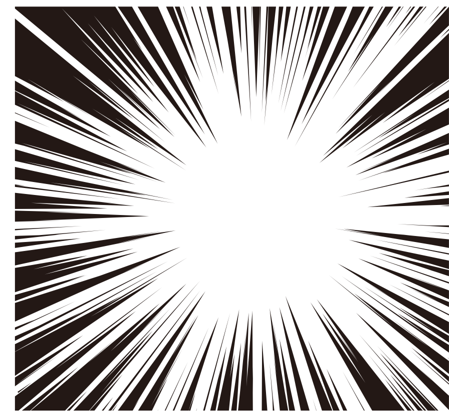
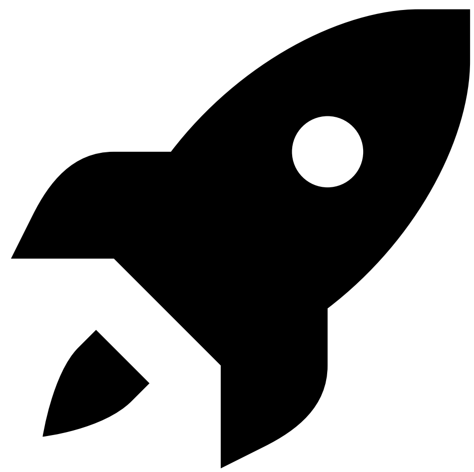
Programming with Data Bootcamp: Lecture 8

http://dsg.csail.mit.edu/6.S079/

Slides courtesy of Sam Madden / Tim Kraska (6.S079)



Key ideas:
Single-node Parallelism
Multi-node Parallelism
- Dask
- Spark
- Ray



Parallelism Goal

- Make a job faster by running on multiple processors
- What do we mean by faster?

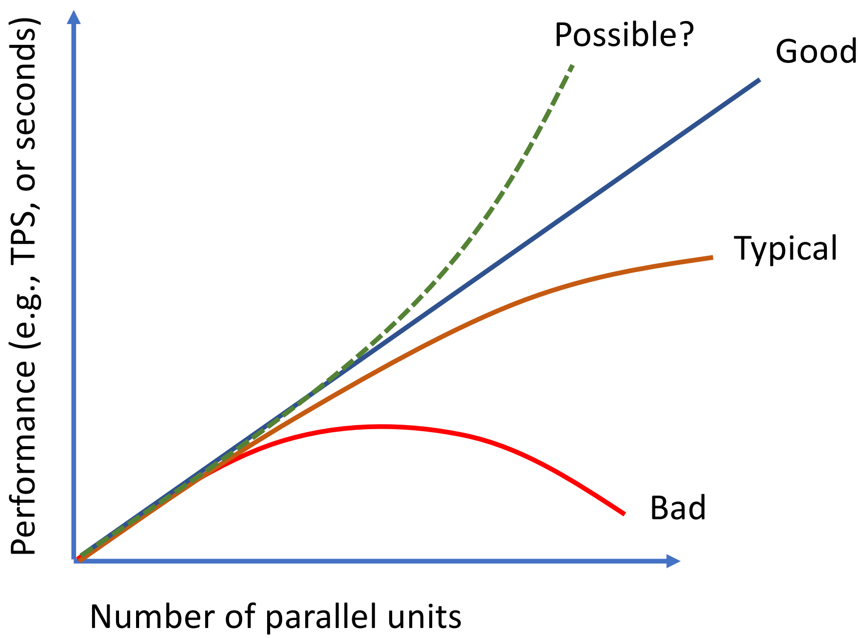
$$speed\ up = \frac{old\ time}{new\ time}\ \text{on same problem, with N times more hardware}$$

$$scale\ up = \frac{1x\ larger\ problem\ on\ 1x\ hardware}{Nx\ larger\ probelm\ on\ Nx\ hardware}$$

- Not necessarily the same: smaller problem may be harder to parallelize

Speedup Goal

- Linear?



Barriers to Linear Scaling

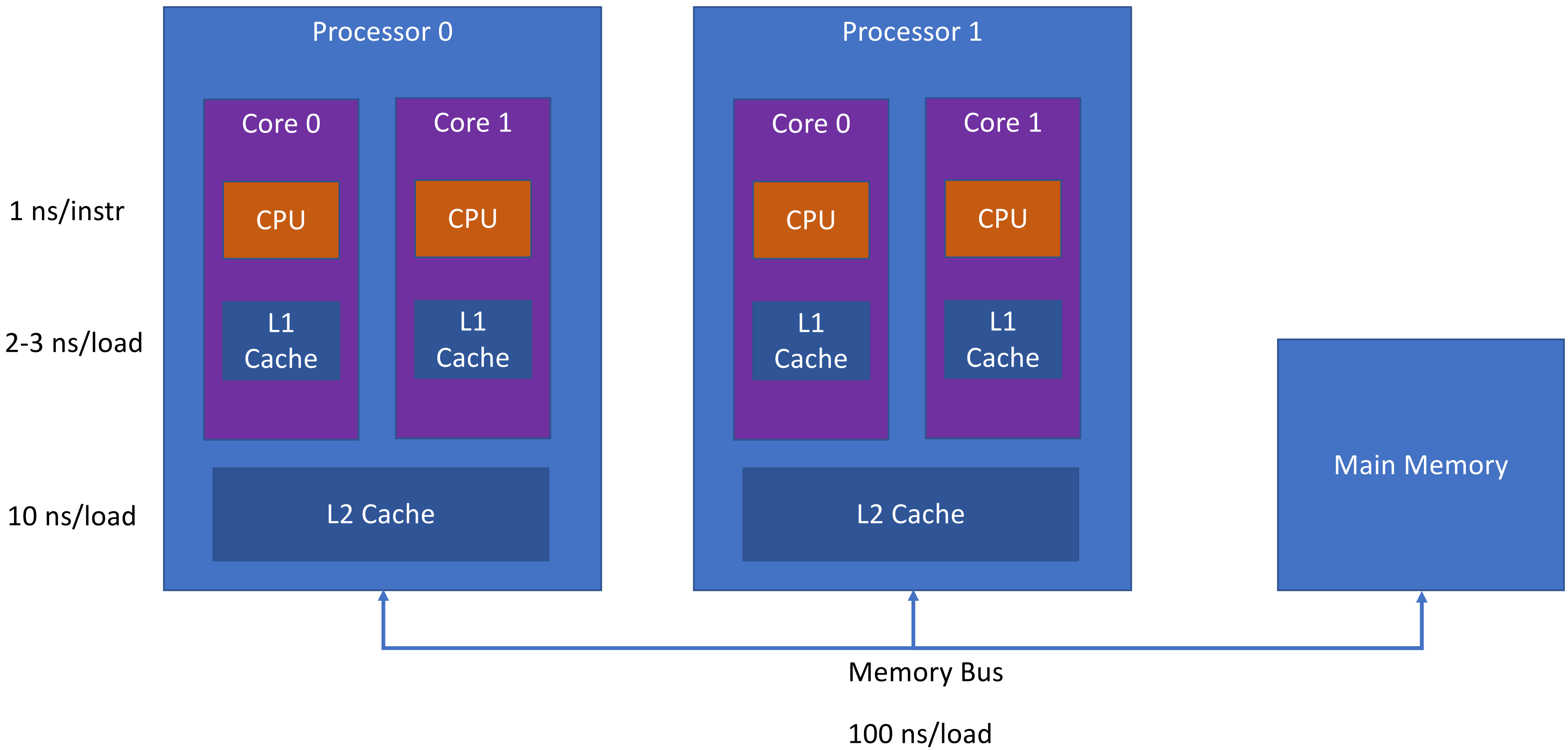
- Startup times
  - e.g., may take time to launch each parallel executor
- Interference
  - processors depend on some shared resource
  - E.g., input or output queue, or other data item
- Skew
  - workload not of equal size on each processor

- *Almost all workloads will stop scaling at some point!*

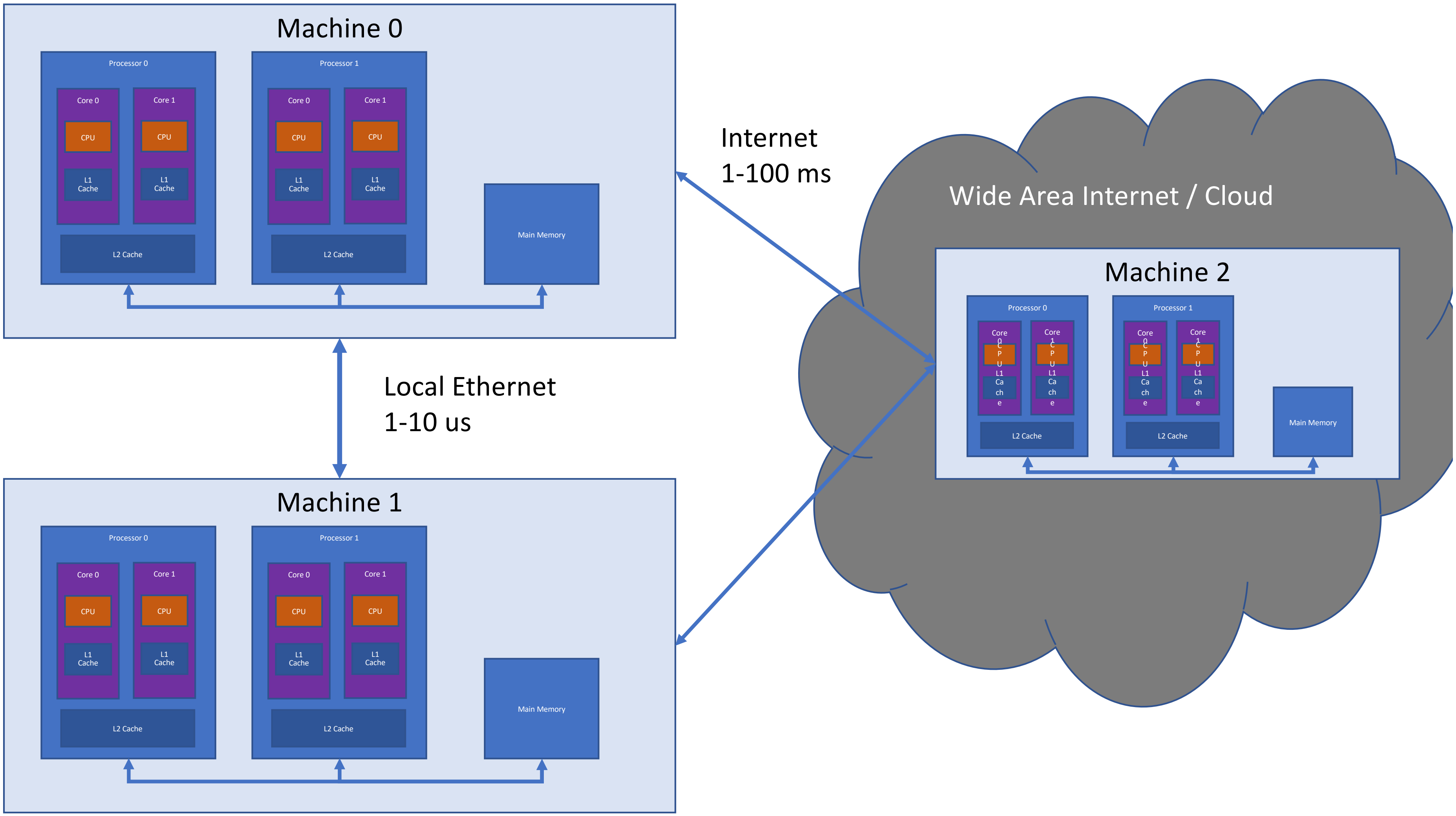
- What are some barriers in data science workloads?

Properties of Parallelizable Workloads

- Provide linear speedup
- Usually can be decomposed into small units that can be executed independently
  - "embarrassingly parallel"
- As we will see, SQL-style operations generally provide this
- Some ML algorithms support it, but often tricky

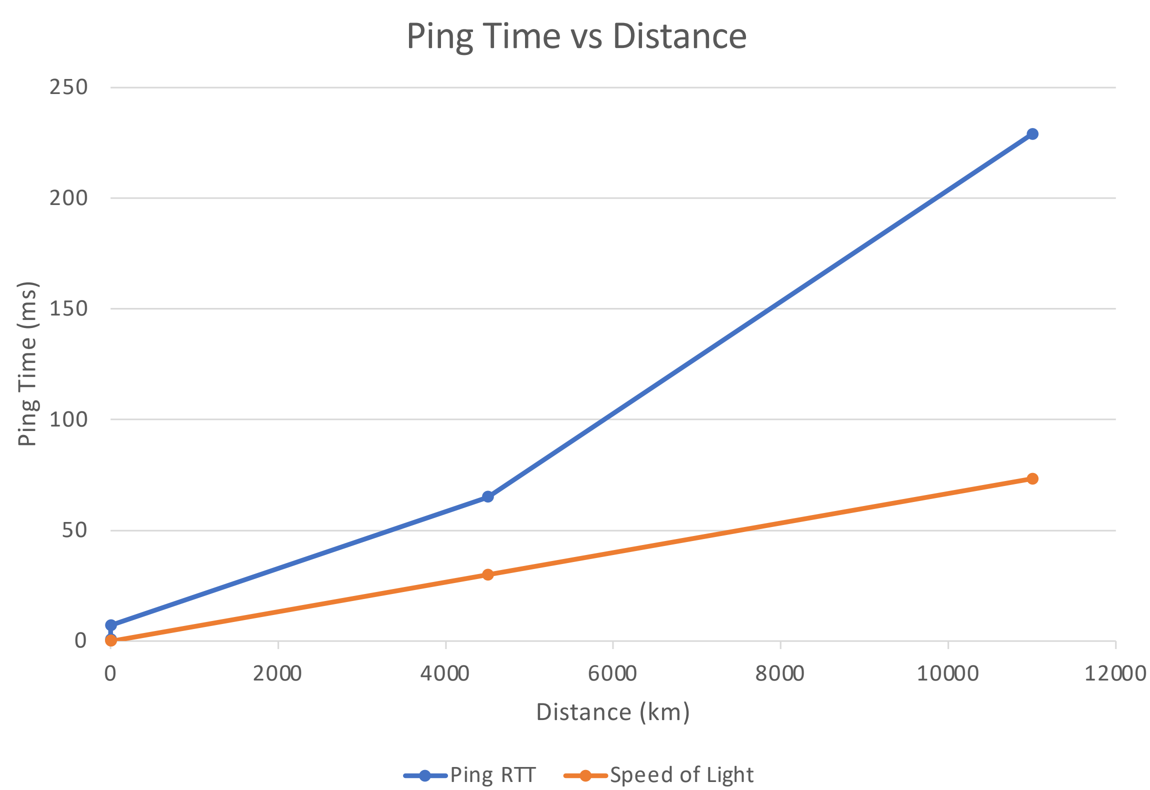


Some machines may have 2 levels of cache per core

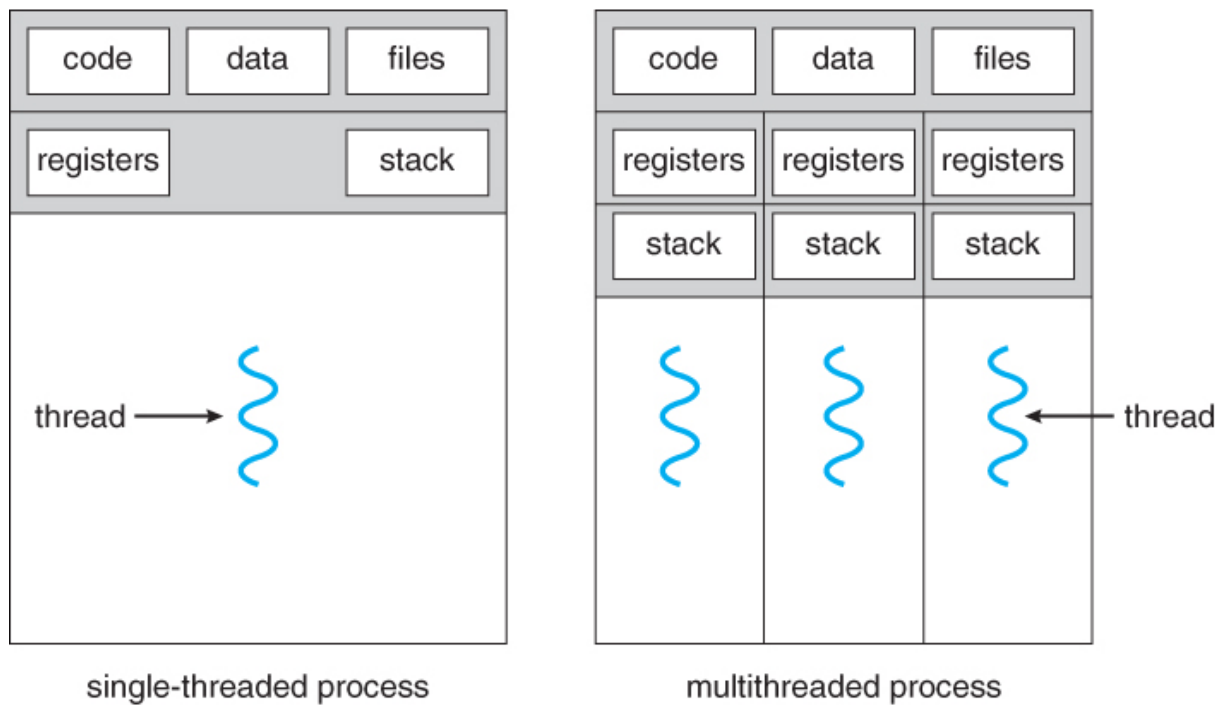

Machine 0
Processor 0
Core 0
CPU
L1 Cache
Core 1
CPU
L1 Cache
L2 Cache
Processor 1
Core 0
CPU
L1 Cache
Core 1
CPU
L1 Cache
L2 Cache
Main Memory
Internet
1-100 ms
Local Ethernet
1-10 us
Wide Area Internet / Cloud
Machine 2
Processor 0
L2 Cache
Processor 1
L2 Cache
Main Memory
Machine 1
Processor 0
Core 0
CPU
L1 Cache
Core 1
CPU
L1 Cache
L2 Cache
Processor 1
Core 0
CPU
L1 Cache
Core 1
CPU
L1 Cache
L2 Cache
Main Memory

Ping Test (Ethernet inside CSAIL)

- csail.mit.edu
  - 0.7 ms
- mit.edu
  - 14.0 ms
- harvard.edu
  - 7.0 ms
- berkeley.edu
  - 65.1 ms
- tsinghua.edu
  - 229.5 ms



Threads vs Processes



https://www.cs.uic.edu/~jbell/CourseNotes/OperatingSystems/4_Threads.html

Python Threads API

```
import threading

t = threading.Thread(target=func_name, args=(a1,a2,…))
t.start()   #start thread running – main thread continues
t.join()   #wait for thread to finish

lock = threading.Lock()   #create a lock object
lock.acquire() #acquire the lock; block if another thread has it
lock.release()  #release the lock
```

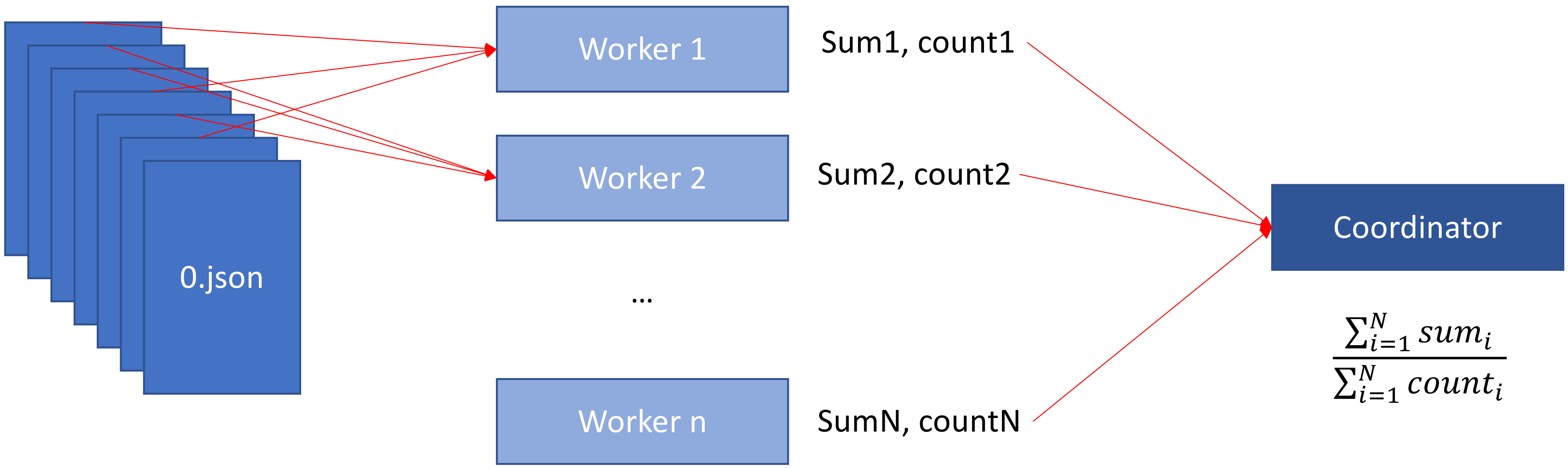
Problem: Python Global Interpreter Lock (GIL)
Only one thread can be executing python code at once

Python Multiprocessing API

```
import multiprocessing

p = multiprocessing.Process(target=func_name, args=(a1,a2,…))
p.start()   #start thread running – main thread continues
p.join()   #wait for thread to finish

lock = multiprocessing.Lock()   #create a lock object
lock.acquire() #acquire the lock; block if another thread has it
lock.release()  #release the lock
```

Parallel Aggregation

Task: compute average age across all people



{"age": 30, "name": ["Michal", "Sharpe"], "occupation": "Archivist", "telephone": "285.290.9033", "address": {"address": "458 Girard Plantation", "city": "Wentzville"}, "credit-card": {"number": "5384 0033 6904 0042", "expiration-date": "06/23"}}

Parallel Aggregation Implementation

- Use multiprocessing, not threading
- Main thread creates a work queue

```
q = multiprocessing.Queue()
```

- Puts work on it, as pointers to files

```
q.put(file1); q.put(file2)
```

- Starts threads, passing them the work queue, as well as a result queue
- Threads pull from queue in a loop:

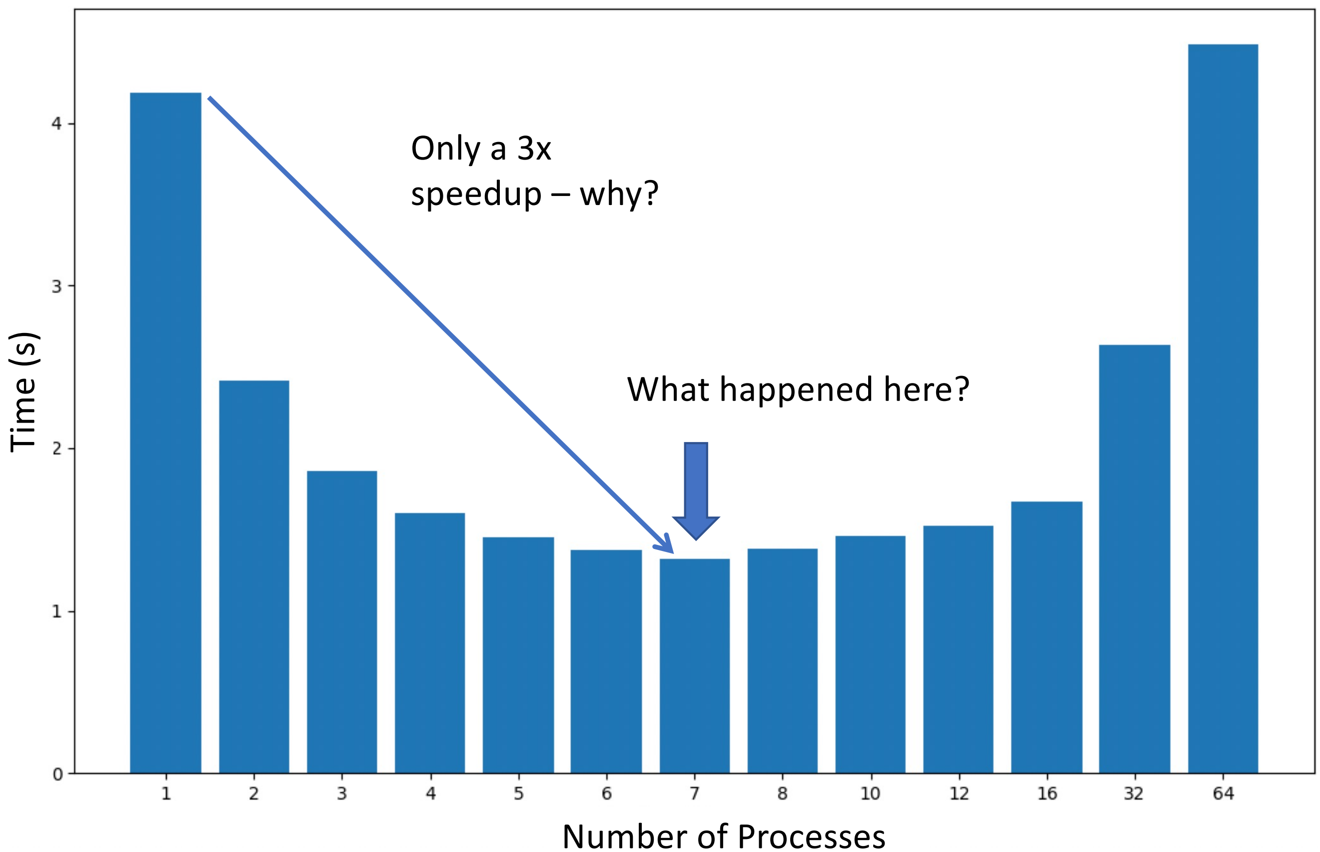
```
while True:
      f = q.get(block=False)
      process(f)
```

- Threads compute running sum and average
- Once complete, threads put their running sum and average on the result queue:

```
out_q.put((age_sum, age_cnt))
```

- Main thread blocks on result queue to read a result from each worker:

```
for p in procs:
      (p_sum,p_count) = out_q.get()
```


Only a 3x
speedup – why?
What happened here?
Time (s)
0
1
2
3
4
1
2
3
4
5
6
7
8
10
12
16
32
64
Number of Processes

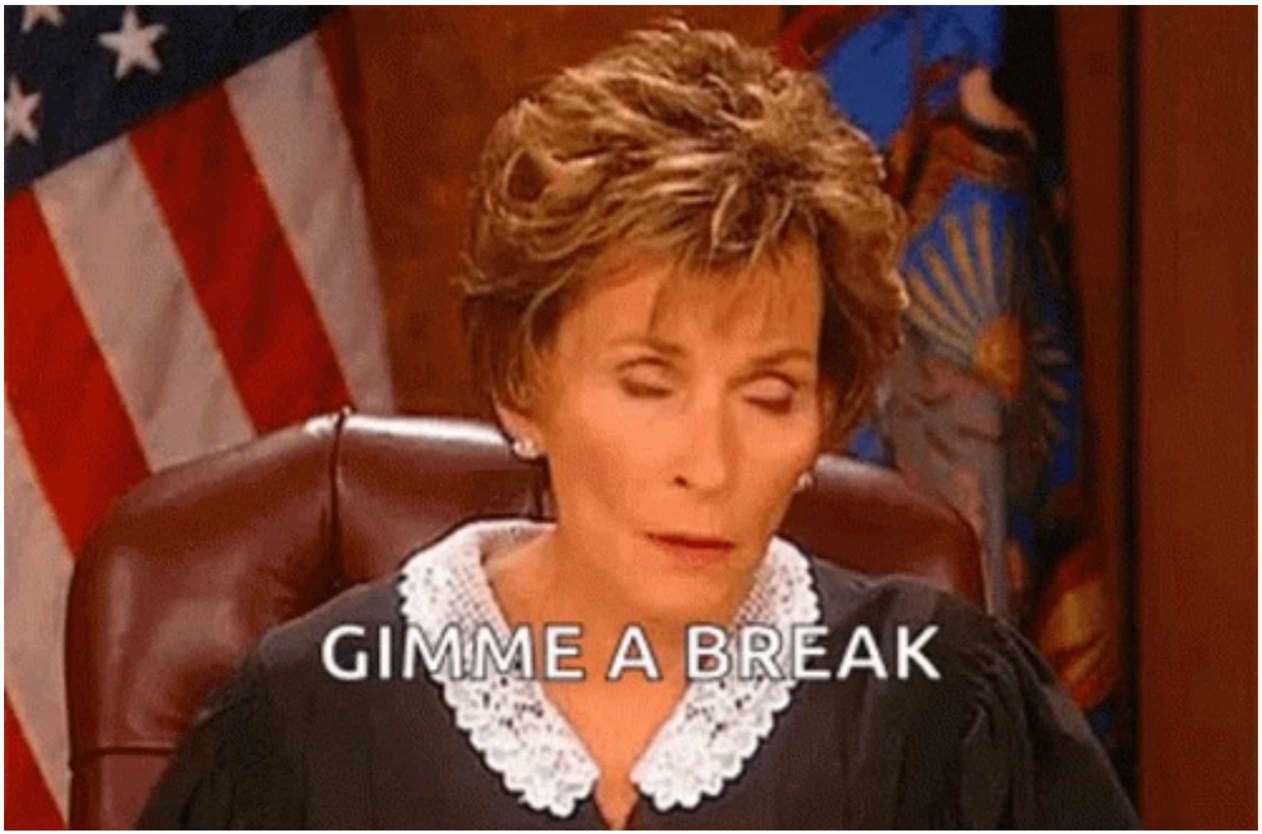
Clicker

Why didn't this program speed up beyond 8 processes? Choose all that apply

a) Not enough memory
b) Not enough processors
c) Startup overheads of launching processes
d) Too much coordination between processes

https://clicker.mit.edu/6.S079

Break



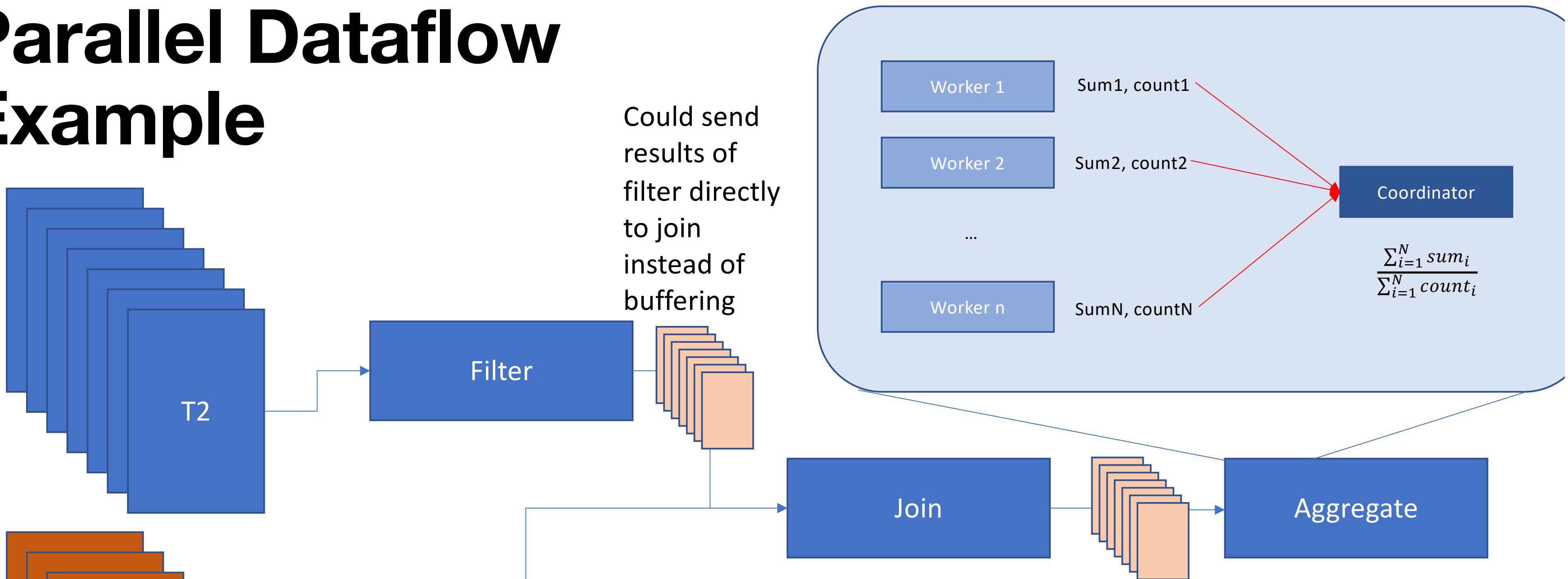
Parallelism Approach

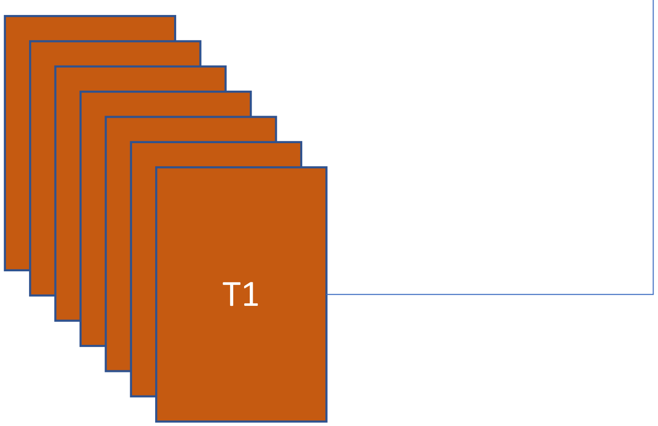
Split given data set split into N partitions

Use M processors to process this data in parallel

We will need to come up with parallel implementations of common operators

Parallel Dataflow Example





- Directed Acyclic Graph of Operators
    - Data flows from files to output
- Internally each operator is a parallel job
- Intermediate results between jobs typically buffered in mem or on disk between tasks
    - May be possible to pipeline directly

Parallel Dataflow Operations

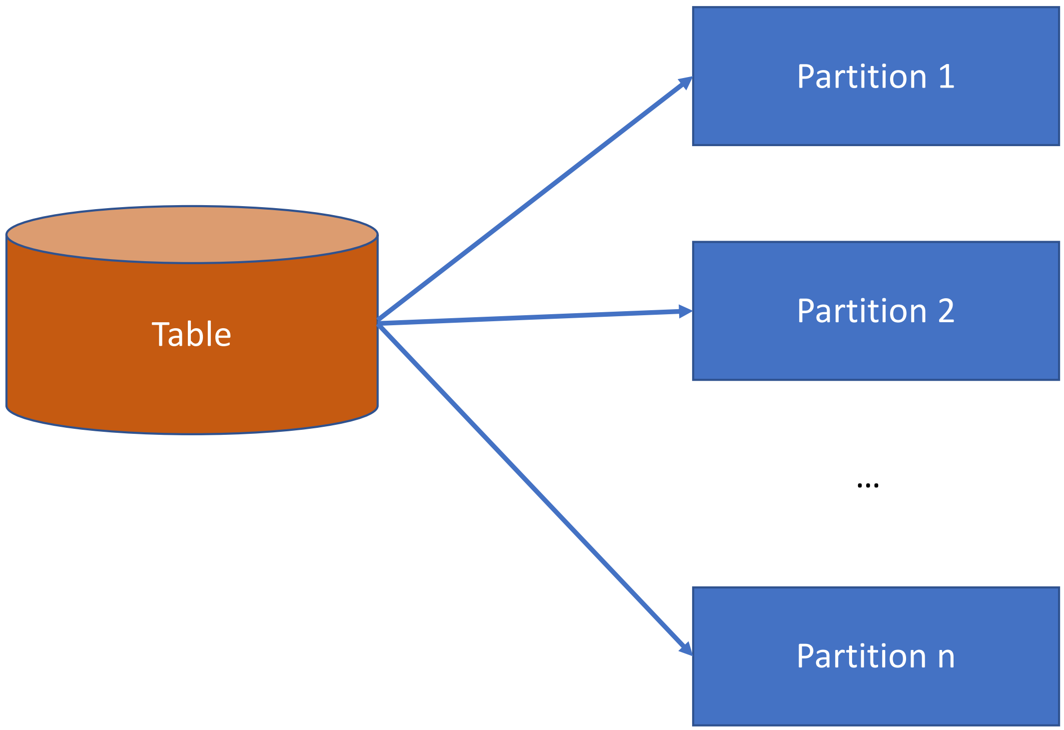
- Filter
- Project
- Element-wise or row-wise transform
- Join
  - Repartition vs broadcast
- Aggregate
- Sort
- Train an ML model
- Arbitrary python "UDF"s

Which of these are easy to parallelize?

Partitioning Strategies

- Random / Round Robin
  - Evenly distributes data (no skew)
  - Requires us to repartition for joins
- Range partitioning
  - Allows us to perform joins/merges without repartitioning, when tables are partitioned on join attributes
  - Subject to skew
- Hash partitioning
  - Allows us to perform joins/merges without repartitioning, when tables are partitioned on join attributes
  - Only subject to skew when there are many duplicate values

Round Robin Partitioning



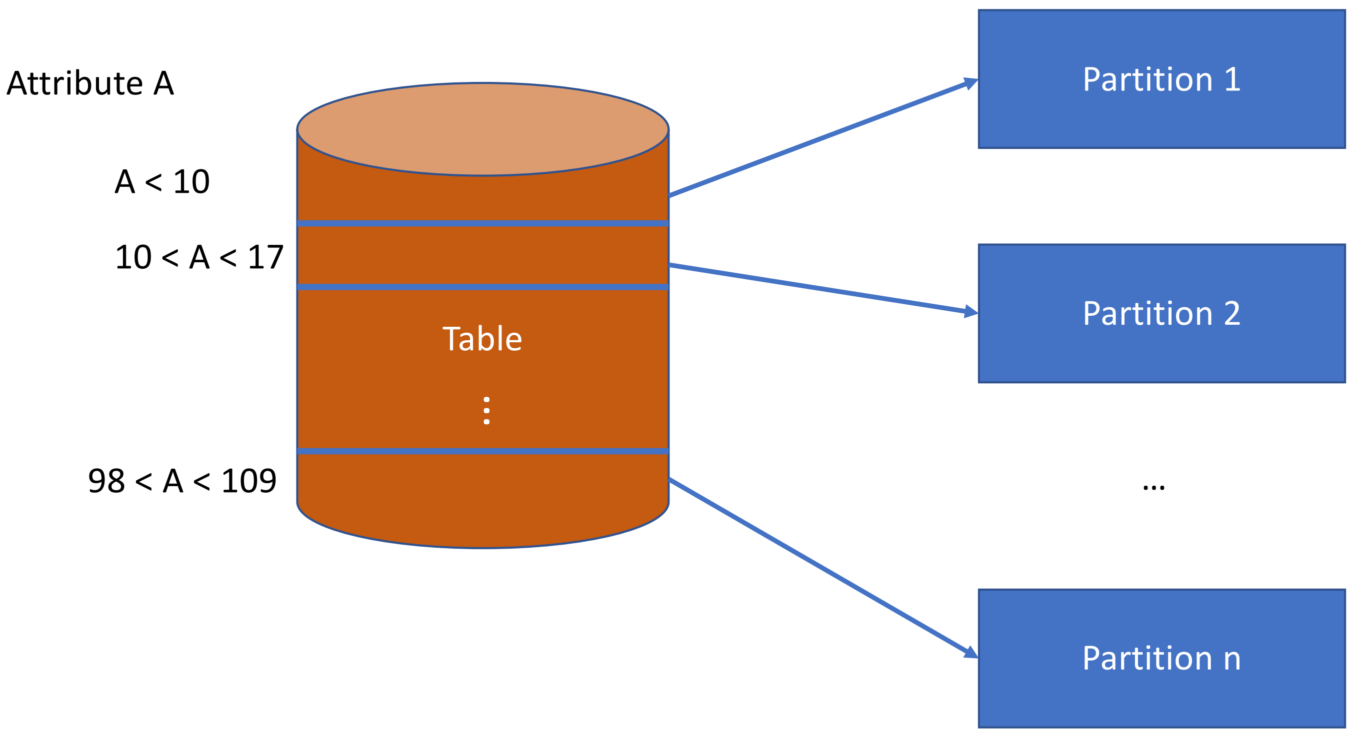
Advantages:

Each partition has the same number of records

Disadvantage:

No ability to push down predicates to filter out some partitions

Range Partitioning



Advantages:

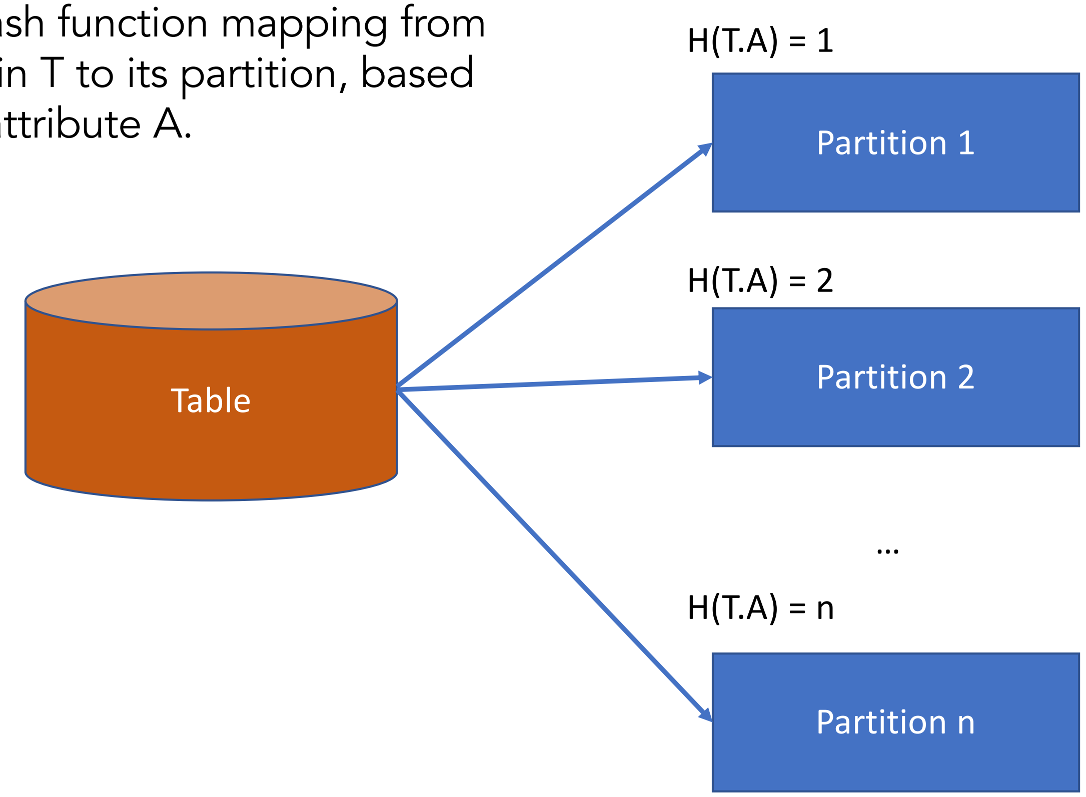
Easy to push down predicates (on partitioning attribute)

Disadvantage:

Difficult to ensure equal sized partitions, particularly in the face of inserts and skewed data

Hash Partitioning

H(T.A) is a hash function mapping from each record in T to its partition, based on value of attribute A.



<u>Advantages:</u>

Each partition has about the same number of records, unless one value is very frequent

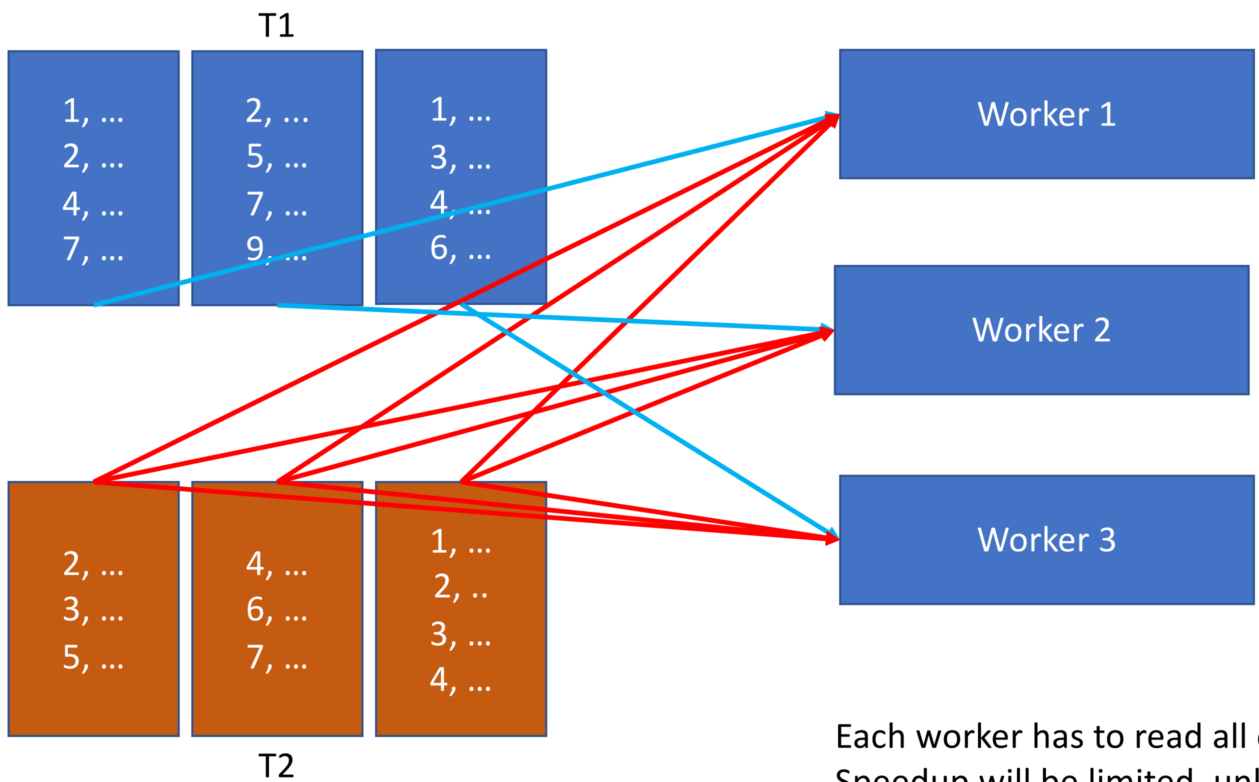
Possible to push down equality predicates on partitioning attribute

<u>Disadvantages:</u>

Can't push down range predicates

Parallel Join – Random Partitioning Naïve Algo

(1, …) indicates value of join attribute



(2, …) ⋈ (2, …)

(4, …) ⋈ (4, …)
(7, …) ⋈ (7, …)

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

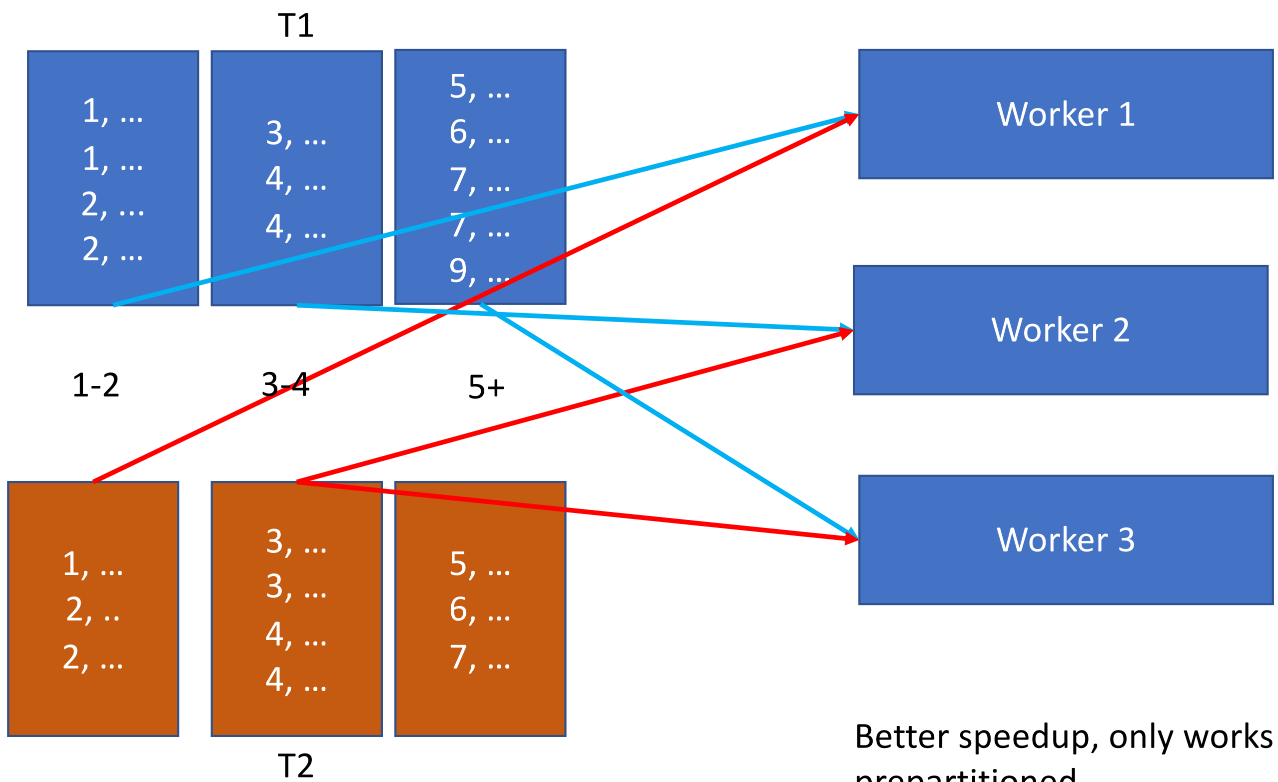
Must join each partition with every other partition

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



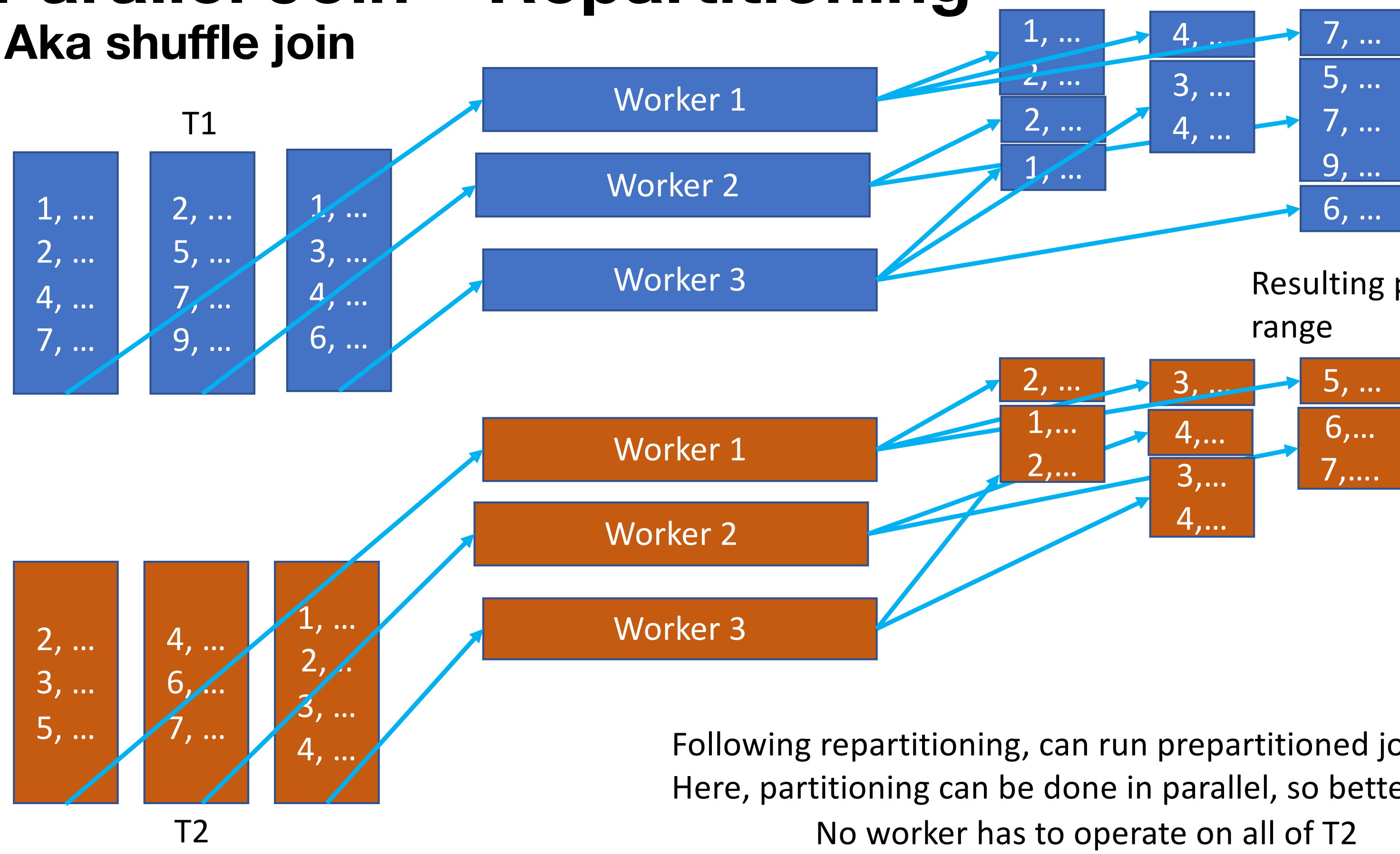
(1, …) ⋈ (1, …)
(1, …) ⋈ (1, …)
(2, …) ⋈ (2, …)
(2, …) ⋈ (2, …)
(2, …) ⋈ (2, …)
(2, …) ⋈ (2, …)
(2, …) ⋈ (2, …)

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

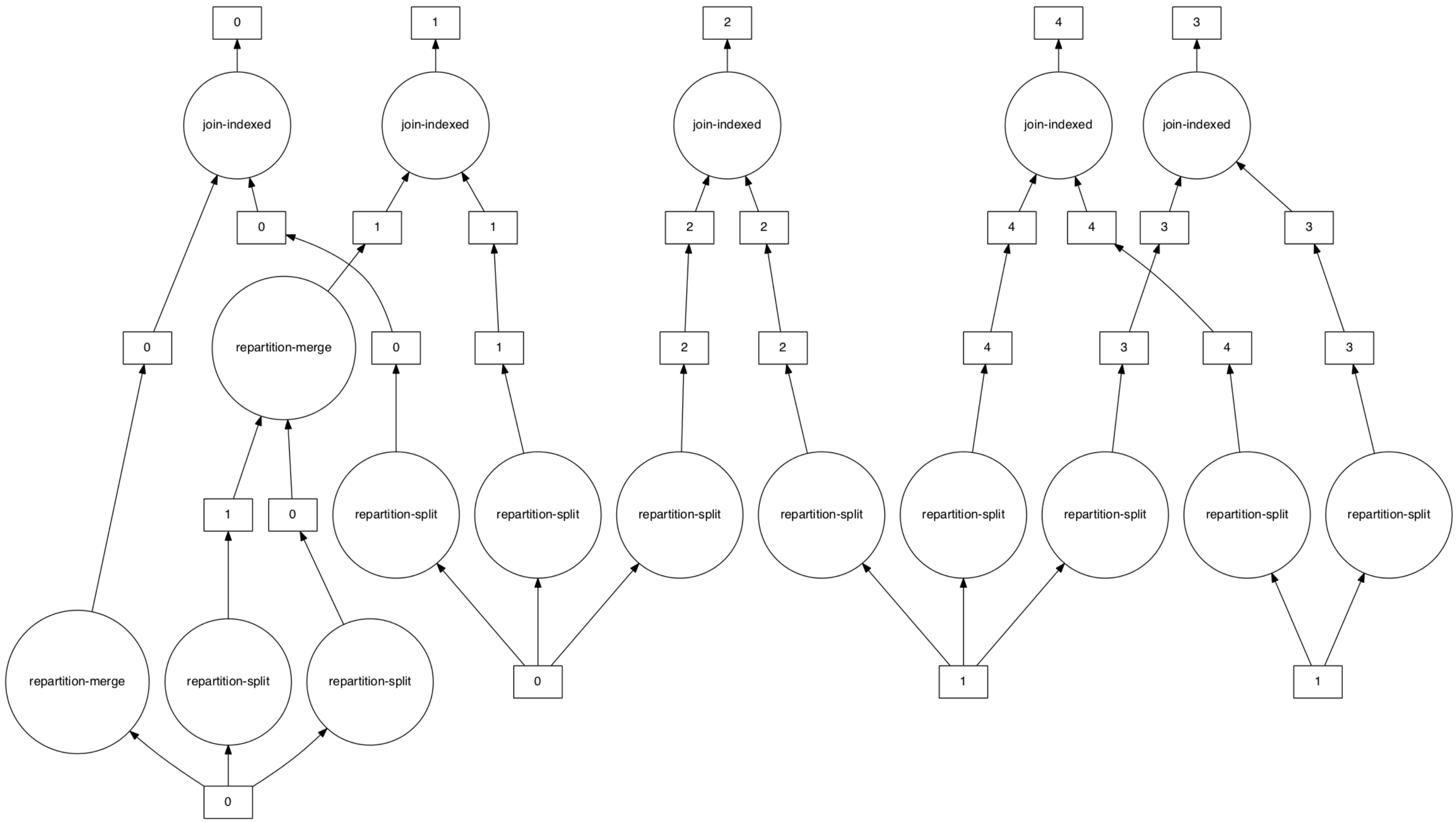
Dask https://dask.org

- General purpose python parallel / distributed computation framework
- Includes parallel implementation of Pandas dataframes
- Usually straightforward to translate a pandas program into a parallel implementation
  - Just use dask.dataframe instead of pandas.dataframe
  - Have to specify a parallel configuration to run on, via Client() object
    - Can be a local machine or distributed cluster
- Also has support for other types of parallelism, e.g., dask.bag class that allows parallel operation on collections of python objects

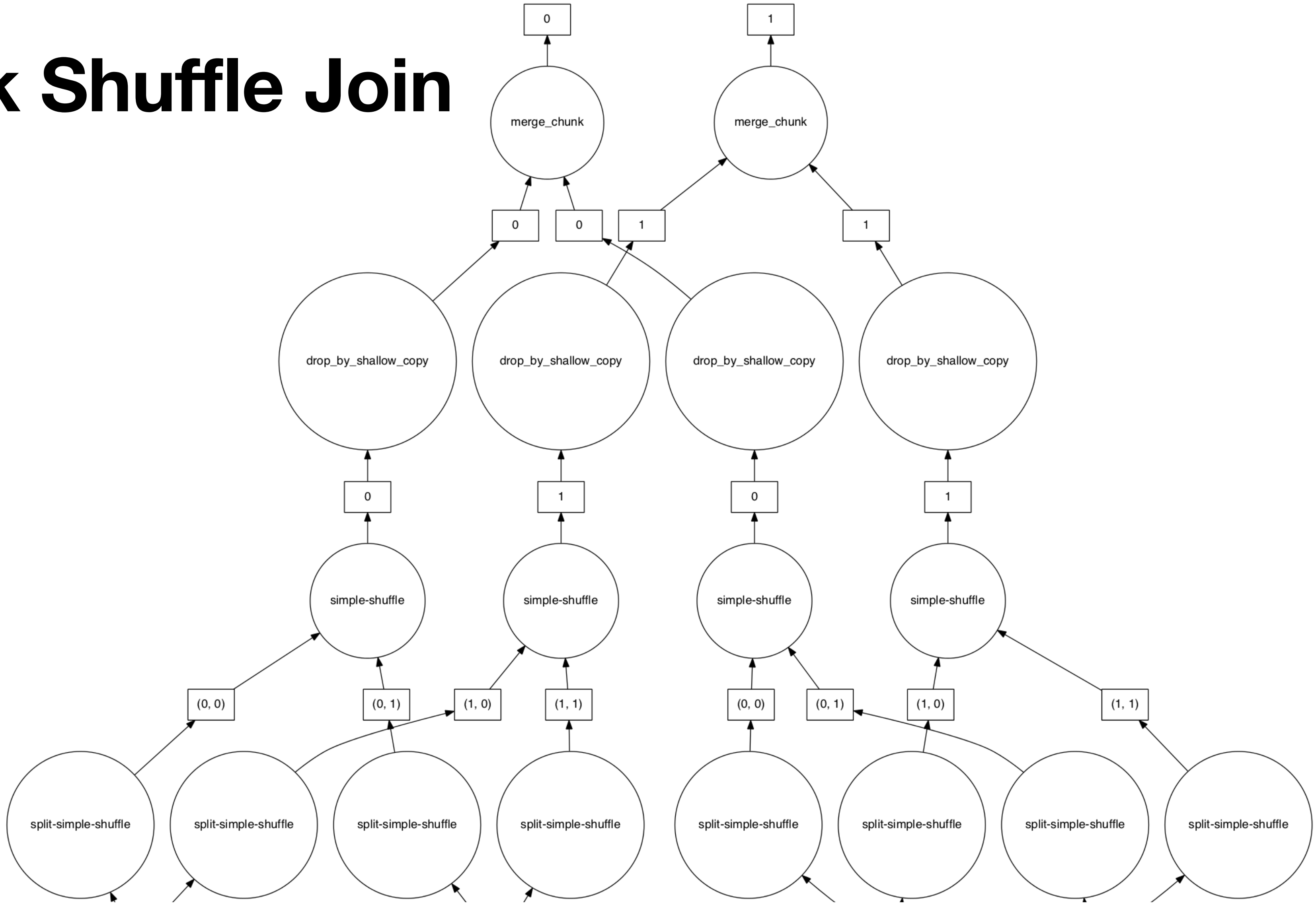
Large Join Demo

- Changing number of nodes
- Changing join algorithm

Dask Partitioned Join



Dask Shuffle Join



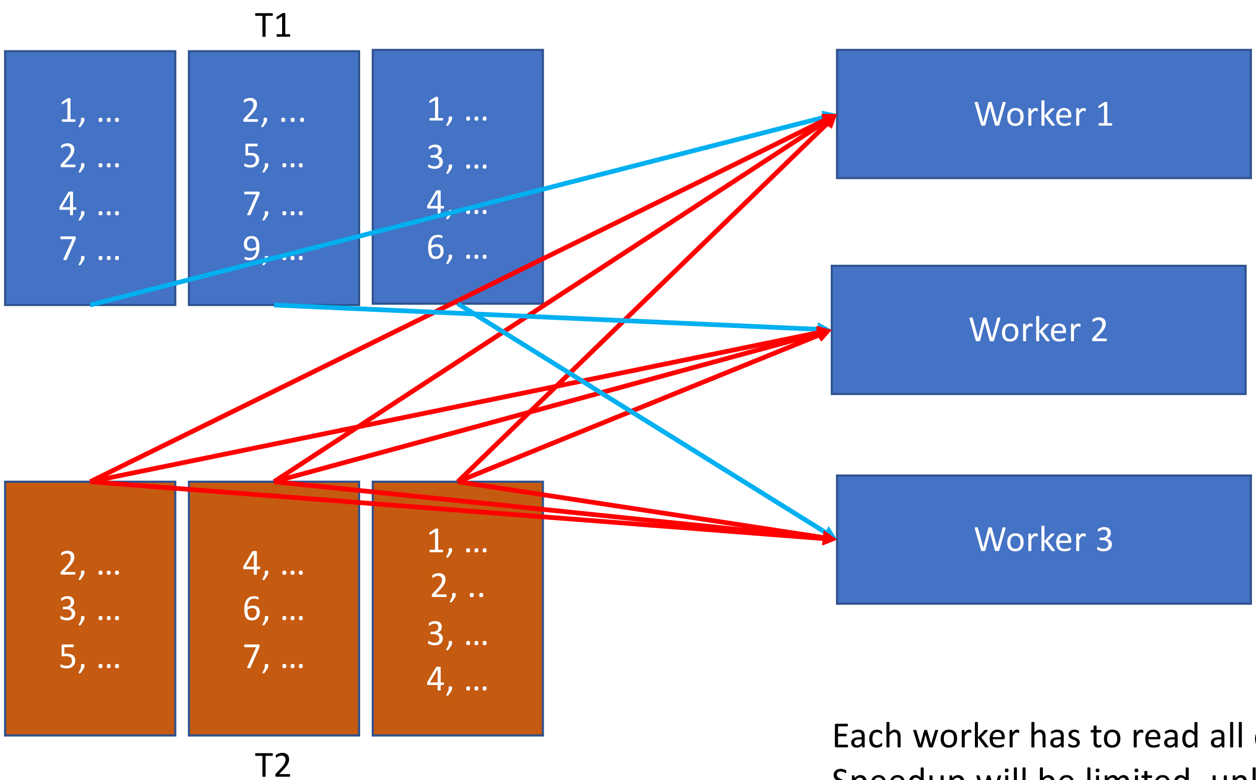
Many alternatives

- MapReduce / Hadoop
  - Rewrite you program as collection of parallel map() and reduce() jobs
  - Hard to do, slow()

- Spark
  - Popular library -- similar to dask, more focused on large scale distributed
  - Includes parallel implementations of ML and other operations
  - Difficult to use

Parallel Join – Random Partitioning Naïve Algo

(1, …) indicates value of join attribute



(2, …) $\bowtie$ (2, …)

(4, …) $\bowtie$ (4, …)
(7, …) $\bowtie$ (7, …)

(1, …) $\bowtie$ (1, …)
(2, …) $\bowtie$ (2, …)
(4, …) $\bowtie$ (4, …)

Must join each partition with every other partition

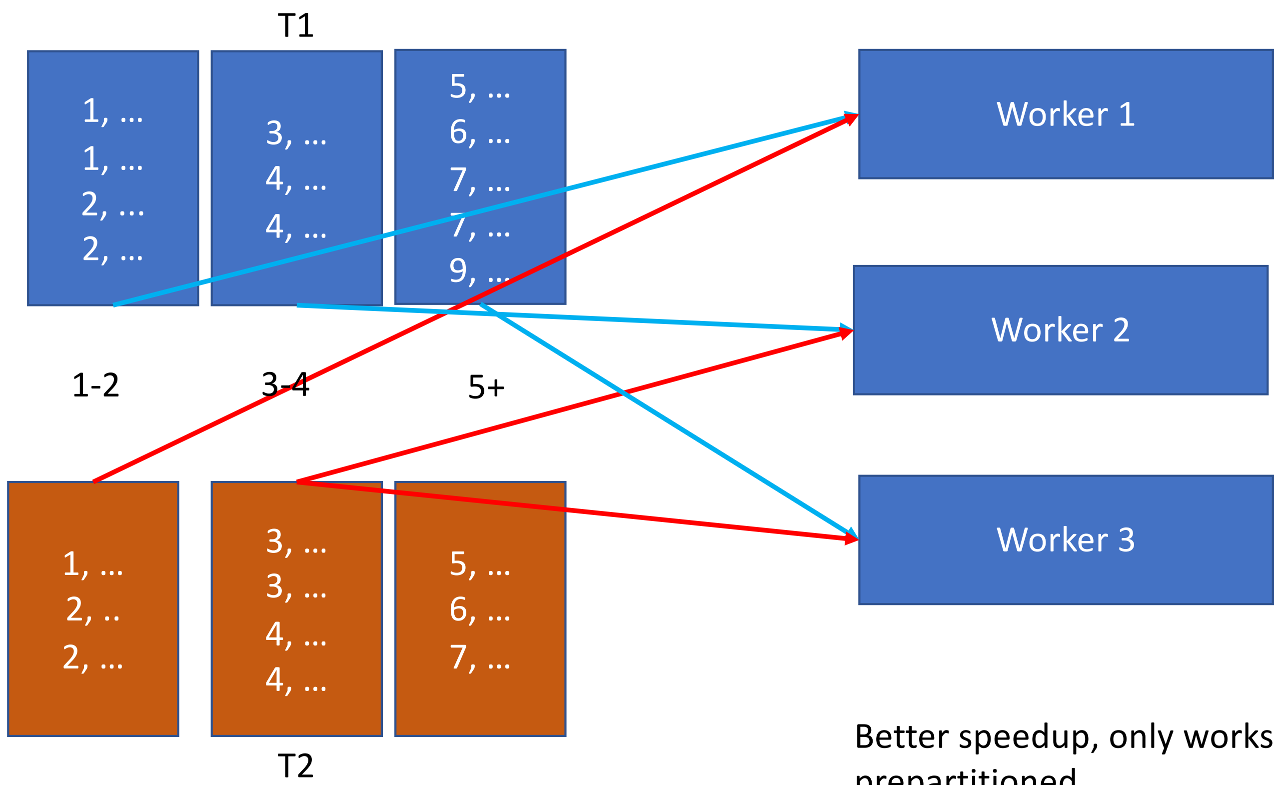
Each worker has to read all of T2
Speedup will be limited, unless T2 is much smaller than T1

Parallel Join – Prepartitioned

(1, …) indicates value of join attribute

Only need to join partitions that match

(1, …) $\bowtie$ (1, …)
(1, …) $\bowtie$ (1, …)
(2, …) $\bowtie$ (2, …)
(2, …) $\bowtie$ (2, …)
(2, …) $\bowtie$ (2, …)
(2, …) $\bowtie$ (2, …)
(2, …) $\bowtie$ (2, …)

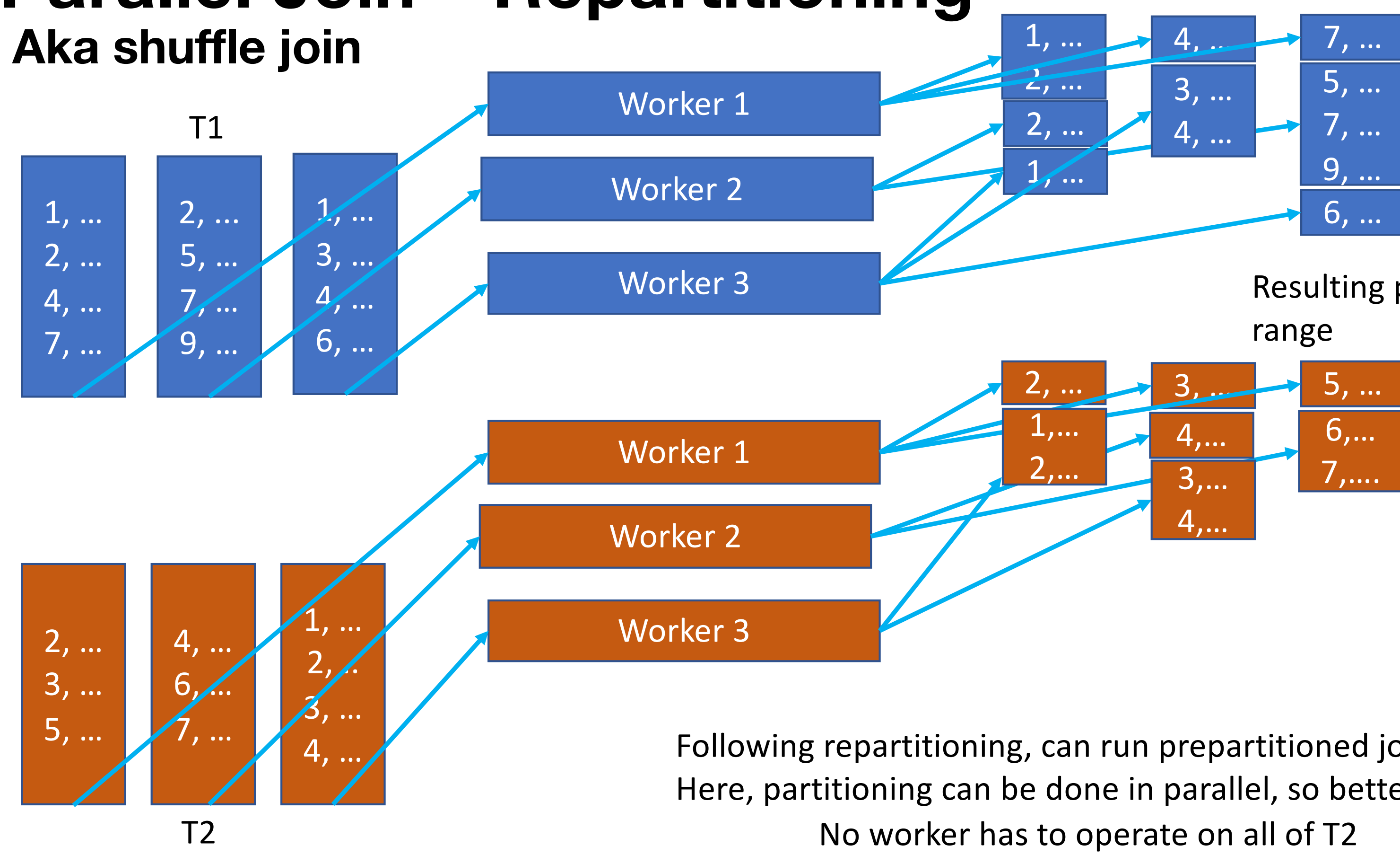


This is what our Postgres example showed

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()
```

Execution is deferred until compute is called

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

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!

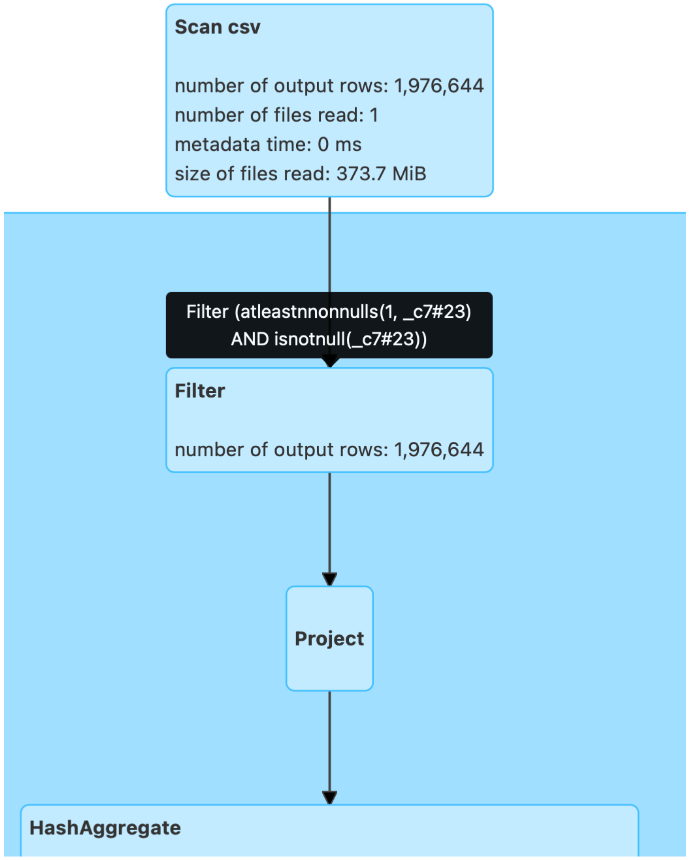
```
spark = SparkSession.builder.appName("SimpleApp").getOrCreate()

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

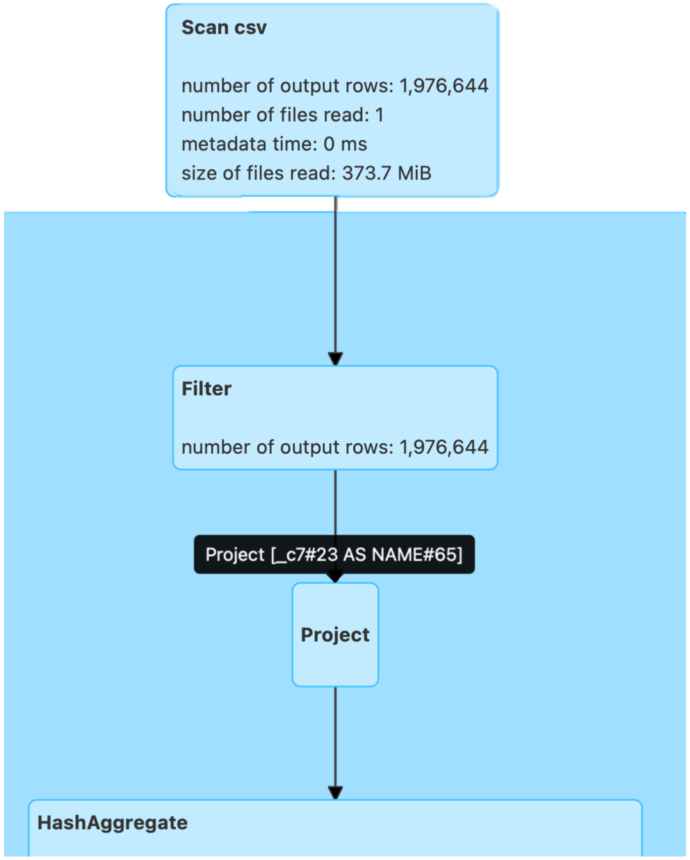
Spark Under the Hood

- 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

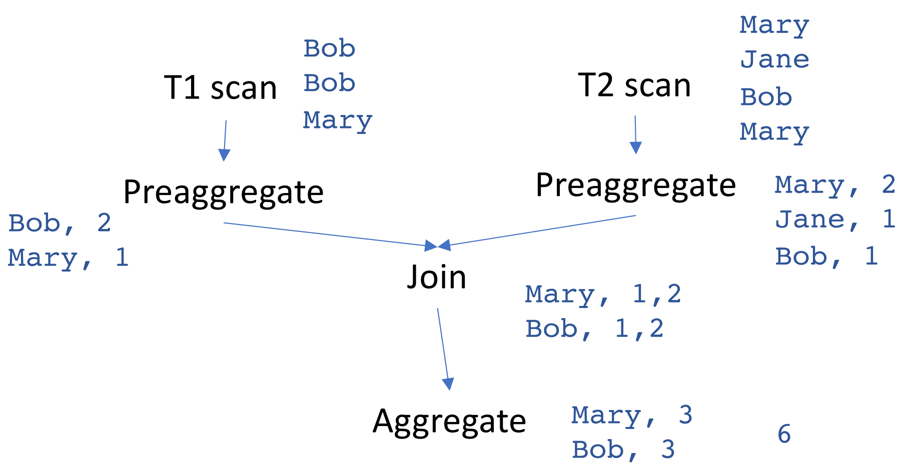


Projection Push Down



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