Using PySpark to Generate a Hash of a Column

Kyle Gibson · Follow
5 min read · Dec 1, 2022

Introduction

When working in data, there are times that we need to generate a hash of a column. Sometimes this is used to mask sensitive information, e.g. Date of Birth, Social Security Number, I.P. Address, etc. Other times it may be needed to derive a repeatable *ID/PrimaryKey* column. Regardless of what the need is, there will almost certainly be a time that we will need to incorporate this into our data transformations.

This article will show some examples of how to generate a hash in PySpark. I used Databricks in my examples, but this would almost certainly apply in other Spark environments as well.

Note 1: We are using the *sha2* PySpark function in these examples. Its documentation can be found here: pyspark.sql.functions.sha2 — PySpark 3.1.2 documentation (apache.org)

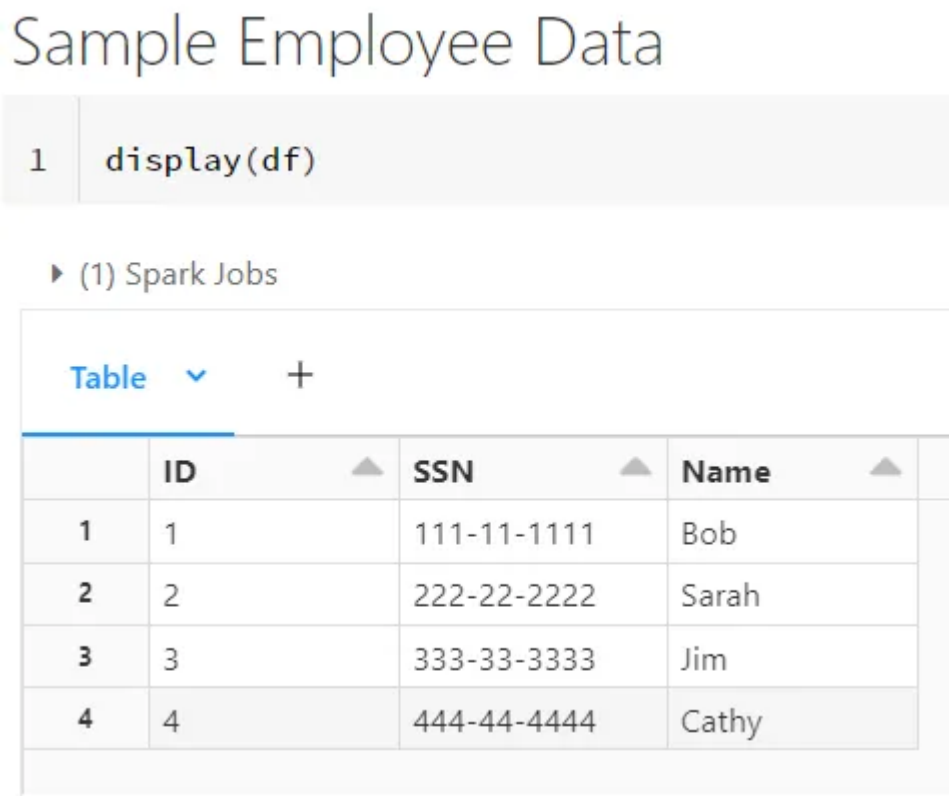
Note 2: For purposes of these examples, there are four PySpark functions that I imported to use in the explanation. Here is the import statement:

```
from pyspark.sql.functions import col, concat_ws, lit, sha2
```

Examples

Example 1: Hashing a Single Column

Let's start with a sample DataFrame of Employees, containing ID, SSN, and Name columns.



Suppose that we did not want end-users to see the actual *ID* values but just their hash value. Using *sha2* function to hash that column would look something like this:

```
df_hashed_1 = df\
    .withColumn('HashedID', sha2(col('ID'), 256))\
    .select('ID', 'HashedID', 'SSN', 'Name')
```

The results of this hash:

ID	HashedID	SSN	Name
1	6b86b273ff34fce19d6b804eff5a3f5747a	111-11-1111	Bob
2	d4735e3a265e16eee03f59718b9b5d030	222-22-2222	Sarah
3	4e07408562bedb8b60ce05c1decfe3ad1	333-33-3333	Jim
4	4b227777d4dd1fc61c6f884f48641d02b4	444-44-4444	Cathy

Now, if we wanted, we could use this *HashedID* column instead of the *ID* column.

Example 2: Hashing a Single Column (Again)

One of the most important things about hashing is that it will generate the same value every time for all the values that are hashed. Let's look at an example of that to confirm.

First, let's create a duplicate of the *ID* column from Example 1:

```
df_duplicate_id = df_hashed_1\
    .withColumn('DuplicateID', col('ID'))\
    .select('ID', 'HashedID', 'DuplicateID', 'SSN', 'Name')
```

The new results with the duplicated ID column:

ID	HashedID	DuplicateID	SSN	Name
1	6b86b273ff34fce19d6b804eff5a3f5747a	1	111-11-1111	Bob
2	d4735e3a265e16eee03f59718b9b5d030	2	222-22-2222	Sarah
3	4e07408562bedb8b60ce05c1decfe3ad16	3	333-33-3333	Jim
4	4b227777d4dd1fc61c6f884f48641d02b4	4	444-44-4444	Cathy

As you can see, each value is the same for *ID* and *DuplicateID*. Let's now confirm that hashing the *DuplicateID* column will give the same results as the *HashedID* column:

```
df_hashed_2 = df_duplicate_id\
    .withColumn('DuplicateHashedID', sha2(col('DuplicateID'), 256))\
    .select('ID', 'HashedID', 'DuplicateID', 'DuplicateHashedID', 'SSN', 'Name'
```

The results of hashing the *DuplicateID* column:

ID	HashedID	DuplicateID	DuplicateHashedID
1	6b86b273ff34fce19d6b804	1	6b86b273ff34fce19d6b804eff5a3f574
2	d4735e3a265e16eee03f59	2	d4735e3a265e16eee03f59718b9b5d03
3	4e07408562bedb8b60ce05	3	4e07408562bedb8b60ce05c1decfe3ad
4	4b227777d4dd1fc61c6f884	4	4b227777d4dd1fc61c6f884f48641d02

I cutoff some of the hashed columns for better visibility, but as you can see, we got the same values for hashing the *1*, *2*, *3*, and *4* values in both columns. This behavior shows that using this function on a value in any context will generate the same values every time.

Example 3: Hashing a Single Column with Salt Value

In our previous examples, we hashed the columns without any manipulation. However, some of our end-users may apply the *sha2* function to some data, and try to back into the underlying values of our hashed column. To prevent this, we could use a "salt" value in our hashing. This example will show a very basic example of this.

Let's say we want to hash the *ID* column, same as Example 1, but we want to add a salt value to it. We will basically just concatenate a random value to the ID column, and then hash it. In our example, we are using *datamadness* as our salt value:

```
salt_value = 'datamadness'

df_salted_1 = df\
    .withColumn('SaltedID', concat_ws('_', lit(salt_value), col('ID')))\
    .select('ID', 'SaltedID', 'SSN', 'Name')
```

The results of this salting:

ID	SaltedID	SSN	Name
1	datamadness_1	111-11-1111	Bob
2	datamadness_2	222-22-2222	Sarah
3	datamadness_3	333-33-3333	Jim
4	datamadness_4	444-44-4444	Cathy

As you can see, we are concatenating (with an underscore separator) the string value *datamadness* to every value in the *ID* column.

Now, let's hash the *SaltedID* column, instead of the *ID* column:

```
df_hashed_3 = df_salted_1\
    .withColumn('HashedSaltedID', sha2(col('SaltedID'), 256))\
    .select('ID', 'SaltedID', 'HashedSaltedID', 'SSN', 'Name')
```

Here are the results:

ID	SaltedID	HashedSaltedID	SSN	Name
1	datamadness_1	549236bfbb64d23cedaac38def	111-11-1111	Bob
2	datamadness_2	bf43cb473dd0b9ba531f8f0ec1	222-22-2222	Sarah
3	datamadness_3	7c0f2dde4d746cdfabd400d3ee	333-33-3333	Jim
4	datamadness_4	7dcf35d6a99706c6f13905b41a	444-44-4444	Cathy

So what have we accomplished? Well let's look at the results of hashing *ID* vs *SaltedID:*

3	4e07408562bedb8b60ce05c1	datamadness_3	7c0f2dde4d746cdfabd400d3ee	333-33-3333	Jim
4	4b227777d4dd1fc61c6f884f4	datamadness_4	7dcf35d6a99706c6f13905b41a	444-44-4444	Cathy

As you can see, we get different values hashing *SaltedID* vs *ID*. So if we have an end-user trying to back into the underlying values for a hashed column, they would also have to know the salt value.

Example 4: Hashing Multiple Columns with Salt Value

This example is probably the one I've used the most in production. Suppose you have a Slowly Changing Dimension table of SCD Type 2 that contains *ID*, *DateEffectiveFrom*, and *DateEffectiveThru* columns, along with any other attributes needed. In SCD Type 2, the *ID* column is not a Primary Key column, as it can appear multiple times in the table with different effective dates. To fix that, you could derive an *EffectiveID* column that is a hash of both the *ID* and *DateEffectiveFrom* columns.

Obviously there are other use cases for hashing multiple columns, but that is one you could potentially use.

In our example, we will assume that we want to hash both the *ID* and *SSN* columns, so that our end-users don't see either value.

Here is the logic to first concatenate (with an underscore separator) the salt value, *ID* column, and *SSN* column:

```
salt_value = 'datamadness'

df_salted_2 = df\
    .withColumn('SaltedCombinationID', concat_ws('_', lit(salt_value), col('ID'
    .select('ID', 'SSN', 'SaltedCombinationID', 'Name')
```

Here are the results of that concatenation:

ID	SSN	SaltedCombinationID	Name
1	111-11-1111	datamadness_1_111-11-1111	Bob
2	222-22-2222	datamadness_2_222-22-2222	Sarah
3	333-33-3333	datamadness_3_333-33-3333	Jim
4	444-44-4444	datamadness_4_444-44-4444	Cathy

The *SaltedCombinationID* column now contains the concatenation of the salt value, *ID* column, and *SSN* column.

The last step now is to hash the *SaltedCombinationID* column:

```
df_hashed_5 = df_salted_2\
    .withColumn('HashedSaltedCombinationID', sha2(col('SaltedCombinationID'), 2
    .select('ID', 'SSN', 'SaltedCombinationID', 'HashedSaltedCombinationID', 'N
```

The final results:

ID	SSN	SaltedCombinationID	HashedSaltedCombinationID	Name
1	111-11-1111	datamadness_1_111-11-1111	5db2fa4eff1d30404fa4c0d1161e5021cd1a9af	Bob
2	222-22-2222	datamadness_2_222-22-2222	e142174d1b342c8cfccdd7acb8cd4c35fdab71	Sarah
3	333-33-3333	datamadness_3_333-33-3333	cd8cb533f92c0d17c07d6bcc4dcf1f610743fb4	Jim
4	444-44-4444	datamadness_4_444-44-4444	60b9d5b325397bf81dfc963fbf1b0e84f047cf3	Cathy

Now, if needed, we could drop the *ID* and *SSN* columns and just use the *HashedSaltedCombinationID* column.

Conclusion

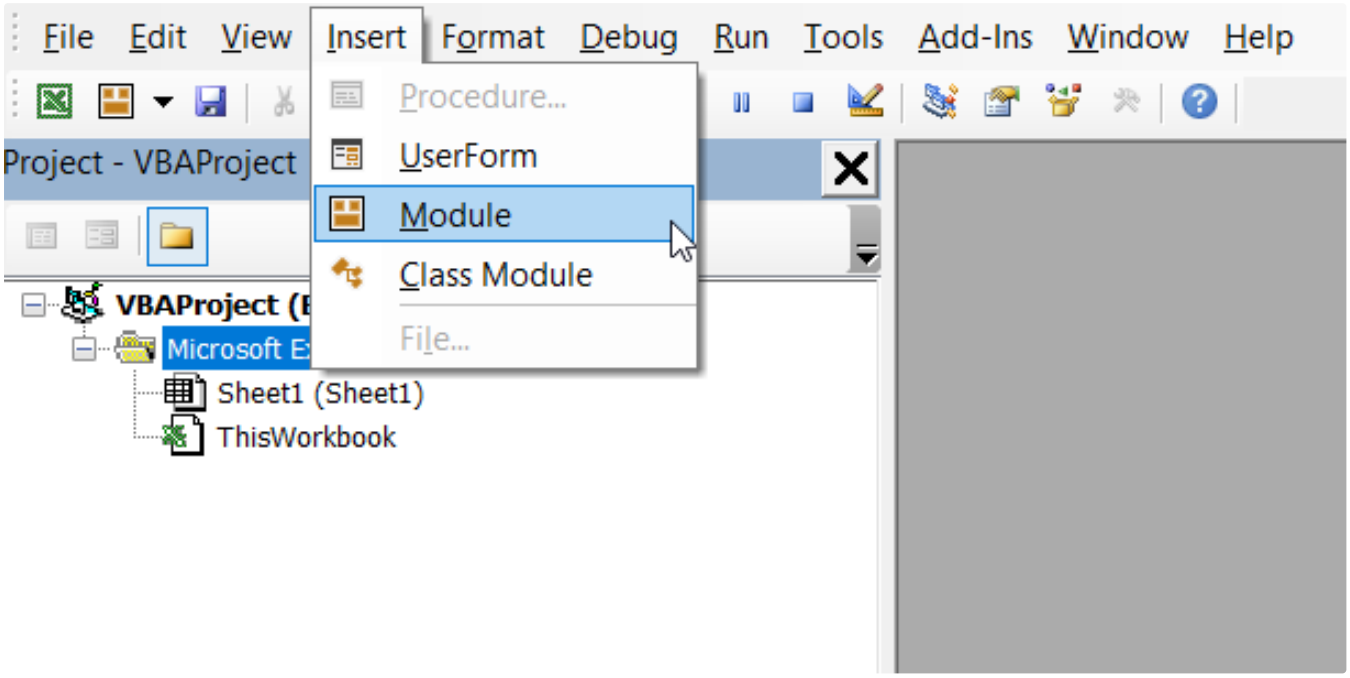
These were pretty basic examples of how to hash a column in PySpark, but hopefully this helps generate some ideas for how you could use it in your job.

Thanks for reading!

Pyspark Hashing Databricks Sha2 Data

Christian, husband, father. Data Engineer at Chick-fil-A.

More from Kyle Gibson

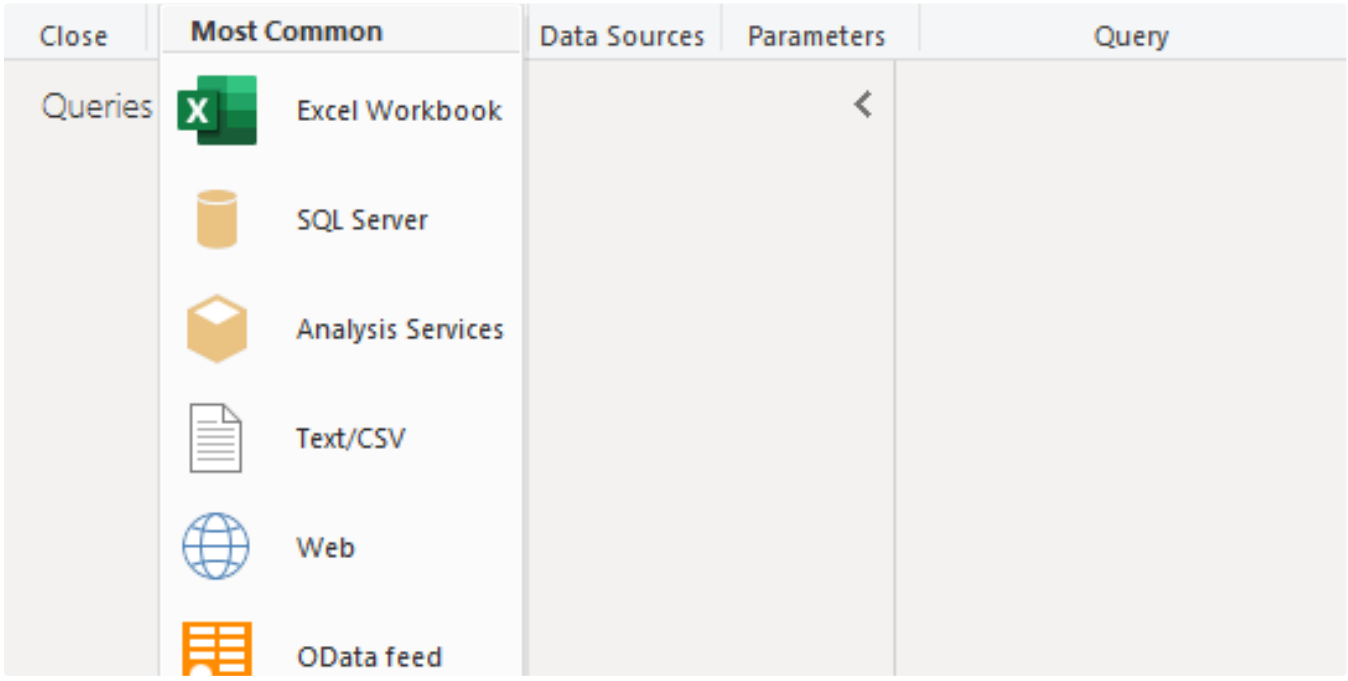


Kyle Gibson

Write Your First VBA Macro in Excel

Impress all your co-workers with your new Excel skills

Jan 18, 2023 22 1



Kyle Gibson

How to Write functions in Power Query

I struggled to learn M Language

Nov 1, 2022 73

```
Date: string
Item: string
Amount: string
IsDiscounted: string
```

Table +

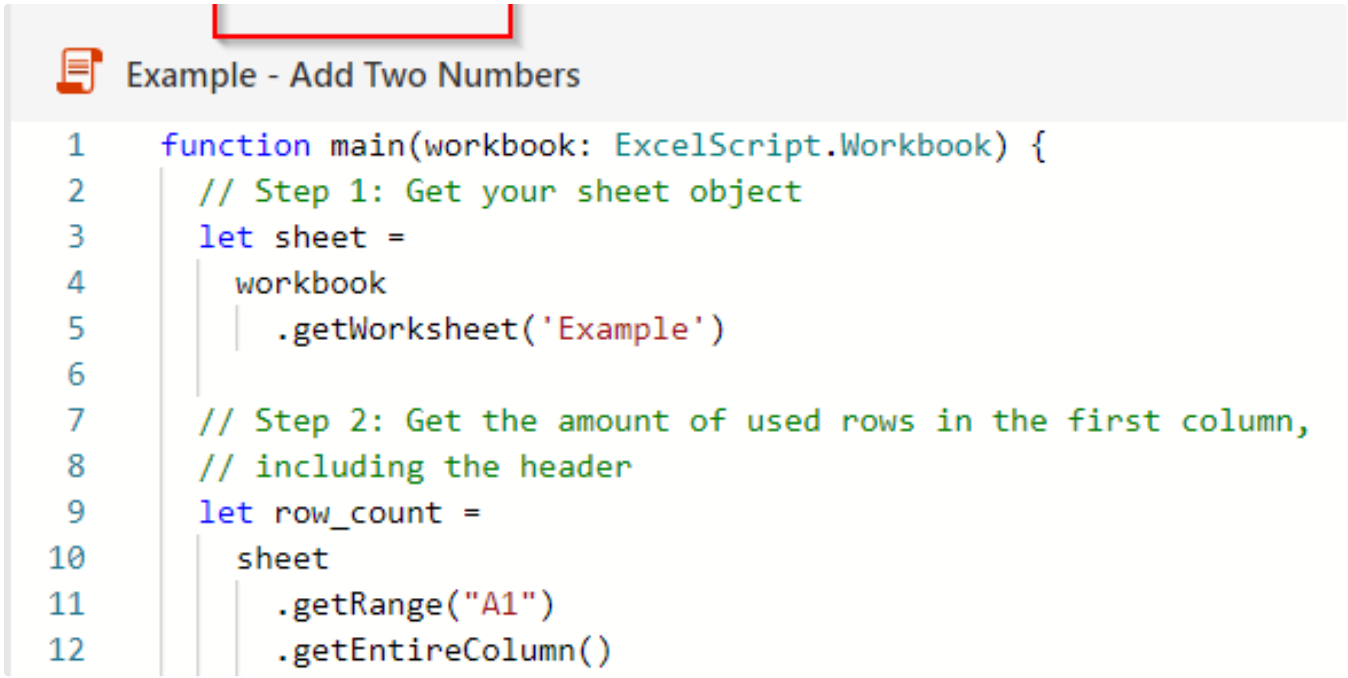
	Date	Item	Amount	IsDiscounted
1	1/1/2023	Burger	5	FALSE
2	1/2/2023	Hot Dog	3	TRUE
3	1/3/2023	Chicken Wings	10	FALSE

Kyle Gibson

Casting Data Types in PySpark

How often have you read data into your Spark DataFrame and gotten schema like this?

Mar 14, 2023 51 1



Kyle Gibson

Write Your First Office Script in Excel

Bye, bye VBA—hello Office Scripts?

Jan 26, 2023 51

See all from Kyle Gibson

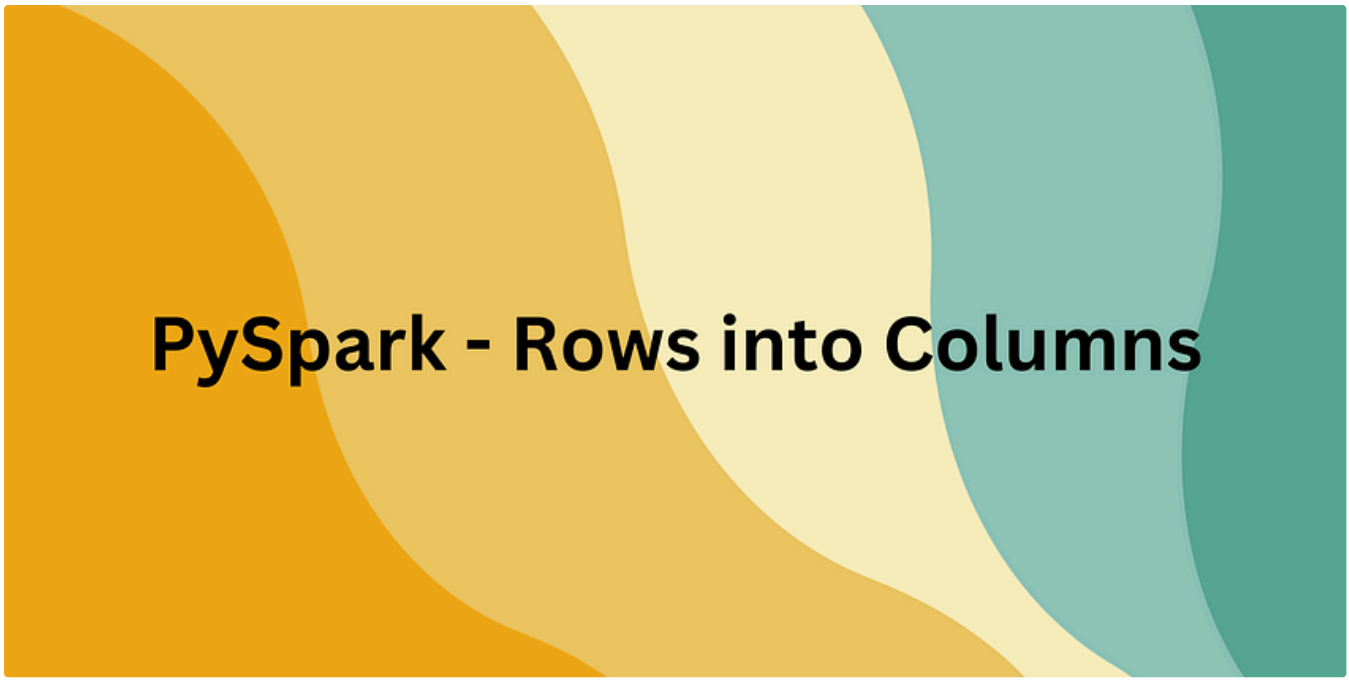
Recommended from Medium

Subham Khandelwal in Dev Genius

PySpark—Run Multiple Jobs in Parallel

Understand How to Execute multiple Jobs in Parallel or Concurrently in PySpark

Sep 2 54 2



Vengateswaran Arunachalam

Pyspark—Transpose Rows into Columns

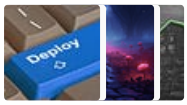
Introduction:

Apr 11 12

Lists

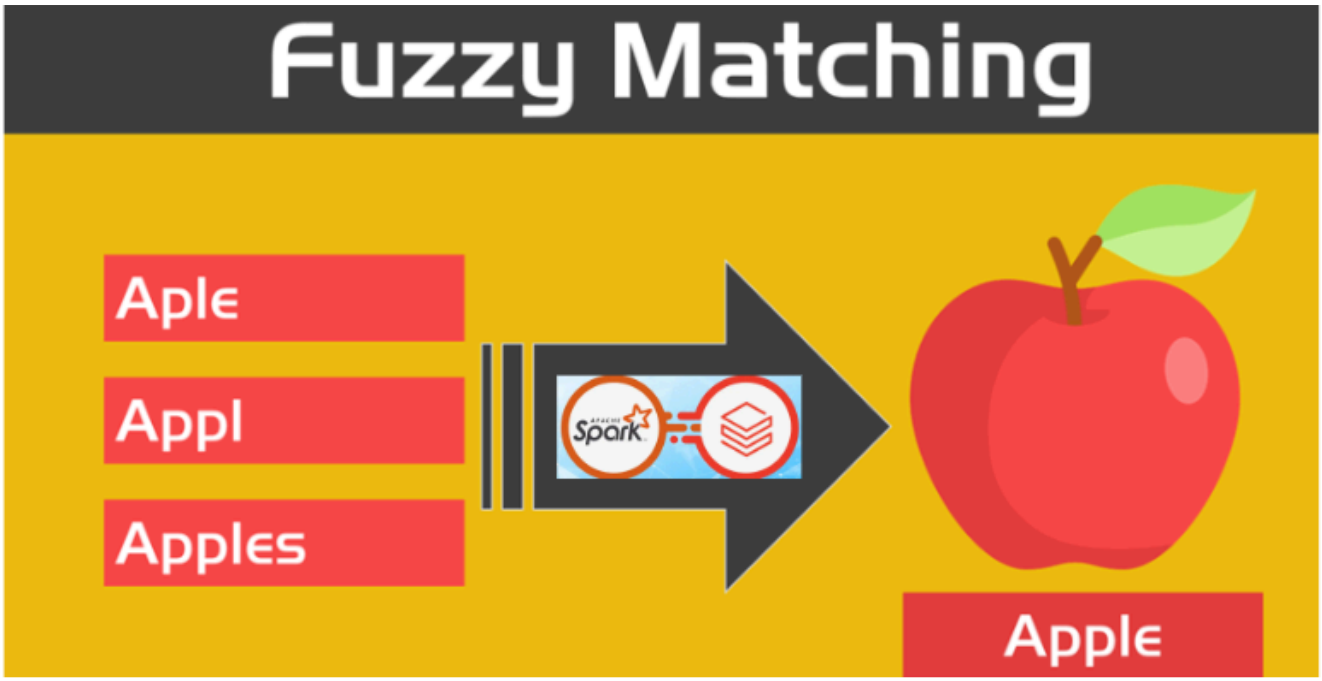
General Coding Knowledge
20 stories · 1626 saves

data science and AI
40 stories · 259 saves



Predictive Modeling w/ Python
20 stories · 1581 saves

ChatGPT
21 stories · 825 saves



Gavaragirijarani

Optimizing Large-Scale Fuzzy Matching with Apache Spark and Databricks

Recently, I got the chance to implement my learnings on an interesting data processing challenge involving large-scale fuzzy matching using...

Jun 8 49

Samhitha Poreddy

Optimizing API Data Ingestion with PySpark

Leveraging Pagination for Performance

Sep 9 13

Meher Ben Ahmed

A Comprehensive Guide to Filtering Array Columns in Apache Spark

Abstract:

Apr 22 5

Deepa Vasanthkumar

Spark Logical and Physical Plan Generation

In Spark, when you submit a SQL query or DataFrame transformation, it goes through several stages of processing before execution. Let us...

Apr 9 124