

Databricks / Photon

ROBIN SCHLAURI – NINO ZANITTI – LINUS SZOKODY

Inhalt

- Übersicht Databrick
- Databrick optimizations
- Photon
- Photon Tests



Databricks



2013 von Apache Spark Entwicklern gegründete amerikanische Firma

“Eine einheitliche Plattform mit welcher Data Scientists, Data Engineers und Analysten zusammenarbeiten können, um Wertschöpfung aus Daten zu generieren”

Basierend auf Apache Spark mit vielen Open Source Projekten

- Apache Spark
- Delta Lake
- Mlflow



Databricks Features

Hauptpunkte:

- Erstellung und Konfiguration von Server-Clustern
- Verbindung zu verschiedenen Dateisystemen
- Programmierschnittstellen für Python, Scala und SQL
- Interaktive und kollaborative Workspaces
- Sehr schneller start möglich durch managed umgebungen von Cloud providern (lokale installation nicht möglich)

User Interface

Workspace

- Notebooks und andere Files für die Zusammenarbeit

Catalog

- Delta Lake

Workflows

- Automations -> z.B. Scheduled Jobs

Compute

- Cluster und Jobs Konfigurieren

The screenshot displays the Databricks user interface, which is divided into two main sections: Workspace and Compute.

Workspace Section:

- Left Sidebar:** Contains navigation links for New, Workspace, Recents, Catalog, Workflows, Compute, Data Engineering, Job Runs, Machine Learning, Playground, Experiments, Features, Models, and Serving.
- Workspace View:** Shows a file explorer with a tree structure: Home > Workspace > Shared > Users > linus.szokody@godiz.com. Below this, there are links for Repos, Favorites, and Trash.
- Right Panel:** Displays the user profile for linus.szokody@godiz.com. Below the profile, there is a table with columns Name and Type. The table contains one entry: Benchmark (Notebook).

Compute Section:

- Left Sidebar:** Similar to the Workspace section, but with the Compute link highlighted.
- Compute View:** Shows a table of compute clusters. The table has columns: State, Name, Runtime, Active m..., Active co..., Active DB..., Source, Creator, Notebooks, and a settings icon. The table contains one entry: Linus Szokody's Cluster (9.1, 28 GB, 8 cores, 3 UI, linus.szokody@...). A 'Create compute' button is visible in the top right corner.

Delta Lake

Open-source storage layer welcher auf ein existierenden Data Lake angewant werden kann

Vorteile:

- ACID-Transactions
 - Datenintegrität bleibt auch bei gleichzeitigen Transaktionen erhalten
- Schema Enforcement and Evolution
 - Erzwingt ein Datenbankschema beim schreiben von Daten
- Scalable Metadata Handling
 - Behebt übliche Probleme beim Skalieren von Metadaten
- Unified Batch and Streaming Source and Sink
 - Delta Lake Tabellen sind Streaming und Batch Quellen und Ziele
- Time Travel
 - Behält eine Historie der Daten

Idee: Vorteile von Data Lake und Data Warehouse in einem übernehmen

MLflow

Open-source platform für Machine Learning

Managing

- Machine learning lifecycle
- Experimentation
- Reproducibility
- Deployment

Databricks

Ein einzelnes Tool für optimierte, effiziente und kollaborative Big-Data-Verarbeitungsumgebung

Vorteile:

- Einfache Einrichtung / weniger installationsaufwand durch Cloud basiert
- Kosteneffizient (je nach Anwendungsfall)
- Features für die Zusammenarbeit
- Integriert mit Cloud Provider (einfacheres Zusammenspiel von Cloud Ressourcen)
- Skalierbar (einfach in Cloud mehr Ressourcen zur Verfügung zustellen)
- Zuverlässig (Backups, Ausfallsicherheit etc. (Cloud))

Nachteile:

- Kosten (je nach Anwendungsfall)
- Anpassungen (Version up- und downgrades können je nachdem schwieriger werden)
- Sicherheit (Daten sind in der Cloud)
- Abhängigkeit (Abhängig vom Cloud Provider)
- 3rd Party Plugins für z.B. Spark können theoretisch nicht unterstützt werden

Databricks optimizations

Starke integration in Cloud (Microsoft Azure, Amazon AWS, Google Cloud)



Ermöglicht Features und Verbesserungen die sonst nicht möglich wären:

- I/O von anderen Services können perfekt abgestimmt werden
 - z.B. Azure Databricks + Azure Data Lake
- optimiertes Caching
- Sowas wie “**Photon**” ist möglich

Photon

“Photon is a high-performance Databricks-native vectorized query engine that runs your SQL workloads and DataFrame API calls faster to reduce your total cost per workload.” (Databricks Dokumentation)

Funktionsweise:

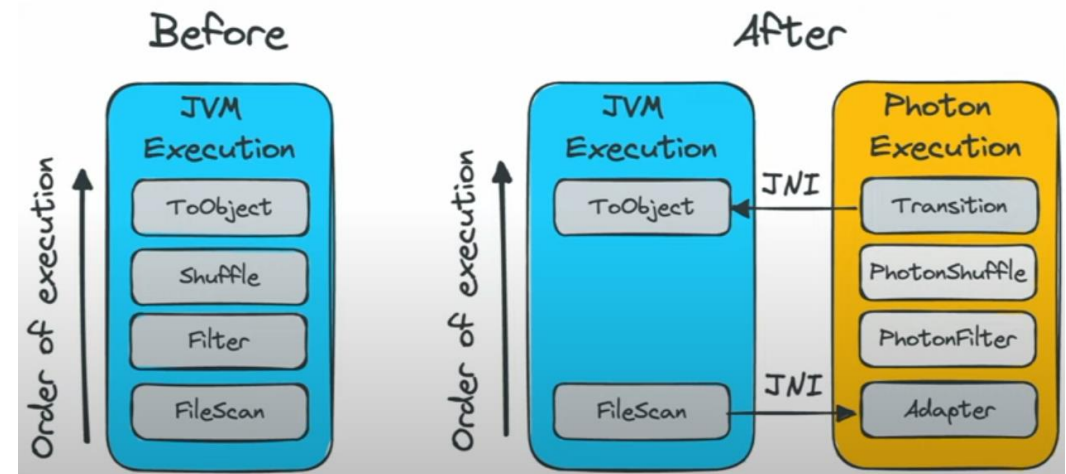
- Photon aktiviert -> Photon bearbeitet die Abfrage (wenn schneller)
- Von Photon nicht unterstützt/langsamer mit Photon -> Spark übernimmt

Grund:

- Spark engine läuft auf einer JVM (Java Virtual Machine) (Scala)
- Durch C++ soll JVM Overhead verringert werden
- Photon arbeitet immer Vektorisiert

Integration:

- Standardmässig in Databricks aktiviert
- Kompatibel mit Apache Spark-APIs (**funktioniert mit bestehendem Code**)



<https://www.youtube.com/watch?v=kL2XWgLeXHE>

Photon – Key Features

- Unterstützung für SQL und entsprechende DataFrame-Operationen mit Delta- und Parquet-Tabellen
- Beschleunigte Abfragen die Daten schneller verarbeiten und Aggregationen sowie Joins beinhalten
- Schnellere Leistung, wenn Daten wiederholt aus dem Disk-Cache abgerufen werden.
- Robuste Scan-Leistung bei Tabellen mit vielen Spalten und vielen kleinen Dateien.
- Schnelleres Schreiben von Delta- und Parquet-Daten mithilfe von UPDATE, DELETE, MERGE INTO, INSERT und CREATE TABLE AS SELECT, einschließlich grosser Tabellen mit Tausenden von Spalten.
- Ersetzt sort-merge Joins durch hash-joins.
 - (Auch in Spark verfügbar aber hashes werden schneller erstellt und können in batches abgefragt werden)
- Es gibt auch Features die Photon benötigen um überhaupt verwendet werden zu können:
 - Predictive I/O for read and write
 - H3 geospatial expressions
 - Dynamic file pruning (Dynamische Dateibereinigung)

Photon – Operatoren, Ausdrücke und Datentypen

Operators

- Scan, Filter, Project
- Hash Aggregate/Join/Shuffle
- Nested-Loop Join
- Null-Aware Anti Join
- Union, Expand, ScalarSubquery
- Delta/Parquet Write Sink
- Sort
- Window Function

Expressions

- Comparison / Logic
- Arithmetic / Math (most)
- Conditional (IF, CASE, etc.)
- String (common ones)
- Casts
- Aggregates (most common ones)
- Date/Timestamp

Data types

- Byte/Short/Int/Long
- Boolean
- String/Binary
- Decimal
- Float/Double
- Date/Timestamp
- Struct
- Array
- Map

Limitationen

- Queries die unter zwei Sekunden dauern werden nicht beeinflusst
- User defined functions und RDD APIs werden nicht unterstützt
- Structured Streaming:
 - Stateless streaming mit Delta, Parquet, CSV und JSON werden unterstützt
 - Stateless Kafka und Kinesis streaming ist supported, wenn nach Delta oder Parquet geschrieben wird

Features die nicht von Photon supported sind, funktionieren normal mit Sparks weiter

Photon - Testen

Setup:

- 1 Databrick installation -> 2 gleiche Cluster
 - 1x mit Photon enabled
 - 1x mit Photon disabled
- 1x Runtime 14.2.x.scala2.12
Spark Version: 3.5.0
- 1x Runtime 9.1.x.scala2.12
Spark Version: 3.1.2
- 1 Driver + 2-8 Worker (gleicher Typ)
- 14 GB Memory
- 4 Cores

Summary

2-8 Workers	28-112 GB Memory 8-32 Cores
1 Driver	14 GB Memory, 4 Cores
Runtime	14.2.x-scala2.12

Standard_DS3_v2 **2-7 DBU/h**

Summary

2-8 Workers	28-112 GB Memory 8-32 Cores
1 Driver	14 GB Memory, 4 Cores
Runtime	14.2.x-scala2.12

Photon **Standard_DS3_v2** **4-14 DBU/h**

☒ Multi node ☐ Single node

Access mode [ⓘ] **Single user access** [ⓘ]

Single user Linus Szokody

Performance

Databricks Runtime Version

9.1 LTS (includes Apache Spark 3.1.2, Scala 2.12)

☒ Use Photon Acceleration [ⓘ]

Worker type [ⓘ] **Min workers** **Max workers**

Standard_DS3_v2 14 GB Memory, 4 Cores 1 1 ☐ Spot instances [ⓘ]

Driver type

Standard_DS3_v2 14 GB Memory, 4 Cores

☒ Enable autoscaling [ⓘ]

☒ Terminate after 30 minutes of inactivity [ⓘ]

Photon – Testen – Setup 1

Setup:

- 1 Databrick installation -> 2 gleiche Cluster
 - 1x mit Photon enabled
 - 1x mit Photon disabled
- Spark Version: 3.1.2
- Runtime 9.1.x-scala2.12
- 1 Driver + 1 Worker (gleicher Typ)
- 14 GB Memory
- 4 Cores

Summary

1-1 Worker 14-14 GB Memory
4-4 Cores

1 Driver 14 GB Memory, 4 Cores

Runtime 9.1.x-scala2.12

Photon **Standard_DS3_v2** **3 DBU/h**

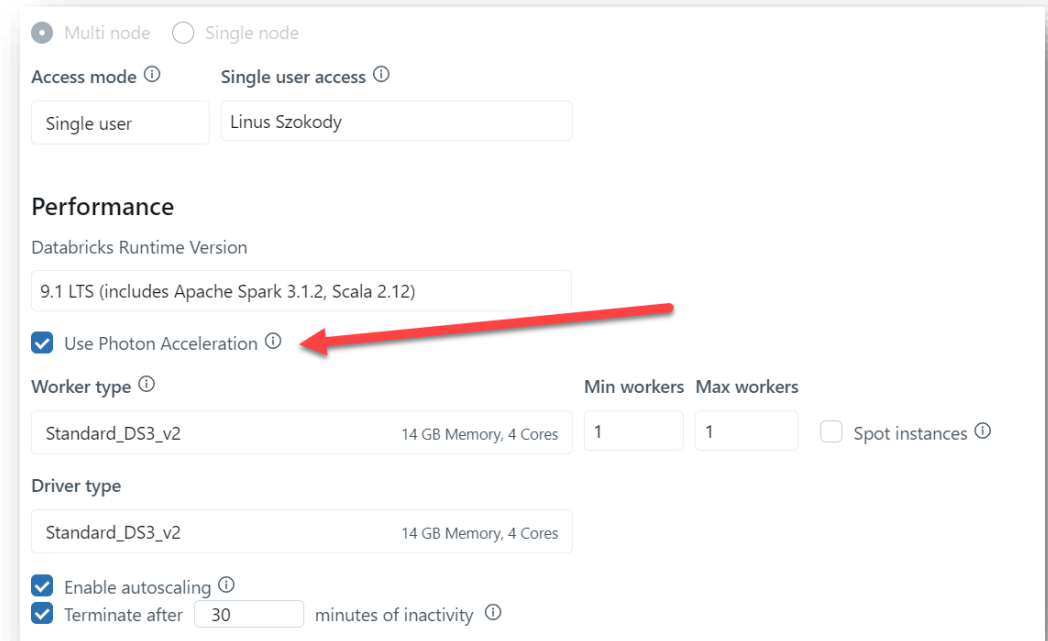
Summary

1-1 Worker 14-14 GB Memory
4-4 Cores

1 Driver 14 GB Memory, 4 Cores

Runtime 9.1.x-scala2.12

Standard_DS3_v2 **1.5 DBU/h**



☒ Multi node ☐ Single node

Access mode ⓘ Single user access ⓘ

Single user Linus Szokody

Performance

Databricks Runtime Version

9.1 LTS (includes Apache Spark 3.1.2, Scala 2.12)

☒ Use Photon Acceleration ⓘ

Worker type ⓘ Min workers Max workers

Standard_DS3_v2 14 GB Memory, 4 Cores 1 1 ☐ Spot instances ⓘ

Driver type

Standard_DS3_v2 14 GB Memory, 4 Cores

☒ Enable autoscaling ⓘ

☒ Terminate after 30 minutes of inactivity ⓘ

Photon – Testen – Setup 2

Setup:

- 1 Databrick installation -> 2 gleiche Cluster
 - 1x mit Photon enabled
 - 1x mit Photon disabled
- Spark Version: 3.5.0
- Runtime 14.2.x.scala2.12
- 1 Driver + 2-8 Worker (gleicher Typ)
- 14 GB Memory
- 4 Cores

Summary

2-8 Workers	28-112 GB Memory 8-32 Cores
1 Driver	14 GB Memory, 4 Cores
Runtime	14.2.x-scala2.12

Standard_DS3_v2

2-7 DBU/h

Summary

2-8 Workers	28-112 GB Memory 8-32 Cores
1 Driver	14 GB Memory, 4 Cores
Runtime	14.2.x-scala2.12

Photon

Standard_DS3_v2

4-14 DBU/h

Do not present

effectiveSparkVersion

Jobs	Stages	Storage	Environment	Executors	SQL / DataFrame
spark.databricks.clusterUsageTags.clusterPinned	false				
spark.databricks.clusterUsageTags.clusterPythonVersion	3				
spark.databricks.clusterUsageTags.clusterResourceClass	default				
spark.databricks.clusterUsageTags.clusterScalingType	autoscaling				
spark.databricks.clusterUsageTags.clusterSizeType	VM_CONTAINER				
spark.databricks.clusterUsageTags.clusterSku	STANDARD_SKU				
spark.databricks.clusterUsageTags.clusterSpotBidMaxPrice	-1.0				
spark.databricks.clusterUsageTags.clusterState	Pending				
spark.databricks.clusterUsageTags.clusterStateMessage	Starting Spark				
spark.databricks.clusterUsageTags.clusterTargetWorkers	1				
spark.databricks.clusterUsageTags.clusterUnityCatalogMode	***** (redacted)				
spark.databricks.clusterUsageTags.clusterWorkers	1				
spark.databricks.clusterUsageTags.containerType	LXC				
spark.databricks.clusterUsageTags.dataPlaneRegion	eastus				
spark.databricks.clusterUsageTags.driverContainerId	5546d048f38341bda7812a9c636e25ed				
spark.databricks.clusterUsageTags.driverContainerPrivately	10.139.64.4				
spark.databricks.clusterUsageTags.driverInstanceId	b30b97319d3845c6b47c63f987c818f0				
spark.databricks.clusterUsageTags.driverInstancePrivately	10.139.0.4				
spark.databricks.clusterUsageTags.driverNodeType	Standard_DS3_v2				
spark.databricks.clusterUsageTags.driverPublicDns	172.191.189.246				
spark.databricks.clusterUsageTags.effectiveSparkVersion	9.1.x-scala2.12				
spark.databricks.clusterUsageTags.enableCredentialPassthrough	***** (redacted)				
spark.databricks.clusterUsageTags.enableDfAcls	false				
spark.databricks.clusterUsageTags.enableElasticDisk	true				
spark.databricks.clusterUsageTags.enableGlueCatalogCredentialPassthrough	***** (redacted)				

Jobs	Stages	Storage	Environment	Executors	SQL	JDBC/ODBC Server	Structured Streaming
spark.databricks.clusterUsageTags.clusterPinned	false						
spark.databricks.clusterUsageTags.clusterPythonVersion	3						
spark.databricks.clusterUsageTags.clusterResourceClass	default						
spark.databricks.clusterUsageTags.clusterScalingType	autoscaling						
spark.databricks.clusterUsageTags.clusterSizeType	VM_CONTAINER						
spark.databricks.clusterUsageTags.clusterSku	STANDARD_SKU						
spark.databricks.clusterUsageTags.clusterSpotBidMaxPrice	-1.0						
spark.databricks.clusterUsageTags.clusterState	Pending						
spark.databricks.clusterUsageTags.clusterStateMessage	Starting Spark						
spark.databricks.clusterUsageTags.clusterTargetWorkers	1						
spark.databricks.clusterUsageTags.clusterUnityCatalogMode	***** (redacted)						
spark.databricks.clusterUsageTags.clusterWorkers	1						
spark.databricks.clusterUsageTags.containerType	LXC						
spark.databricks.clusterUsageTags.dataPlaneRegion	eastus						
spark.databricks.clusterUsageTags.driverContainerId	777ee3cf48bc45f9800119e0ced85b17						
spark.databricks.clusterUsageTags.driverContainerPrivately	10.139.64.5						
spark.databricks.clusterUsageTags.driverInstanceId	3224d213c8e74cacb8cec8a1ceb43d1f						
spark.databricks.clusterUsageTags.driverInstancePrivately	10.139.0.5						
spark.databricks.clusterUsageTags.driverNodeType	Standard_DS3_v2						
spark.databricks.clusterUsageTags.driverPublicDns	40.88.33.107						
spark.databricks.clusterUsageTags.effectiveSparkVersion	9.1.x-photon-scala2.12						
spark.databricks.clusterUsageTags.enableCredentialPassthrough	***** (redacted)						
spark.databricks.clusterUsageTags.enableDfAcls	false						
spark.databricks.clusterUsageTags.enableElasticDisk	true						
spark.databricks.clusterUsageTags.enableGlueCatalogCredentialPassthrough	***** (redacted)						

Test 1

1GB Datei mit Random Integern wurde erstellt und soll sortiert werden

Der Blob Storage wurde gemounted um direkt daraus zu lesen

Die Sortierung wurde 10 mal ausgeführt und der Durchschnitt der Laufzeit als Resultat genommen

Read from mounted storage and Sort X-Times

Cmd 4

```
1 from pyspark.sql import SparkSession
2 from pyspark.sql.functions import rand
3 import time
4
5 file_path = "/mnt/sample/sort_benchmark_data_1GB.txt"
6 df = spark.read.format("csv").load(file_path)
7
8 num_iterations = 10
9 durations = []
10
11 for i in range(num_iterations):
12     start_time = time.time()
13     sorted_df = df.sort("_c0")
14     sorted_df.show()
15     end_time = time.time()
16
17     duration = end_time - start_time
18     durations.append(duration)
19     print(f"Sorting Run {i+1} took {duration} seconds.")
20
21 average_duration = sum(durations) / num_iterations
22 print(f"Average sorting time over {num_iterations} runs: {average_duration} seconds.")
```

► (11) Spark Jobs

► df: pyspark.sql.dataframe.DataFrame = [_c0: string]

► sorted_df: pyspark.sql.dataframe.DataFrame = [_c0: string]

```
10000028|
10000028|
10000037|
10000042|
10000008|
10000081|
10000090|
10000010|
100000103|
100000113|
100000128|
100000138|
100000139|
10000014|
100000144|
100000193|
```

+-----+

only showing top 20 rows

Sorting Run 10 took 54.98161792755127 seconds.

Average sorting time over 10 runs: 58.48239943981171 seconds.

Command took 10.26 minutes -- by linus.szokody@godiz.com at 10.1.2024, 20:39:06 on 14.2 Photon enabled

Do not present

Test 2

Datenset wurde in Memory erstellt mit Zufallszahlen und einer Kategorie (A, B oder C)

Aggregation -> Durchschnitt der Kategorien berechnen

Sortieren -> Sortieren der Daten nach "Value"

Die Aggregationen und Sortierungen wurden 50x durchgeführt und der Durchschnitt berechnet

Aggregate and Sort

Create Data

```
1 from pyspark.sql import SparkSession
2 from pyspark.sql.functions import col, expr
3 import random
4
5 # Number of rows for the DataFrame
6 num_rows = 25000000 # 25 Mio
7
8 # Generate random data
9 data = [(random.randint(1, 1000), random.random(), random.choice(['A', 'B', 'C'])) for _ in range(num_rows)]
10
11 # Create DataFrame
12 df = spark.createDataFrame(data, ["id", "value", "category"])
13
14 df.cache()
15 df.count() # Trigger the caching
16
```

▶ (3) Spark Jobs

▶ df: pyspark.sql.dataframe.DataFrame = [id: long, value: double ... 1 more field]

25000000

Command took 9.12 minutes -- by linus.szokody@godiz.com at 10.1.2024, 21:00:54 on 14.2 Photon enabled

Run Aggregation and Sorting

```
1 import time
2
3 num_iterations = 50
4
5 # Aggregation
6 durations = []
7 for i in range(num_iterations):
8     start_time = time.time()
9     aggregated_df = df.groupBy("category").avg("value")
10    aggregated_df.show()
11    end_time = time.time()
12    duration = end_time - start_time
13    durations.append(duration)
14    print(f"Aggregation Run {i+1} took {duration} seconds.")
15
16 average_duration_agg = sum(durations) / num_iterations
17 print(f"Average aggregation time over {num_iterations} runs: {average_duration_agg} seconds.")
18
19 # Sorting
20 durations = []
21 for i in range(num_iterations):
22     start_time = time.time()
23     sorted_df = df.orderBy(col("value").desc())
24     sorted_df.show()
25     end_time = time.time()
26     duration = end_time - start_time
27     durations.append(duration)
28     print(f"Sorting Run {i+1} took {duration} seconds.")
29
30 average_duration = sum(durations) / num_iterations
31 print(f"Average sorting time over {num_iterations} runs: {average_duration} seconds.")
32 print(f"Average aggregation time over {num_iterations} runs: {average_duration_agg} seconds.")
33
34
```

▶ (50) Spark Jobs

▶ aggregated_df: pyspark.sql.dataframe.DataFrame = [category: string, avg(value): double]

▶ sorted_df: pyspark.sql.dataframe.DataFrame = [id: long, value: double ... 1 more field]

```
[207|0.9999997470881982| C|
[658|0.9999995958075151| B|
[514|0.9999995926141899| B|
[972|0.9999995141424379| A|
[303|0.9999994830698105| B|
[335|0.9999994649795151| B|
[514|0.9999994100883675| C|
[599|0.9999993753872134| C|
[504|0.9999992030613987| B|
[158|0.9999991889710428| B|
[643|0.9999991157591201| A|
[551|0.999999118998829| C|
[585|0.9999990829349577| C|
[474| 0.999999043313599| A|
[596|0.9999990333394736| B|
```

+---+-----+
only showing top 20 rows

Sorting Run 50 took 5.189778566360474 seconds.

Average sorting time over 50 runs: 5.25242018699646 seconds.

Average aggregation time over 50 runs: 4.606600594520569 seconds.

Command took 8.22 minutes -- by linus.szokody@godiz.com at 10.1.2024, 21:00:55 on 14.2 Photon enabled

Test 3

Parquet Datei Erstellen

Die in Test 2 erstellten Daten wurden in eine Parquet Datei umgewandelt

Durchschnitt von 10 Runs

Cmd 5

Write File to Parquet X-Times

Cmd 6

```
1  num_iterations = 10
2  durations = []
3
4  for i in range(num_iterations):
5      start_time = time.time()
6      df.write.mode("overwrite").parquet("/sample/photon_enabled/1GB_Ints.parquet")
7      end_time = time.time()
8
9      duration = end_time - start_time
10     durations.append(duration)
11     print(f"Writing Run {i+1} took {duration} seconds.")
12
13     average_duration = sum(durations) / num_iterations
14     print(f"Average parquet writing time over {num_iterations} runs: {average_duration} seconds.")
```

► (10) Spark Jobs

```
Writing Run 1 took 63.24141263961792 seconds.
Writing Run 2 took 60.456552267074585 seconds.
Writing Run 3 took 67.50304508209229 seconds.
Writing Run 4 took 63.92989182472229 seconds.
Writing Run 5 took 65.89549589157104 seconds.
Writing Run 6 took 67.2645046710968 seconds.
Writing Run 7 took 71.41995644569397 seconds.
Writing Run 8 took 69.89065933227539 seconds.
Writing Run 9 took 62.45558476448059 seconds.
Writing Run 10 took 60.49013876914978 seconds.
Average parquet writing time over 10 runs: 65.25472416877747 seconds.
```

Command took 10.88 minutes -- by linus.szokody@godiz.com at 10.1.2024, 20:39:08 on 14.2 Photon enabled

Photon Testen

Tests sind in Präsentation genauer beschrieben

- Sortieren / Aggregieren aus generierten Daten (25Mio Records)
 - Mit Photon: 5.252s (Sort); 4.607s (Agg)
 - Ohne Photon: 4.151s (Sort); 4.342s (Agg)
- Parquet erstellen
 - Mit Photon: 65.255 Sekunden
 - Ohne Photon: 63.757 Sekunden
- 1GB Ints sortieren (aus Azure Blob Storage)
 - Mit Photon: 58.482 Sekunden
 - Ohne Photon: 51.046 Sekunden

Mögliche Erklärung:

- Zu kleine Datenmenge
 - Photon overhead durch mehr möglichkeiten für Catalyst?

→ Neuer Test mit komplexen JOINS

Test 4

Relationale Datenbank erstellt

Mit Random Daten befüllt

```
Cmd 1
Create DB

Cmd 2
1 # Import necessary libraries
2 from pyspark.sql import SparkSession
3 from pyspark.sql.functions import col
4
5 # Create a Spark session
6 spark = SparkSession.builder.appName("DatasetCreation").enableHiveSupport().getOrCreate()
7
8 # Drop databases if they exist
9 spark.sql("DROP DATABASE IF EXISTS db1 CASCADE")
10 spark.sql("DROP DATABASE IF EXISTS db2 CASCADE")
11
12 # Create databases
13 spark.sql("CREATE DATABASE IF NOT EXISTS db1")
14 spark.sql("CREATE DATABASE IF NOT EXISTS db2")
15
16 # Create tables in db1
17 spark.sql("""
18     CREATE TABLE IF NOT EXISTS db1.customers (
19         customer_id INT,
20         name STRING,
21         age INT
22     )
23 """)
24
25 spark.sql("""
26     CREATE TABLE IF NOT EXISTS db1.orders (
27         order_id INT,
```

```
Cmd 3
Create Data

Cmd 4
1 from pyspark.sql import SparkSession, Row
2 from pyspark.sql.functions import rand, concat, lit, monotonically_increasing_id, col
3 import random
4 import datetime
5
6 # Initialize Spark session
7 spark = SparkSession.builder.appName("EnhancedRandomDataGeneration").enableHiveSupport().getOrCreate()
8
9 # Sample size parameters
10 num_customers = 100000
11 num_orders = 500000
12 num_products = 10000
13 num_order_details = 1500000
14
15 def random_date(start, end):
16     """Generate a random datetime between start and end"""
17     delta = end - start
18     int_delta = (delta.days * 24 * 60 * 60) + delta.seconds
19     random_second = random.randrange(int_delta)
20     return start + datetime.timedelta(seconds=random_second)
21
22 # Generate random data for customers
23 df_customers = spark.range(num_customers) \
24     .withColumn("customer_id", (monotonically_increasing_id() + 1).cast("int")) \
25     .withColumn("name", concat(lit("Customer_"), col("customer_id"))) \
26     .withColumn("age", (rand() * 40 + 18).cast("int"))
27
28 # Generate random data for orders
29 df_orders = spark.range(num_orders) \
30     .withColumn("order_id", (monotonically_increasing_id() + 1).cast("int")) \
31     .withColumn("customer_id", df_customers.select("customer_id").collect().sample(withReplacement=True, size=num_orders)) \
32     .withColumn("product_id", (rand() * 10000).cast("int")) \
33     .withColumn("amount", (rand() * 100).cast("int")) \
34     .withColumn("timestamp", random_date(datetime.datetime.now(), datetime.datetime.now() + datetime.timedelta(days=30)))
```

Test mit JOINS

1 Abfrage mit:

- 3 Inner Joins
- 1 Left Join auf Subquery
- Gruppierung
- Durchschnitt
- Summe
- Min/Max
- Count
- Sortiert

Durchschnitt von 50 Runs auf grösseres Dataset

Join Data

Cmd 6

```
1 import time
2
3 num_iterations = 50
4 durations = []
5
6 # Example of a complex join query across databases
7 for i in range(num_iterations):
8     start = time.time()
9     result_df = spark.sql("""
10         SELECT
11             c.name AS CustomerName,
12             c.age AS CustomerAge,
13             o.order_year,
14             o.order_month,
15             COUNT(DISTINCT o.order_id) AS TotalOrders,
16             SUM(od.quantity) AS TotalProductsOrdered,
17             AVG(o.order_total) AS AverageOrderValue,
18             SUM(o.order_total) AS TotalOrderValue,
19             MAX(p.price) AS MaxProductPrice,
20             MIN(p.price) AS MinProductPrice,
21             p_top.product_name AS MostExpensiveProductOrdered
22         FROM
23             db1.customers c
24         JOIN
25             (SELECT order_id, customer_id, YEAR(order_date) AS order_year, MONTH(order_date) AS order_month, order_total
26              FROM db1.orders) o ON c.customer_id = o.customer_id
27         JOIN
28             db1.order_details od ON o.order_id = od.order_id
29         JOIN
30             db2.products p ON od.product_id = p.product_id
31         LEFT JOIN
32             (SELECT product_id, product_name FROM db2.products ORDER BY price DESC LIMIT 1) p_top ON od.product_id = p_top.product_id
33         GROUP BY
34             c.name, c.age, o.order_year, o.order_month, p_top.product_name
35         ORDER BY
36             TotalOrderValue DESC, TotalProductsOrdered DESC
37         LIMIT 10
38     """)
39     # Show the result
40     result_df.show()
41     end = time.time()
42     duration = end - start
43     durations.append(duration)
44     print(duration)
45
46 average_duration = sum(durations) / num_iterations
47 print(f"Average query time over {num_iterations} runs: {average_duration} seconds.")
48
49
```

Resultat

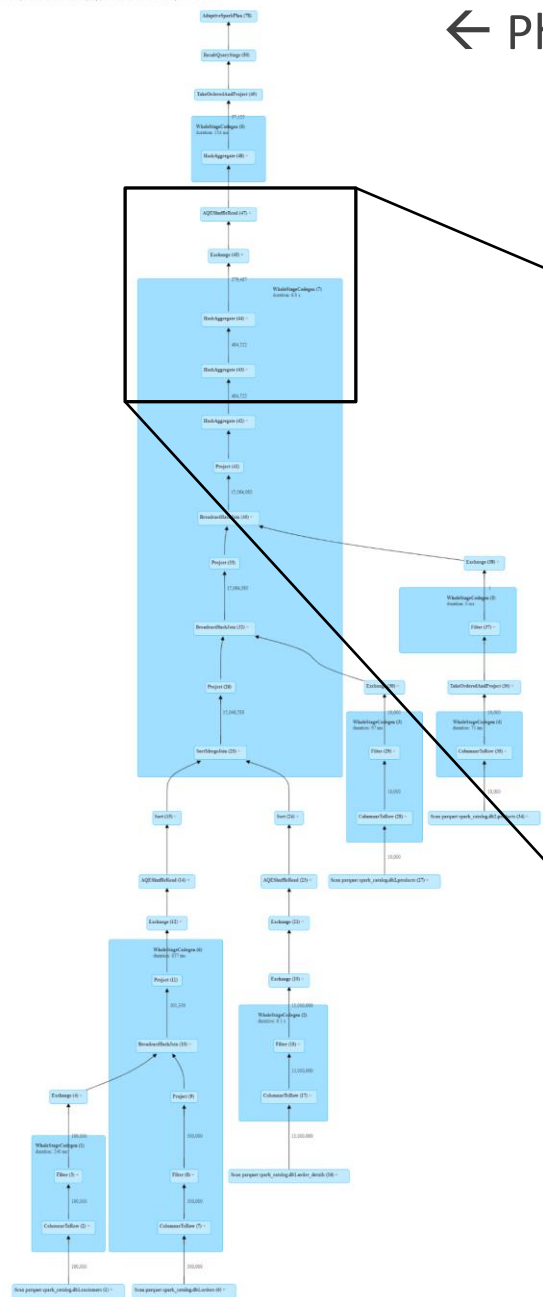
Ohne Photon:

Average query time over 50 runs: **11.28707139968872** seconds.

Mit Photon:

Average query time over 50 runs: **3.189061794281006** seconds.

-> Query fast 4x schneller mit Photon enabled



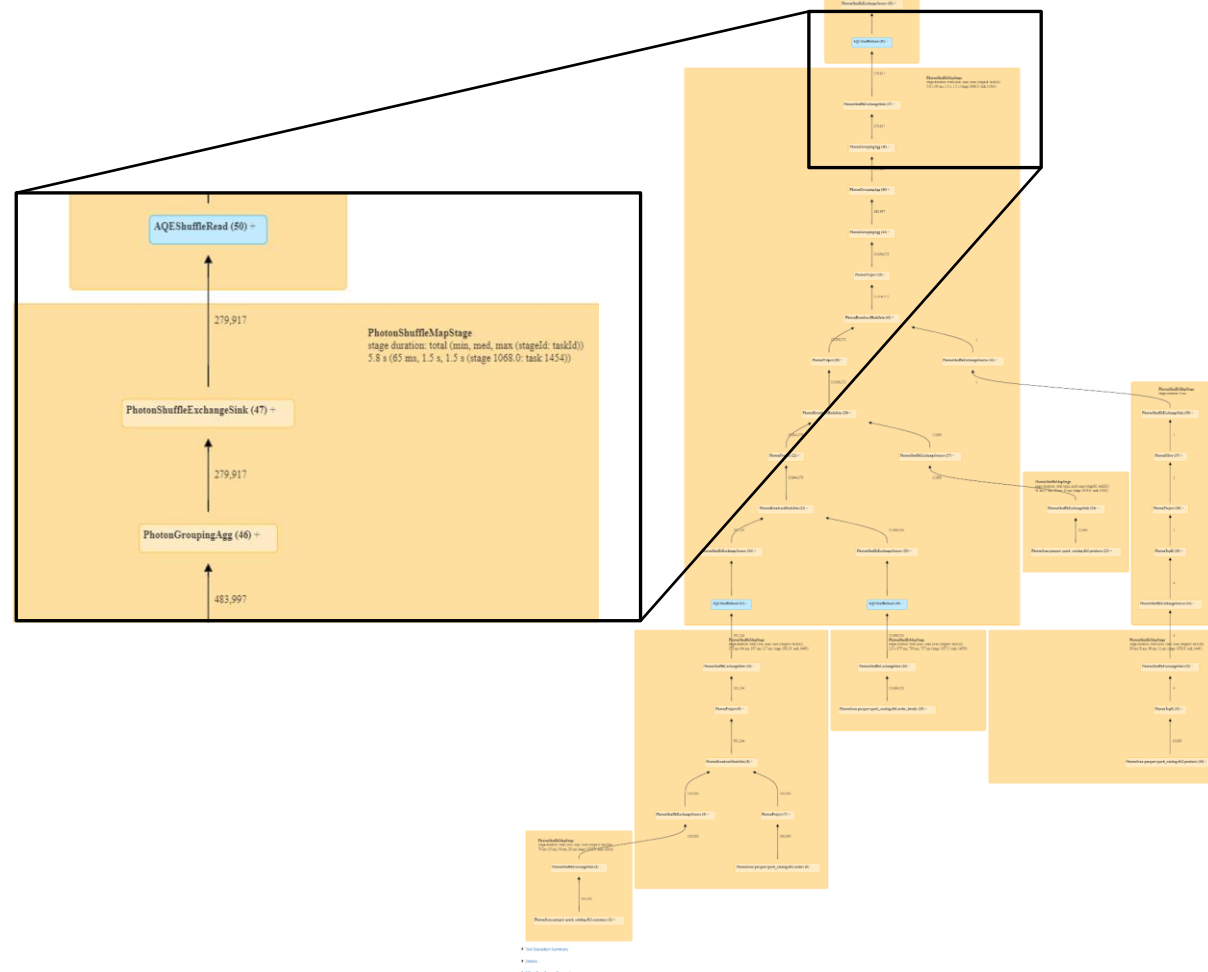
← Photon disabled

Spark UI

Blau = Spark ausführung

Gelb = Photon ausführung

Photon enabled →



Photon – nicht unterstützte Abfragen

Anwenden einer UDF (User defined Function)

Im Execution Plan sehen wir,
Photon unterstützt keine UDFs

```

1  from pyspark.sql.functions import udf
2  from pyspark.sql.types import StringType
3  import re
4
5  # a user defined function
6  def complex_text_processing(text):
7      processed_text = re.sub(r'\s+', ' ', text)
8      if "special_keyword" in processed_text:
9          processed_text = processed_text.replace("special_keyword", "replacement")
10         return processed_text[::-1] # Reversing the text for the sake of complexity
11
12     complex_text_udf = udf(complex_text_processing, StringType())
13
14     # Sample DataFrame operation
15     df_result = df.filter(df.id > 100) \
16         .withColumn('processed_text', complex_text_udf(df.text)) \
17         .orderBy(df.timestamp)
18
19     # Execution plan
20     print(df_result.explain())
21
22     # to check spark UI
23     df_result.show()

```

▼ (1) Spark Jobs

► Job 21 [View](#) (Stages: 1/1)

► df_result: pyspark.sql.dataframe.DataFrame = [id: integer, text: string ... 2 more fields]

== Physical Plan ==

AdaptiveSparkPlan isFinalPlan=false

+-- Sort [timestamp#104 ASC NULLS FIRST], true, 0

+-- Exchange rangepartitioning(timestamp#104 ASC NULLS FIRST, 200), ENSURE_REQUIREMENTS, [plan_id=586]

+-- Project [id#102, text#103, timestamp#104, pythonUDF0#114 AS processed_text#109]

+-- BatchEvalPython [complex_text_processing(text#103)#108], [pythonUDF0#114]

+-- Filter (isnotnull(id#102) AND (id#102 > 100))

+-- Scan ExistingRDD[id#102,text#103,timestamp#104]

== Photon Explanation ==

Photon does not fully support the query because:

Unsupported node: Scan ExistingRDD[id#102,text#103,timestamp#104].

Reference node:

Scan ExistingRDD[id#102,text#103,timestamp#104]

None

```

+-----+-----+-----+-----+
| id|    text|    timestamp|processed_text|
+-----+-----+-----+-----+

```

Command took 7.17 seconds -- by linus.szokody@godiz.com at 10.1.2024, 20:55:39 on 14.2 Photon enabled

Abschluss

Photon kann ein Query schneller machen aber nicht bei kleinen Datenmengen.

Wenn Photon eingeschaltet wird, kosten die Cluster auch mehr

→ Abwägung nötig, ob höhere Kosten und dafür schneller Verarbeitung lohnenswert sind

Github mit Logs und Folien: https://github.com/Szokody/ZHAW_BigData_Photon

Wer selbst mit Databricks spielen will:

-> PPTX Anleitung wie man auf Azure selbst eine Databricks Umgebung aufbaut

Links

Databrick on Azure:

- <https://learn.microsoft.com/en-us/azure/databricks>

Databrick Benchmarks:

- <https://docs.databricks.com/en/index.html>

Databrick Dokumentation:

- <https://docs.databricks.com/en/index.html>

Photon Kosten vs. Nutzen:

- <https://synccomputing.com/databricks-photon-and-graviton-instances-worth-it/>

Photon Whitepaper:

- https://people.eecs.berkeley.edu/~matei/papers/2022/sigmod_photon.pdf

Quellen

<https://datasolut.com/was-ist-databricks>

<https://learn.microsoft.com/en-us/azure/databricks/compute/photon>

<https://learn.microsoft.com/en-us/azure/databricks/spark/>

<https://docs.databricks.com/en/compute/photon.html>