Hands-On Introduction to Delta Lake with (py)Spark

Concepts, theory, and functionalities of this modern data storage framework

João Pedro · Follow
Published in Towards Data Science
10 min read · Feb 16, 2023

Listen Share More

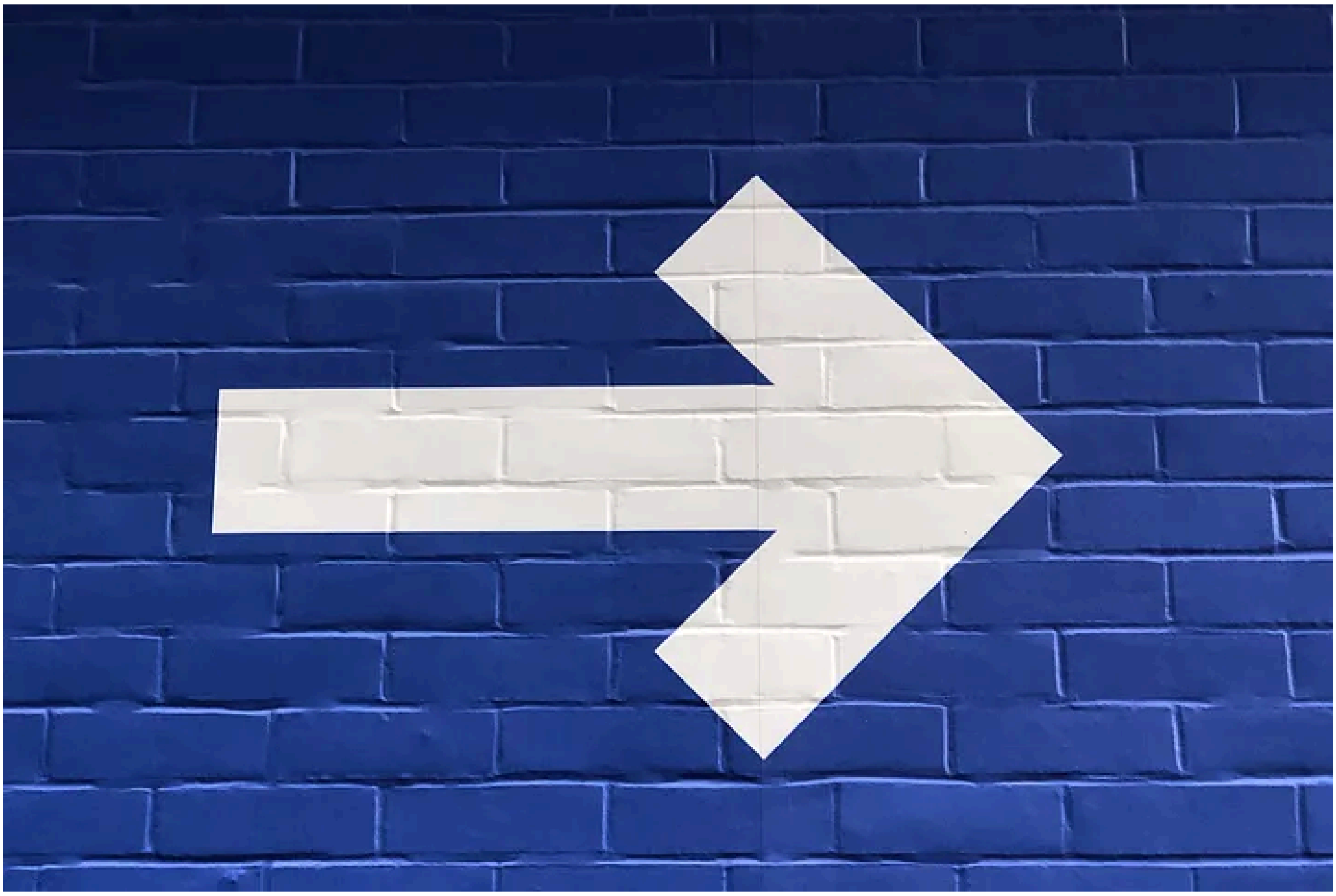


Photo by Nick Fewings on Unsplash

Introduction

I think it's now perfectly clear to everybody the value data can have. To use a hyped example, models like ChatGPT could only be built on a huge mountain of data, produced and collected over years.

I would like to emphasize the word "can" because there is a phrase in the world of programming that still holds, and probably ever will: garbage in, garbage out. Data by itself has no value, it needs to be organized, standardized, and clean. Governance is needed. In this context, data management in an organization is a key point for the success of its projects involving data.

One of the main aspects of correct data management is the definition of a data architecture. Data architecture is the set of practices, technologies, and services that meet the data demand of a given organization, both technical (speed, volume, frequency, availability) and non-technical (business rules, compliance with data legislation) needs.

Nowadays, almost by default, organizations will have to deal with data in different formats (CSV, pdf, video, parquet, etc), hence the success of blob storage like amazon's S3. However, this type of approach can bring some problems due to the absence of management tools on raw files (especially in tabular data), such as schema enforcement, versioning, and data lineage.

With that in mind (and a bunch of other things), Delta Lake was developed, an open-source data storage framework that implements/materializes the Lakehouse architecture and the topic of today's post.

What is Delta Lake?

Before going into further details on Delta Lake, we need to remember the concept of Data Lake, so let's travel through some history.

The Data Lake architecture was proposed in a period of great growth in the data volume, especially in non-structured and semi-structured data, when traditional Data Warehouse systems start to become incapable of dealing with this demand.

The proposal is simple — "Trow everything you have here inside and worry later". The main player in the context of the first data lakes was Hadoop, a distributed file system, with MapReduce, a processing paradigm built over the idea of minimal data movement and high parallelism. In theory, was just throwing everything inside Hadoop and later on writing jobs to process the data into the expected results, getting rid of complex data warehousing systems.

Legend says, that this didn't go well. The files were thrown with no quality worries, no versioning, and no management. The data became useless. The problem was so big that the terms "data swamp", a joke on very messy data lakes, and "WORN paradigm", Write Once Read Never, were created. In practice, the guarantees imposed by traditional Data Warehouse systems, especially RDBMS were still needed to assure data quality. (I was a child at the time, I read all this history recently from modern literature)

Time has passed and, based on the hits and misses of the past, new architectures were proposed. The Lakehouse architecture was one of them. In a nutshell, it tries to mix the advantages of both Data Lakes (flexibility) and Data Warehouses (guarantees).

Delta Lake is nothing more than a practical implementation of a storage framework/solution with a Lakehouse vision.

Let's go to it:

A table in Delta Lake (aka Delta Table) is nothing more than a *parquet* file with a transaction log in JSON that stores all the change history on that file. In that way, even with data stored in files, it is possible to have total control over all that happened to it, including reading previous versions and reverting operations. Delta Lake also works with the concept of ACID transactions, that is, no partial writing caused by job failures or inconsistent readings. Delta Lake also refuses writes with wrongly formatted data (schema enforcement) and allows for schema evolution. Finally, it also provides the usual CRUD functionalities (insert, update, merge, and delete), usually not available in raw files.

This post will tackle these functionalities in a hands-on approach with pyspark in the following sections.

The data

The data used in this post is the list of traffic accidents that occurred on Brazillian highways, collected by the PRF (Polícia Rodoviária Federal, our highway police) and publicly available in the Brazillian Open Data Portal [Link][License — CC BY-ND 3.0].

The data covers the period from 2007 up to 2021 and contains various information about the accidents: place, highway, km, latitude and longitude, number of people involved, accident type, and so on.

The implementation

0. Setting Up the environment

As always, the project is developed using docker containers:

```
version: '3'
services:
  spark:
    image: bitnami/spark:3.3.1
    environment:
      - SPARK_MODE=master
    ports:
      - '8080:8080'
      - '7077:7077'
    volumes:
      - ./data:/data
      - ./src:/src
  spark-worker:
    image: bitnami/spark:3.3.1
    environment:
      - SPARK_MODE=worker
      - SPARK_MASTER_URL=spark://spark:7077
      - SPARK_WORKER_MEMORY=4G
      - SPARK_EXECUTOR_MEMORY=4G
      - SPARK_WORKER_CORES=4
    ports:
      - '8081:8081'
    volumes:
      - ./data:/data
      - ./src:/src
  jupyter:
    image: jupyter/pyspark-notebook:spark-3.3.1
    ports:
      - '8890:8888'
    volumes:
      - ./data:/data
```

All the code is available in this GitHub repository.

1. Creating a Delta Table

The first thing to do is instantiate a Spark Session and configure it with the Delta-Lake dependencies.

```
# Install the delta-spark package.
```

```
!pip install delta-spark
```

```
from pyspark.sql import SparkSession
from pyspark.sql.types import StructField, StructType, StringType, IntegerType, DoubleType
import pyspark.sql.functions as F

from delta.pip_utils import configure_spark_with_delta_pip

spark = (
    SparkSession
    .builder.master("spark://spark:7077")
    .appName("DeltaLakeFundamentals")
    .config("spark.sql.extensions", "io.delta.sql.DeltaSparkSessionExtension")
    .config("spark.sql.catalog.spark_catalog", "org.apache.spark.sql.delta.catalog.DeltaCatalog")
)

spark = configure_spark_with_delta_pip(spark).getOrCreate()
```

Creating a Delta Table is very simple, it's just like writing a new file in a specific format. The code below reads the CSV with the 2020's accidents and writes the data as a delta table.

```
SCHEMA = StructType(
    [
        StructField('id', StringType(), True),          # ACCIDENT ID
        StructField('data_inversa', StringType(), True),# DATE
        StructField('dia_semana', StringType(), True),  # DAY OF WEEK
        StructField('horario', StringType(), True),     # HOUR
        StructField('uf', StringType(), True),          # BRAZILIAN STATE
        StructField('br', StringType(), True),          # HIGHWAY
        # AND OTHER FIELDS OMITTED TO MAKE THIS CODE BLOCK SMALL
    ]
)
```



```
    .option("header", "true")
    .option("encoding", "ISO-8859-1")
    .schema(SCHEMA)
    .load("/data/acidentes/datatran2020.csv")
)

df_acidentes.show(5)
```

```
+------+------------+-------------+--------+---+---+-----+--------------------+--------------------+--------------------+----------------------+---------
-+-----------+-------------------------+-----------+-----------+---------+-------+------+-------------+--------------+------+----------+-------+--------
--------+------------+--------+---------+---------------+
|    id|data_inversa|   dia_semana| horario| uf| br|   km|           municipio|      causa_acidente|       tipo_acidente|classificacao_acidente| fase_di
a|sentido_via|condicao_metereologica|tipo_pista|tracado_via|uso_solo|pessoas|mortos|feridos_leves|feridos_graves|ilesos|ignorados|feridos|veiculos|
latitude|   longitude|regional|delegacia|            uop|
+------+------------+-------------+--------+---+---+-----+--------------------+--------------------+--------------------+----------------------+---------
-+-----------+-------------------------+-----------+-----------+---------+-------+------+-------------+--------------+------+----------+-------+--------
--------+------------+--------+---------+---------------+
|260068|  2020-01-01|quarta-feira|05:40:00| PA|316|   84|SAO FRANCISCO DO ...|Falta de Atenção ...|Saída de leito ca...|   Com Vítimas Feridas|Pleno di
a|Decrescente|              Céu Claro|   Simples|       Reta|     Não|      2|     0|            2|             0|     0|        0|      2|       1|  -
1.3101929|-47.74456398| SPRF-PA| DEL01-PA|UOP02-DEL01-PA|
|260073|  2020-01-01|quarta-feira|06:00:00| MG|262|  804|             UBERABA|Falta de Atenção ...| Colisão transversal|   Com Vítimas Feridas|Pleno di
a|Decrescente|              Céu Claro|     Dupla|       Reta|     Sim|      4|     0|            1|             0|     3|        0|      1|       2|-19.
76747537|-47.98725511| SPRF-MG| DEL13-MG|UOP01-DEL13-MG|
|260087|  2020-01-01|quarta-feira|06:00:00| BA|116|  191|             CANUDOS|   Condutor Dormindo|Saída de leito ca...|    Com Vítimas Fatais|Pleno di
a|  Crescente|                Nublado|   Simples|       Reta|     Não|      1|     1|            0|             0|     0|        0|      0|       1|-10.
32002103|-39.06425211| SPRF-BA| DEL07-BA|UOP02-DEL07-BA|
|260116|  2020-01-01|quarta-feira|10:08:00| SP|116|   71|           APARECIDA|Não guardar distâ...|    Colisão traseira|   Com Vítimas Feridas|Pleno di
a|  Crescente|                    Sol|     Dupla|       Reta|     Sim|      3|     0|            2|             0|     1|        0|      2|       2|-22.
85651665|-45.23114328| SPRF-SP| DEL08-SP|UOP01-DEL08-SP|
|260129|  2020-01-01|quarta-feira|12:10:00| MG|262|380,9|             JUATUBA|   Condutor Dormindo|Saída de leito ca...|   Com Vítimas Feridas|Pleno di
a|  Crescente|              Céu Claro|     Dupla|      Curva|     Não|      1|     0|            1|             0|     0|        0|      1|       1|  -1
9.947864|  -44.381226| SPRF-MG| DEL01-MG|UOP03-DEL01-MG|
+------+------------+-------------+--------+---+---+-----+--------------------+--------------------+--------------------+----------------------+---------
-+-----------+-------------------------+-----------+-----------+---------+-------+------+-------------+--------------+------+----------+-------+--------
--------+------------+--------+---------+---------------+
only showing top 5 rows
```

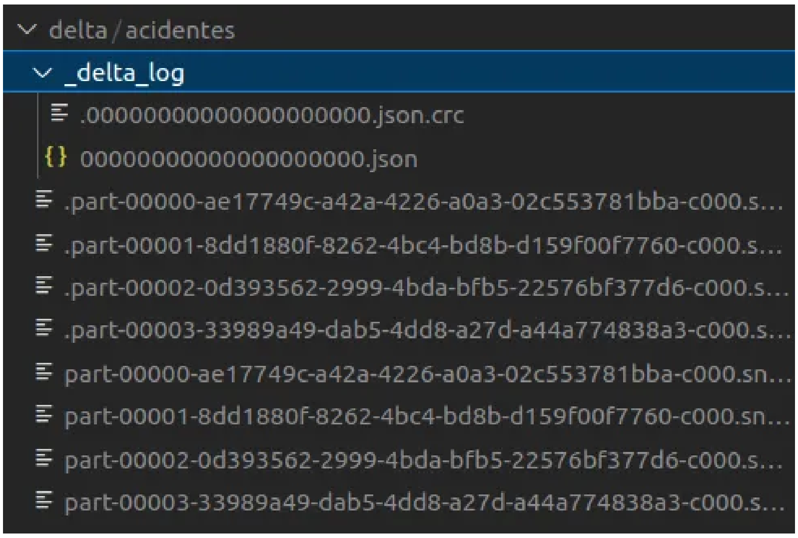
First 5 rows of 2020.

Writing the delta table.

```
df_acidentes\
    .write.format("delta")\
    .mode("overwrite")\
    .save("/data/delta/acidentes/")
```

And that's all.

As mentioned, a Delta-Lake table (in terms of files) is just the traditional *parquet* file with a transaction log in JSON that stores all the changes made.



Delta Table with the JSON Transaction Log.

2. Read from a Delta Table

Again, there is nothing (yet) special about reading a Delta Table.

```
df_acidentes_delta = (
    spark
    .read.format("delta")
    .load("/data/delta/acidentes/")
)
```

```
df_acidentes_delta.select(["id", "data_inversa", "dia_semana", "horario", "uf"]).show(5)
```

```
+------+------------+----------+--------+---+
|    id|data_inversa|dia_semana| horario| uf|
+------+------------+----------+--------+---+
|263804|  2020-01-19|   domingo|12:30:00| SC|
|263806|  2020-01-19|   domingo|14:50:00| SC|
|263807|  2020-01-19|   domingo|14:00:00| RJ|
|263809|  2020-01-19|   domingo|15:30:00| RS|
|263812|  2020-01-19|   domingo|15:15:00| GO|
+------+------------+----------+--------+---+
only showing top 5 rows
```

Let’s count the number of rows

```
df_acidentes_delta.count()

>> Output: 63576
```

3. Add new data to the Delta Table

Delta Tables support the “append” write mode, so it’s possible to add new data to the already existing table. Let’s add the readings from 2019.

```
# READING THE 2019 DATA
df_acidentes_2019 = (
    spark
    .read.format("csv")
    .option("delimiter", ";")
    .option("header", "true")
    .schema(SCHEMA)
    .load("/data/acidentes/datatran2019.csv")
)
```

Appending to the Delta Table

```
df_acidentes_2019\
    .write.format("delta")\
```

```
        .mode("append")\
        .save("/data/delta/acidentes/")
```

It’s important to empathize: Delta Tables will perform schema enforcement, so it’s only possible to write data that have the same schema as the already existing table, otherwise, Spark will throw an error.

Let’s check the number of rows in the Delta Table

```
df_acidentes_delta.count()

>> Output: 131132
```

4. View the history (logs) of the Delta Table

The Log of the Delta Table is a record of all the operations that have been performed on the table. It contains a detailed description of each operation performed, including all the metadata about the operation.

To read the log, we need to use a special python object called `DeltaTable`

```
from delta.tables import DeltaTable

delta_table = DeltaTable.forPath(spark, "/data/delta/acidentes/")
delta_table.history().show()
```

```
|version|           timestamp|userId|userName|operation| operationParameters| job|notebook|clusterId|readVersion|isolationLevel|isBlindAppend|    opera
tionMetrics|userMetadata|          engineInfo|
|      1|2023-02-14 01:12:...|  null|    null|    WRITE|{mode -> Append, ...|null|    null|     null|          0|  Serializable|         true|{numFiles
-> 4, n...|        null|Apache-Spark/3.3....|
|      0|2023-02-14 01:11:...|  null|    null|    WRITE|{mode -> Overwrit...|null|    null|     null|       null|  Serializable|        false|{numFiles
-> 4, n...|        null|Apache-Spark/3.3....|
```

The history object is a Spark Data Frame.

```
delta_table.history().select("version", "timestamp", "operation", "operationParameters").show(10, F
```

```
+-------+-----------------------+---------+-----------------------------------------+
|version|timestamp              |operation|operationParameters                      |
+-------+-----------------------+---------+-----------------------------------------+
|1      |2023-02-14 01:12:04.86 |WRITE    |{mode -> Append, partitionBy -> []}      |
|0      |2023-02-14 01:11:28.625|WRITE    |{mode -> Overwrite, partitionBy -> []}|
+-------+-----------------------+---------+-----------------------------------------+
```

As we can see, there are currently two table versions, one for each operation performed: the overwrite write when the table was created and the append write made previously.

5. Read a specific version of the Delta Table

If nothing is specified, Spark will read the latest version of the Delta Table.

```
df_acidentes_latest = (
    spark
    .read.format("delta")
    .load("/data/delta/acidentes/")
)
df_acidentes_latest.count()

>> Output: 131132
```

But it's also possible to read from a specific version by just adding a single line of code:

```
df_acidentes_version_0 = (
    spark
    .read.format("delta")
    .option("versionAsOf", 0)
    .load("/data/delta/acidentes/")
)
df_acidentes_version_0.count()

>> Output: 63576
```

The counts dropped because we're reading from version 0, before the 2019's data was inserted.

6. Revert to a previous version

It's possible to revert to a previous version of a table. This is very useful to quickly solve errors made by a pipeline. This operation is also performed via the DeltaTable object created earlier.

Let's restore the table to version 0:

```
delta_table.restoreToVersion(0)
```

Now, the latest counts will be =63576 again, because we reverted to a version when the data from 2019's have yet not been included.

```
# Counting the number of rows in the latest version
df_acidentes_latest.count()
```

The **RESTORE** operation is also stored in the log. So, in practice, no information is lost:

```
delta_table.history().select("version", "timestamp", "operation", "operationParameters").show(10, F
```

```
+-------+-----------------------+---------+-------------------------------------+
|version|timestamp              |operation|operationParameters                  |
+-------+-----------------------+---------+-------------------------------------+
|2      |2023-02-14 01:33:27.373|RESTORE  |{version -> 0, timestamp -> null}    |
|1      |2023-02-14 01:12:04.86 |WRITE    |{mode -> Append, partitionBy -> []}  |
|0      |2023-02-14 01:11:28.625|WRITE    |{mode -> Overwrite, partitionBy -> []}|
+-------+-----------------------+---------+-------------------------------------+
```

Let’s restore back to version 1.

```
delta_table.restoreToVersion(1)
```

7. Update

The update operation can also be done by the `DeltaTable` object, but we will perform it with the SQL syntax, just to try a new approach.

First, let’s write the data from 2016 to the delta table. This data contains the “data_inversa” (date) column wrongly formatted: dd/MM/yy instead of yyyy-MM-dd

```
df_acidentes_2016 = (
    spark
    .read.format("csv")
    .option("delimiter", ";")
    .option("header", "true")
    .option("encoding", "ISO-8859-1")
    .schema(SCHEMA)
    .load("/data/acidentes/datatran2016.csv")
)

df_acidentes_2016.select("data_inversa").show(5)
```

```
+------------+
|data_inversa|
+------------+
|    10/06/16|
|    01/01/16|
|    01/01/16|
|    01/01/16|
|    01/01/16|
+------------+
only showing top 5 rows
```

Let’s save the data:

```
df_acidentes_2016\
    .write.format("delta")\
    .mode("append")\
    .save("/data/delta/acidentes/")

df_acidentes_latest.count()
>> Output: 227495
```

But, because our data_inversa field is of type string, no errors occur. Now, we have bad data inserted on our table that we need to fix. Of course, we could just REVERT this last operation and insert the data again correctly, but let's use the UPDATE operation instead.

The SQL code below fixes the data formatting just for the year = 2016.

```
df_acidentes_latest.createOrReplaceTempView("acidentes_latest")

spark.sql(
    """
    UPDATE acidentes_latest
    SET data_inversa = CAST( TO_DATE(data_inversa, 'dd/MM/yy') AS STRING)
    WHERE data_inversa LIKE '%/16'
    """
)
```

And the number of rows with wrongly formatted data is 0:

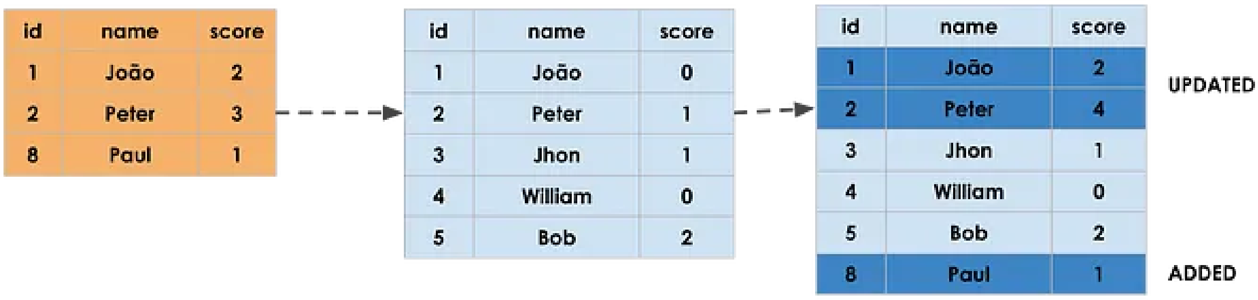
```
df_acidentes_latest.filter( F.col("data_inversa").like("%/16") ).count()
>> Output: 0
```

8. Merge

The last operation that'll be covered is the MERGE (a.k.a UPSERT) operation. It's a mix of INSERT and UPDATE,

It will try to insert new rows to a target table considering some columns as keys. If the row to be inserted already exists in the target table (i.e. the row keys are already present in the target table), it will just update the row (with some logic specified), other else, it will insert the new row.

In a nutshell: if exists, then update, if not, then insert.



id	name	score
1	João	2
2	Peter	3
8	Paul	1

id	name	score
1	João	0
2	Peter	1
3	Jhon	1
4	William	0
5	Bob	2

id	name	score	
1	João	2	UPDATED
2	Peter	4	
3	Jhon	1	
4	William	0	
5	Bob	2	
8	Paul	1	ADDED

Merge example. Image by Author.

To demonstrate this method let's insert some data from 2018 with the number of *people* = 0 (*pessoas —* number of people involved in the accident) for all rows, simulating a partial report with incomplete data.

```
# FULL DATA FROM 2018
df_acidentes_2018 = (
    spark
    .read.format("csv")
    .option("delimiter", ";")
    .option("header", "true")
    .option("encoding", "ISO-8859-1")
    .schema(SCHEMA)
    .load("/data/acidentes/datatran2018.csv")
)

# SAMPLE WITH pessoas=0
df_acidentes_2018_zero = (
  df_acidentes_2018
  .withColumn("pessoas", F.lit(0))
  .limit(1000)
)

df_acidentes_2018_zero\
    .write.format("delta")\
    .mode("append")\
    .save("/data/delta/acidentes/")
```

If we now want to update the table with the complete data from 2018, we must assure that the already inserted rows have just the *people* column updated and all the new rows are inserted.

This can be performed with the following MERGE operation, which considers the accident's id and date as keys:

```
df_acidentes_latest.createOrReplaceTempView("acidentes_latest")
df_acidentes_2018.createOrReplaceTempView("acidentes_2018_new_counts")

spark.sql(
    """
    MERGE INTO acidentes_latest
    USING acidentes_2018_new_counts

    ON acidentes_latest.id = acidentes_2018_new_counts.id
    AND acidentes_latest.data_inversa = acidentes_2018_new_counts.data_inversa
```

```
        WHEN MATCHED THEN
            UPDATE SET pessoas = acidentes_latest.pessoas + acidentes_2018_new_counts.pessoas

        WHEN NOT MATCHED THEN
            INSERT *
        """
    )
```

Conclusion

Defining a data architecture is extremely important to all organizations that aim at creating data-driven products, like BI reports and Machine Learning applications. A data architecture defines the tools, technologies, and practices that will ensure that the technical and non-technical data needs of an organization are met.

In private companies, it can help to speed up the development of such products, improve their quality and efficiency, and give commercial advantages that turn into profit. In public organizations, the benefits of a data architecture turn into better public policies, a better understanding of the current situation in a specific area like transport, safety, budget, and improvement in transparency and management.

Many architectures have been proposed in the last decades, each one with its own benefits in each context. The Lakehouse paradigm tries to mix together the benefits of Data Lakes and Data Warehouses. The Delta Lake is a framework for storage based on the Lakehouse paradigm. In a nutshell, it brings many of the guarantees usually only available in classical RDBMS (ACID transactions, logs, revert operations, CRUD operations) on top of file-based storage (based on *parquet*).

In this post, we explored a few of these functionalities using data from traffic accidents on Brazilian highways. I hope I helped you somehow, I am not an expert in any of the subjects discussed, and I strongly recommend further reading (see some references below) and discussion.

Thank you for reading! ;)

References

> All the code is available in this GitHub repository.

[1] Chambers, B., & Zaharia, M. (2018). *Spark: The definitive guide: Big data processing made simple*. " O'Reilly Media, Inc."
[2] Databricks. (2020, March 26). *Tech Talk | Diving into Delta Lake Part 1: Unpacking the Transaction Log* [Video]. YouTube.
[2] *How to Rollback a Delta Lake Table to a Previous Version with Restore*. (2022, October 3). Delta Lake. Link
[3] *Delta Lake Official Page*. (n.d.). Delta Lake. https://delta.io/
[4] Databricks. (2020a, March 12). *Simplify and Scale Data Engineering Pipelines with Delta Lake* [Video]. YouTube.
[5] Databricks. (2020c, September 15). *Making Apache SparkTM Better with Delta Lake* [Video]. YouTube.
[6] Reis, J., & Housley, M. (2022c). *Fundamentals of Data Engineering: Plan and Build Robust Data Systems* (1st ed.). O'Reilly Media.

Data Science · Data Engineering · Spark · Data Lake · Pyspark

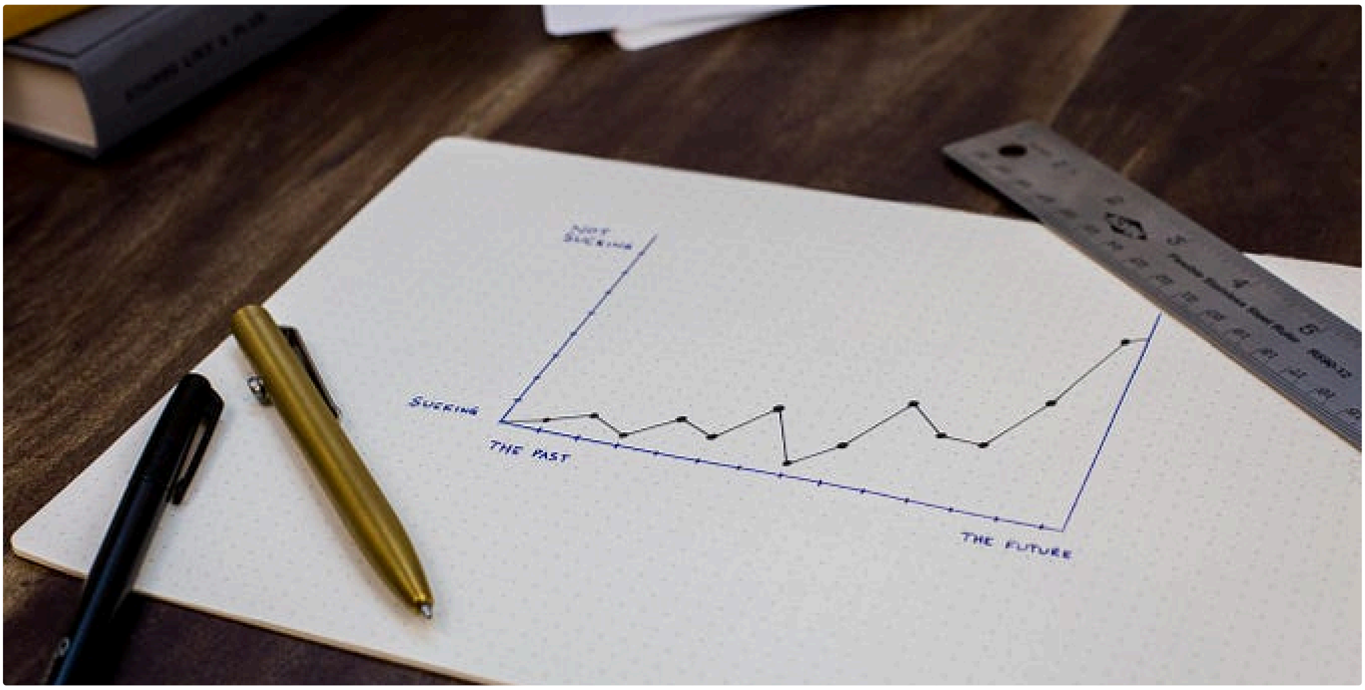
Follow

Written by João Pedro

390 Followers · Writer for Towards Data Science

Bachelor of IT at UFRN. Graduate of BI at UFRN — IMD. Strongly interested in Machine Learning, Data Science and Data Engineering.

More from João Pedro and Towards Data Science

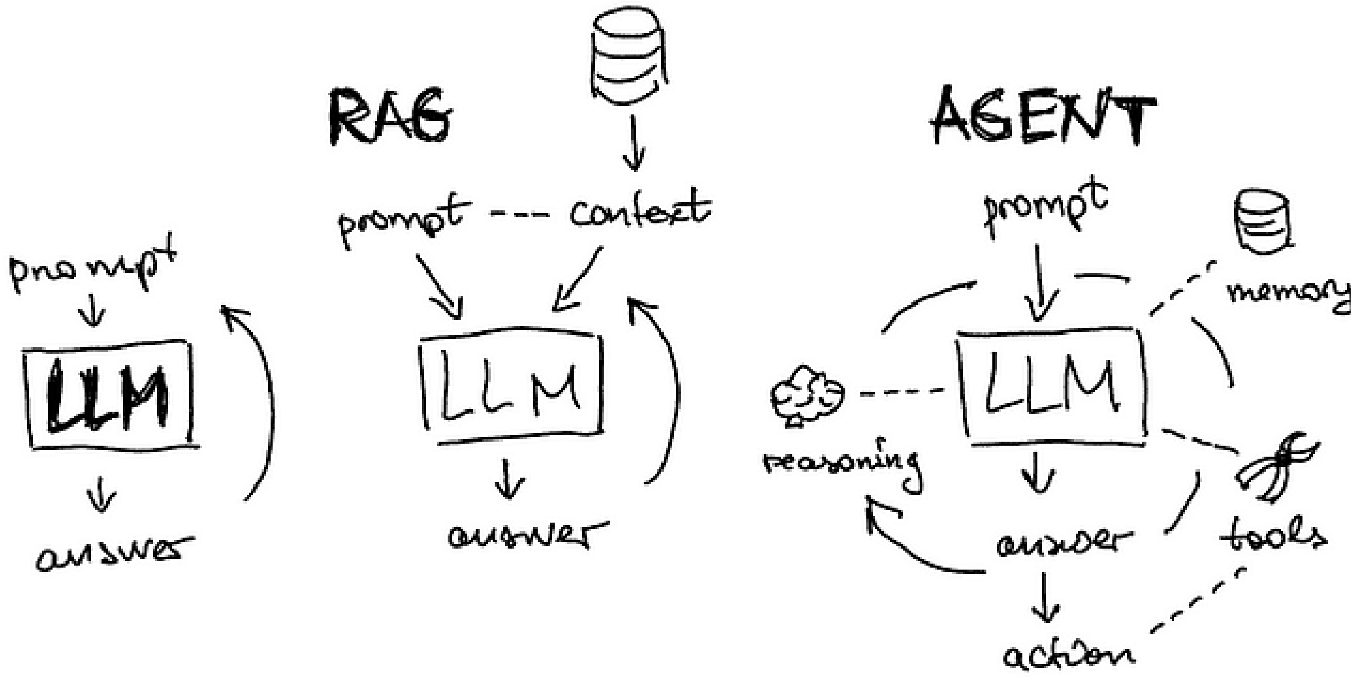


João Pedro in Towards Data Science

Understanding Topic Coherence Measures

11 min read · Jan 10, 2022

182 4



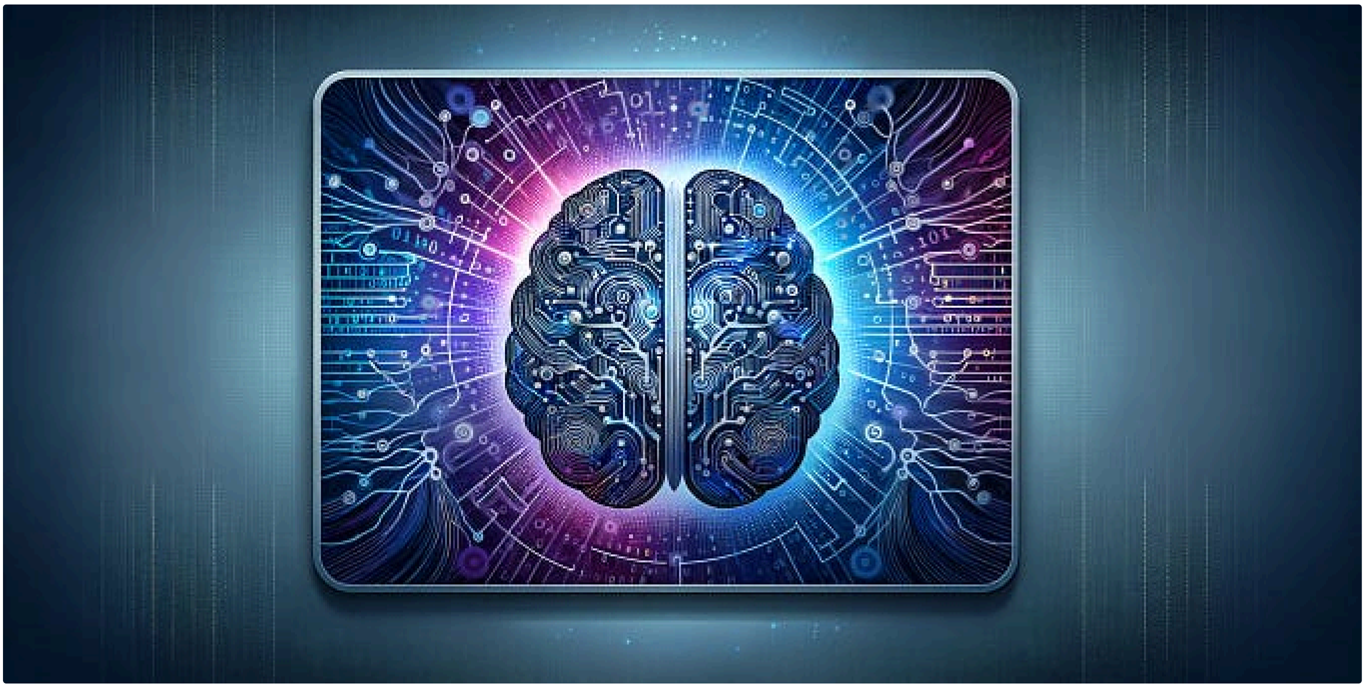
Alex Honchar in Towards Data Science

Intro to LLM Agents with Langchain: When RAG is Not Enough

First-order principles of brain structure for AI assistants

7 min read · Mar 15, 2024

1.6K 8



Cristian Leo in Towards Data Science

The Math Behind Neural Networks

Dive into Neural Networks, the backbone of modern AI, understand its mathematics, implement it from scratch, and explore its applications

28 min read · 5 days ago

João Pedro

Creating a Simple ETL Pipeline With Apache Spark

Transforming raw data into a star schema with simple (py)spark code

10 min read · Aug 7, 2022

See all from João Pedro

See all from Towards Data Science

Recommended from Medium

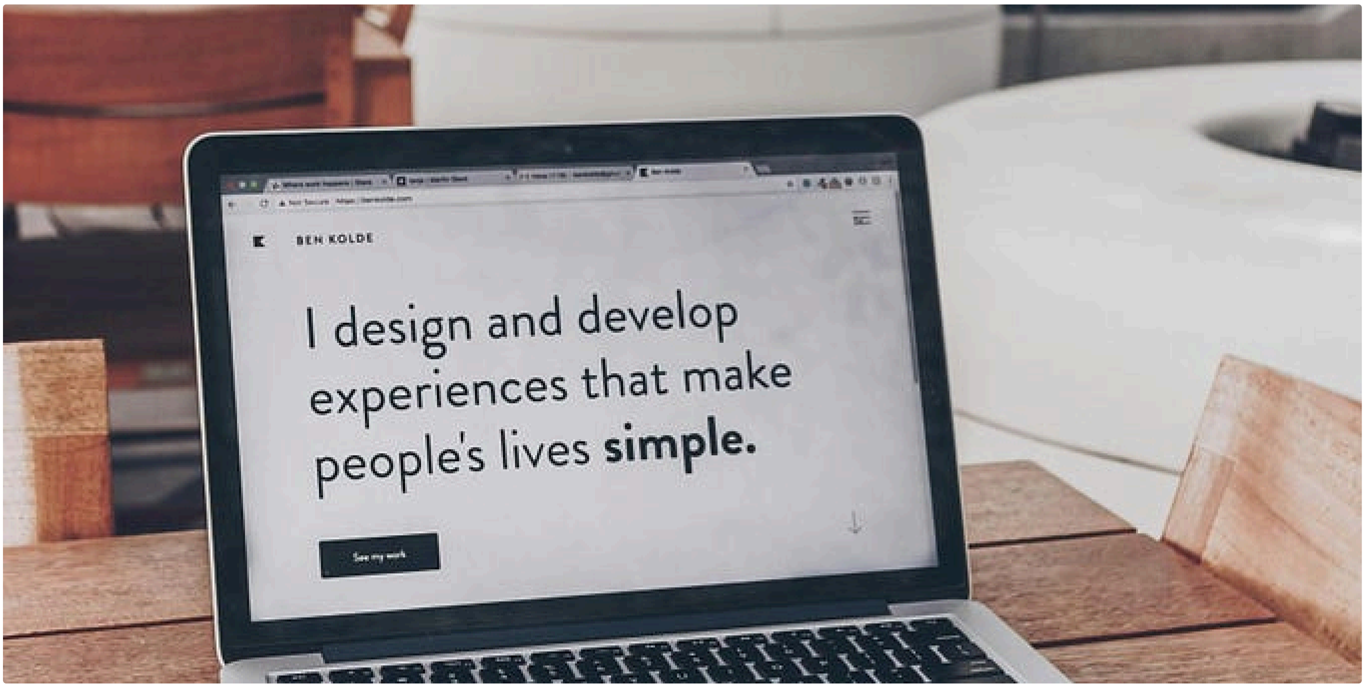
Nick Hass

How to Connect Local PySpark to AWS S3 and Read a Delta Table

While you could use AWS EMR and automatically have access to the S3 file system, you can also connect Spark to your S3 file system on your...

2 min read · Oct 4, 2023

5



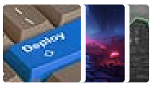
Rahul Madhani in Data Engineer Things

4 Proven Methods to Download Files from Databricks Locally

A Detailed Guide to Fast and Effective File Downloads from Databricks to Your Local Machine with examples and code

4 min read · Feb 20, 2024

Lists



Predictive Modeling w/ Python

20 stories · 1060 saves

Practical Guides to Machine Learning

10 stories · 1265 saves

Coding & Development

11 stories · 538 saves

ChatGPT prompts

47 stories · 1365 saves

ISHAN PRADHAN

How to read a .snappy.parquet file in databricks

In Databricks, learn how to read .snappy.parquet files of your delta tables.

5 min read · Oct 11, 2023

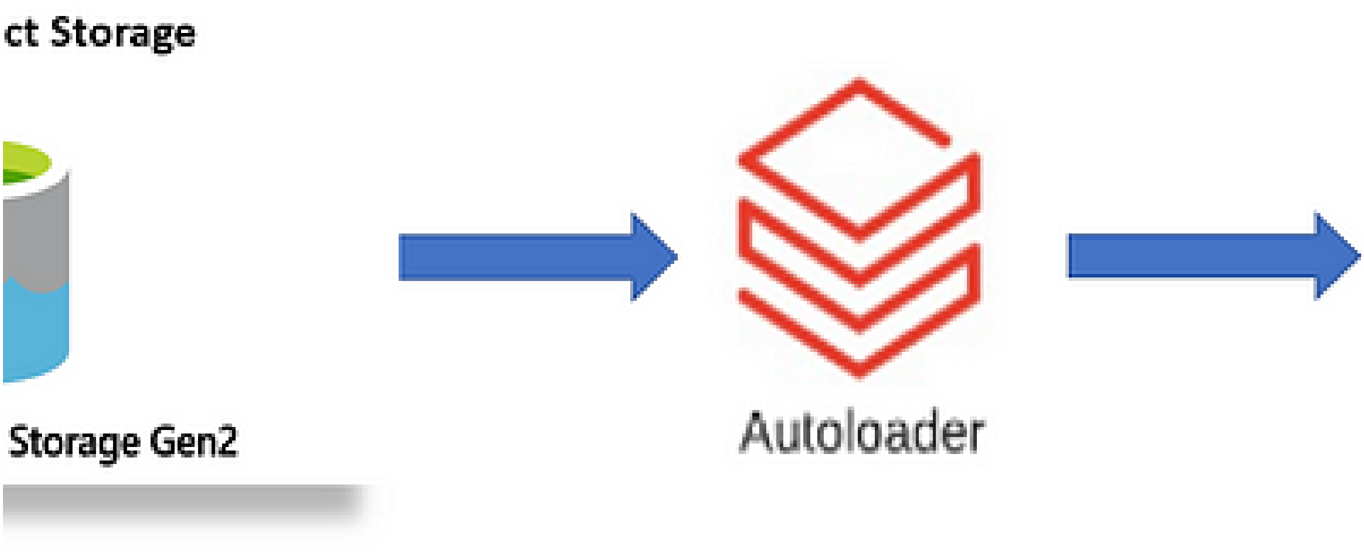
Muqtada Hussain Mohammed

Efficient Change Data Capture (CDC) on Databricks Delta Tables with Spark

In today's data-driven applications, organizations face a critical challenge: ensuring near-real-time data aggregation and accuracy for...

5 min read · Oct 20, 2023

72



Shubhodaya Hampiholi

Streaming Data Ingestion with Databricks Auto Loader

Use case: As part of out Data Ingestion framework we wanted to adapt to a robust, scalable and reusable ingestion mechanism which can cater...

5 min read · Nov 27, 2023

1

dezimaldata

Incremental Data Loading in Azure Databricks: A Comprehensive Guide

Data engineering processes often involve handling large datasets efficiently, and incremental loading is a key strategy to optimize these...

3 min read · Jan 18, 2024

2

See more recommendations