CMSI 620 Group Project: Database Engine with Distributed Computing, Apache Spark

Mariam Joan

Date: Nov 28, 2023

Table of Contents:

Technical Overview

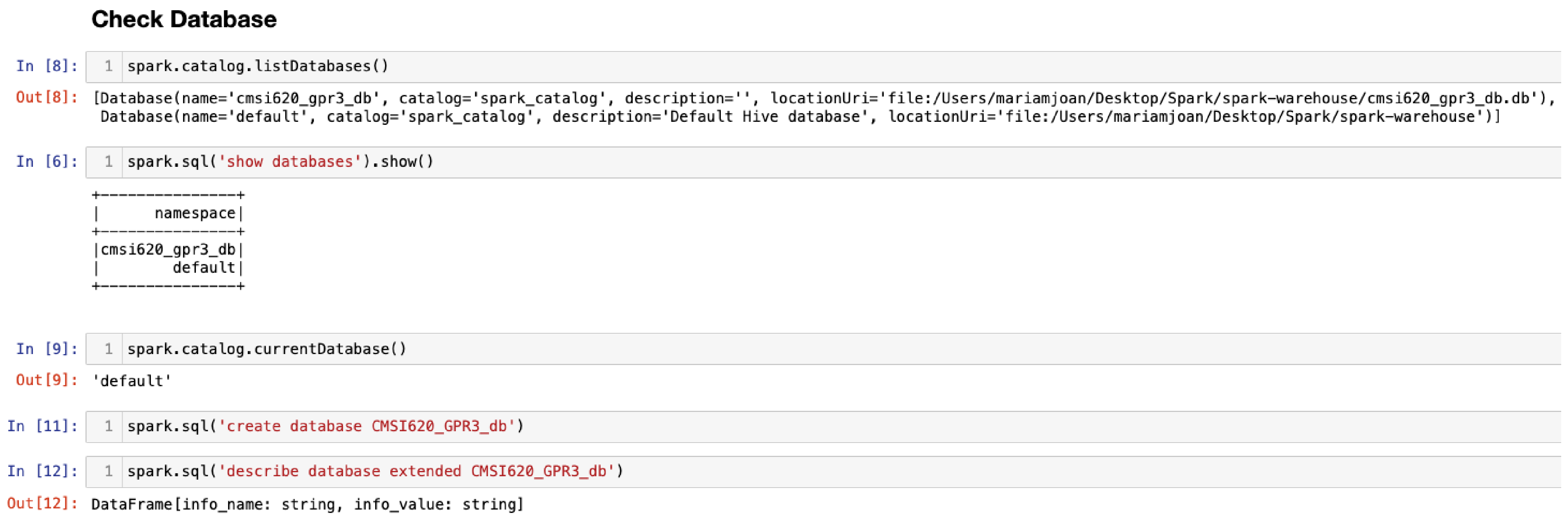
Initial Setup:

1. Download Apache Spark 3.4.1 using Spark via Homebrew
   a. By running `brew --prefix apache-spark` we can see where Homebrew installed spark
   b. We can see Spark is installed here: `/usr/local/opt/apache-spark`
   c. We also have to apply variables to our `HOME` or ~ directory to ensure Spark is configured correctly locally:
      i. `export SPARK_HOME="/usr/local/opt/apache-spark/"`
      ii. `export PATH=$SPARK_HOME/bin:$PATH`
2. Setup the Spark default or Hive metastore:
   a. Hive: Instructions from Git
3. Developer environment used to support project:
   a. Python 3.11.5
   b. Spark SQL and PySpark programming (Jupyter Notebooks)
   c. Visual Studio IDE

Setup Learnings:

```python
warehouse_location = abspath('spark-warehouse')
dataset_size_in_bytes = 1073741824

def calculate_partitions(dataset_size_in_bytes, bytes_per_partition=10000000):
    return max(1, int(dataset_size_in_bytes / bytes_per_partition))

spark = (SparkSession \
    .builder \
    .appName("CMSI 620: Database Engine") \
    .config("spark.executor.memory", "10g") \
    .config("spark.executor.memoryOverhead", "5g") \
    .config("spark.driver.memory", "5g") \
    .config("spark.driver.memoryOverhead", "3g") \
    .config("spark.memory.fraction", "0.1") \
    .config("spark.executor.instances", "5") \
    .config("spark.executor.cores", "5") \
    .config("spark.sql.autoBroadcastJoinThreshold", "100m") \
    .config("spark.sql.inMemoryColumnarStorage.compressed", "true") \
    .config("spark.sql.warehouse.dir", warehouse_location) \
    .config("spark.sql.catalogImplementation", "hive") \
    .config("spark.sql.shuffle.partitions", calculate_partitions(dataset_size_in_bytes)) \
    .enableHiveSupport() \
    .getOrCreate())
```

By default, Spark's metastore is called 'default' for the user's session. However, you can configure a different metastore such as Hive, in order to either access Hive tables, or create Hive tables using SparkSQL. I learned Hive comes bundled with the Spark library as HiveContext, which inherits from SQLContext[1] so Spark by default uses Apache Hive metastore located at /user/hive/warehouse which will persist all metadata from your tables, which you can change via configuration to the spark.sql.warehouse.dir. Hive is a data warehousing DBMS, and is built on top of Hadoop Distributed File System. Hive now called Apache, was built by Facebook[2]. Spark can be set up using the Hive metastore with a few configurations in your bash .bashrc or .zshrc files. SparkSQL comes with a default database while you can also create a database:



Check Database

```
In [8]:  spark.catalog.listDatabases()

Out[8]: [Database(name='cmsi620_gpr3_db', catalog='spark_catalog', description='', locationUri='file:/Users/mariamjoan/Desktop/Spark/spark-warehouse/cmsi620_gpr3_db.db'),
         Database(name='default', catalog='spark_catalog', description='Default Hive database', locationUri='file:/Users/mariamjoan/Desktop/Spark/spark-warehouse')]

In [6]:  spark.sql('show databases').show()

+---------------+
|      namespace|
+---------------+
|cmsi620_gpr3_db|
|        default|
+---------------+

In [9]:  spark.catalog.currentDatabase()

Out[9]: 'default'

In [11]: spark.sql('create database CMSI620_GPR3_db')

In [12]: spark.sql('describe database extended CMSI620_GPR3_db')

Out[12]: DataFrame[info_name: string, info_value: string]
```

[1] Tutorialspoint.com, 2023, *Spark SQL - Hive Tables*, Retrieved from: https://www.tutorialspoint.com/spark_sql/spark_sql_hive_tables.htm#:~:text=Hive%20comes%20bundled%20with%20the,can%20still%20create%20a%20HiveContex

[2] Berman, D., 2023, Comparing Apache Hive vs. Spark, Retrieved from: https://logz.io/blog/hive-vs-spark/#:~:text=Hive%20and%20Spark%20are%20both,more%20modern%20alternative%20to%20MapReduce

Regarding table creation, SparkSQL supports two kinds of tables, managed and unmanaged or otherwise known as external. With managed tables, if you create a table as managed, Spark will store this inside the database directory location, while alternatively, if you drop that same table it will simply delete the file from that location along with the table subdirectory (Damji, et. al., 2020). Additionally, for unmanaged or external tables, this means upon creating these tables the location will be outside of the default database directory which works better in cases where you have the JDBC connection setup such as with Hive, etc. In these cases, you can't simply drop external tables as you'd need to have the configurations applied during setup to properly drop and or manipulate data within these tables. For our project, we will be using managed tables.

Introduction to Spark

Basic Terminology:

1. Application: This is the program operating for the user which has a driver and coordinates with executor nodes on a cluster.
2. SparkSession: This is an object that allows the user to interact with the Spark APIs, and also the Spark driver, a "session" is instantiated for your program.
3. Job: This consists of multiple tasks upon a Spark action like .collect().
4. Stage: Dependent on jobs that make up a stage.
5. Task: One operation that gets sent to the Spark executor.

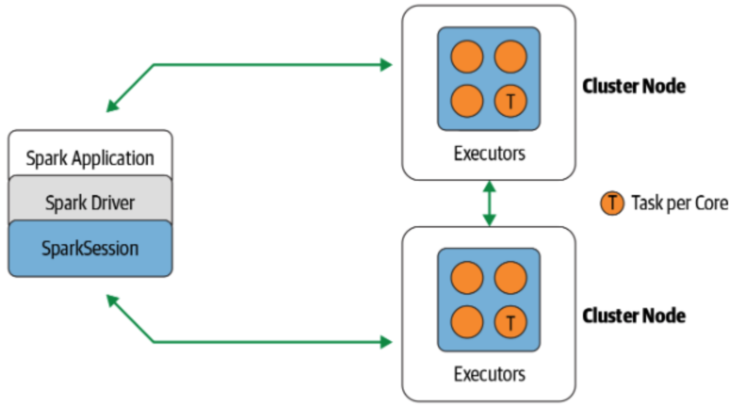


Figure 2-2. Spark components communicate through the Spark driver in Spark's distributed architecture [3]

[3] Damji, J.S., et al., 2020, Learning Spark, Downloading Apache Spark and Getting Started, Step 3: Understanding Spark Application Concepts, 2nd Ed., O'Reilly Media

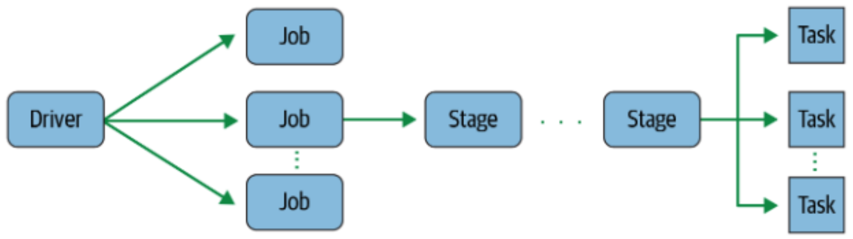


Figure 2-5. Spark stage creating one or more tasks to be distributed to executors [4]

Building Data Model and Schema from DataTypes

The core dataset we are using is called GlobalLandTemperaturesByCity sourced via Kaggle providing data on temperature changes by city. We will also look at a CO2 emissions dataset sourced from Our World in Data which is a project of the Global Change Data Lab, a non-profit organization based in the United Kingdom (Registered Charity Number 1186433). Finally we will also load in the Sea Level dataset, which accounts for sea level rising across the globe. In all cases we will build a schema and import the csv data.

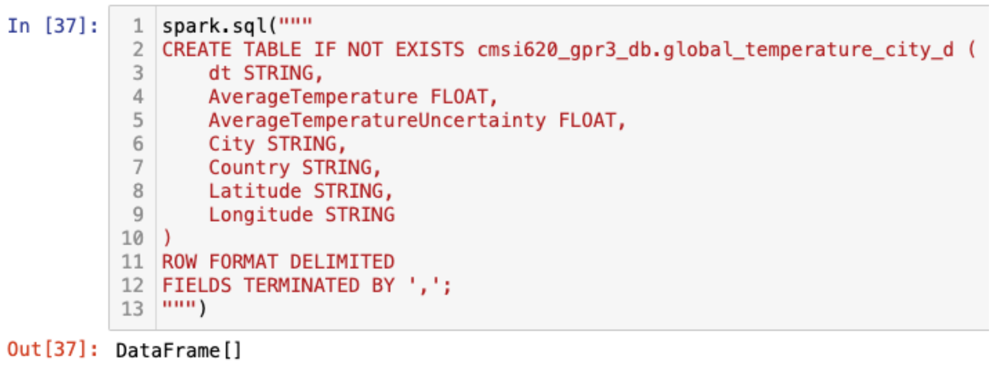
Extract and Explore:

The beginning of the notebook and images below show DDL and required methods to show databases, use the newly created CMSI620_GRP3_DB database for this project, list current tables and finally describe the tables in the database.

Implementing DDL and DML

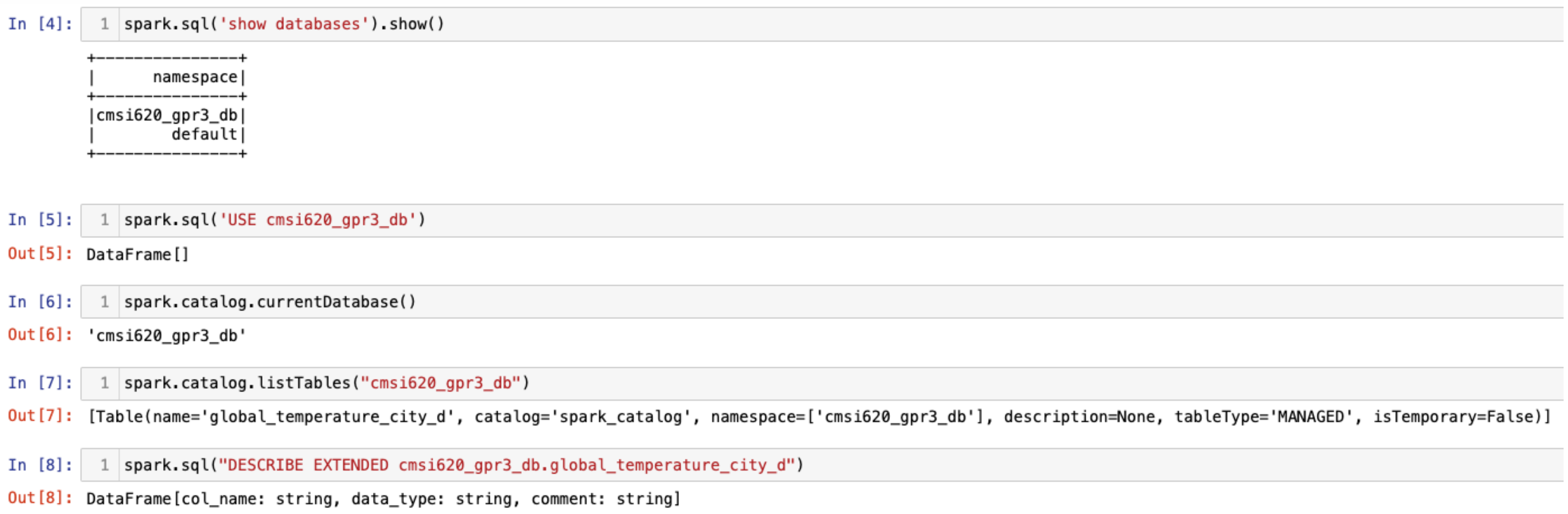
The DDL and DML language used for this project were SparkSQL which are very similar to MySQL. See below:

[4] Damji, J.S., et al., 2020, Learning Spark, Downloading Apache Spark and Getting Started, Step 3: Understanding Spark Application Concepts, 2nd Ed., O'Reilly Media


```
In [37]:
spark.sql("""
CREATE TABLE IF NOT EXISTS cmsi620_gpr3_db.global_temperature_city_d (
    dt STRING,
    AverageTemperature FLOAT,
    AverageTemperatureUncertainty FLOAT,
    City STRING,
    Country STRING,
    Latitude STRING,
    Longitude STRING
)
ROW FORMAT DELIMITED
FIELDS TERMINATED BY ',';
""")

Out[37]: DataFrame[]
```

DML and DDL used in the notebook:

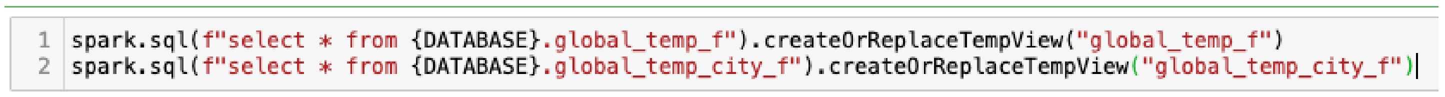
1. CREATE TABLE
2. USE SCHEMA
3. DESCRIBE
4. DROP TABLE IF EXISTS
5. DROP DATABASE IF EXISTS
6. INSERT INTO TABLE_NAME
7. SELECT * FROM WHERE TRUE



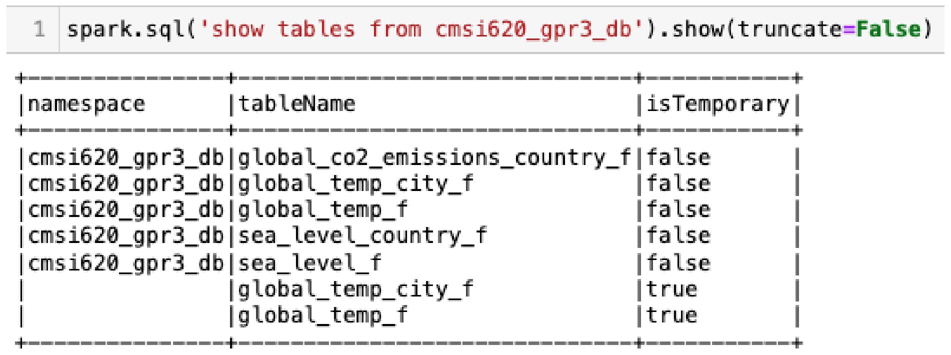
```
In [4]: spark.sql('show databases').show()

+--------------+
|     namespace|
+--------------+
|cmsi620_gpr3_db|
|       default|
+--------------+

In [5]: spark.sql('USE cmsi620_gpr3_db')
Out[5]: DataFrame[]

In [6]: spark.catalog.currentDatabase()
Out[6]: 'cmsi620_gpr3_db'

In [7]: spark.catalog.listTables("cmsi620_gpr3_db")
Out[7]: [Table(name='global_temperature_city_d', catalog='spark_catalog', namespace=['cmsi620_gpr3_db'], description=None, tableType='MANAGED', isTemporary=False)]

In [8]: spark.sql("DESCRIBE EXTENDED cmsi620_gpr3_db.global_temperature_city_d")
Out[8]: DataFrame[col_name: string, data_type: string, comment: string]
```

Implementing Views

This is a very useful way for flexibility in querying across business needs because you can store data as views but not actually have it stored in our database. For example below you can see we created a temporary view of our two global temperature and city tables to be able to run queries against them and or perform query operations which will run more efficiently because the views are stored within local memory. Below you can see also when we query our database, these views show up as “temporary” tables which means once you quit your SparkSessions these views are gone.



```
spark.sql(f"select * from {DATABASE}.global_temp_f").createOrReplaceTempView("global_temp_f")
spark.sql(f"select * from {DATABASE}.global_temp_city_f").createOrReplaceTempView("global_temp_city_f")
```

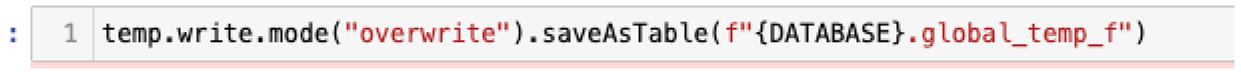


```python
spark.sql('show tables from cmsi620_gpr3_db').show(truncate=False)
```

```
+---------------+--------------------------------+-----------+
|namespace      |tableName                       |isTemporary|
+---------------+--------------------------------+-----------+
|cmsi620_gpr3_db|global_co2_emissions_country_f|false      |
|cmsi620_gpr3_db|global_temp_city_f              |false      |
|cmsi620_gpr3_db|global_temp_f                   |false      |
|cmsi620_gpr3_db|sea_level_country_f             |false      |
|cmsi620_gpr3_db|sea_level_f                     |false      |
|               |global_temp_city_f              |true       |
|               |global_temp_f                   |true       |
+---------------+--------------------------------+-----------+
```

Storing Data to Tables

There are many ways to store data in the Spark database, of which below is one option. There are different WRITE modes, such as “append”, or “overwrite”.



```python
temp.write.mode("overwrite").saveAsTable(f"{DATABASE}.global_temp_f")
```

Project Rubric Criteria:

Apache Spark is a fast distributed computing system. It provides high-level APIs in Java, Scala, Python and R, and an optimized engine that supports general execution graphs. It also supports a rich set of higher-level tools including Spark SQL for SQL and structured data processing, MLlib for machine learning, GraphX for graph processing, and Spark Streaming.

Performance:

Spark performance is efficient and fast due to a number of components within its architecture such as:

1. In memory processing: Reduces need to read/write to disk (minimizes I/O overhead)
2. Distributed computing: Otherwise known as parallel processing, to scale horizontally across multiple nodes.
3. RDDs: Fault tolerant immutable data objects distributed across nodes,
4. Lazy Evaluation: Reduces reads, and optimizes execution plan, e.g. splits operations
5. Catalyst Optimizer: Advanced analysis on the execution plan, generating a physical execution plan for Spark queries.
6. Data Locality: Keeps data localized first reducing impact of network latency until necessary.

Scalability

All Spark applications as the example shown in the notebook, starts with a driver program that runs the functions and configurations provided, then executes these configurations on your particular cluster of nodes, or in this case, one cluster, locally.

The primary way in which Spark is able to scale is that it runs your data on a resilient distributed dataset or RDD, which is a collection partitioned across nodes of the cluster that can be operated on in parallel. RDDs begin from an external storage system such as a Hadoop file system or any data source offering a Hadoop InputFormat, which is what the database is sourced from, and transforms your data in a series of parallel operations, either in memory locally or across multiple machines. One important note about RDDs is that they are fault tolerant and recover from any node failure. Finally, Spark also uses a global variable environment, as either broadcast or accumulators, in which it can compute these parallel operations on a set of tasks, in addition to running methods or functions literally copied over all cluster nodes

One requirement that makes a Spark application scalable is persistence or consistency across clusters, which describes the fault tolerance, meaning any node failure will trigger the RDD to automatically be recomputed and stored. Every RDD gets persisted across each node when using cache() or persist() method. This means that all your data from the RDD is stored and kept in memory, recomputing any transformations or actions on the data, allowing future actions to be faster (more than 10x). In addition you can specify the storage levels across nodes, and where the data persists.

Flexibility

Apache Spark is a fast and general-purpose cluster computing system. It provides high-level APIs in Java, Scala, Python and R, and an optimized engine that supports general execution graphs. It also supports a rich set of higher-level tools including Spark SQL for SQL and structured data processing.

Because Spark computes parallel operations on your RDDs across the cluster, this creates a flexibility with the data that no traditional RDBMS can do. Through either transformations or actions, which are the two primary ways in which RDDs operate, it either creates a new dataset transformed or only computes functions when actions are triggered which makes your program more efficient. By default, each RDD can be recomputed and transformed multiple times either in memory using cache or on disk replicated across multiple nodes. Spark is still rigid in terms of schema flexibility because of the Hive relational database under the hood.

Complexity

Spark is highly complex if you are new or a beginner to SQL or Python and distributed computing. Spark has a UI but, it’s not like the UI for Postgres or MySQL, the UI is only to monitor the job once you have set it up programmatically which includes configuration for Spark executor, driver, and JVM memory along with executor and driver function. The Spark UI is meant for monitoring and configuring so you are always aware of your application efficiency and the Spark UI offers details into database architecture that is not the kind of visibility you get into database management systems UIs like MySQL and Postgres.

Functionality

Once the user can establish a configuration for their dataset, Spark API offers multiple options of interaction programmatically with Python, R, Scala and Java which is very helpful for the Spark developer community. Spark does not offer indexing as an option nor does it officially offer ACID properties or CRUD operations however, it does offer aspects of them. All in all, Spark offers many configurations and customizations to support your application including integrations with data lakes and data warehouses that can offer traditional RDBMS properties.

Spark can run idempotent or isolated operations on an RDD, which means that an operation can be repeated multiple times but the result is the same each time. In addition, for consistency, or fault tolerance Spark offers “checkpointing” which means that your data is resilient to node failures because if a failure occurs, it automatically recomputes the data to each respective location, example below.

-
  ```
  spark.sparkContext.setCheckpointDir("hdfs://my/checkpoint/directory")
  df.checkpoint()
  ```

CRUD Operations → See Notebook Output for Examples

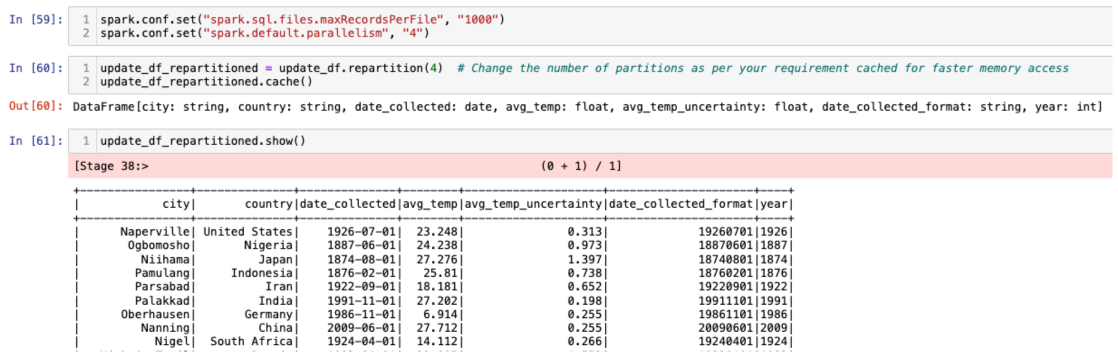
Because Spark is a distributed processing framework, we aren’t necessarily doing CRUD operations within the database in traditional RDBMS sense but, rather performing these operations on the Spark Dataframe or RDD (resilient distributed dataset) and then writing the new DataFrame or RDD to the existing table as an overwrite or a new table. The reasoning for this is because of Spark’s internal architecture being immutable and distributed, because it is distributed processing a table in place, can happen across the nodes, in an efficient way so instead the Spark DataFrame API allows for parallel and distributed processing to be able to modify the data in place, then these representations of the data are replicated across nodes.

ACID Operations → See Notebook Output for Examples

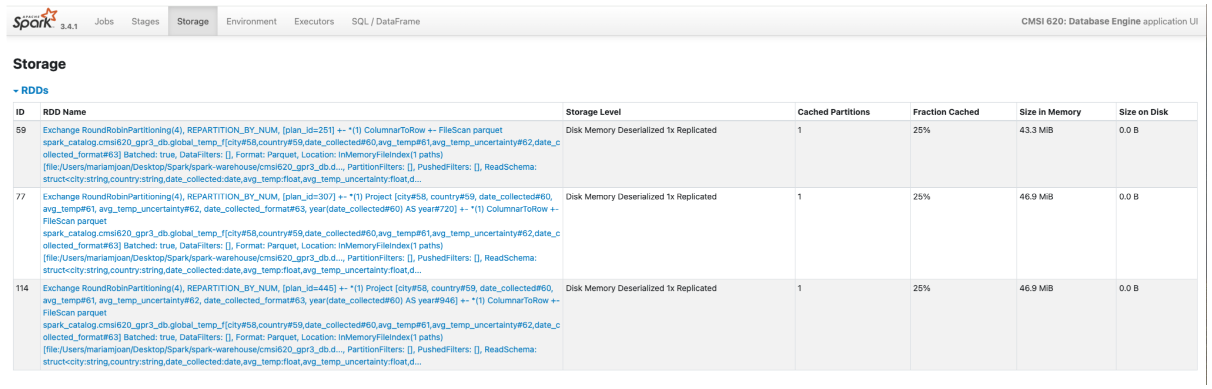
Spark is mainly designed for distributed processing of data, and transactional guarantees are handled at data storage level like as with HADOOP or distributed file systems, or databases that can be used in conjunction with Spark not by Spark alone however, Spark does by nature resolve many of these operations such as consistency, atomicity and durability.

Replication

This configuration spark.sql.files.maxRecordsPerFile controls the maximum number of records to write out to a single file when saving a DataFrame to a distributed file system like HDFS. Replication in Spark is a default component of fault tolerance and also the replication requirements of RDDs or resilient distributed datasets, which by nature are replicated across nodes for easy data processing which is also controlled by the spark.default.parallelism configuration, which sets the number of partitions for distributed operations. Please see the last cell in the notebook and Replication video in Slides for full images.



```
In [59]: spark.conf.set("spark.sql.files.maxRecordsPerFile", "1000")
         spark.conf.set("spark.default.parallelism", "4")

In [60]: update_df_repartitioned = update_df.repartition(4)  # Change the number of partitions as per your requirement cached for faster memory access
         update_df_repartitioned.cache()

Out[60]: DataFrame[city: string, country: string, date_collected: date, avg_temp: float, avg_temp_uncertainty: float, date_collected_format: string, year: int]

In [61]: update_df_repartitioned.show()

         [Stage 38:>                                                         (0 + 1) / 1]

         +---------------+-------------+--------------+--------+--------------------+---------------------+----+
         |           city|      country|date_collected|avg_temp|avg_temp_uncertainty|date_collected_format|year|
         +---------------+-------------+--------------+--------+--------------------+---------------------+----+
         |     Naperville| United States|    1926-07-01|  23.248|               0.313|             19260701|1926|
         |      Ogbomosho|      Nigeria|    1887-06-01|  24.238|               0.973|             18870601|1887|
         |        Niihama|        Japan|    1874-08-01|  27.276|               1.397|             18740801|1874|
         |       Pamulang|    Indonesia|    1876-02-01|   25.81|               0.738|             18760201|1876|
         |       Parsabad|         Iran|    1922-09-01|  18.181|               0.652|             19220901|1922|
         |       Palakkad|        India|    1991-11-01|  27.202|               0.198|             19911101|1991|
         |     Oberhausen|      Germany|    1986-11-01|   6.914|               0.255|             19861101|1986|
         |        Nanning|        China|    2009-06-01|  27.712|               0.255|             20090601|2009|
         |          Nigel| South Africa|    1924-04-01|  14.112|               0.266|             19240401|1924|
```



Storage

RDDs

ID	RDD Name	Storage Level	Cached Partitions	Fraction Cached	Size in Memory	Size on Disk
59	Exchange RoundRobinPartitioning(4), REPARTITION_BY_NUM, [plan_id=251] +- *(1) ColumnarToRow +- FileScan parquet spark_catalog.cmsi620_gpr3_db.global_temp_f[city#58,country#59,date_collected#60,avg_temp#61,avg_temp_uncertainty#62,date_collected_format#63] Batched: true, DataFilters: [], Format: Parquet, Location: InMemoryFileIndex(1 paths)[file:/Users/mariamjoan/Desktop/Spark/spark-warehouse/cmsi620_gpr3_db.d..., PartitionFilters: [], PushedFilters: [], ReadSchema: struct<city:string,country:string,date_collected:date,avg_temp:float,avg_temp_uncertainty:float,d...	Disk Memory Deserialized 1x Replicated	1	25%	43.3 MiB	0.0 B
77	Exchange RoundRobinPartitioning(4), REPARTITION_BY_NUM, [plan_id=307] +- *(1) Project [city#58, country#59, date_collected#60, avg_temp#61, avg_temp_uncertainty#62, date_collected_format#63, year(date_collected#60) AS year#720] +- *(1) ColumnarToRow +- FileScan parquet spark_catalog.cmsi620_gpr3_db.global_temp_f[city#58,country#59,date_collected#60,avg_temp#61,avg_temp_uncertainty#62,date_collected_format#63] Batched: true, DataFilters: [], Format: Parquet, Location: InMemoryFileIndex(1 paths)[file:/Users/mariamjoan/Desktop/Spark/spark-warehouse/cmsi620_gpr3_db.d..., PartitionFilters: [], PushedFilters: [], ReadSchema: struct<city:string,country:string,date_collected:date,avg_temp:float,avg_temp_uncertainty:float,d...	Disk Memory Deserialized 1x Replicated	1	25%	46.9 MiB	0.0 B
114	Exchange RoundRobinPartitioning(4), REPARTITION_BY_NUM, [plan_id=445] +- *(1) Project [city#58, country#59, date_collected#60, avg_temp#61, avg_temp_uncertainty#62, date_collected_format#63, year(date_collected#60) AS year#946] +- *(1) ColumnarToRow +- FileScan parquet spark_catalog.cmsi620_gpr3_db.global_temp_f[city#58,country#59,date_collected#60,avg_temp#61,avg_temp_uncertainty#62,date_collected_format#63] Batched: true, DataFilters: [], Format: Parquet, Location: InMemoryFileIndex(1 paths)[file:/Users/mariamjoan/Desktop/Spark/spark-warehouse/cmsi620_gpr3_db.d..., PartitionFilters: [], PushedFilters: [], ReadSchema: struct<city:string,country:string,date_collected:date,avg_temp:float,avg_temp_uncertainty:float,d...	Disk Memory Deserialized 1x Replicated	1	25%	46.9 MiB	0.0 B

References:

1. Damji, J. S., et al., 2020, Learning Spark, 2nd Ed., Ch. 2