Parallelism Continued
6.S079 Lecture 19

Sam Madden

4/13/2022

Lab 5 Due

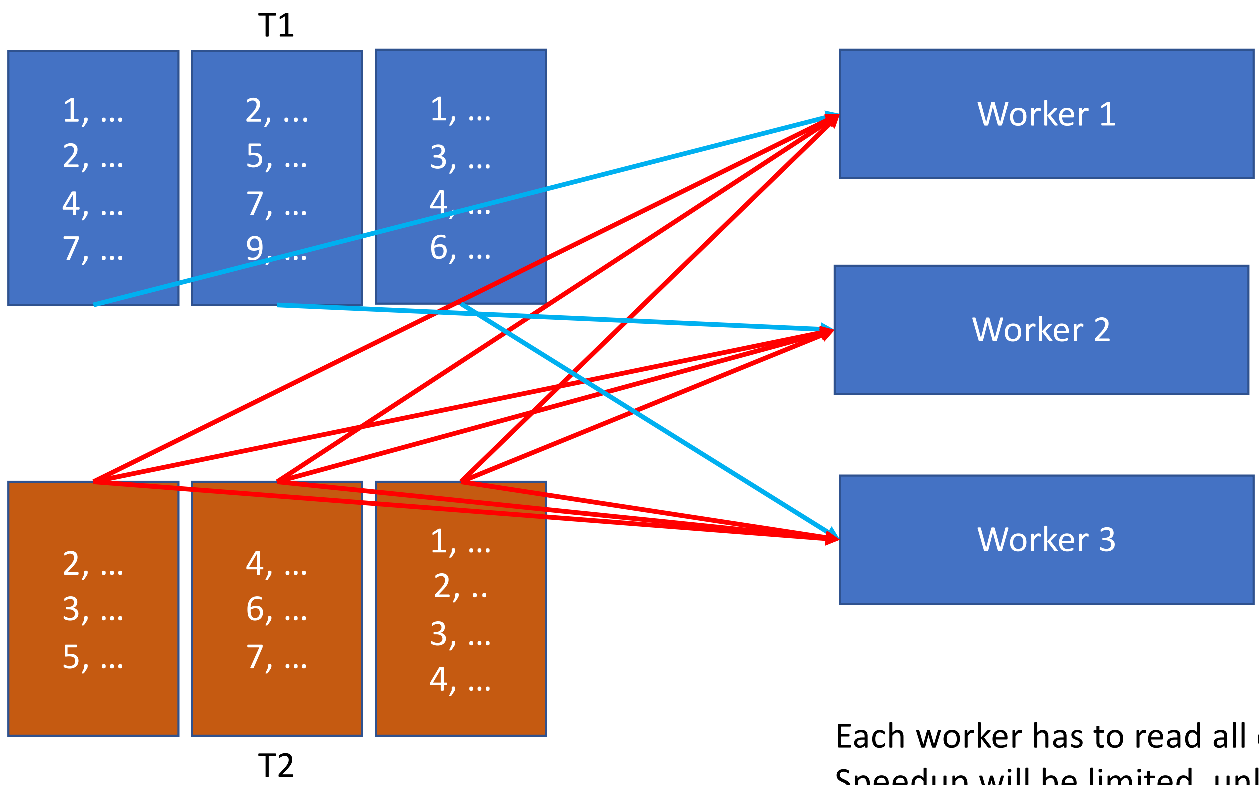
Topics:
Dask distributed
Spark
Pushdown & preaggregation
Hands-on Dask

Last Time

- Introduced Parallel Processing
- Look at Parallel Dataflow as a common set of operations that can be readily parallelized
- Studied parallel join and parallel aggregation
- Introduced Dask, a parallel implementation of Python pandas (and numpy and scikit learn)

Parallel Join – Random Partitioning Naïve Algo

(1, …) indicates value of join attribute



(2, …) ⋈ (2, …)

(4, …) ⋈ (4, …)
(7, …) ⋈ (7, …)

(1, …) ⋈ (1, …)
(2, …) ⋈ (2, …)
(4, …) ⋈ (4, …)

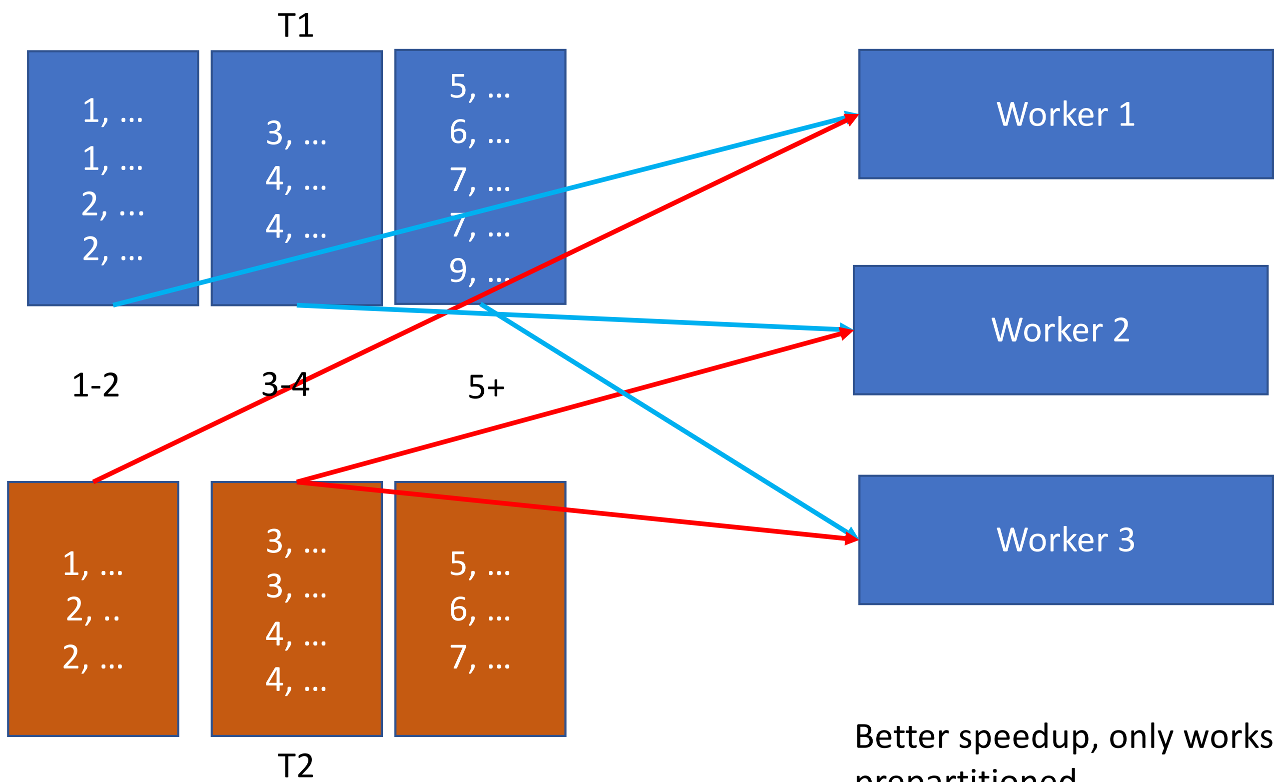
Must join each partition with every other partition

Each worker has to read all of T2
Speedup will be limited, unless T2 is much smaller than T1

Parallel Join – Prepartitioned

(1, …) indicates value of join attribute

Only need to join partitions that match



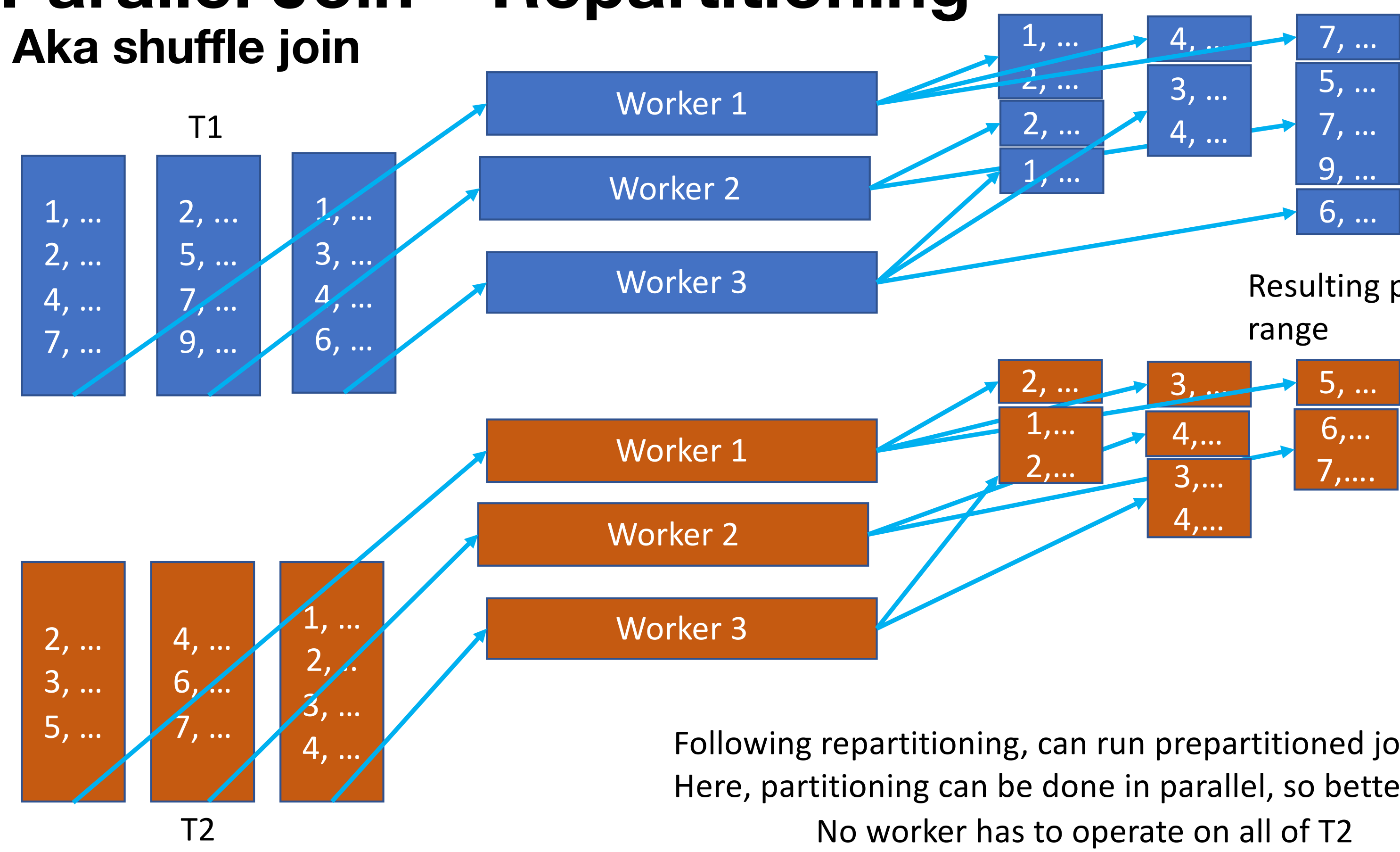
(1, …) ⋈ (1, …)
(1, …) ⋈ (1, …)
(2, …) ⋈ (2, …)
(2, …) ⋈ (2, …)
(2, …) ⋈ (2, …)
(2, …) ⋈ (2, …)
(2, …) ⋈ (2, …)

This is what our Postgres example showed

Better speedup, only works if data is properly prepartitioned
Should be 3x faster than single node join
Skew problem (hashing may help)

Parallel Join – Repartitioning

Aka shuffle join



Resulting partitions are divided by range

Following repartitioning, can run prepartitioned join

Here, partitioning can be done in parallel, so better than naïve

No worker has to operate on all of T2

Recap: Large Join In Dask

```
client = Client(n_workers=8, threads_per_worker=1, memory_limit='16GB')

header = "CMTE_ID,AMNDT_IND,RPT_TP,TRANSACTION_PGI,IMAGE_NUM,TRANSACTION_TP ...
PATH = "indiv20/by_date/itcont_2020_20010425_20190425.txt"
PATH2 = "indiv20/by_date/itcont_2020_20190426_20190628.txt"

df = dask.dataframe.read_csv(PATH, low_memory=False, delimiter='|', header=None ...
df2 = dask.dataframe.read_csv(PATH2, low_memory=False, delimiter='|', header=None ...
df = df.dropna(subset=['NAME']).drop_duplicates(subset=['NAME'])
df2 = df2.dropna(subset=['NAME']).drop_duplicates(subset=['NAME'])

# make 3 copies
df = df.append(df)
df = df.append(df)
df = df.append(df)

df2 = df2.append(df2)
df2 = df2.append(df2)
df2 = df2.append(df2)

ans = df.merge(df2, on='NAME').count()

ans = ans.compute()

print(f"found {ans} matches")
```

Execution is deferred until compute is called

Dask Distributed

"Distributed" = multiple machine
"Parallel" = multiple processors on same machine

- Demo on Amazon
  - Much slower than laptop, t3.large machines (8GB RAM, 2x vCPU ~30% performance / CPU)

- Single local executor: 174.3 s
- Single distributed worker: 200.5
- Three distributed workers: 78.5 s (2.2x/2.6 speedup)

Subgraph Caching via “Persist”

- Can “persist” a subresult to cause it to be stored in memory
- Avoids recomputing

```
n1 = df.loc[:,["NAME"]].persist()
n2 = df2.loc[:,["NAME"]].persist()

#will compute the count and persist n1 and n2
ans = n1.merge(n2, on='NAME').count()
print(ans.compute())

#will resuse previously peristed rsult
ans2 = n1.merge(n2, on='NAME').max()
print(ans2.compute())
```

Fault Tolerance Model

- Retries tasks that fail
- Resilient to the failure of any one worker

- Demo

Spark

- Distributed / parallel data processing system

- pyspark.sql engine very similar to dask in functionality
  - Slightly different API
  - Other pands-on-spark projects, e.g., koalas provide pandas API compatibility

Example

Demo!

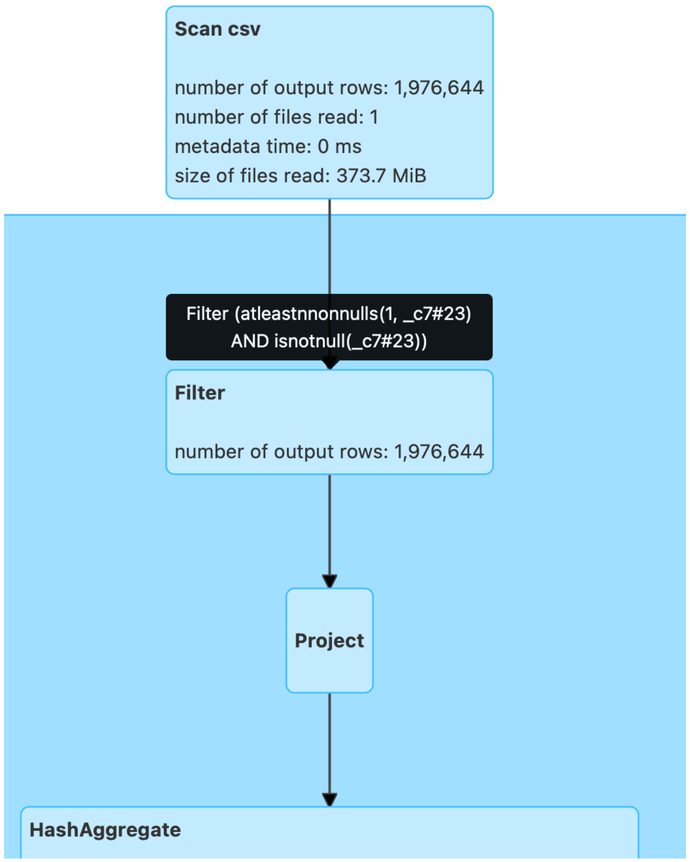
```
spark = SparkSession.builder.appName("SimpleApp").getOrCreate()

path = "indiv20/by_date/itcont_2020_20010425_20190425.txt"
path2 = "indiv20/by_date/itcont_2020_20190426_20190628.txt"
header = "CMTE_ID,AMNDT_IND,RPT_TP,TRANSACTION_PGI,IMAGE_NUM,TRANSACTION_TP, …

df_spark = spark.read.csv(path, sep ='|', header = False)
df_spark = df_spark.toDF(*header)
df_spark = df_spark.dropna(subset=["NAME"]).dropDuplicates(subset=["NAME"])
df_spark = df_spark.union(df_spark)
df_spark = df_spark.union(df_spark)
df_spark = df_spark.union(df_spark)

df_spark2 = spark.read.csv(path2, sep ='|', header = False)
df_spark2 = df_spark2.toDF(*header)
df_spark2 = df_spark2.dropna(subset=["NAME"]).dropDuplicates(subset=["NAME"])
df_spark2 = df_spark2.union(df_spark2)
df_spark2 = df_spark2.union(df_spark2)
df_spark2 = df_spark2.union(df_spark2)

ans = df_spark.join(df_spark2, on='NAME').count()
print(ans)
```

This is a way to run spark locally; most people run a cluster of machines and submit jobs, like the dask distributed demo before

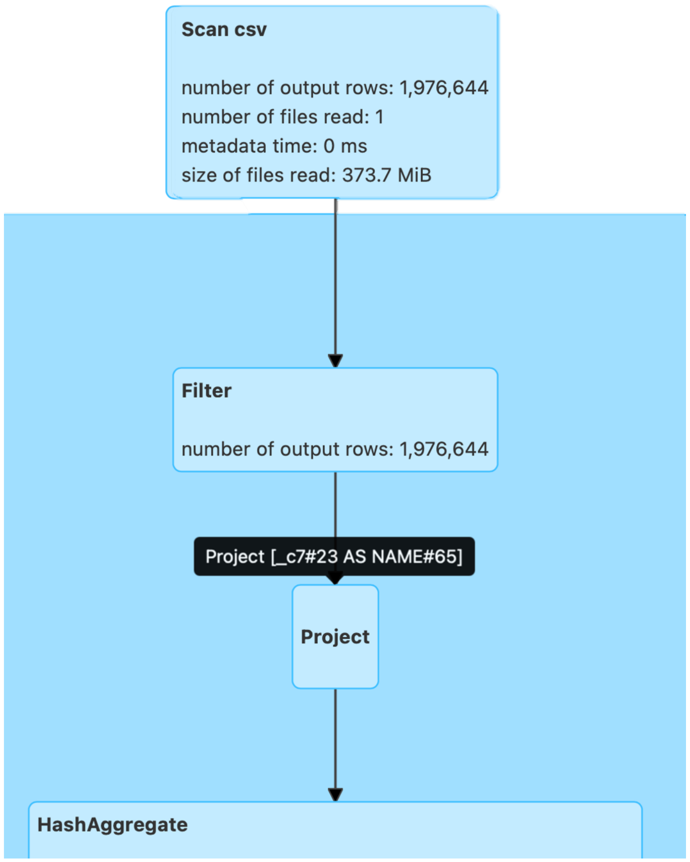
Spark Under the Hood

- Compiles to Java/Scala
  - Makes understand what tasks are doing and debugging messages somewhat confusing
- Query optimizer much smarter than Dask
  - Projection push down
  - Pre-aggregation

Projection Push Down

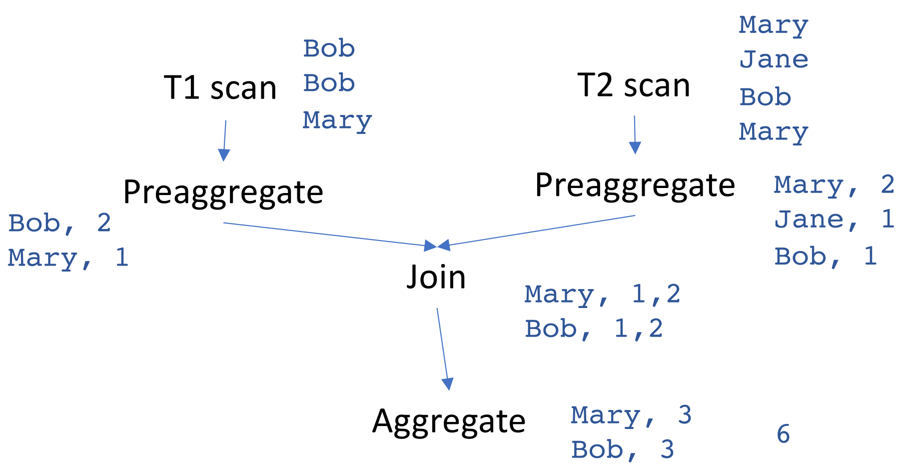


Projection Push Down



Preaggregation

- Goal: count the number satisfying records in the join
- Idea: count records in each table before the join
- Join {record, count} pairs from tables to compute final join
- Eliminates the number of records that need to join



In spark, preaggregate, join and aggregate can all be done massively in parallel

Spark vs Dask

- Dask is much smaller, more pythonic
- Spark generally performs better
  - More optimized for very large datasets on S3 / cloud storage
  - Dask lacks query optimization
- Spark is harder to use and debug
  - Compilation down to Java makes it hard to understand what is happening, sometimes

- Many other packages in spark, including
  - SparkML
  - Spark Streaming
  - A variety of data lake / storage tools

Summary

- Dask and Spark both support parallel and distributed computation over data
  - Both scale from a few processors to hundreds of machines
- Dask is good for parallelizing pandas/numpy code
- Spark more sophisticated, less tied to python ecosystem