Lab 6.2: Joins and Broadcasts

Overview

In this lab, we will explore some slightly more complex queries, including possible alternatives that use a join.

We'll also look at Catalyst's broadcast optimizations when doing joins, and examine performance with those optimizations in place and without them.

Builds on

None

Run time

30-40 minutes

Lab Preparation: Load Github Data

You've already worked with the data file (***spark-labs/data/github.json***). In previous labs you should already have done the below. If you haven't, then do it now.

Tasks

- Load the githup data in the Zeppelin shell, as below.

```
// Scala
> val githubDF=spark.read.json("spark-labs/data/github.json")
```

```
# Python
> githubDF=spark.read.json("spark-labs/data/github.json")
```

Top 20 Contributors - Straightforward Query

We want to find out the 20 contributors who have the most entries in the github data. Let's first do this in a straightforward way, as done in an earlier lab.

Tasks

- Find the top 20 contributors as follows.

```
// Scala
> val scanQuery =
githubDF.groupBy("actor.login").count.orderBy('count.desc).limit(20)
> scanQuery.show
```

```
# Python
> scanQuery = githubDF.groupBy("actor.login").count().orderBy("count",
ascending=False).limit(20)
> scanQuery.show()
```

- Open the Web UI (localhost:4040) and look at the jobs tab. Note
- the execution time (0.6s on our system)
- Drill down on this job, and note the shu#e data (304KB on our system)

Top 20 Contributors - Join Query

We've decided that we want to track 20 specific contirbutors (the top 20 contributors from the previous month), instead of the top 20 contributors at any given moment. We keep the login values of these contributors-of-interest in another data file we supply (***spark-labs/data/github-top20.json***)

Write a join query that gives the actor login and count from the entries in *github.json* for the login values in *github-top20.json*.

Tasks

- Review the data in *github-top20.json*. Accomplish our
- query needs by:
    - Loading *github-top20.json* into a dataframe (`githubTop20DF`)
    - Joining `githubDF` and `githubTop20DF`
    - Grouping by `actor.login`
    - Counting the results, and ordering by descending count.

```
// Scala
> val githubTop20DF = spark.read.json("spark-labs/data/github-
top20.json")
> val topContributorsJoinedDF = // Figure this out
> topContributorsJoinedDF.show // Show the results
```

```
# Python
> githubTop20DF = spark.read.json("spark-labs/data/github-top20.json")
> val topContributorsJoinedDF = // Figure this out
> topContributorsJoinedDF.show() // Show the results
```

- If you can't figure out this transformation in a reasonable amount of time, we supply the query in the helper file **joinStatements.txt**.
    - Use the Scala or Python version as appropriate.

Join Query Performance

Let's review the performance of this join query, and the optimizations that are being done by Catalyst.

Tasks

- Open a browser window on the Web UI - localhost:4040.
    - Note the time taken for the last `show` performed above, drill through on the job for it, and note the shu#e data.
    - On our system it took 0.6s and shu#ed 7.9K of data.
    - This is much less data shu#ing than the scanQuery done earlier (40 times reduction).
- Execute `topContributorsJoinedDF.explain` to see what is going on.
    - We illustrate this below. Note the broadcasts which we have highlighted. Catalyst has
    - optimized the join to use broadcast.

```
> topContributorsJoinedDF.explain()
== Physical Plan ==
*Sort [count#581L DESC NULLS LAST], true, 0
+- Exchange rangepartitioning(count#581L DESC NULLS LAST, 200)
 +- *HashAggregate(keys=[actor#80.login#587], functions=[count(1)])
   +- Exchange hashpartitioning(actor#80.login#587, 200)
    +- *HashAggregate(keys=[actor#80.login AS actor#80.login#587],
                                    functions=[partial_count(1)])
     +- *Project [actor#80]
************
      +- *BroadcastHashJoin [actor#80.login], [login#98], Inner, ...
************
       :- *FileScan json [actor#80] ...
struct<actor:struct<avatar_url:string,gravatar_id:string,id:bigint,
                        login:string,url:string>>
************
   +- BroadcastExchange HashedRelationBroadcastMode(
                                List(input[0, string, true]))
************
    +- *Project [login#98]
     +- *Filter isnotnull(login#98)
      +- *FileScan json [login#98] ... struct<login:string>
```

Join Performance without Broadcast

Let's disable the Catalyst broadcast optimization and look at what happens.

Tasks

- Disable automatic broadcasting, by seting the appropriate configuration property as shown below.
- Next, **create and show** the `topContributorsJoinedDF` query again
  - You must create the DataFrame after setting the config property, to make sure it conforms to the new setting.

```
> spark.conf.set("spark.sql.autoBroadcastJoinThreshold",-1)
```

- Open a browser window on the Web UI - localhost:4040.
  - Note the time taken, drill through on the job for this query, and note the

shu#e data.
    - On our system it took 1s and shu#ed 771K of data. This is
    - almost **100 times** the amount of shu#e data!
- `explain` the transformation
    - We illustrate this below. Note that the broadcasts are gone
    - We highlighted the replacements - a SortMergeJoin, and an Exchange (shu#e).
    - We've turned o**!** the broadcast Catalyst optimization.

```
> topContributorsJoinedDF.explain()
== Physical Plan ==
*Sort [count#634L DESC NULLS LAST], true, 0
+- Exchange rangepartitioning(count#634L DESC NULLS LAST, 200)
 +- *HashAggregate(keys=[actor#80.login#683], functions=[count(1)])
  +- Exchange hashpartitioning(actor#80.login#683, 200)
   +- *HashAggregate(keys=[actor#80.login AS actor#80.login#683],
                                          functions=[partial_count(1)])
    +- *Project [actor#80]
************
     +- *SortMergeJoin [actor#80.login], [login#98], Inner
************
        :- *Sort [actor#80.login ASC NULLS FIRST], false, 0
        :  +- Exchange hashpartitioning(actor#80.login, 200)
        :     +- *FileScan json [actor#80] ...
struct<actor:struct<avatar_url:string,gravatar_id:string,id:bigint,
                                 login:string,url:string>>
  +- *Sort [login#98 ASC NULLS FIRST], false, 0
************
   +- Exchange hashpartitioning(login#98, 200)
************
    +- *Project [login#98]
     +- *Filter isnotnull(login#98)
      +- *FileScan json [login#98] ... struct<login:string>
```

- Renable automatic broadcasting, by seting the appropriate configuration property as shown below.

```
> spark.conf.set("spark.sql.autoBroadcastJoinThreshold",1024*1024*10)
```

[Optional] Do the Same Join Query using SQL

Hints:

- You'll need to have tables for both `githubDF` and `githubTop20DF` .
- You'll need a subquery.
- The results should be the same, and the resources used roughly the same.
- You can `explain` this query also to examine any di!erences from the join version.

Summary

Joins are powerful tools, but distributed joins can be expensive in terms of amount of data shuffled. We've worked with them here, and seen how Catalyst can optimize by broadcasting a small data set.

Note that we've used a common strategy here - we've replaced a query that was resource intensive (our straight scan query), with a different query (querying on a specific group of logins that are provided). The new query is not exactly equivalent, but fills our needs, and is much less resource intensive.