Discovering complex event stream processing with Spark structured streaming & the 'Lakehouse'

Indranil Tarafdar · Follow

11 min read · Jun 2, 2023

A guide to solve complex stateful stream data processing problems using Spark, Kafka & Delta Lake.

Streaming data practitioners dealing with Apache Spark tend to solve the streaming

problems with Structured Streaming's (SS) inbuilt transformation or aggregation APIs (most of the time). However, in order to achieve that, the problem statement has to be straightforward enough to fit in the SS ready-to-use high-level API landscape.

> *Unfortunately, some real-life streaming problems can be a bit more complex than they can be solved using the SS inbuilt and ready-to-use aggregation/analytical APIs and may require a custom approach to supply curated data to the downstream consumers in near real-time. Those sorts of stream problems solving can be referred to as '**Complex Event Processing**' (CEP).*

Fortunately, such stream problems can be solved using the Spark SS along with the Delta Lake integration for efficient processed event storage and consumption.Let's understand it through a real-life business problem I faced a while back (with a pinch of salt added to make it more appropriate for the given context 😉).

Note: The blog has been written with the assumption that the reader has a fundamental understanding of how Spark (or a distributed computing) framework works and some stream processing basics. If that's not the case, then let me know in the comments, and I will cover the prerequisites in some other blogs.

Let's say a business called "XYZ" deals with IOT sensors and needs a near real-time data pipeline able to ingest, process, and store the events flowing continuously through an event messaging system (Kafka) from the sensor APIs. The business wants to extract the required events based on some sampling and observed anomalies and send those out into the designated sinks. Now, as part of the requirements, below are some of the salient points.

1. Events need to be sampled on the event time windowing group of 1 minute sliding over every 30 seconds.
2. Some events can arrive in the system up to 30 seconds later than expected.
3. For every moving time window needs to have a sampling on the **first event**, the **middle event** and the **last event** with respect to that window's event timings.
4. Need to capture any anomaly events (sensors sending unusual frequency

readings) that needs to be sent out to the event messaging system (Kafka) asap for alerting.

5. All the filtered-out samplings and the anomalies are to be sent to the Delta Lake system for analytics and querying with the minimal possible latency.

Now that we have the requirements clearly laid out, let's try to understand the event data model. For the sake of brevity, I have tried to keep it very simple. The sample event 'json' shall look something like this:

```
{"device_id": 377412, "sensor_freq": 44.491581725355466, "event_time": "2023-04
```

and its equivalent object representation can be defined as below (all code examples use Scala), along with the Spark Dataframe abstraction mapping from the observed JSON event. It also demonstrates how events can be read out of a Kafka topic.

```
// Reading stream events from a Kafka topic and constructs a dataframe object
def sparkReadStreamFromKafka(spark:SparkSession,KAFKA_BOOTSTRAP_SERVERS: String
  val df=spark.readStream.format("kafka")
          .option("kafka.bootstrap.servers", KAFKA_BOOTSTRAP_SERVERS)
          .option("subscribe", KAFKA_TOPIC)
          .option("startingOffsets", "latest")
          .load()
          .selectExpr( "CAST(value AS STRING) as value")

  df
}

val sensorSchema = StructType(Seq(
    StructField("device_id", StringType, true), StructField("sensor_freq", Doub
    ,StructField("event_time",StringType,true)
  ))

val kafkaDF=sparkReadStreamFromKafka(spark,BOOTSTRAP_SERVERS,TOPIC_NAME)
      .withColumn("json_data_sample", from_json(col("value"),sensorSchema))
      .select(col("json_data_sample.*"))
```

For sampling aggregation (from requirements 1–3), we have to define a time-based (event time) sliding window that should accept data as late as up to 30 seconds using a watermarking threshold.

```
val kafkaAggDF=kafkaDF
              .withColumn("event_time",to_timestamp(col("event_time")))
              .withColumn("time_window",window($"event_time","60 seconds","30 s
              .withColumn("time_window",concat_ws("-",$"time_window.start",$"ti
```

Requirement — Samplings aggregation:

Let's now come to the most important part, which is how we should define our aggregation logic to meet the functional requirement (first, middle, and last element sampling). There are a few possible solutions that quickly come to mind.

- *Can we sort or somehow rank these events based on the event time?* — But SS allows you to perform such operations only in complete output mode after the aggregation. In 'complete' mode, the whole output result set has to be kept in memory and needs to be refreshed with the new-incoming event data. So needless to say, it is very impractical for production-like systems with continuously growing data.

- *Alternatively, if we are to write out the events to the Delta table, which gives us enough flexibility to perform those aggregations and samplings (from another pipeline), can't we then simply leverage that after the un-sampled events are written out to the Delta lake?*— This is a viable approach but deviates from the philosophy of CEP, where the data curation has to happen in near real-time, and from the streaming pipeline itself with minimum possible latency.

So can we think of some way to define our own custom stateful aggregation logic within the stream pipeline itself? Fortunately, Spark SS provides two APIs to accomplish the same thing — "**[map/flatMap]GroupsWithState**". It's a gateway to enter the custom-developed stateful stream aggregation world. This way of events processing is also referred to as "**arbitrary stateful stream**" processing. Let's see

how this can be utilized in practice to solve the problem that we are discussing.

In the below snippet, the **flatMapGroupsWithState** (**FMGWS**) API along with the watermarking threshold has been used. When comparing the mapGroupsWithState api, the basic difference is that FMGWS allows to emit 0 or any number of records in contrast to only 1 record allowed to emit for the mapGroupWithState api. Thus, it enables us to emit a group of desired sampled & anomalous (if any) records for every event time sliding window group key. Additionally, it is noteworthy that the FMGWS API works with Spark's dataset abstraction, hence conversion from Dataframe to Dataset is required before we start using it.

```
val streamedDS:Dataset[sensor_event]=kafkaAggDF.as[sensor_event]
// group key is defined based on the calculated sliding window
val aggFilteredDS=streamedDS
      .withWatermark("event_time","30 seconds")
      .groupByKey(_.time_window)
      .flatMapGroupsWithState(OutputMode.Append(),GroupStateTimeout.EventTimeTi
```

The FMGWS API takes the following attributes to be constructed:

- **Output mode (append/update):** We choose to send the events in append mode to Delta Lake.
- **State timeout semantic (event time/processing time):** We already have a designated field regarding the event occurrence time, and hence we go for the EventTimeOut semantic.
- **The custom function call to define the aggregation & timeout logic:** *transformSensorState* is the function here that performs the major part here.

Alright, now the fun part begins! Let's take a deep dive into the "***transformSensorState***" function implementation, which solves the majority of the functional requirements. It expects the following parameters to work on the stateful aggregation logic:

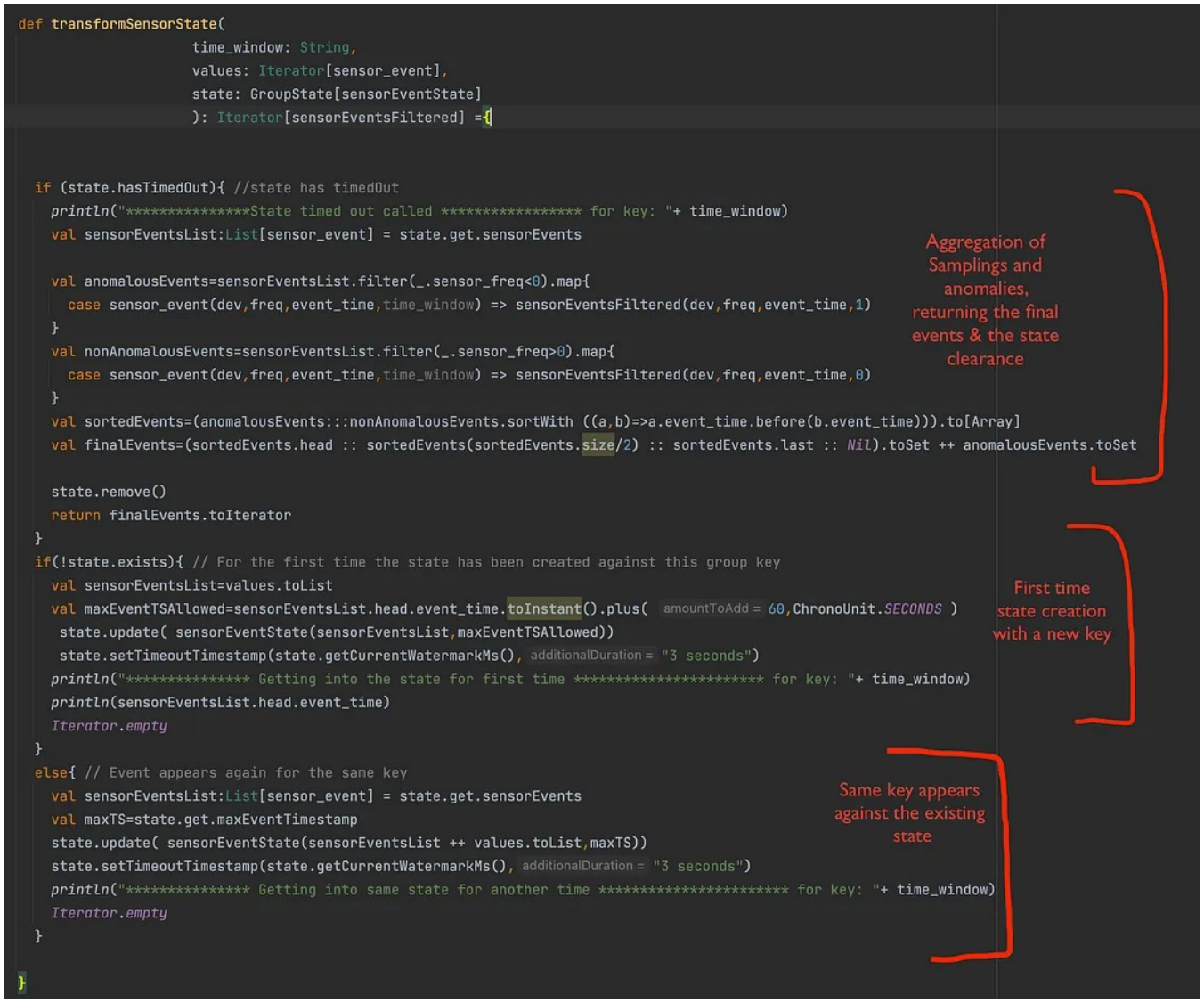
- **Grouping key:** We defined the sliding window as the key here, and it was passed to this function implicitly.
- **The values passed for the group key:** Its data type is a collection (Scala iterator) of data event object types.
- **Intermediate state object:** The object type to hold the intermediate state of the keys across the microbatches (until the state timeout).
- **Returning data object:** The aggregated data object that is to be sent to the sink.

The following code snippet shows the state and aggregated (sampled and anomalous) type objects defined.

```
// object model to hold the intermediate state of the aggregation
case class sensorEventState(sensorEvents:List[sensor_event],maxEventTimestamp:
// object model to be returned once the aggregates are finalized
case class sensorEventsFiltered(device_id:String
                                ,sensor_freq:Double,event_time:java.sql.Times
                               ,is_anomalous:Int)
```

Finally the function definition will look like below.

```
  /* takes the following parameters
    1. time_window as the key
    2. iterables of sensor_events
    3. Groupstate object to hold the intermediate state of 'sensorEventState' t
    4. Returns: 'sensorEventsFiltered' type iterator to be sent to the sink.
  */
```



I will briefly demonstrate the code block here. It can be decomposed into the following parts (or rather questions).

- **What to do when a new window key enters to the system?** — We retrieve it, construct a new state object, and store the values against the key. Finally, we define the state timeout logic (to define how long we should hold the state in the worker's memory).
- **What to do when a key is repeated & the state already exists?** — We identify it and update the state object (with additional values added) and finally update the state timeout logic (as per the defined watermark threshold) to wait for the late events.

- **What to do when there are no more entries for the same key and a state timeout is called?** — That's when/where we shall build our aggregation logic after ensuring that no more values can appear for the key or no more late events are expected to arrive. The desired aggregated events are returned through an iterator to the sink, and finally, we call the remove API for the state to free it up from the state store.

Note: The custom state aggregate function will be called from every executor as per the configured number of shuffle partitions and after the shuffle operation is invoked by the group by key. This function call will be made only for those groups present in that particular microbatch and not for other group keys stored in the state manager.The state timeouts are initiated based on the global watermark defined for the stream query, with an additional 3 seconds to just ensure that timeouts are always greater than the defined watermark.

Now, since our aggregated events are returned, we can go for writing them to the "*foreachBatch*" sink, within which we simply construct a Delta writer that intends to write the output to a partitioned Delta path with 15-second trigger intervals. Below is the snippet, which is pretty much a no-brainer.

```
def delta_writer(df:DataFrame,path:String,partition_by:String): Unit ={
  df.write.partitionBy(partition_by).mode("append").option("mergeSchema", "true
    .delta(path)
}

val streamingQueryForAggregation=aggFilteredDS.toDF().writeStream.foreachBatch(
     (outputDF:DataFrame,batch_id:Long)=>{

         delta_writer(outputDF.drop($"time_window").withColumn("part_date", to_
           deltaDir, "part_date")

       } ).option("checkpointLocation", aggCheckpointDir)
     .queryName("Sampling_Agg_Stream")
     .trigger(Trigger.ProcessingTime("10 seconds"))
     .start()
```

So if we tie-back again with the requirements list discussed before, we are now left

with the anomaly tracker part. Let's smash that.

Requirement — Anomaly alerts:

This one is very straightforward, unlike the previous one. No aggregation, state management, late event waits, etc. We just pick the events in a microbatch, filter them based on the unusual sensor frequencies observed, and just send the filtered ones to a Kafka topic asap.

```
val kafkaAnomalyTrackerDF=kafkaDF.filter("sensor_freq<0")
val streamingQueryForAnomalies=kafkaAnomalyTrackerDF.writeStream.foreachBatch(
      (anomalousDF:DataFrame,batch_id:Long)=>{

        if(!anomalousDF.head(1).isEmpty){
          kafka_writer(anomalousDF,ANOMALY_TOPIC_NAME,BOOTSTRAP_SERVERS)
        }
      }
    ).option("checkpointLocation", anomalyCheckpointDir)
     .queryName("Anomaly_Tracker_Stream")
     .trigger(Trigger.ProcessingTime("5 seconds"))
     .start()
```

Co-existing multiple stream queries:

You might have noticed that to serve both sampling aggregation (writing to Delta) and anomaly tracking (writing to Kafka), two stream queries have been initiated. The stream queries start immediately with independent query execution threads to serve the respective consumers asap (based on the respective trigger intervals defined) with relevant pieces of desired data.

But if we start both of these streams, even though they will try to run in parallel, they will still be in the same Spark default pool in a FIFO queue in which compute resource sharing isn't that great (where an independent job has to wait for its previous job to finish to get resource sharing). To avoid that, we can specify dedicated Spark pools for both of the stream queries, respectively.

```
 spark.sparkContext.setLocalProperty("spark.scheduler.pool", "agg_pool")
// start streamingQueryForAggregation stream query
```

```
 spark.sparkContext.setLocalProperty("spark.scheduler.pool", "anomaly_pool")
// start streamingQueryForAnomalies stream query
```

As coded above, setting the dedicated pool right before the stream query is started will ensure that it runs on that specific pool.

Some streaming performance improvement measures:

1. **State store manager:** The default state store used by 'spark' is "HDFSBackedStateStore", where the state keys are maintained in executor JVM heap memory and go for frequent GC. Configuring the RocksDB state store ensures keys are maintained in the executor's non-heap memory (prune to GC). It is also able to hold more keys than the default because it is a specialized in-memory key-value database from its implementation.In cases where there is high memory pressure in the cluster the number of keys is often the problem and enabling this option should improve performance, however, sometimes it does add additional latency to processing. The simple way to configure it for the **Databricks** run time—> *spark.conf.set("spark.sql.streaming.stateStore.providerClass","com.databricks.sql.streaming.state.RocksDBStateStoreProvider")*

2. **Shuffle partitions:** For stream aggregation queries, the data shuffling is an unavoidable occurrence. The default number of shuffled partitions is 200. But you may not need 'this number'. Identify the volume of events to be processed in a microbatch trigger. Up to 50 MB of data per partition can be a good starting point to experiment with. Based on that, it is better to keep shuffle partitions equal to the total number of cores in the cluster for optimal utilization of the compute slots. *Note: shuffle partition number can't be changed for a running stream query (on restarts) with an active checkpoint.*

3. **Async state commit:** For stateful stream queries, after the partitions are done with their respective set of operations for a microbatch, they tend to log the state information along with the processed Kafka offset range (a separate offsets sub

path) in a persistent storage like a cloud object store (ADLS, S3, etc.) as a checkpoint. The state checkpoint location pattern is *"<root_checkpoint_path>/state/operatorId=0/partitionId=0"*. The operatorId is the state operator used (flatMapGroupsWithState for our example), and the partitionId is the partition index of the total number of shuffle partitions (starting from 0 index). However, by default, this state checkpointing happens as part of the microbatch activity, and as long as the state commit operation is not done, the microbatch is not considered to be completed.If the state commit operation time is longer (as observed from the driver logs), then making it an asynchronous activity can minimise the batch duration to some extent without having to wait for the state commit to be completed. As of now, this option is available for **Databricks run time** with the RocksDB state store only. Just a single configuration set will do the job — *spark.conf.set("spark.databricks.streaming.statefulOperator.asyncCheckpoint.enable,"true")*

4. **Right trigger interval:** SS works in a micro-batch way with a configurable trigger interval (how often micro-batches spawn). This micro-batch architecture adds an acceptable or negligible delay in the stream processing and event consumption, making SS a near-real-time stream processing framework. However, with a desire to reduce the latency, it may not always be beneficial to reduce the trigger interval and spawn the micro-batches more often. Too frequent micro-batches frequently attempt to launch tiny tasks to look for data from the source, making the read-stage throughput not that great. Also, it affects the write sink's health, especially if they are persistent stores like data lakes or lakehouses with possibly too many small files. Trying to know the acceptable latency and event load (for example, the expected average number of events per second) from the source and, based on that, deciding the micro-batch interval should be the way to go. Setting the right trigger interval is tricky and use-case-based. There is always a trade-off between latency and throughput, and stressing too much on one can hurt the other.

That is pretty much all (at least for the time being). We tried to understand what a complex event processing thing really is and how to approach it using Spark structured streaming with some performance enhancement best practices. Hope it is able to help the readers and the community in someway in their data streaming exploration path.

I am hopeful to come back with some more findings and lookouts around the same subject line in the future. Until then...

happy learning! 😊

References:

Structured Streaming Programming Guide

Structured Streaming is a scalable and fault-tolerant stream processing engine built on the Spark SQL engine. You can...

spark.apache.org

Asynchronous state checkpointing for Structured Streaming

For stateful streaming queries bottlenecked on state updates, enabling asynchronous state checkpointing can reduce...

docs.databricks.com

Streaming in Production: Collected Best Practices

Releasing any data pipeline or application into a production state requires planning, testing, monitoring, and...

www.databricks.com

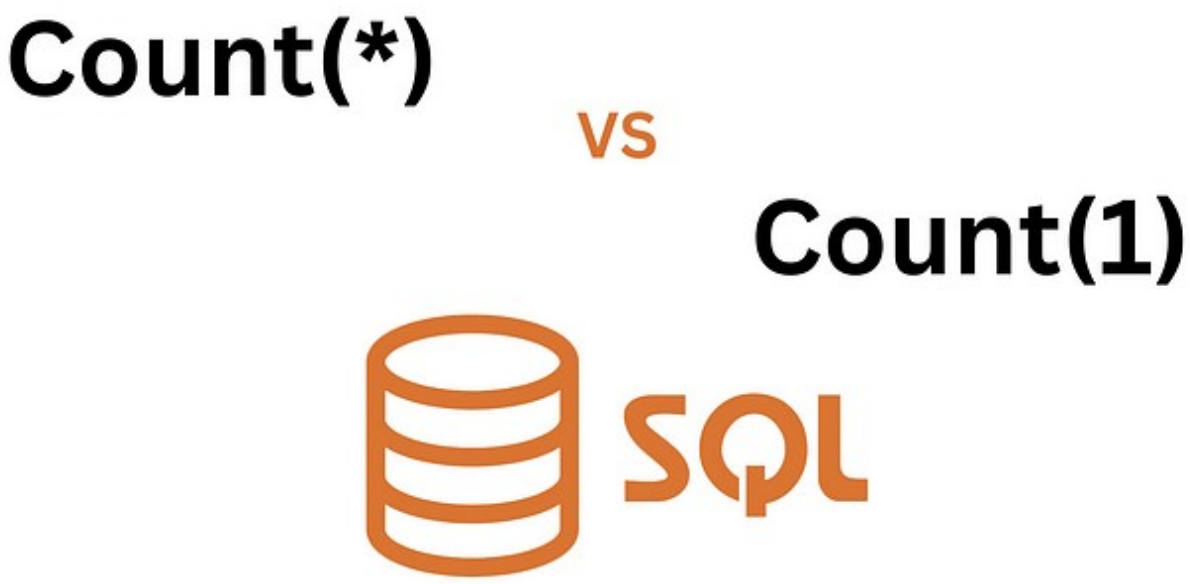
Follow

Written by Indranil Tarafdar

9 Followers

A data enthusiast

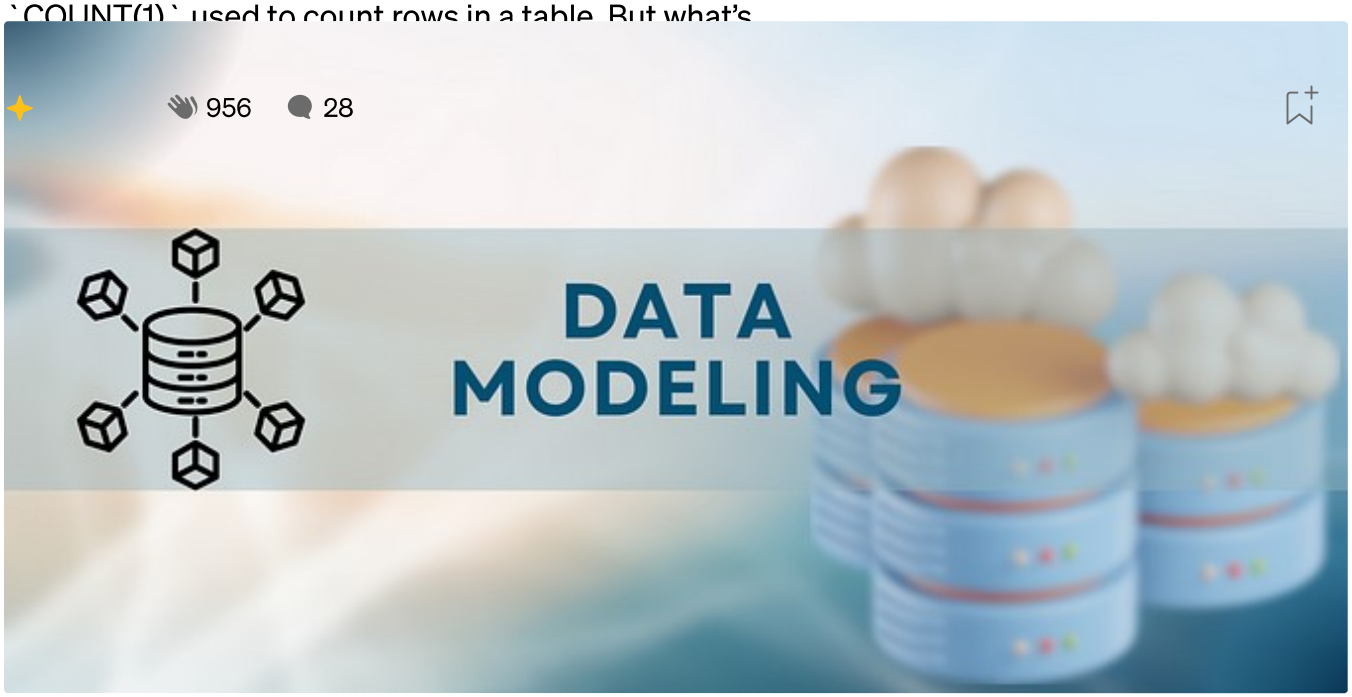
Recommended from Medium



Vishal Barvaliya in Data Engineer

Count(*) vs Count(1) in SQL.

If you've spent any time writing SQL queries, you've probably seen both `COUNT(*)` and `COUNT(1)` used to count rows in a table. But what's



Feruz Urazaliev

Data Modeling Techniques: Relational vs. Dimensional

This article explores the critical differences and applications of relational and dimensional data modeling techniques. It provides an...

6d ago 1

Lists

Staff Picks

714 stories · 1222 saves

Stories to Help You Level-Up at Work

19 stories · 739 saves

Self-Improvement 101

20 stories · 2530 saves

Productivity 101

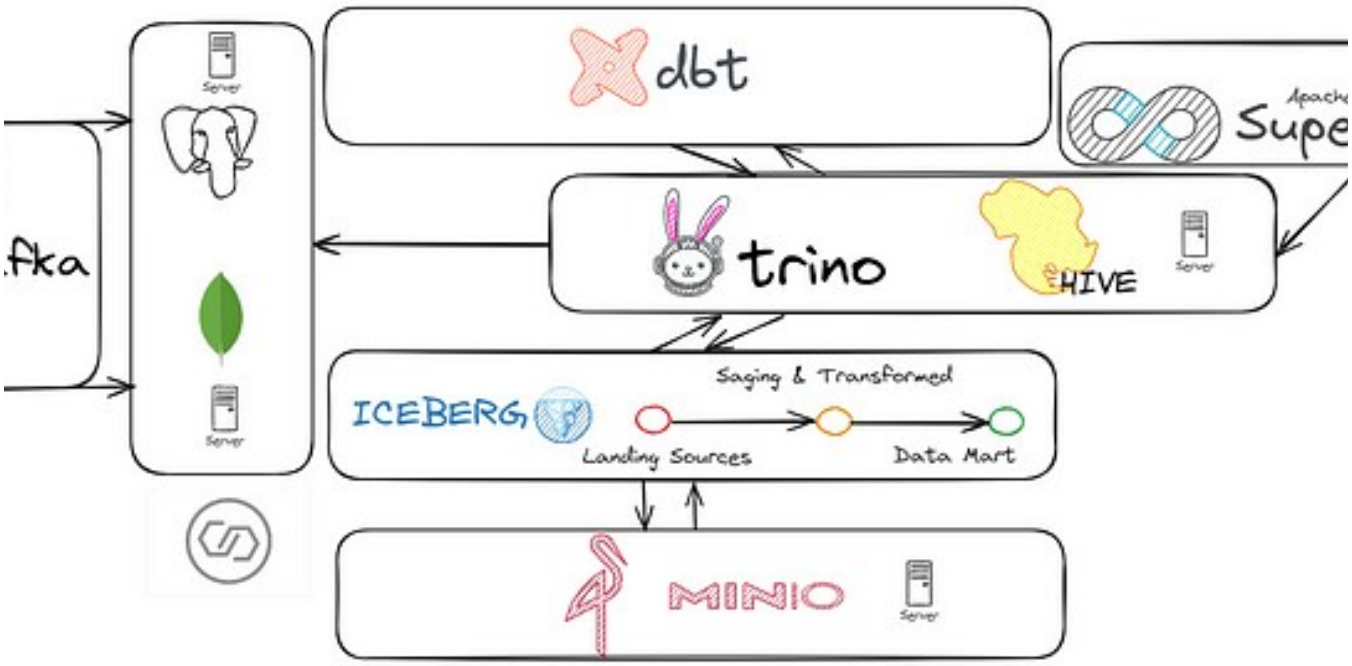
20 stories · 2195 saves

Nethaji Kamalapuram

Spark Data Caching, Repartition, Coalesce, Data Frame Hints

Let's deep dive into the data caching concept in spark, why it's needed, different caching techniques and parameters, what is best fit for...

May 15 5



Stefentaime

Iceberg + Dbt + Trino + Hive : modern, open-source data stack

To provide a deeper understanding of how the modern, open-source data stack consisting of Iceberg, dbt, Trino, and Hive operates within a...

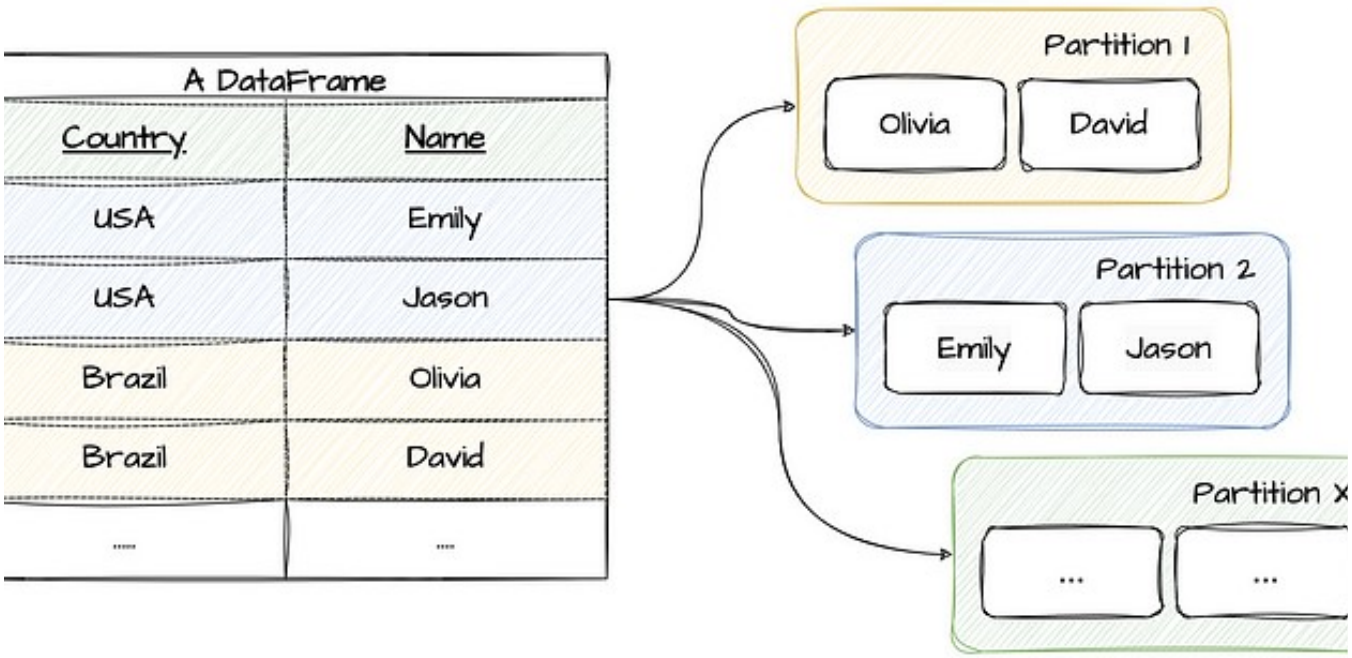
Mar 7 56 1

Lenon Rodrigues

A Detailed Comparison between Spark Structured Streaming and Apache Flink: Comparison of Features...

Introduction

May 16 105 1



Lubomir Franko in Python in Plain English

The Truth About PySpark's Repartition: Prepare to Be Surprised!

Paritioning concept, Image by Author

Jul 4 293 3

See more recommendations