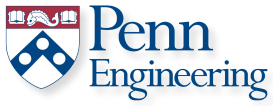
Improving Algorithmic Efficiency

Susan B. Davidson and Zachary G. Ives

University of Pennsylvania

CIS 545 – Big Data Analytics



Portions of this lecture have been contributed to the OpenDS4All project, piloted by Penn, IBM, and the Linux Foundation

https://tinyurl.com/cis545-lecture-02-14-22

Two Key Ideas

- To this point, we've looked at reducing data

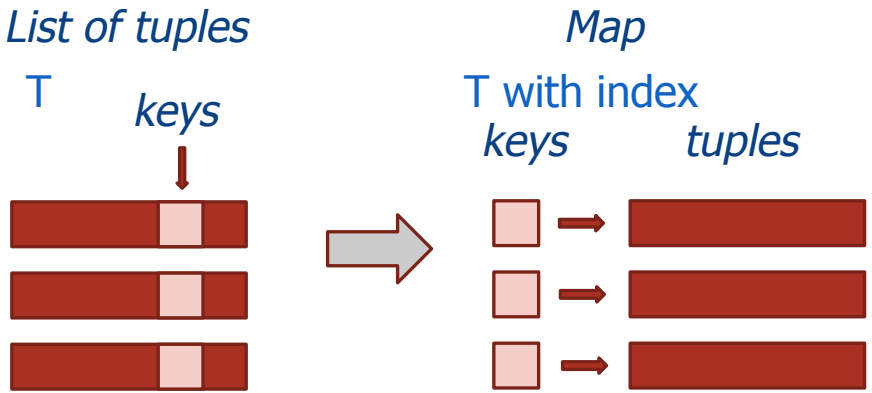
- Now: two ideas for improving performance by changing how we access data
  - Dictionaries / in-memory indices / maps
  - Buffering / blocks

https://tinyurl.com/cis545-lecture-02-14-22

Pandas Merge Is Based on Exact-Matches

Can we find exact-matches between the values in one tuple (from some table S) with another tuple (from some table T)?

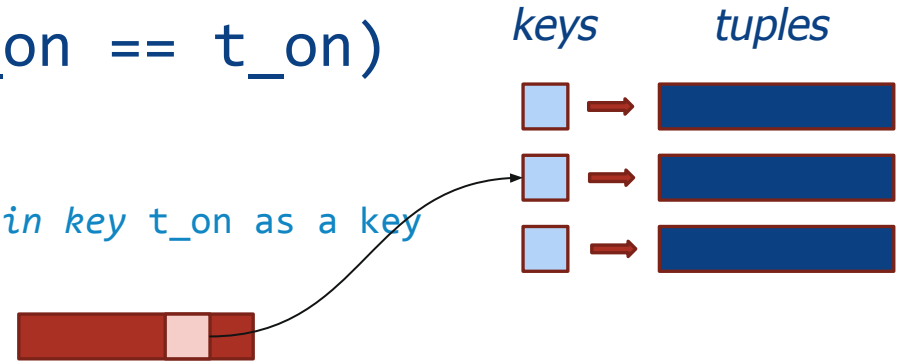
We had a way of doing fast lookups: maps from keys to values (i.e., dicts, indices)



https://tinyurl.com/cis545-lecture-02-14-22

Rethinking Join (For Equality)

Given `S join T on (s_on == t_on)`




```
index Dataframe T, using the join key t_on as a key
for each tuple s in S:
   Pull out s_on.
   Find matches in the index where t_on = s_on.
   Combine s with the matches and return.
```

https://tinyurl.com/cis545-lecture-02-14-22

A More Efficient Equality Join

```
def merge_map(S,T,s_on,t_on):
    ret = pd.DataFrame()
    T_map = {}
    # Take
    # make
    for t_i
      T_map[

    # Now f
    for s_i
        cou
        if S.loc[s_index, s_on] in T_map:
                ret = ret.append(S.loc[s_index].\
        appe
        igno

return ret
```

```
%%time

# Here's a test join, with people and their experiences.  We can see how many
# comparisons are made
merge_map(experience_df, people_df, 'person', '_id')
```

```
Merge compared 5500 tuples
CPU times: user 7.12 s, sys: 0 ns, total: 7.12 s
Wall time: 7.14 s
```

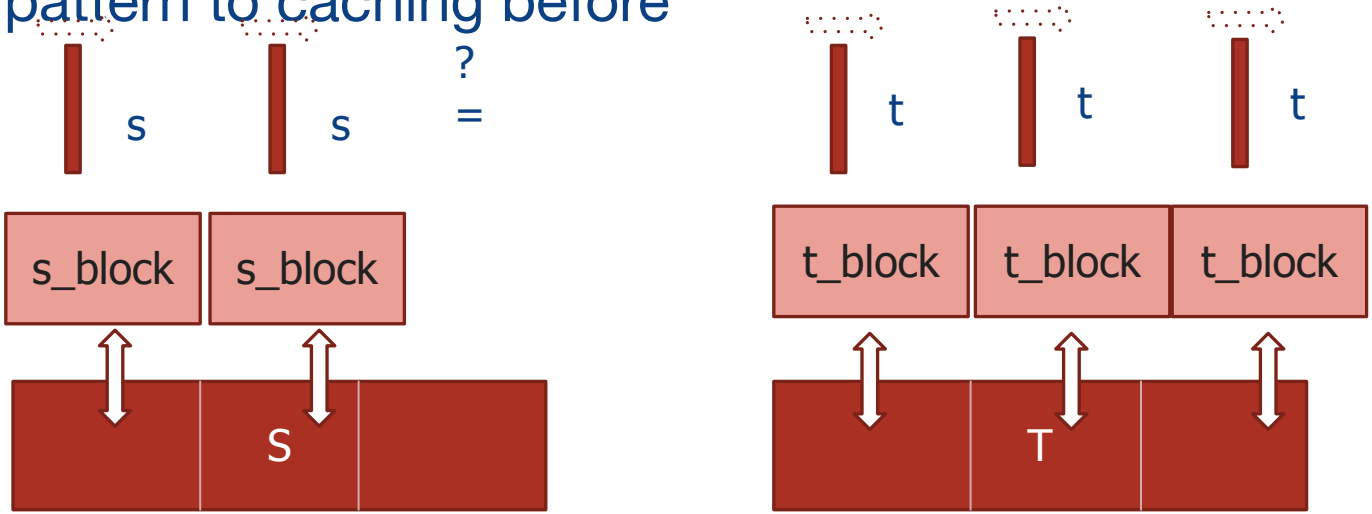
```
merge(people_df, experience_df, '_id', 'person')
```

```
Merge compared 2500000 tuples
CPU times: user 30.8 s, sys: 0 ns, total: 30.8 s
Wall time: 30.9 s
```

https://tinyurl.com/cis545-lecture-02-14-22

Suppose We Have Big Data, so S and T Don't Fit in Memory!

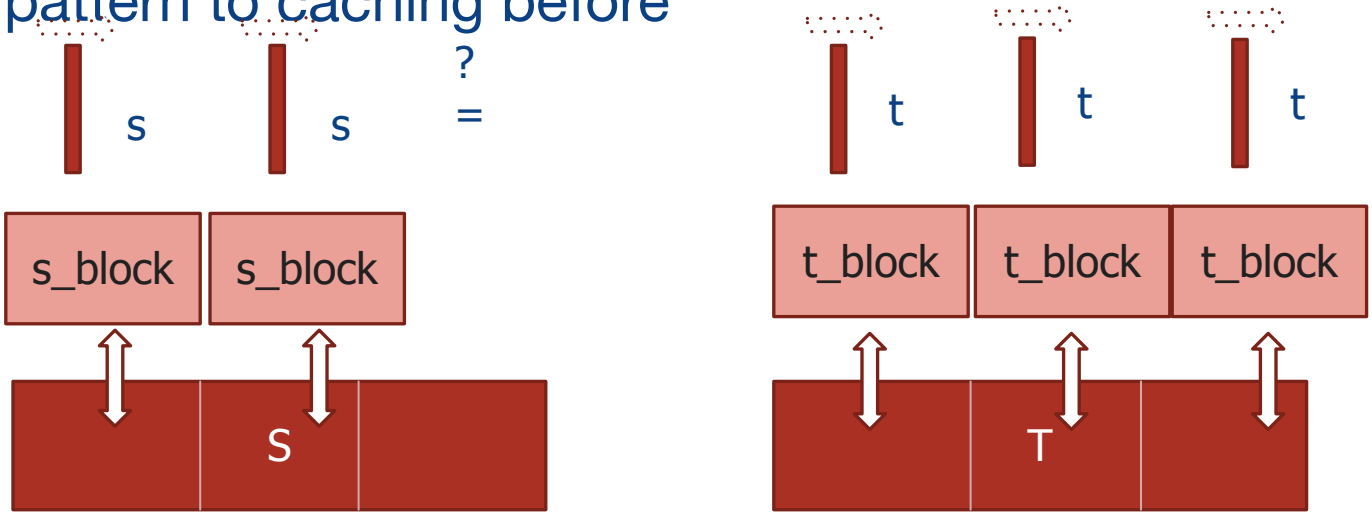
We need to read a *block* at a time, following a similar pattern to caching before



https://tinyurl.com/cis545-lecture-02-14-22

Suppose We Have Big Data, so S and T Don't Fit in Memory!

We need to read a *block* at a time, following a similar pattern to caching before



https://tinyurl.com/cis545-lecture-02-14-22

Joins with Big Data
Need to Read a Table in Blocks

```
# Pseudocode
def merge_on_disk(S,T,s_on,t_on):
    ret = []
    for s_block in blocks(S):
         for t_block in blocks(T):
              for s in s_block:
                  for t in t_block:
                   if s[s_on] == t[t_on]:
                           ret.add(s.append(t)))

    return ret
```

Now we read blocks(S) * blocks(T) pages, and compare each s and t from these

100us per block read, 75ns per comparison!!!

https://tinyurl.com/cis545-lecture-02-14-22

Beyond the Relational Algebra

- For big data, sometimes we'll need to supply our own operations
  - Functions to be called via apply (or applymap)
  - Functions that take collections of data

- Again, we'll want to rely on using maps and indices to reduce data usage, and intelligent buffering

https://tinyurl.com/cis545-lecture-02-14-22

Brief Review

https://canvas.upenn.edu/courses/1636888/quizzes/2771493 (07E)

If we use a dictionary / map / hash table to join two tables `S` and `T`, our cost is proportional to

a. the product of the cardinalities (numbers of rows) of `S` and `T`
b. the cardinality (number of rows) of the bigger of `S` and `T`
c. the sum of the cardinalities (numbers of rows) of `S` and `T`
d. the square root of the sums of the sizes of the tables, squared

How does buffering or blocked access help speed performance when tables are bigger than memory?

e. we reduce the size of the index
f. we reduce the overall number of disk fetches
g. we reduce the size of the join output
h. we reduce the overall number of join comparisons

https://tinyurl.com/cis545-lecture-02-14-22

Summary:
Making Dataframes / Queries Efficient

General rule of thumb for efficiency: consider order of evaluation

- Minimize intermediate results
- For Pandas, “ballpark estimate” which order is better
- SQL database optimizes using statistical information it collects on tables!

For some joins, can use faster algorithm:

- Joins, by default, iterate over both tables
- An *index* makes the lookups more efficient IF the join is on the *index key*

https://tinyurl.com/cis545-lecture-02-14-22

Processing Data at Scale

Susan B. Davidson and Zachary G. Ives

University of Pennsylvania

CIS 545 – Big Data Analytics

Portions of this lecture have been contributed to the OpenDS4All project, piloted by Penn, IBM, and the Linux Foundation

https://tinyurl.com/cis545-lecture-02-14-22

How Do We Handle Large Volumes of Data?

1. Reduce the amount of data

2. Use more efficient algorithms or data structures

3. Split work among multiple processors / workers!

https://tinyurl.com/cis545-lecture-02-14-22

Multiple Processors Are Important!

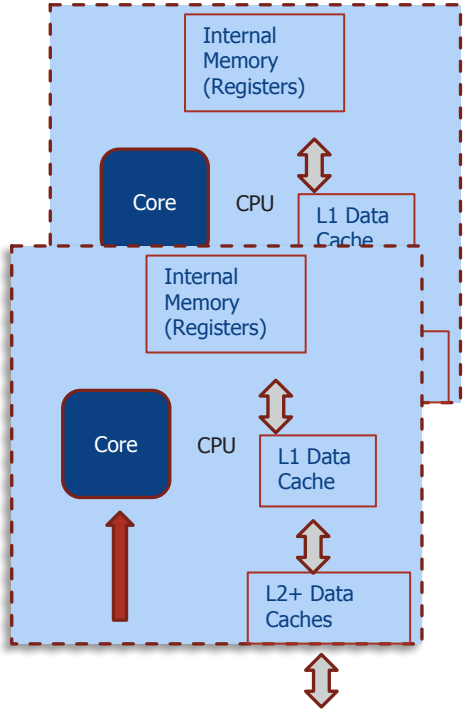
For decades, CPU designers made cores dramatically faster each generation:

- Roughly 2x performance every 18 months – "Moore's Law" (and "Dennard Scaling")

No longer true – but now we can put often more *cores* into each CPU generation

But: how to *coordinate* among the cores?

https://tinyurl.com/cis545-lecture-02-14-22



Bulk Operations and Scale

Relational and other **bulk** operations are key:

- They express how to do **the same thing over many values**
- Selections, projects, joins, apply, etc. can often be done on different rows in parallel!
- (Also true for certain other operations, as we'll see later)

https://tinyurl.com/cis545-lecture-02-14-22

Road Map: Variations on "Data Parallelism"

"Data parallelism" = same instructions, multiple data items

- Dask: multicore processing of relational algebra on a single machine
- Cluster computing and the cloud
- Data processing in a cluster
- Apache Spark and cluster-based processing of dataframes

https://tinyurl.com/cis545-lecture-02-14-22

Multicore Processing of Dataframes

Susan B. Davidson and Zachary G. Ives

University of Pennsylvania

CIS 545 – Big Data Analytics

Portions of this lecture have been contributed to the OpenDS4All project, piloted by Penn, IBM, and the Linux Foundation

https://tinyurl.com/cis545-lecture-02-14-22

Multicore and Pandas

- Relational operators inherently apply the same value-based computation to each row

- There's a natural way of assigning different data to different CPU cores, and processing them in parallel

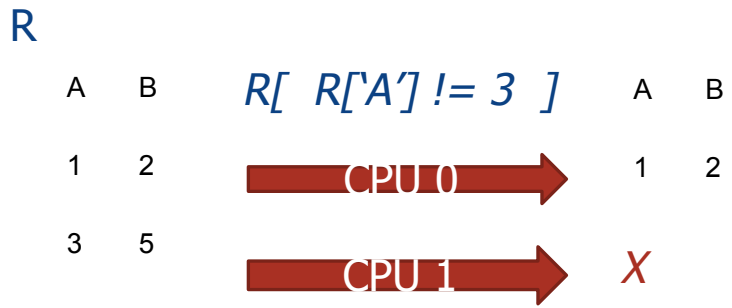
- Not done by Pandas, but there are alternatives

https://tinyurl.com/cis545-lecture-02-14-22

Parallel Processing of Relational Algebra

So far: we did our work one step at a time!

Relational operators can be done *in parallel* across records



https://tinyurl.com/cis545-lecture-02-14-22

Dask: Parallel Execution

Dask provides its own replacement dataframes for Pandas (and other libraries) that automatically supports parallelism on a single machine

Almost identical code to Pandas and Apache Spark,
which we'll talk about instead (and next)…

https://tinyurl.com/cis545-lecture-02-14-22

Let's Revisit our LinkedIn Data Example

https://tinyurl.com/cis545-005

```
# 100K records from linkedin
linked_in = open('/content/linkedin_small.json.txt' )

people = []

for line in linked_in:
    person = json.loads(line)
    people.append(person)

people_df = pd.DataFrame(people)
print ("%d records"%len(people_df))

people_df
```

We should split out the nested lists for skills, education, experience, and honors:

```
skills_df = people_df[['_id','skills']].\
  explode('skills')
education_df = people_df[['_id','education']].\
  explode('education')
experience_df = people_df[['_id','experience']]
honors_df = people_df[['_id', 'honors']]

linkedin_df = people_df.copy().drop(\
  columns=['skills','education','experience',\
  'honors'])
```

https://tinyurl.com/cis545-lecture-02-14-22

Rejoining the Tables Could Be Expensive!

```
%%time

linkedin_df.merge(experience_df, on='_id').\
  merge(skills_df, on='_id').\
  merge(honors_df, on='_id').\
  merge(education_df, on='_id')
```

```
Wall time: 3.71 s
```

Full LinkedIn data is 90x bigger!

https://tinyurl.com/cis545-lecture-02-14-22

Dask Has Its Own Dataframes

```
import dask
import dask.dataframe as dd
linkedin_ddf = dd.from_pandas(linkedin_df,npartitions=100)
skills_ddf = dd.from_pandas(skills_df,npartitions=100)
experience_ddf = dd.from_pandas(experience_df,npartitions=100)
education_ddf = dd.from_pandas(education_df,npartitions=100)
honors_ddf = dd.from_pandas(honors_df, npartitions=10)
%%time
linkedin_ddf.merge(experience_ddf, on='_id').\
  merge(skills_ddf, on='_id').merge(honors_ddf, on='_id').\
  merge(education_ddf, on='_id')
```

```
Wall time: 160 ms
```

https://tinyurl.com/cis545-lecture-02-14-22

We Saw Dask Does Parallel Dataframes – What about Parallel SQL?

- Many DBMSs like PostgreSQL can support parallel processing, but on a single machine disk I/O is typically the bottleneck

- So they largely benefit with *clusters* as we'll see soon!

https://tinyurl.com/cis545-lecture-02-14-22

Summary of Multicore Processing

Through parallel libraries like Dask, we can process data items independently on different cores

Not true for every operation, but for some relational operations we can conceptually get speedups up to min(# cores, # rows)

But for big data, we often have too much data to handle on a single (even multi-core) computer!

https://tinyurl.com/cis545-lecture-02-14-22

Brief Review

https://canvas.upenn.edu/courses/1636888/quizzes/2771523 (08B)

How does Dask speed up common dataframe operations?

a. it optimizes the cache
b. it partitions dataframe by row so they can be processed by different CPU cores
c. it replaces Pandas code with native Python
d. it partitions dataframes by column so they can be more efficiently executed

Why don't DBMSs generally show big benefits from multicore processors?

e. they are just fundamentally slower
f. SQL can't be executed in parallel
g. typically most of their processing is limited by I/O
h. they need to incorporate Apache Spark

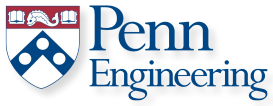
https://tinyurl.com/cis545-lecture-02-14-22

Beyond a Single Machine: Clusters and Data Centers

Susan B. Davidson and Zachary G. Ives

University of Pennsylvania

CIS 545 – Big Data Analytics

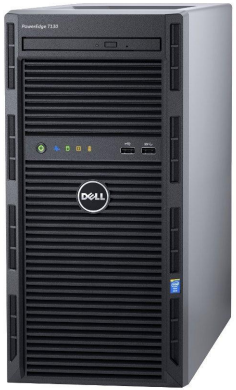


Portions of this lecture have been contributed to the OpenDS4All project, piloted by Penn, IBM, and the Linux Foundation

https://tinyurl.com/cis545-lecture-02-14-22

The First Approach: Bigger Machines and Dask / Swifter

2019 laptop:
16GB RAM
6 4GHz cores
2TB SSD



2019 Server:
1TB RAM
GPU
28 4GHz cores
1PB SSD

Cluster:
30 servers,
fast network
among them

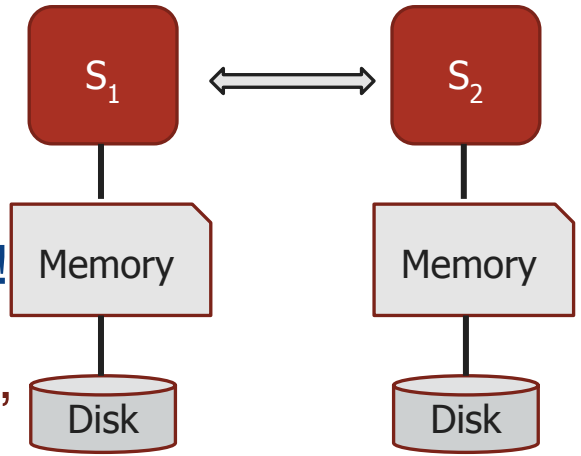
https://tinyurl.com/cis545-lecture-02-14-22

Non-uniform Costs in Clusters and Parallel Systems

Recall we needed to pay a lot of attention to algorithms when we had a disk

- Internal CPU state (registers, cache)
- Memory (latency 1000s of times slower)
- SSD / disk (latency 1000+ times slower)

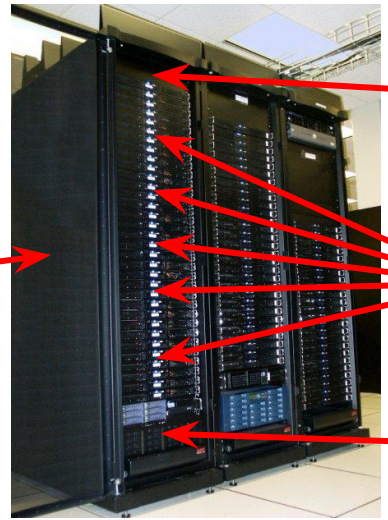
- Same is true if we have multiple computers!



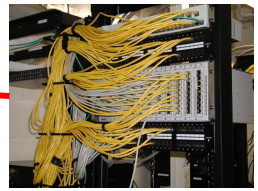
Each processor should work "independently" and "locally" as much as possible

https://tinyurl.com/cis545-lecture-02-14-22

Building Clusters of Computers



Rack



Network switch
(connects nodes with each other and with other racks)

Many nodes/blades
(often identical)

Storage device(s)

- Many similar machines, close interconnection (same room?)
- Often special, standardized hardware (racks, blades)
- Usually owned and used by a single organization

https://tinyurl.com/cis545-lecture-02-14-22

Scaling Out

Desktop

Server

Cluster

Data center

What if your cluster is too big (hot, power hungry) for an office building?

- Build a separate building for the cluster – a data center!
- Building can have lots of cooling and power

https://tinyurl.com/cis545-lecture-02-14-22

"The Cloud"

Commercial cloud: **data centers run by commercial providers**

Amazon Web Services, Microsoft Azure, Google Cloud, IBM Cloud, Rackspace, Linode, …

- Machines you can access "elastically", maintained by the cloud provider's staff
- MANY services, ranging from "configurable Linux machines" to "Storage" or "Jupyter"
- Often less expensive than maintaining local resources – due to economies of scale

Hybrid cloud: data centers run by commercial providers, combined with some local clusters administered the same way

https://tinyurl.com/cis545-lecture-02-14-22

Google data centers

https://tinyurl.com/cis545-lecture-02-14-22

Brief Review

https://canvas.upenn.edu/courses/1636888/quizzes/2771570 (08C)

On average, how much system memory (RAM) is typical in a laptop today?

a. 1 PB
b. 1TB - 4TB
c. 64 - 128 TB
d. single-digit to 3-digit GB

When might we need a data center?

e. we need a server with more memory
f. we want to set up a compute cluster
g. we don't have enough power and cooling for a machine room
h. we don't have enough Internet bandwidth and would need to upgrade to fiber

https://tinyurl.com/cis545-lecture-02-14-22

Summary: Clusters and Data Centers Provide the Hardware for Big Data

- Today we build parallel computation by linking together many compute nodes
- Each has its own local CPU cache, memory, and disk
- Communication costs vary dramatically at different levels
- The challenge: we can't program these *just* using Python and Pandas!
  - How do we get 30 multicore servers to work together?
  - At scale, machines crash all the time – how do we handle failures?

https://tinyurl.com/cis545-lecture-02-14-22

Programming for Clusters: Relational Algebra + Partitioning

Susan B. Davidson and Zachary G. Ives

University of Pennsylvania

CIS 545 – Big Data Analytics

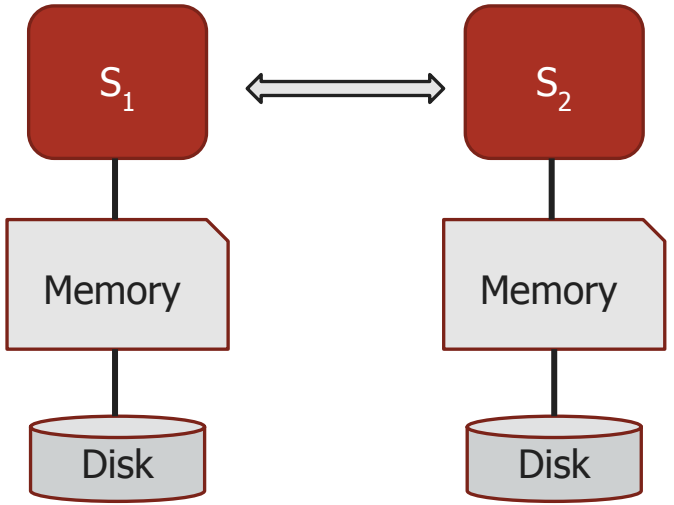
Portions of this lecture have been contributed to the OpenDS4All project, piloted by Penn, IBM, and the Linux Foundation

https://tinyurl.com/cis545-lecture-02-14-22

So How Do We Use Compute Clusters to Handle Big Data?

Two main issues:

- Making computation *parallel*
- Minimizing communication and coordination



https://tinyurl.com/cis545-lecture-02-14-22

The Basic Approach

- We'll write programs using relational operations – Pandas-style or using SQL

- Data will reside on multiple compute nodes

- A *coordinator* will tell multiple nodes on the cluster which relational algebra operators to apply, and when

https://tinyurl.com/cis545-lecture-02-14-22

“Horizontal” Partitioning of Data

Much like in the multicore setting, we can randomly put different rows on different computers – “horizontal partitioning”

A	B	C	
1	2	3	Server 0
4	8	22	
9	23	45	Server 1
4	12	44	

https://tinyurl.com/cis545-lecture-02-14-22

"Horizontal" Partitioning of Data

Much like in the multicore setting, we can randomly put different rows on different computers – "horizontal partitioning"

A	B	C	
1	2	3	Server 0
4	8	22	
9	23	45	Server 1
4	12	44	

Tuple-at-a-time operators work great!

- Select
- Project
- Apply

https://tinyurl.com/cis545-lecture-02-14-22

"Horizontal" Partitioning of Data

Much like in the multicore setting, we can randomly put different rows on different computers – "horizontal partitioning"

A	B	C	
1	2	3	Server 0
4	8	22	
9	23	45	Server 1
4	12	44	

Doesn't work for everything –

What about **groupby('A').count()** ?

https://tinyurl.com/cis545-lecture-02-14-22

"Horizontal" Partitioning of Data

Much like in the multicore setting, we can randomly put different rows on different computers – "horizontal partitioning"

A	B	C	
1	2	3	Server 0
4	8	22	Server 1
9	23	45	Server 0
4	12	44	Server 1

Let's *shuffle* the data so all A's with the same value are on the same machine!

Then we can do **groupby('A').count()** !

https://tinyurl.com/cis545-lecture-02-14-22

The Intuition

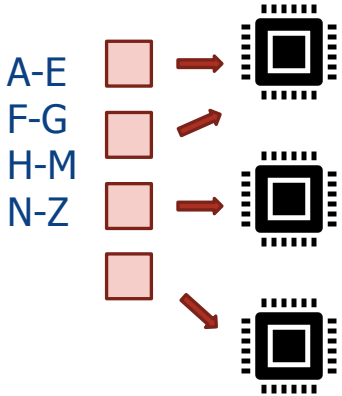
- If we want to “shard” the data by value, we need a mapping from values to servers

https://tinyurl.com/cis545-lecture-02-14-22

Sharding: A Generalization of Maps and Indices

We saw an index as a *map*: key → the set of tuple(s) with the key

Suppose we take a set of computers in a cluster, and build a map from *key range* to machine ID



Then we send to that machine all tuples whose key matches this

https://tinyurl.com/cis545-lecture-02-14-22

Doing this More Intelligently and Generally

What if we want to partition based on arbitrary datatypes and numbers of nodes?

Hashing: takes a value ("a hash key") and returns a large integer

For a good hash function *h*, for any k_1, k_2,

$h(k_1) = h(k_2)$ if $k_1 = k_2$

$h(k_1) \neq h(k_2)$ with **high probability** if $k_1 \neq k_2$!

h(k)	h(k) % 3
0	0
1	1
2	2
3	0
4	1
5	2
6	0

Then, if we have *n* machines, we typically put the data for key *k* at node:

h(k) (modulo n)

or, in Python, h(k) % n

https://tinyurl.com/cis545-lecture-02-14-22

Brief Review

https://canvas.upenn.edu/courses/1636888/quizzes/2771496 (08E)

Which relational operator does NOT work with random placement of rows across machines?

a. selection
b. group-by
c. apply (to row axis)
d. projection

We can define hash functions that return large integers from what kinds of datatypes:

e. ints only
f. ints or floating point numbers
g. strings only
h. strings, ints, or floating point numbers

https://tinyurl.com/cis545-lecture-02-14-22

Recap

- Relational operators, if done in parallel across multiple computers, can give us a model of scaling our computation

- A key issue is how to *partition* the data – we can do it randomly for simple operators, but need to *shard on an attribute* for others
  - We typically use *hash functions* to determine where to put the data

https://tinyurl.com/cis545-lecture-02-14-22

Apache Spark

Susan B. Davidson and Zachary G. Ives

University of Pennsylvania

CIS 545 – Big Data Analytics

Portions of this lecture have been contributed to the OpenDS4All project, piloted by Penn, IBM, and the Linux Foundation

https://tinyurl.com/cis545-lecture-02-14-22

Platforms for Big Data

- There are many tools for big data processing

- Key ideas developed in Google's MapReduce (Jeff Dean and Sanjay Ghemawat)

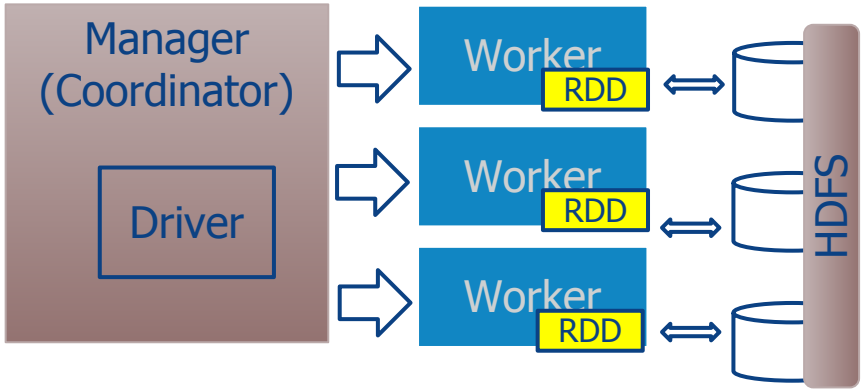
  https://www.newyorker.com/magazine/2018/12/10/the-friendship-that-made-google-huge

- Inspired Apache Hadoop, many other systems – including Spark, Flink, Flume, etc.

- We're focused on Spark but ideas are general!

https://tinyurl.com/cis545-lecture-02-14-22

Apache Spark
A Platform for Sharded Big Data



"Shards" tables as we discussed: "Spark DataFrames" similar to Pandas DataFrames

https://tinyurl.com/cis545-lecture-02-14-22

Spark-Jupyter Practicalities

https://tinyurl.com/cis545-notebook-04

Spark is not written in Python, making for some quirks:

- Spark dataframes have a **typed schema** – Spark can't determine on-the-fly whether input fields are strings vs integers like Python does

(See sample notebook for example code)

Jupyter and Spark may be on different machines – Colab vs Spark cluster coordinator

https://tinyurl.com/cis545-lecture-02-14-22

Writing Code for Spark

In Colab, every cell will need to *execute remotely* on a Python/Spark machine

Prefix the cell with `%%spark`

Spark supports Dataframes style operations, as in Pandas

```
linked_df.createOrReplaceTempView("linked_in")
```

Or SQL-style operations as with relational DBMSs:

```
sqlContext.sql("select * from linked_in")
```

It also does something non-intuitive: *lazy evaluation*

… Nothing gets computed until you try to **save**, **show**, or **collect** it!

https://tinyurl.com/cis545-lecture-02-14-22

Simple Spark Dataframe Operations

Basic Spark operations look like Pandas – mostly:

- `df['x']` and `df.x` both reference field `x` in dataframe `df`
- Projection via double-brackets: `df[['field1', 'field2']]`
- Selection via "filter" function: `df.filter(df.field` *op* `value)`

https://tinyurl.com/cis545-lecture-02-14-22

Sharding and Tables

Given a cluster with *n* workers, running remotely, Spark creates a table with *at least n* partitions (here, 200, where 100 are stored on each machine)

Spark will partition "automatically" but it's best to *repartition* on the key you want!

```
%%spark
# 10
link
my_l
link
  re
```

```
+------------------+--------------------+--------------------+--------------------+-
|               _id|                name|            locality|              skills|
+------------------+--------------------+--------------------+--------------------+-
|      in-00000001|[given_name -> Dr...|       United States|[Key Account Deve...|
|          in-00001|[given_name -> An...|Antwerp Area, Bel...|[Molecular Biolog...|
|          in-00006|[given_name -> Sh...|San Francisco, Ca...|[DNA, Nanotechnol...|
|  in-000montgomery|[given_name -> Ed...|San Francisco Bay...|                null|I
|in-000vijaychauhan|[given_name -> Vi...| Chennai Area, India|[Program Manageme...|A
+------------------+--------------------+--------------------+--------------------+-
only showing top 5 rows
```

https://tinyurl.com/cis545-lecture-02-14-22

Computation in a Sharded System: Selection, Projection

_id	name	locality	skills
in-00000001	[given_name -> Dr...	United States	[Key Account Deve...
in-00001	[given_name -> An...	Antwerp Area, Bel...	[Molecular Biolog...
in-00006	[given_name -> Sh...	San Francisco, Ca...	[DNA, Nanotechnol...
in-000montgomery	[given_name -> Ed...	San Francisco Bay...	null
in-000vijaychauhan	[given_name -> Vi...		

only showing top 5 rows

Selection + projection is "farme

```
linked_df.filter(linked_df.l
'name', 'locality']].show(5)
```

_id	name	locality
in-00000001	[given_name -> Dr...	United States
in-100percenthair	[given_name -> Su...	United States
in-1solone	[given_name -> Ha...	United States
in-2raviagarwal	[given_name -> Ra...	United States
in-aarongatescarlton	[given_name -> Aa...	United States

only showing top 5 rows

https://tinyurl.com/cis545-lecture-02-14-22

Apply (with Python Functions) in Spark

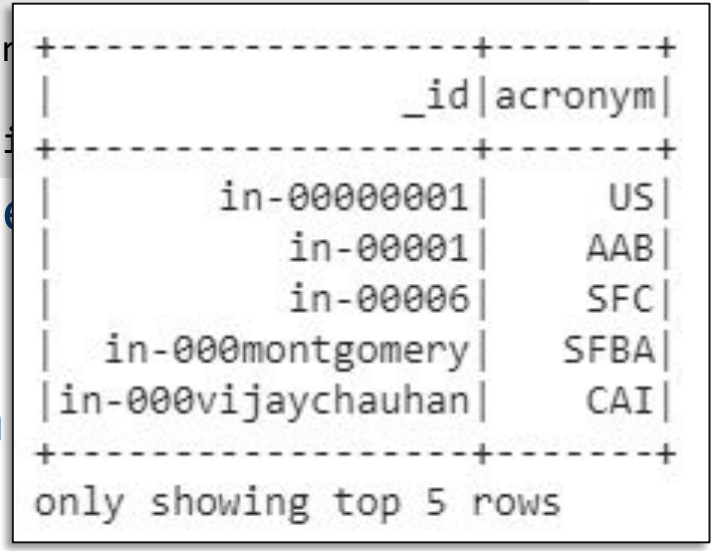
https://docs.databricks.com/spark/latest/spark-sql/udf-python.html

```
%%spark
from pyspark.sql.functions import udf
from pyspark.sql.types import StringType

acro = udf(lambda x: ''.join([n[0] for n in

linked_df.select("id", acro("locality").ali
```

Note also that we used Spark's sele
arguments looks much like a list for

As with select / project, apply is run
parallel!

https://tinyurl.com/cis545-lecture-02-14-22

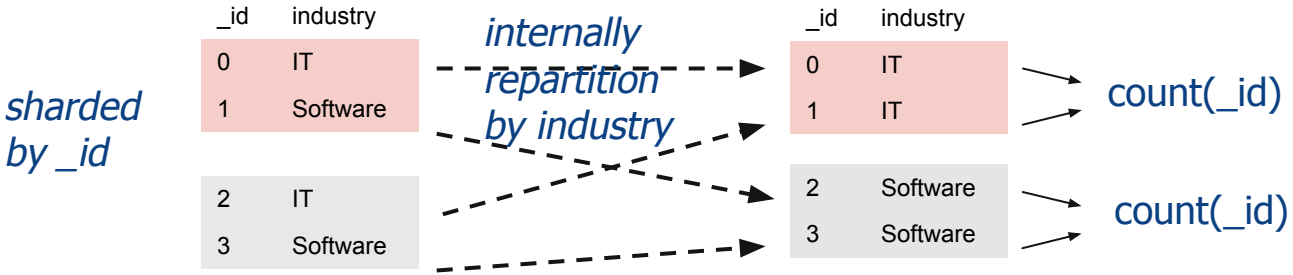


```
+------------------+-------+
|               _id|acronym|
+------------------+-------+
|       in-00000001|     US|
|          in-00001|    AAB|
|          in-00006|    SFC|
|  in-000montgomery|   SFBA|
|in-000vijaychauhan|    CAI|
+------------------+-------+
only showing top 5 rows
```

Grouping

Grouping needs *all* of the tuples in a group to be on the same machine, in order to do a computation over the group!

```
%%spark
# Which industries are most popular?
sqlContext.sql('select count(_id), industry '+\
'from linked_in group by industry '+\
'order by count(_id) desc').show(5)
```



https://tinyurl.com/cis545-lecture-02-14-22

Failures

- In a large cluster running for a long time – machines may die or software may crash

- Spark actually handles such failures transparently
  - It periodically "checkpoints" or snapshots what has happened
  - And if a node dies, it can restart the computation elsewhere!

https://tinyurl.com/cis545-lecture-02-14-22

Brief Review

https://canvas.upenn.edu/courses/1636888/quizzes/2771539 (08F)

Spark is not written in Python, which means;

a. Spark is slower than Pandas
b. schemas are strongly typed
c. we can't use Python strings
d. Spark must be written in C

How do we handle (a small number of) worker failures when Spark tasks are executed?

e. Spark handles this transparently
f. We need to buy new servers
g. We have to retry our jobs
h. We have to write try/except blocks in Python

https://tinyurl.com/cis545-lecture-02-14-22

Recap

Summary of Big Data so far:

- We need to partition or *shard* data by keys, allowing machines to work in parallel across different shards
- Apache Spark is the most popular system for doing this right now
  - Supports Spark dataframes or Spark SQL
  - Some variations from Pandas: strong typing, syntax variations, special **udf** function
  - You can control sharding via **repartition**
  - Select, project, **apply** all work in parallel across shards
  - Grouping typically requires the machines to exchange or repartition data

https://tinyurl.com/cis545-lecture-02-14-22