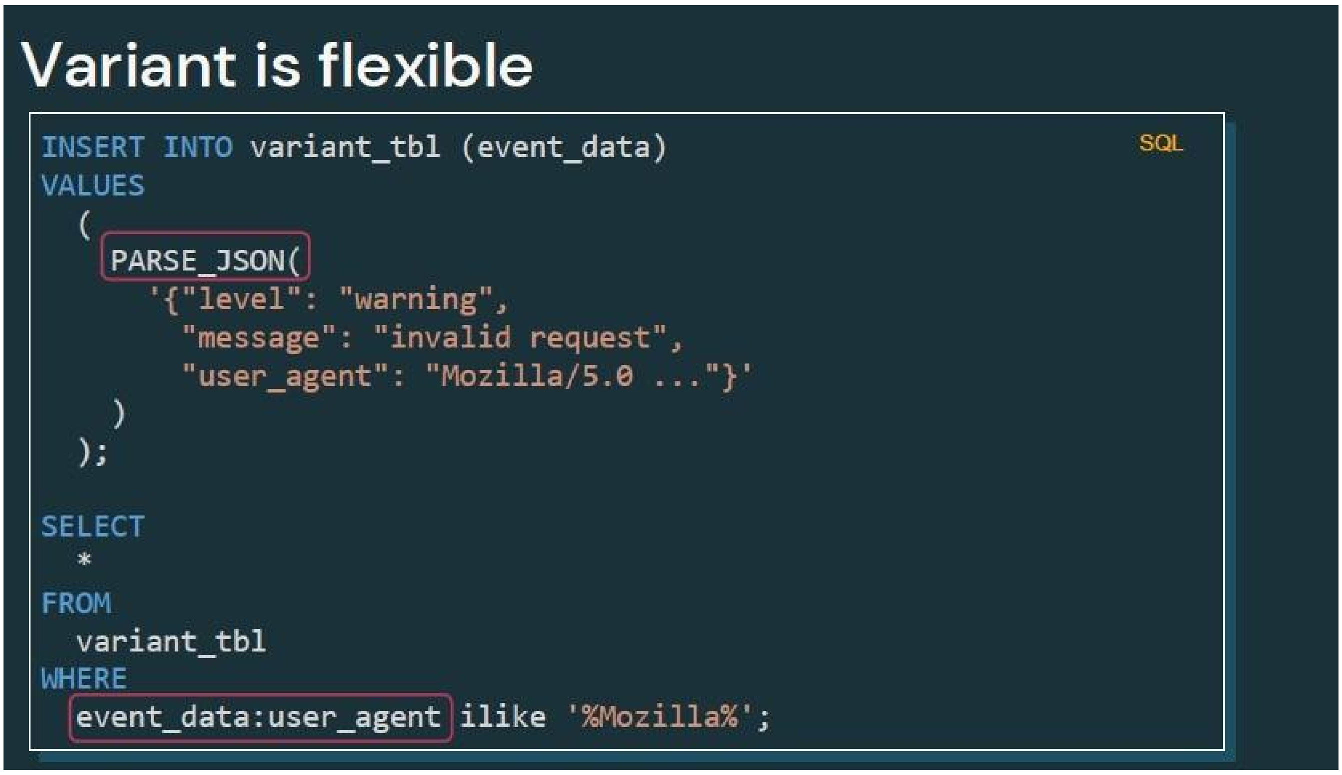
What is a Variant Data Type?

A Variant data type is a flexible and dynamic data type that can hold different types of data within a single column or field, such as integers, strings, arrays, JSON, or other semi-structured formats. This is particularly useful for working with semi-structured data like JSON, XML, Avro, or Parquet, where the schema or structure of the data may vary or be nested.

Motivation For the Update

- Traditionally, when dealing with semi-structured data, you often have to compromise between flexibility, speed, and open standards. You could usually only choose two out of three.
- The introduction of the Variant Data Type proposes that you can now ingest JSON and maintain flexibility, performance, and open standards together.
- This solves a big challenge for data engineers trying to optimize semi-structured data processing.

Example:



Variant is flexible

```
INSERT INTO variant_tbl (event_data)                    SQL
VALUES
  (
    PARSE_JSON(
      '{"level": "warning",
        "message": "invalid request",
        "user_agent": "Mozilla/5.0 ..."}'
    )
  );

SELECT
  *
FROM
  variant_tbl
WHERE
  event_data:user_agent ilike '%Mozilla%';
```

- In this example, the SQL query inserts JSON data into a table using the **PARSE_JSON** function.
- event_data represents a column in a table where each entry holds semi-structured data, specifically JSON objects. In this case, the table is used to store event logs, such as user interactions, warnings, or system messages. The event_data column stores these event logs as JSON strings but parsed into a Variant data type.
- The **PARSE_JSON** function is used to convert a JSON string into a Variant data type. Here, the JSON object contains the following fields:

A. **level**: Indicates the severity of the event (e.g., "warning").

B. **message**: Provides a description of the event (e.g., "invalid request").

C. **user_agent**: Stores information about the user's browser or device (e.g., "Mozilla/5.0 ...").

- The **INSERT INTO** statement places the parsed JSON into the event_data column of the variant_tbl table. Since event_data is of type Variant, it can accept and store the structured JSON without needing to break it into individual fields (columns).

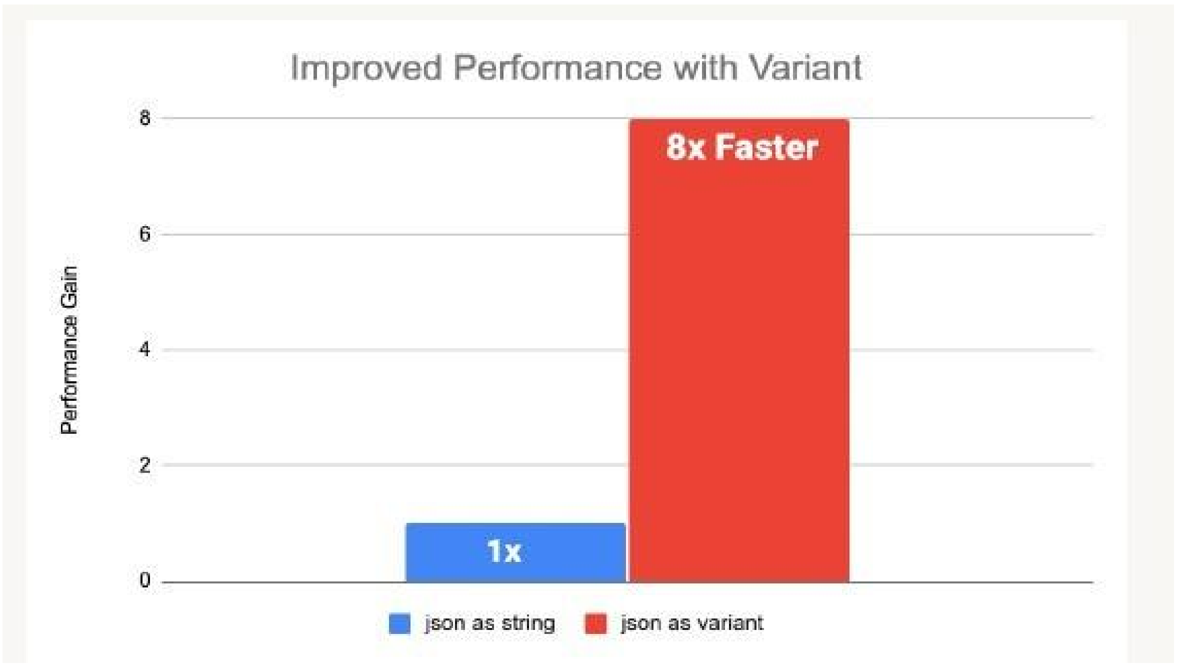
Querying the JSON Data Stored as Variant:

- The SELECT query retrieves data from the variant_tbl where the user_agent field in the event_data (the JSON object) contains the text "Mozilla."
- The use of ILIKE (case-insensitive LIKE) allows for pattern matching on the user_agent value, enabling you to find entries where the user was using a Mozilla-based browser.
- :user_agent notation: This is a specific way to reference nested fields inside the JSON stored in the Variant column. In this case, you’re directly accessing the user_agent key inside the JSON structure.

Summary of Flexibility:

- Inserting JSON as-is: You can store raw JSON directly into the database without transforming it into a rigid schema.
- Querying individual JSON fields: You can easily query specific fields inside the JSON (like user_agent) without needing to extract them into separate columns.
- Handling changing data structures: You don't have to worry about the schema changing when the JSON structure evolves (e.g., new fields being added).

Performance:Json As Variant

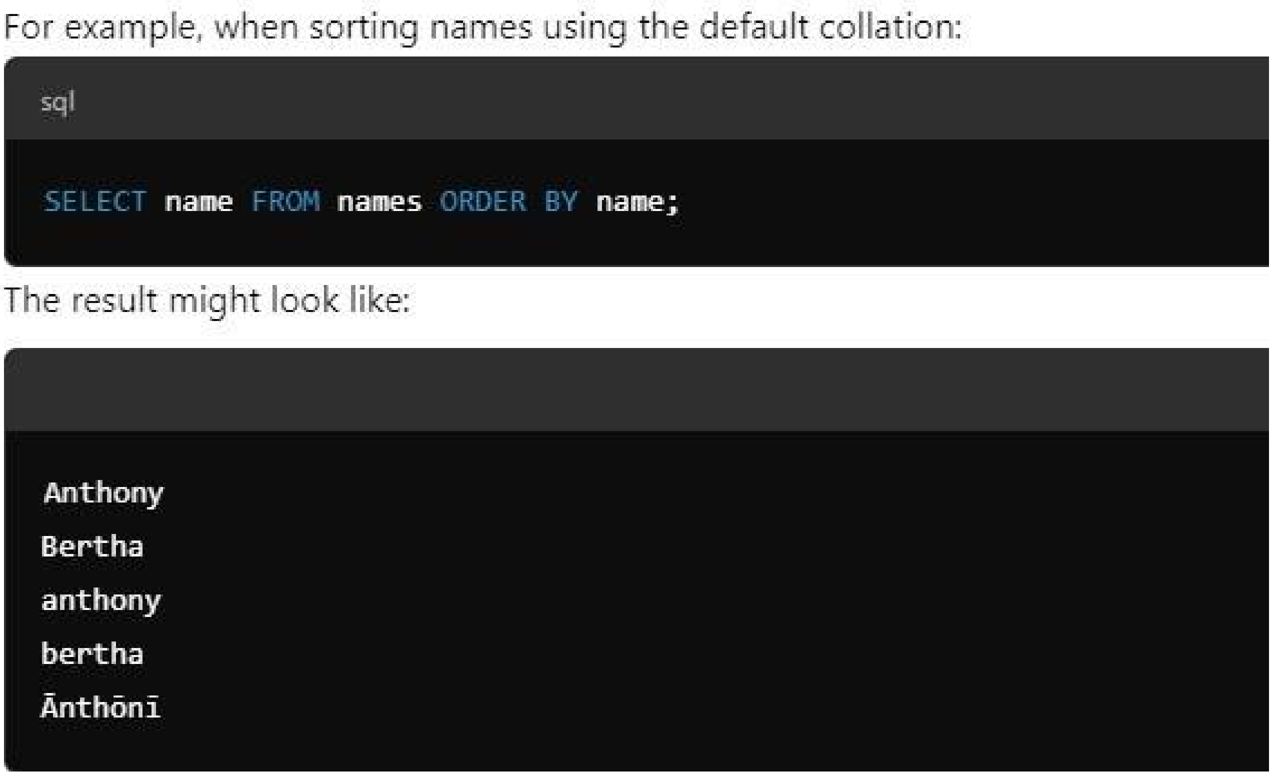


- The performance chart highlights a significant improvement when using the Variant data type to store and query JSON compared to storing JSON as a string.
- **8x Faster**: Storing JSON as a Variant leads to a dramatic increase in performance (8x faster) compared to treating the JSON as just a string. This is because the Variant type allows Spark to efficiently parse and access the JSON fields without needing to do expensive string manipulations.

Improved String Sorting and Comparison with Unicode Collation: Fixing Case and Accent Sensitivity Issues in SQL Queries

Problem (Before Unicode Collation):

Default collation in SQL doesn't handle case sensitivity, accent sensitivity, or locale awareness very well.


For example, when sorting names using the default collation:

sql

```sql
SELECT name FROM names ORDER BY name;
```

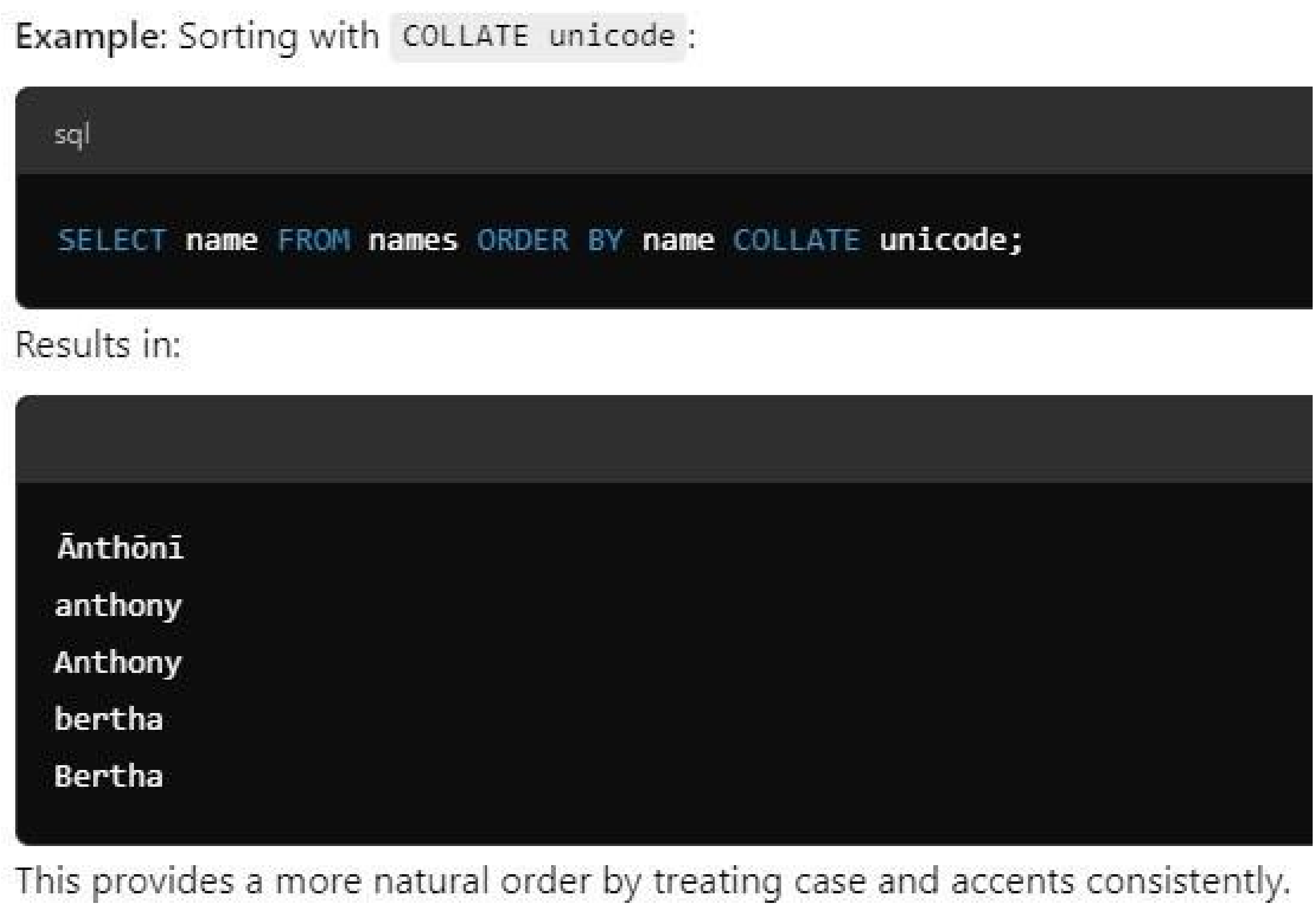
The result might look like:

```
Anthony
Bertha
anthony
bertha
Ānthōnī
```

This is not ideal because lowercase letters are sorted after uppercase letters, and accented characters are treated differently, which may not match natural language sorting expectations.

Unicode Collation (COLLATE unicode):

- The COLLATE unicode clause provides better sorting by handling case and accents more naturally.


Example: Sorting with `COLLATE unicode`:

```sql
SELECT name FROM names ORDER BY name COLLATE unicode;
```

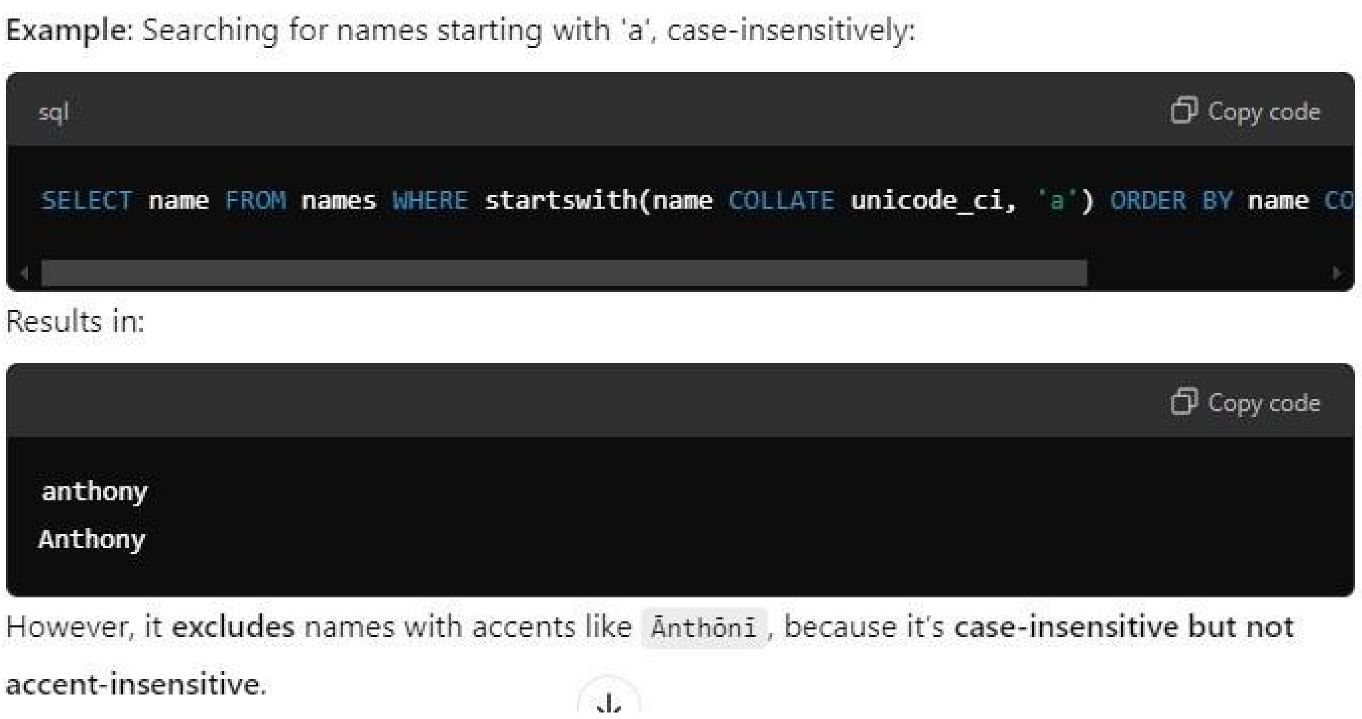
Results in:

```
Ānthōnī
anthony
Anthony
bertha
Bertha
```

This provides a more natural order by treating case and accents consistently.

Unicode Case-Insensitive Collation (COLLATE unicode_ci):

This collation makes case-insensitive comparisons but does not handle accents.


Example: Searching for names starting with 'a', case-insensitively:

```sql
SELECT name FROM names WHERE startswith(name COLLATE unicode_ci, 'a') ORDER BY name CO
```

Results in:

```
anthony
Anthony
```

However, it **excludes** names with accents like `Ānthōnī`, because it's **case-insensitive but not accent-insensitive.**

Key Differences:

- Unicode collation (COLLATE unicode) provides natural sorting for case and accents.
- Unicode case-insensitive collation (COLLATE unicode_ci) ignores case but still differentiates based on accents.

Streaming State Data Source

- **Purpose**: This feature provides a way to access and inspect the internal state of a streaming application. It is particularly useful for debugging, profiling, testing, and troubleshooting. **Key Use Cases**: Debugging issues
- in streaming queries. Profiling the state to optimize performance. Testing state-related logic. Manipulating internal states in urgent situations (e.g.,

recovering from a failure).

Stateful Stream Processing Overview

- **State Management**:
  - Streaming jobs often maintain state, such as windowed aggregations or deduplications.
  - This state is stored locally (e.g., in memory or local storage) for quick access.
- **Checkpointing**:
  - To ensure fault tolerance, the state is periodically checkpointed to remote storage (e.g., HDFS or S3). This allows the system to recover state after a failure.

State Reader API

Spark 4.0 introduces two APIs to inspect the state:

a. **High-Level API**: state-metadata

- Purpose: Provides metadata about

the state being maintained by the streaming application, such as operator-level details.

- Key Information:
    - **operatorId**: Unique identifier for the streaming operator.
    - **operatorName**: The name or type of operation (e.g., deduplication, aggregation).
    - **stateStoreName**: The name of the state store used.
    - **numPartitions**: Number of partitions used for the state.
    - **minBatchId/maxBatchId**: The range of batch IDs currently using the state.
- **Use Case**: This API is useful for getting a high-level understanding of the streaming state, which can help in optimizing resources or debugging.

b. **Granular API**: statestore

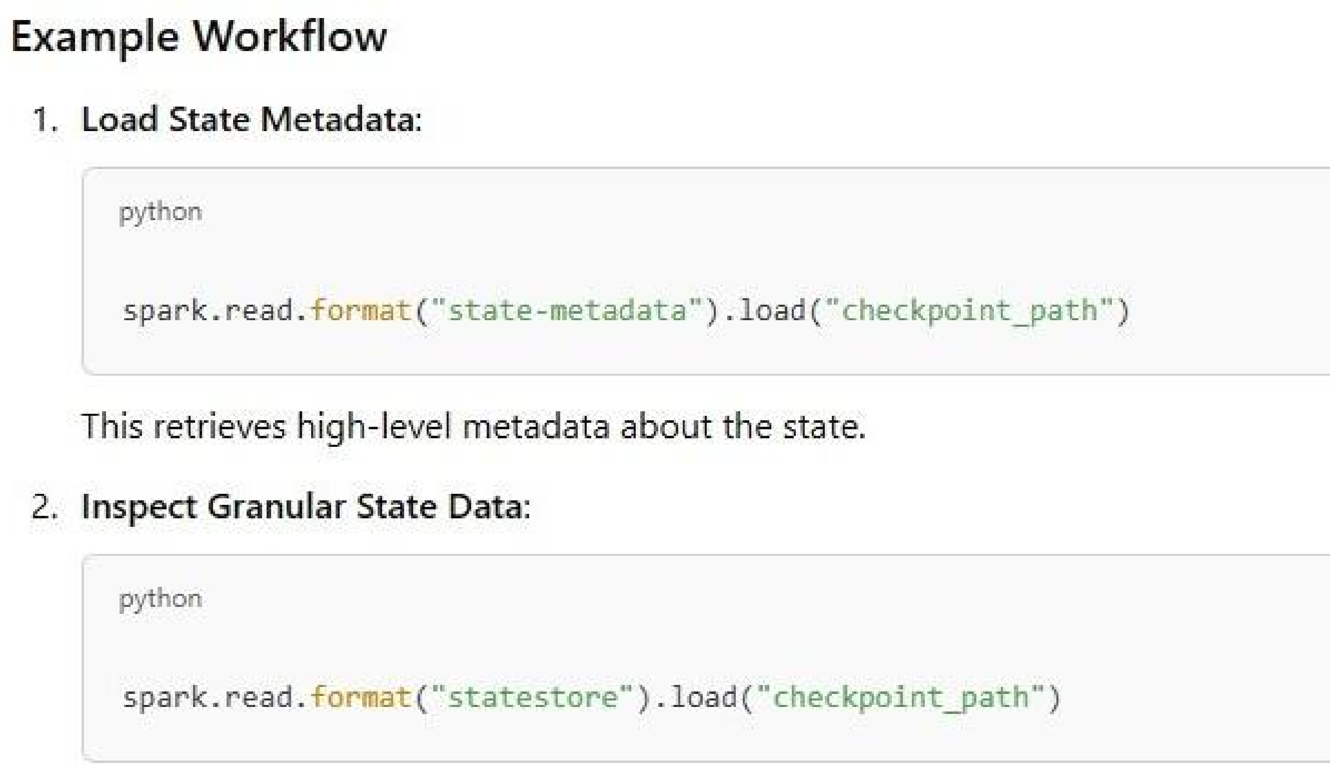
- Purpose: Provides detailed, granular access to the state data itself, allowing inspection of individual state records.
- **Key Information**:

a. key: The key for the state entry (e.g., specific window or entity).

b. value: The associated state information (e.g., counts, aggregated values).

c. partition_id: The partition where this state is stored.

Use Case: This API is beneficial for examining specific state entries to debug issues or understand the data flow through the application.



Example Workflow

1. **Load State Metadata:**

   ```python
   spark.read.format("state-metadata").load("checkpoint_path")
   ```

   This retrieves high-level metadata about the state.

2. **Inspect Granular State Data:**

   ```python
   spark.read.format("statestore").load("checkpoint_path")
   ```

Arbitrary Stateful Processing V2

Before discussing the update,Lets understand whats the problem was and how this new update addresses the problems.

Lets first discuss, what does it mean by stateful processing in spark streaming.

In Spark Streaming, **stateful processing** means keeping a memory of past events (state) and updating it with new incoming data.

Example:

Imagine you're tracking the number of purchases made by users over time.
Every time a user makes a purchase, you update their purchase count (state).

For user A:

State initially = 0
After 1st purchase, state = 1
After 2nd purchase, state = 2

Problem with flatMapGroupsWithState (V1 API)

Limitation 1: No Composite State Types
flatMapGroupsWithState only allows simple state types like integers, not complex structures.
Example Problem: If you're tracking a user’s purchase count and last purchase date, you can't store them both easily. You’d have to combine them manually into a single string or object, making updates and debugging harder.

Limitation 2: No State Cleanup (Eviction)
State keeps growing forever unless you manually remove old, unused states. This is bad for long-running streaming jobs.
Example Problem: If user A hasn’t made a purchase in 1 year, their state still stays in memory, wasting resources.

Limitation 3: Schema Cannot Evolve

If you decide to add another field to the state (e.g., average purchase value), you must restart the streaming query, which is disruptive in production.

How transformWithState (V2 API) Fixes These Problems

Here's how transformWithState improves things, with easy-to-follow examples:

Advantage 1: Support for Composite State Types

With transformWithState, you can define a complex state structure.

Example: Let's say you want to track:

Purchase count

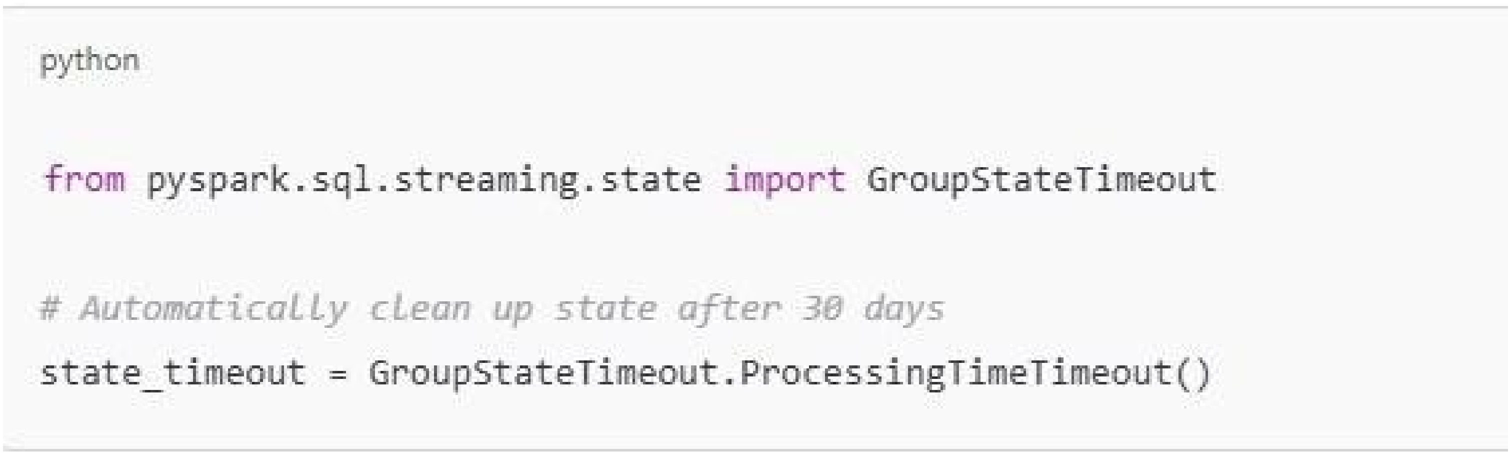
Last purchase date

Total purchase amount

Advantage 2: State Cleanup with TTL

With transformWithState, you can automatically remove old state (e.g., if a user hasn't made a purchase in 30 days).

Example: Let's say user A hasn't made a purchase in 30 days. You can configure a Time-to-Live (TTL) policy to remove A's state from memory automatically



python

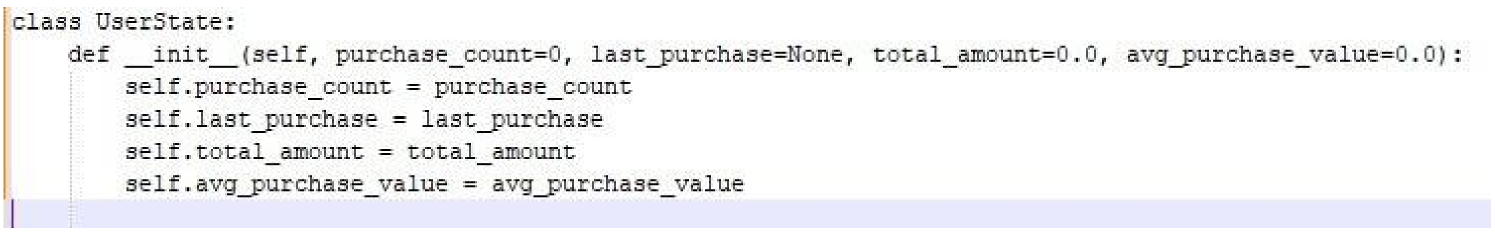
```python
from pyspark.sql.streaming.state import GroupStateTimeout

# Automatically clean up state after 30 days
state_timeout = GroupStateTimeout.ProcessingTimeTimeout()
```

Advantage 3: Schema Evolution

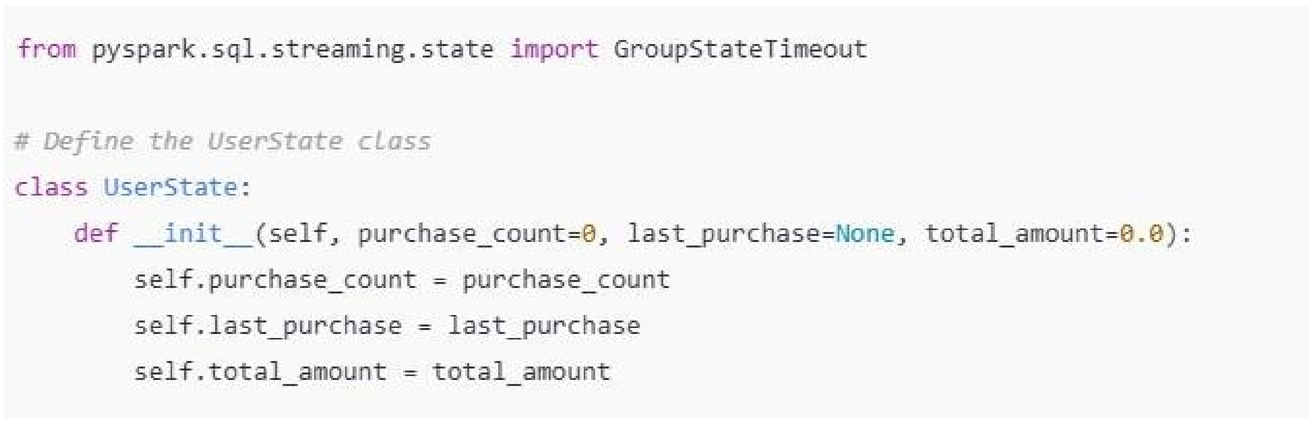
You can add or change fields in your state structure without restarting the query.

Example: If you later decide to add average purchase value to UserState, you can simply update your class:



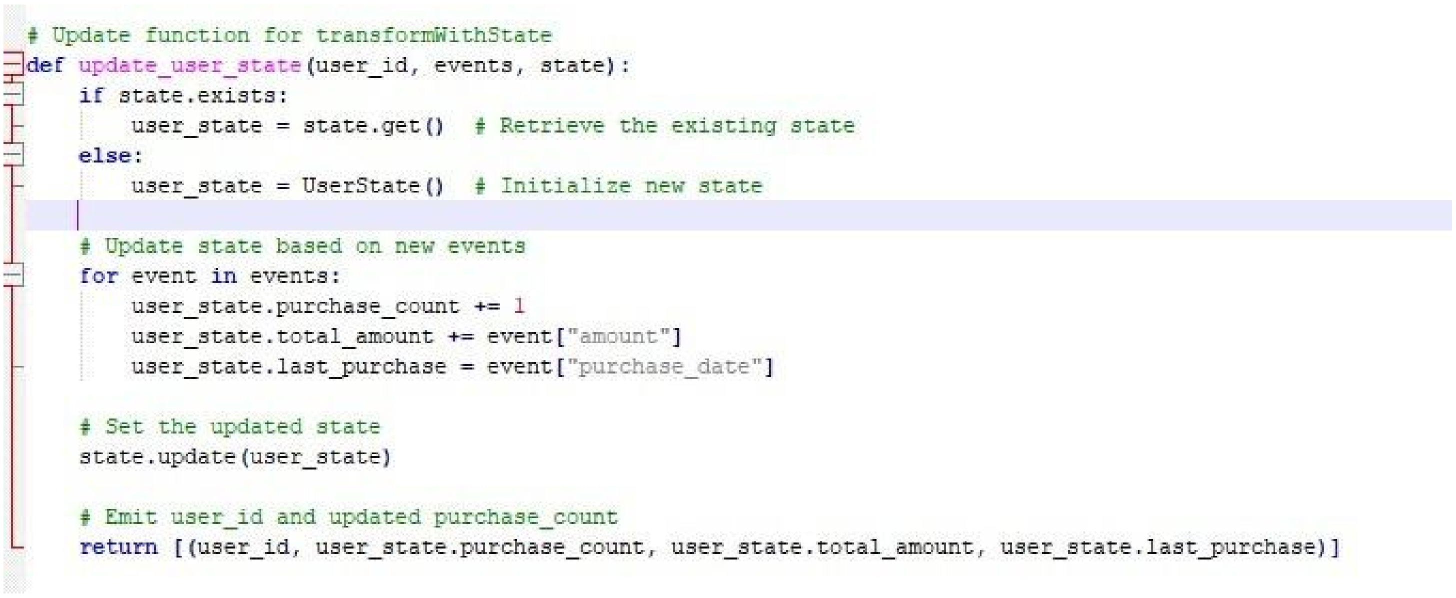
```python
class UserState:
    def __init__(self, purchase_count=0, last_purchase=None, total_amount=0.0, avg_purchase_value=0.0):
        self.purchase_count = purchase_count
        self.last_purchase = last_purchase
        self.total_amount = total_amount
        self.avg_purchase_value = avg_purchase_value
```

Here's a complete example showing all benefits:



```python
from pyspark.sql.streaming.state import GroupStateTimeout

# Define the UserState class
class UserState:
    def __init__(self, purchase_count=0, last_purchase=None, total_amount=0.0):
        self.purchase_count = purchase_count
        self.last_purchase = last_purchase
        self.total_amount = total_amount
```

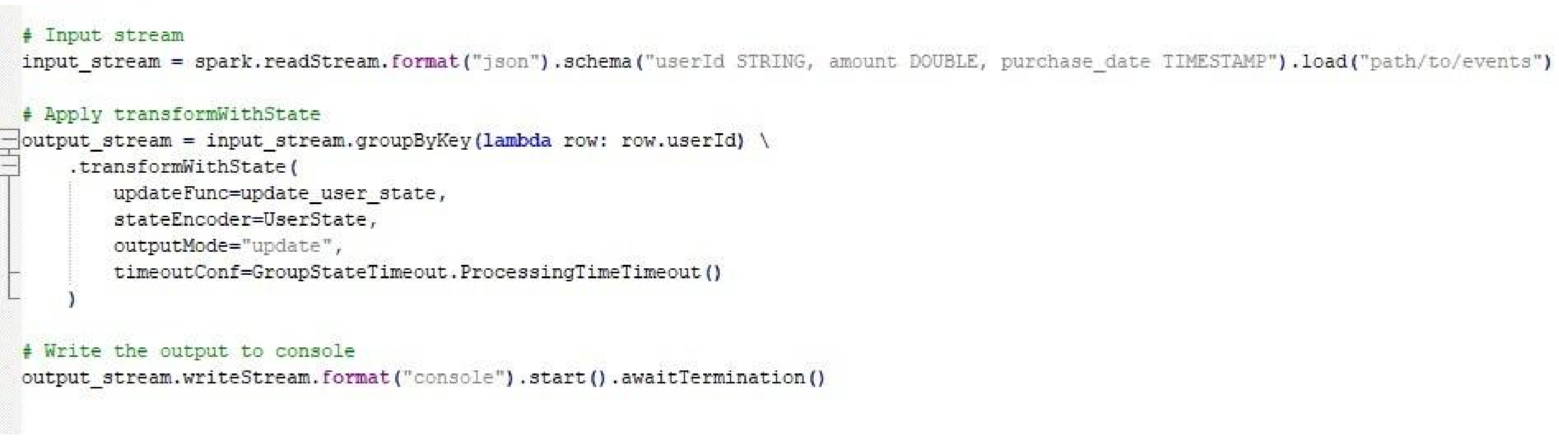


```python
# Update function for transformWithState
def update_user_state(user_id, events, state):
    if state.exists:
        user_state = state.get()  # Retrieve the existing state
    else:
        user_state = UserState()  # Initialize new state

    # Update state based on new events
    for event in events:
        user_state.purchase_count += 1
        user_state.total_amount += event["amount"]
        user_state.last_purchase = event["purchase_date"]

    # Set the updated state
    state.update(user_state)

    # Emit user_id and updated purchase_count
    return [(user_id, user_state.purchase_count, user_state.total_amount, user_state.last_purchase)]
```



```
# Input stream
input_stream = spark.readStream.format("json").schema("userId STRING, amount DOUBLE, purchase_date TIMESTAMP").load("path/to/events")

# Apply transformWithState
output_stream = input_stream.groupByKey(lambda row: row.userId) \
    .transformWithState(
        updateFunc=update_user_state,
        stateEncoder=UserState,
        outputMode="update",
        timeoutConf=GroupStateTimeout.ProcessingTimeTimeout()
    )

# Write the output to console
output_stream.writeStream.format("console").start().awaitTermination()
```

What Happens Here?

State is Complex:

Tracks purchase_count, total_amount, and last_purchase for every user.

Each field can be updated independently.

Automatic Cleanup:

States are removed after the configured timeout (e.g., 30 days of inactivity).

Evolving Schema:

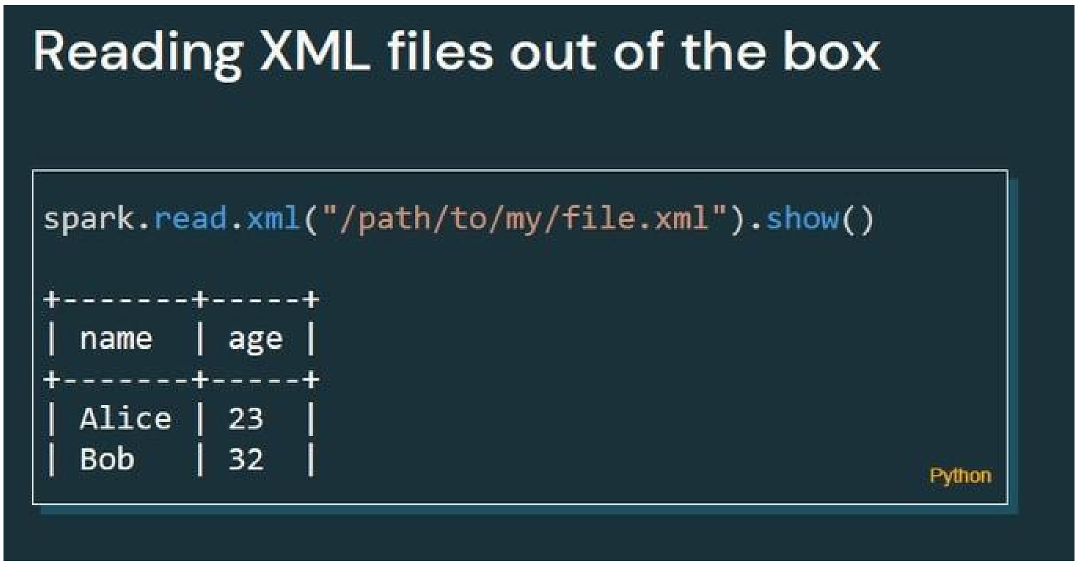
You can add fields (like average purchase value) to UserState later without restarting the streaming job.

Why Is This Useful?

1. Easier to handle real-world, complex use cases (e.g., user behavior tracking). Multiple state change can be captured in one go with ease compared to previous version
2. Prevents memory overload with state cleanup for inactive users.
3. No downtime in production for schema changes.

XML Connectors

Now xml can be read directly with spark read API



Reading XML files out of the box

```python
spark.read.xml("/path/to/my/file.xml").show()

+-------+-----+
| name  | age |
+-------+-----+
| Alice | 23  |
| Bob   | 32  |
```

Except all these spark 4.0 offers several minor update in Logging,memory Profiling and databricks specific optimizations.

Keeping it simple here are few important updates. Scroll to next page.

PySpark UDF Unified Profiling

Unified Profiling: Tracks execution time, memory usage, and function calls for PySpark UDFs.

Benefits:

Combines performance and memory profiling.

Simplifies bottleneck analysis with show, dump, and clear commands.

Configuration:

1. E**nable performance profiling**:
   spark.conf.set("spark.sql.pyspark.udf.profiler", "perf").

Enable memory profiling:

spark.conf.set("spark.sql.pyspark.udf.profiler", "memory").

2. **Structured Spark Logging**

JSON Logs: Spark 4.0 uses structured JSON logs for easier parsing and filtering.

Log Processing:

Use Spark to query logs like any dataset for error messages, exceptions, and more.

Example: spark.read.json("/var/spark/logs.json").filter(col("level") == "ERROR").

3. **System Log Directories**

Query logs directly from system files using Spark's DataFrame APIs:

Filter errors by host: logs.filter(col("context.host") == "100.116.29.4").

.

Find Spark-specific exceptions:
logs.filter(col("exception.class").startswith("org.apache.spark")
).
Identify executor loss logs: logs.filter(col("msg").contains("Lost executor")).
Focus on a specific executor:
logs.filter(col("context.executor_id") == 289).

4. **Enhanced Debugging with Unified Profiling**

Provides deeper insight into UDF performance:
Example: Analyze runtime for each function call and memory usage per line of code.
Commands:
View results: spark.profile.show(type="perf") or
spark.profile.show(type="memory").
Save results: spark.profile.dump("/your_path/").
Clear results: spark.profile.clear().

5. **Advantages Over Legacy Profiling**

Unified profiling works with Spark Connect and allows toggling during runtime.
Simplifies the previously complex, unstructured logging and debugging process.
These updates collectively enhance Spark's capabilities for profiling, debugging, and log analysis, making it easier to build and troubleshoot robust streaming and data processing applications