Branch: master ▾ Find file Copy path

CS167 / Labs / Lab7 / CS167_Lab7.md

akilsevim Update CS167_Lab7.md

698041b 26 days ago

2 contributors

Raw Blame History

274 lines (236 sloc) 12.7 KB

Lab 7

Objectives

- Run analytic queries on Parquet files.
- Understand the performance gains of working with Parquet files.
- Run SQL queries using SparkSQL

Prerequisites

- Setup the development environment as explained in Lab 6.
- Download the following sample file Chicago Crimes.

Lab Work

I. Project Setup (10 minutes)

Setup a new Scala project similar to Lab 6. Make sure to change the project name to Lab 7.

II. Initialize a SparkSession (5 minutes)

In this part, you will initialize your project with SparkSession to access SparkSQL and the DataFrame API.

1. In `App` class, add the following stub code.

```
import org.apache.spark.sql.{DataFrame, SparkSession}
import org.apache.spark.SparkConf

object App {

  def main(args : Array[String]) {
    val operation: String = args(0)
    val inputfile: String = args(1)

    val conf = new SparkConf
    if (!conf.contains("spark.master"))
      conf.setMaster("local[*]")
    println(s"Using Spark master '${conf.get("spark.master")}'")

    val spark = SparkSession
      .builder()
      .appName("CS167 Lab7")
      .config(conf)
```

```
        .getOrCreate()

      try {
        import spark.implicits._
        // TODO Load input

        input.show()
        input.printSchema()

        val t1 = System.nanoTime
        operation match {
          case "write-parquet" =>
            // TODO Write the input dataset to a parquet file. The file name is passed in args(2)
          case "write-parquet-partitioned" =>
            // TODO Write the input dataset to a partitioned parquet file by District. The file name is passed in ar
          case "top-crime-types" =>
            // TODO Print out the top five crime types by count "Primary_Type"
          case "find" =>
            // TODO Find a record by Case_Number in args(2)
          case "stats" =>
            // TODO Compute the number of arrests, domestic crimes, and average beat per district.
            // Save the output to a new parquet file named "stats.parquet"
          case "stats-district"  =>
            // TODO Compute number of arrests, domestic crimes, and average beat for one district (args(2))
            // Write the result to the standard output
        }

        val t2 = System.nanoTime
        println(s"Operation $operation on file '$inputfile' finished in ${(t2-t1)*1E-9} seconds")
      } finally {
        spark.stop
      }
    }
  }
```

III. Explore the data (15 minutes)

Check out the dashboard provided by the City of Chicago to explore how the data can be used. (Q1) What are the top five crime types?

IV. Load the input file (15 minutes)

The first step is to load the input file. In this lab, we will work with two types of files, CSV and Parquet files. We will distinguish them by inputfile extension (`.csv` or `.parquet`).

1. Replace the TODO item of loading the input using the correct format, either CSV or Parquet. To read a Parquet file is loaded using the command `spark.read.parquet()` .
2. In addition, since Parquet cannot deal with attribute names with spaces in them, you will need to rename all attibutes with spaces. In this lab, we will replace any space in an attribute name with an underscore `_` . You can do that using the operation `input.withColumnRenamed(old_name, new_name)` . The attributes that need to be renamed are `Case Number` , `Primary Type` , `Location Description` , `Community Area` , `FBI Code` , `X Coordinate` , `Y Coordinate` , and `Updated On` . Make sure to do that only when loading the CSV file.

V. Convert a CSV file to Parquet (20 minutes)

To see the difference between processing the CSV file and processing Parquet, the first step is to convert the CSV file to Parquet.

1. Implement the command `write-parquet` which writes the input as a Parquet file. The output filename is passed in `args(2)` .
2. (Q2) Compare the sizes of the CSV file and the resulting Parquet file? What do you notice? Explain.

Note: You will be able to see the inferred schema of the Parquet file at the output. It will look similar to the following snippet:

```
message spark_schema {
  optional int32 ID;
  optional binary Case_Number (UTF8);
  optional binary Date (UTF8);
  optional binary Block (UTF8);
  optional binary IUCR (UTF8);
  ...
```

3. (Q3) How many times do you see the schema the output? How does this relate to the number of files in the output directory? What do you make of that?

VI. Create a partitioned Parquet file (10 minutes)

SparkSQL allows you to write a partitioned file to speed up filters on specific attributes. Consider this a cheap and poor index that might sometimes help.

1. Implement the operation `write-parquet-partitioned` that writes the input to a parquet file after partitioning it by `District`.
2. The code will look like the following but you will need to specify the partitioning attribute and the output file name `input.write.partitionBy().parquet`.
3. (Q4) How does the output look like? How many files were generated?

VII. Create a view to run SQL queries (20 miutes)

1. For this lab, you need to implement all analysis queries using SQL. To do so, you first need to create a temporary view out of your input and run the SQL query on it.
2. To create a temporary view, use the following statement with your input data frame `.createTempView()`.
3. After creating the temp view, you can run the SQL query using the method `spark.sql(...)`.
4. Try a simple SQL query to make sure it works correct, e.g., `SELECT COUNT(*) FROM ...`.

Note: For each operation from this point on, you should run it three times, on the original CSV file, on the non-partitioned Parquet file, and on the partitioned Parquet file. Make sure that they all produce the same output. Record all the running times and include them in your README file.

VIII. `top-crime-types` (10 minutes)

1. Add the operation `top-crime-types` that lists the top five crime types by count.
2. You can use SparkSQL to process run the top-k query as we did in the previous lab.
3. Compare your results to the one shown in the dashboard.

VI. Find a record (10 minutes)

1. Add a operation `filter` that locates a crime record by `Case-Number` and writes the entire record.
2. (Q5) Explain an efficient way to run this query on a column store.
3. Try your code with the Case Number `HY413923` on the three file variations.

VII. Stats (10 minutes)

1. Add a operation `stats` that computes the total number of arrests, total number of domestic crimes, and the average beat for each district.
2. Write the output of this operation to a file named `stats.parquet`. Needless to say, it is a parquet file.

Hint: In SparkSQL, when you write the result to a file, you can automatically overwrite the file using the method `mode("overwrite")` when writing the file. I'll let you figure out how to use it. If you do not want to use this option, you can still delete the file after each run.

VIII. Stats for one district (20 minutes)

1. Similar to the previous operation, this operation will compute the same statistics but for only one district. The district ID is given as a command line argument `args(2)` .
2. (Q6) Which of the three input files you think will be processed faster using this operation?
3. Test your command with District ID 8.

IX. Submission (30 minutes)

1. Add a `README` file with all your answers.
2. Include the running time of all the operations you ran on the three files. The table will look similar to the following.

Command	Time on CSV	Time on non-partitioned Parquet	Time on partitioned Parquet
top-crime-types			
find			
stats			
stats-district			

3. Add a `run` script that compiles your code and then runs the following set of operations in order.

- Convert the CSV file to a non-partitioned Parquet file.
- Convert the CSV file to a partitioned Parquet file.
- Run the `top-crime-types` operation on the three files.
- Run the `find` operation with Case Number `HY413923` on the three files.
- Run the `stats` operation on the three files.
- Run the `stats-district` operation with District ID `8` on the three files.

Further Readings

The folllowing reading material could help you with your lab.

- Spark SQL Programming Guide
- Dataset API Docs

FAQ

- Q: My code does not compile using `mvn package` .
- Q: IntelliJ IDEA does not show the green run arrow next to the `App` class.
- A: Check your `pom.xml` file and make sure that the following sections are there in your file.

```
<properties>
  <maven.compiler.source>1.8</maven.compiler.source>
  <maven.compiler.target>1.8</maven.compiler.target>
  <encoding>UTF-8</encoding>
  <scala.version>2.12.6</scala.version>
  <scala.compat.version>2.12</scala.compat.version>
  <spec2.version>4.2.0</spec2.version>
  <spark.version>2.4.5</spark.version>
</properties>
```

```
  <dependencies>
    <dependency>
      <groupId>org.scala-lang</groupId>
      <artifactId>scala-library</artifactId>
      <version>${scala.version}</version>
    </dependency>
    <dependency>
      <groupId>org.apache.spark</groupId>
      <artifactId>spark-core_2.12</artifactId>
      <version>${spark.version}</version>
      <scope>compile</scope>
    </dependency>

    <!-- Test -->
    <dependency>
      <groupId>junit</groupId>
      <artifactId>junit</artifactId>
      <version>4.12</version>
      <scope>test</scope>
    </dependency>
    <dependency>
      <groupId>org.scalatest</groupId>
      <artifactId>scalatest_${scala.compat.version}</artifactId>
      <version>3.0.5</version>
      <scope>test</scope>
    </dependency>
    <dependency>
      <groupId>org.specs2</groupId>
      <artifactId>specs2-core_${scala.compat.version}</artifactId>
      <version>${spec2.version}</version>
      <scope>test</scope>
    </dependency>
    <dependency>
      <groupId>org.specs2</groupId>
      <artifactId>specs2-junit_${scala.compat.version}</artifactId>
      <version>${spec2.version}</version>
      <scope>test</scope>
    </dependency>
  </dependencies>

  <build>
    <sourceDirectory>src/main/scala</sourceDirectory>
    <testSourceDirectory>src/test/scala</testSourceDirectory>
    <plugins>
      <plugin>
        <!-- see http://davidb.github.com/scala-maven-plugin -->
        <groupId>net.alchim31.maven</groupId>
        <artifactId>scala-maven-plugin</artifactId>
        <version>3.3.2</version>
        <executions>
          <execution>
            <goals>
              <goal>compile</goal>
              <goal>testCompile</goal>
            </goals>
            <configuration>
              <args>
                <arg>-dependencyfile</arg>
                <arg>${project.build.directory}/.scala_dependencies</arg>
              </args>
            </configuration>
          </execution>
        </executions>
      </plugin>
      <plugin>
        <groupId>org.apache.maven.plugins</groupId>
        <artifactId>maven-surefire-plugin</artifactId>
        <version>2.21.0</version>
        <configuration>
```

```
            <!-- Tests will be run with scalatest-maven-plugin instead -->
            <skipTests>true</skipTests>
          </configuration>
        </plugin>
        <plugin>
          <groupId>org.scalatest</groupId>
          <artifactId>scalatest-maven-plugin</artifactId>
          <version>2.0.0</version>
          <configuration>
            <reportsDirectory>${project.build.directory}/surefire-reports</reportsDirectory>
            <junitxml>.</junitxml>
            <filereports>TestSuiteReport.txt</filereports>
            <!-- Comma separated list of JUnit test class names to execute -->
            <jUnitClasses>samples.AppTest</jUnitClasses>
          </configuration>
          <executions>
            <execution>
              <id>test</id>
              <goals>
                <goal>test</goal>
              </goals>
            </execution>
          </executions>
        </plugin>
      </plugins>
    </build>
```