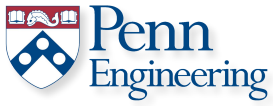
Essentials of Computer Architecture

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-02-09-22

Understanding a Bit of Computer Architecture Is Key

A few ideas we'll need from Computer Architecture:

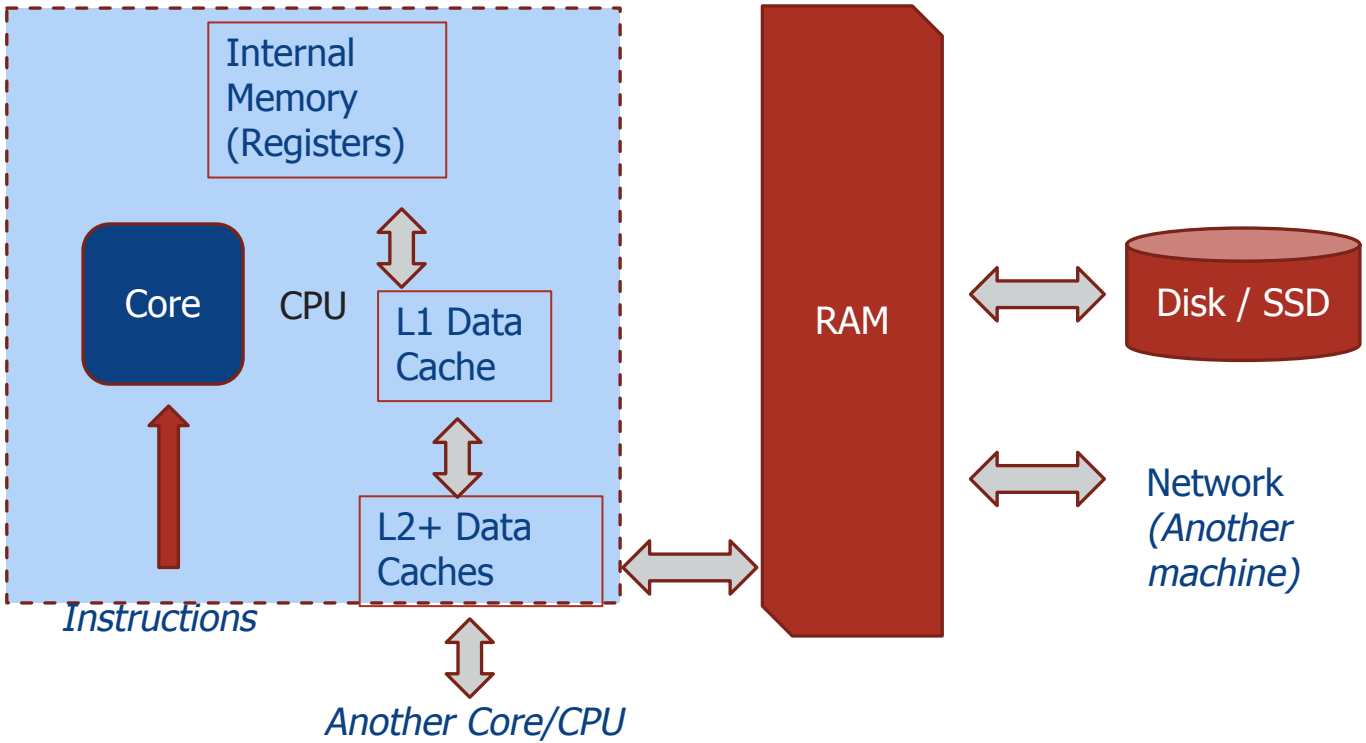
1) Different program instructions take different amounts of work and time
   - Lines of code (in Python / C / etc) turn into multiple *machine instructions*
   - Machine instructions don't all have equal performance!

2) The computer's *memory hierarchy* means accessing data is also not uniform!

3) Modern hardware includes special optimizations to:
   - Do the same operation on multiple data items simultaneously
   - Run multiple independent pieces of code at the same time

https://tinyurl.com/cis545-lecture-02-09-22

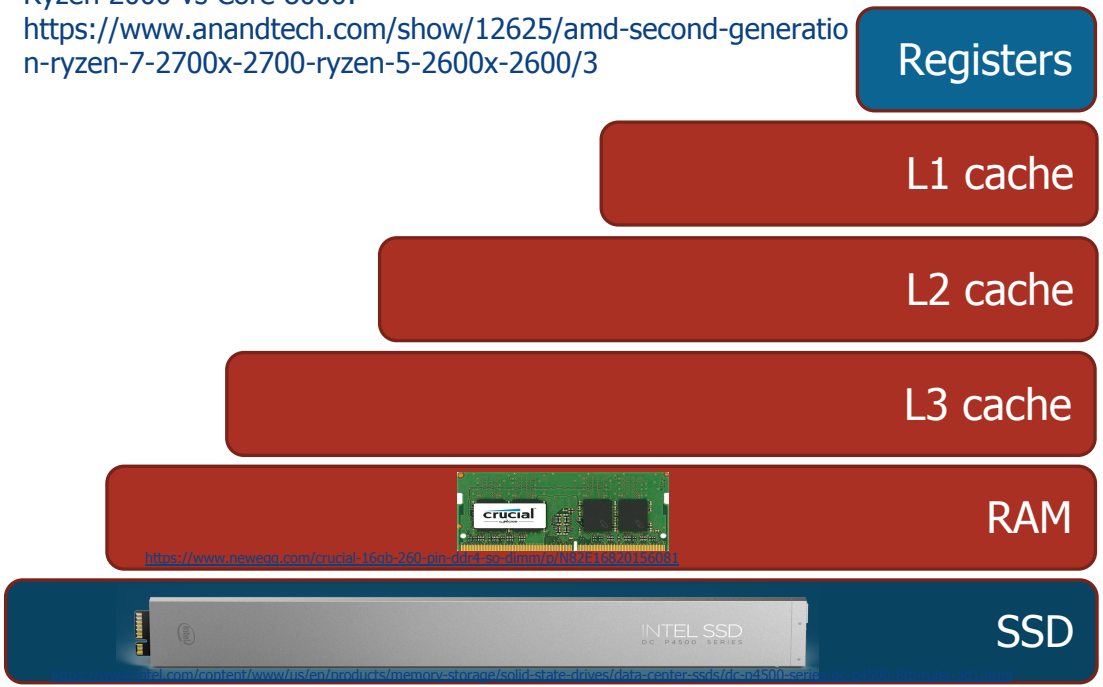
Key Aspects of a Computer: Simplified View of a Microprocessor



https://tinyurl.com/cis545-lecture-02-09-22

The Memory Hierarchy from One X86's Perspective

Ryzen 2000 vs Core 8000:
https://www.anandtech.com/show/12625/amd-second-generation-ryzen-7-2700x-2700-ryzen-5-2600x-2600/3



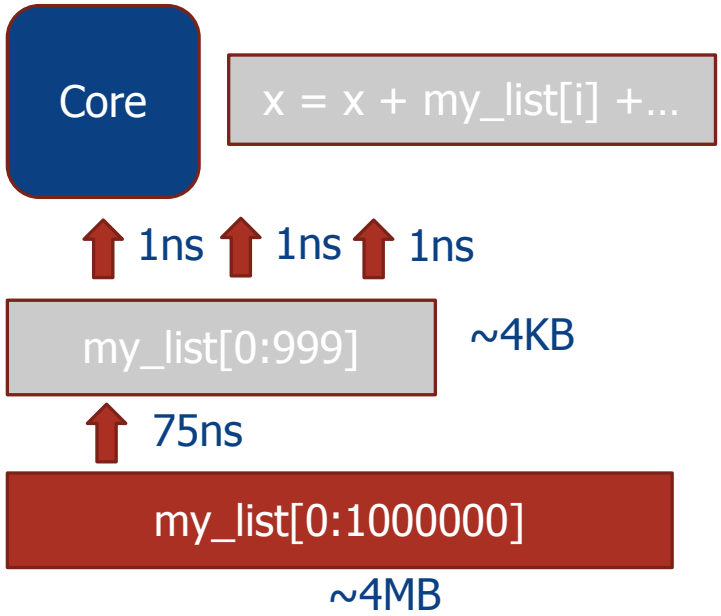
https://www.newegg.com/crucial-16gb-260-pin-ddr4-so-dimm/p/N82E16820156001

https://tinyurl.com/cis545-lecture-02-09-22

How Are Caches Used? (We Are Simplifying to 1 Level)

```
# Create an array 1,000,000 by 1
my_list = np.empty(1000000)

for i in range(0, len(my_list)):
  x = x + my_list[i] + \
    random.random()
```

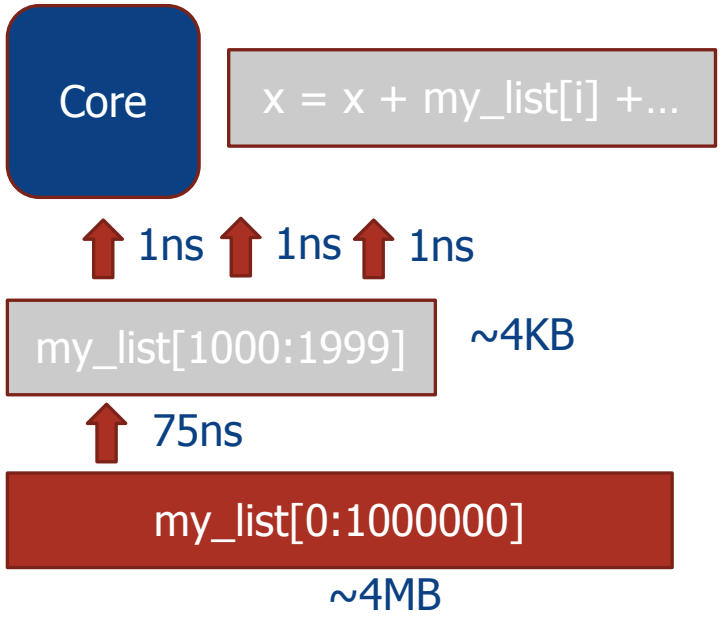


https://tinyurl.com/cis545-lecture-02-09-22

How Are Caches Used? (We Are Simplifying to 1 Level)

```
# Create an array 1,000,000 by 1
my_list = np.empty(1000000)

for i in range(0, len(my_list)):
  x = x + my_list[i] + \
    random.random()
```

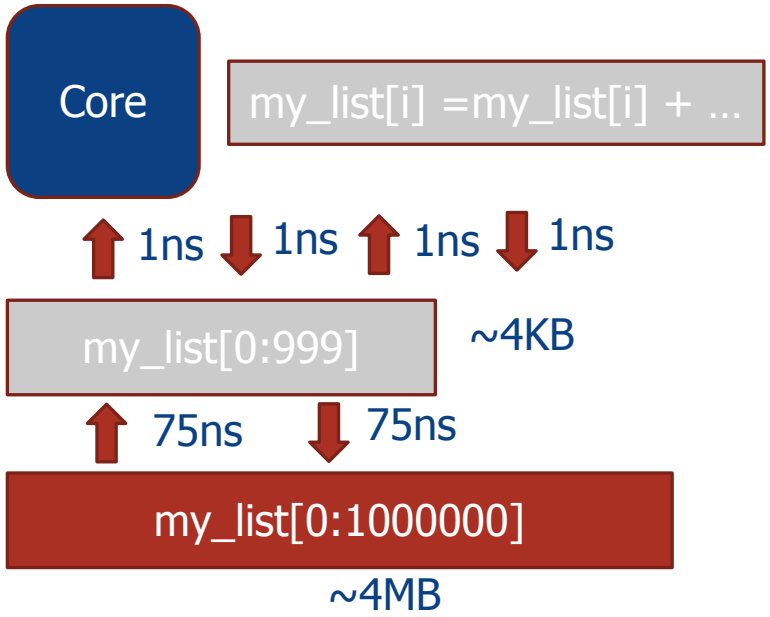


https://tinyurl.com/cis545-lecture-02-09-22

How Are Caches Used? (We Are Simplifying to 1 Level)

```
# Create an array 1,000,000 by 1
my_list = np.empty(1000000)

for i in range(0, len(my_list)):
  my_list[i] = my_list[i] + \
    random.random()
```

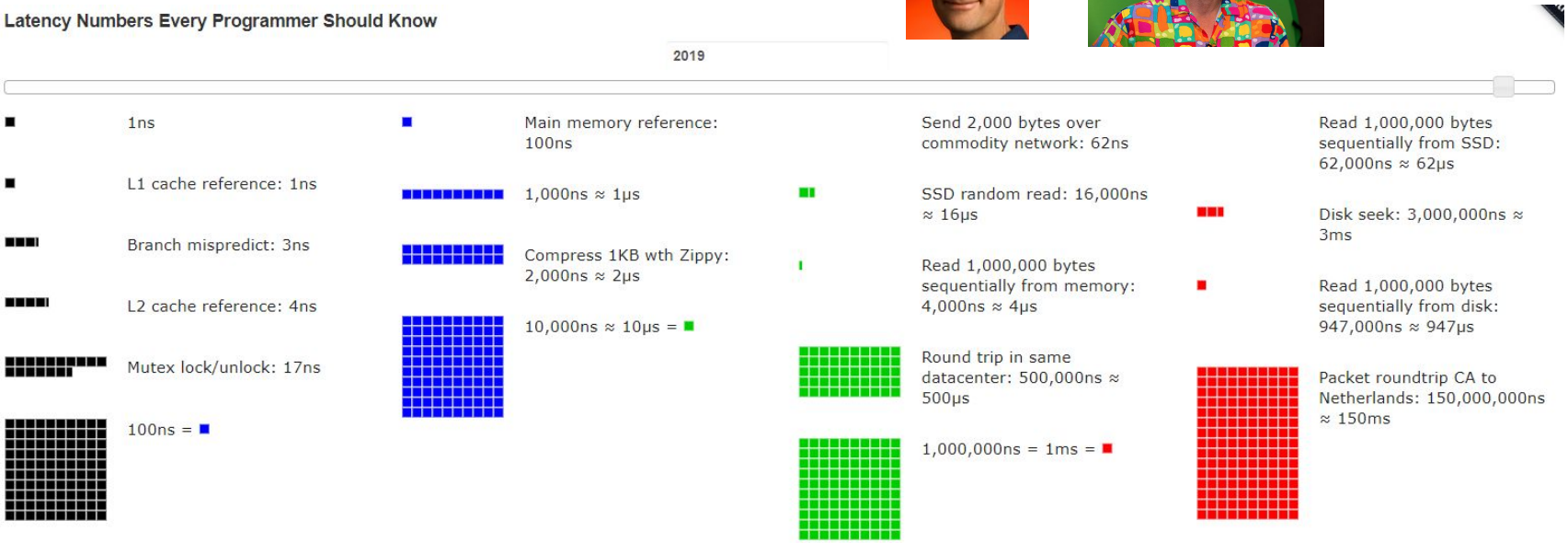


https://tinyurl.com/cis545-lecture-02-09-22

Interactive Visualizer (Colin Scott):

https://people.eecs.berkeley.edu/~rcs/research/interactive_latency.html

Original list from Jeff Dean and Peter Norvig (Google)



How Do We Take Advantage of the Memory Hierarchy?

- Focus on data reduction – filter rows and columns
- Put the most frequently used items together
- Find ways to pass over the data once, instead of many times
- Some of this is done by our big data engines – but there are other factors in *optimization* we'll see shortly

https://tinyurl.com/cis545-lecture-02-09-22

Recap: Consider the Memory Hierarchy When Thinking about Scale

- Accessing data in **predictable ways** is better
  - CPU predicts and "pre-fetches" the data into caches
  - Repeated requests to related data keep memory in the caches
- Smaller "memory footprints" are better (at tension with the previous)
- Packing multiple data values into the same memory, and applying a repeated computation, is better

https://tinyurl.com/cis545-lecture-02-09-22

Brief Review

https://canvas.upenn.edu/courses/1636888/quizzes/2771492/ (07B)

How many instructions can a typical machine run in one second, at peak?

a. billions (10^9)
b. trillions (10^{12})
c. thousands (10^3)
d. millions (10^6)

How does a cache help with performance?

e. it stores all of system memory in a faster place
f. it stores the contents of the registers
g. it allows us to amortize expensive memory fetches across multiple requests
h. it stores all of the disk state

https://tinyurl.com/cis545-lecture-02-09-22

Optimizing Relational Expressions to Improve Performance

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-02-09-22

Roadmap for Data Processing

https://tinyurl.com/cis545-notebook-03

- How computer architecture and memory affect performance
- Optimizing relational algebra to minimize memory costs
- Optimizing join orders
- Algorithmic techniques

https://tinyurl.com/cis545-lecture-02-09-22

Big Data Takes A Long Time to Process

Even simple operations are expensive at scale…

```
%%time
# 100,000 records from linkedin
linked_in = open('linkedaa')

people = []

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

people_df = pd.DataFrame(people)
people_df[people_df['industry'] \
          == 'Medical Devices']
```

_id	name	locality	skills	industry
in-00000001	{'family_name': 'M ', 'given_name': 'D...	United States	[Key Account Development, Strategic Planning, ...	Medical Devices
in-13806219531	{'family_name': 'G ', 'given_name': 'T '}	China	[ISO 13485, Medical Devices]	Medical Devices

```
CPU times: user 58.2 s, sys: 1min 57s, total: 2min 55s
Wall time: 3min 19s
```

What makes this so slow?

- Scanning through memory
- Parsing strings
- Updating the list data structure
- Converting to Pandas DF

https://tinyurl.com/cis545-lecture-02-09-22

Optimizing for Costs

1. Reduce data as soon as possible… "Push down" operations!
   Select (filter) at the earliest point possible
   - Less fetching from disk (if relevant), updating of data structures, processing of rows!

   Project once columns aren't needed: Data better fits into the CPU cache

```
%%time
linked_in = open('linkedaa')
people = []
for line in linked_in:
    person = json.loads(line)
    if 'industry' in person and \
       person['industry'] == \
              'Medical Devices':
        people.append(person)

people_df = pd.DataFrame(people)
```

_id	name	locality	skills	industry
in-00000001	{'family_name': 'M… ', 'given_name': 'D…	United States	[Key Account Development, Strategic Planning, …	Medical Devices
in-13806219531	{'family_name': 'G… ', 'given_name': 'T… '}	China	[ISO 13485, Medical Devices]	Medical Devices

```
CPU times: user 30.2 s, sys: 11.5 s, total: 41.7 s
Wall time: 43.2 s
```

https://tinyurl.com/cis545-lecture-02-09-22

SQL Does this Automatically (and Doesn't Parse Every Time)

```
%%time


pd.read_sql_query('select * from
people where industry="Medical
Devices"', conn)
```

```
CPU times: user 15.6 ms, sys: 109 ms, total: 125 ms
Wall time: 137 ms
```

_id	locality	industry	summary
in-00000001	United States	Medical Devices	SALES MANAGEMENT / BUSINESS DEVELOPMENT / PROJ...
in-13806219531	China	Medical Devices	

https://tinyurl.com/cis545-lecture-02-09-22

Even Better: Indexing Speeds Selection

Can we speed up fetches for specific data, e.g., people by industry?

An **index** is a map from an **index key** to a **set of values**

- It allows us to directly find matches to the key without scanning the data
- Can be in-memory or on disk

Two types of indices:

- **Tree** indices (B+ Trees) allow us to find all values <= key, >= key, = key
- **Hash** indices allow us to look up all values equal to the key

Q: Which is a **dict** in Python?

https://tinyurl.com/cis545-lecture-02-09-22

```
conn = sqlite3.connect('linkedin.db')


conn.execute('begin transaction')
conn.execute("create index
people_industry on people(industry)")
conn.execute('commit')
```

```
%%time

pd.read_sql_query('select * from
people where industry="Medical
Devices"', conn)
```

```
CPU times: user 0 ns, sys: 0 ns, total: 0 ns
Wall time: 26.9 ms
```

To Consider

Can indexing also speed projection?

https://tinyurl.com/cis545-lecture-02-09-22

The Summary So Far: Selections and Projections

- A good rule of thumb is to "push down" selection and projection operations
- SQL DBMSs do this automatically
- Index data structures allow us to evaluate predicates on a **key** and directly return the matches
  - Hash indices help with exact-matches (we'll revisit this soon)
  - Tree indices (B+ Trees in relational DBMSs) help with equality and inequality

https://tinyurl.com/cis545-lecture-02-09-22

Brief Review

https://canvas.upenn.edu/courses/1636888/quizzes/2771528 (07C)

How does "pushing down" filter conditions typically help performance? Choose the best answer.

a. Reduces memory "footprint" and number of operations
b. Requires an index
c. Reduces number of operations only
d. Reduces memory "footprint" only

A typical index:

e. requires time to access data that's linear in the number of rows in the table
f. requires roughly constant time for every access
g. requires time to access data that's linear in the number of columns in the table
h. doesn't work unless all data fits in system memory

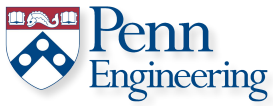
https://tinyurl.com/cis545-lecture-02-09-22

Join Ordering and Optimization

Susan B. Davidson and Zachary G. Ives

University of Pennsylvania

CIS 545 – Big Data Analytics

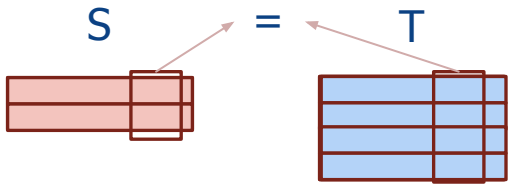


Portions of this lecture have been contributed to the OpenDS4All project, piloted by Penn, IBM, and the Linux Foundation

https://tinyurl.com/cis545-lecture-02-09-22

Recall Join
Compares all Pairs of Tuples for a Match

S join T on s_on = t_on:



How we implement it:

```
for every tuple s in S
   for every tuple t in T
     if s[s_on] == t[t_on] then combine & return
```

https://tinyurl.com/cis545-lecture-02-09-22

Joins: Expensive Operations

```
def merge(S,T,s_on,t_on):
    ret = pd.DataFrame()

    for s_index in range(0, len(S)):            # Rows in S
        for t_index in range(0, len(T)):        # Rows in T
          if S.loc[s_index, s_on] == T.loc[t_index, t_on]:
                ret = ret.append(S.loc[s_index].append(T.loc[t_index]),\
                                 ignore_index=True)
    return ret
```

How many operations does this require?

How big is the output?

https://tinyurl.com/cis545-lecture-02-09-22

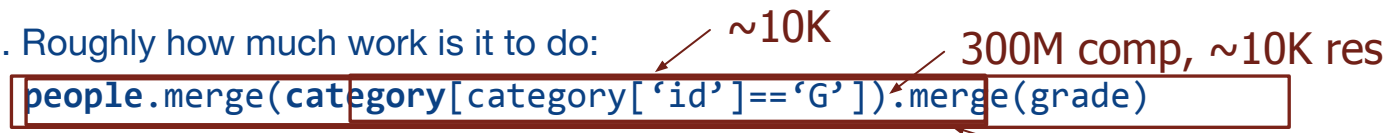
Using Our Intuitions in Order of Evaluation (Abstract Example)

Suppose we have three simple Dataframes describing people at Penn (roughly 10.1K undergrad, 10K grad/professional, 0.1K both, 10K faculty/staff):

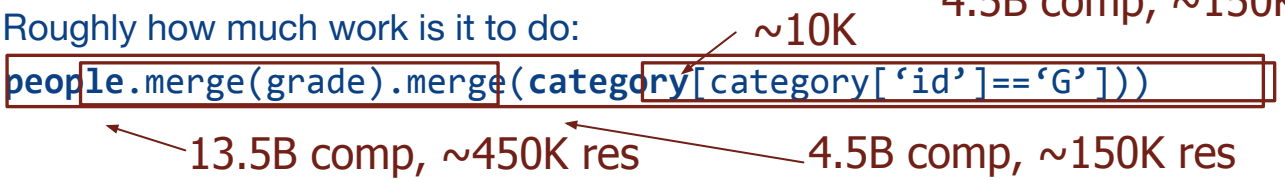
people(id, name) **category**(id, grad_undergrad_facstaff) **grade**(id, sem, score)

30K people *30.1K entries (G, U, F/S)* *450K entries*

1. Roughly how much work is it to do:



2. Roughly how much work is it to do:

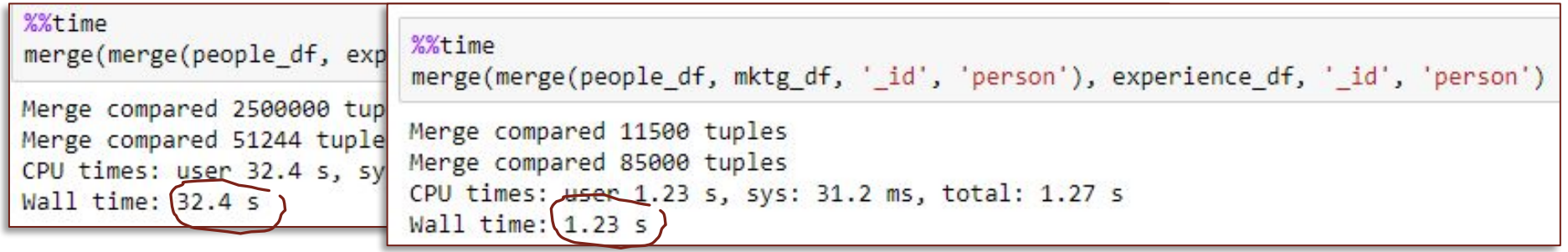


https://tinyurl.com/cis545-lecture-02-09-22

Looking at Order of Evaluation on Our Real, LinkedIn Dataset

```
people_df = pd.read_sql_query('select * from people limit 500', conn)
experience_df = pd.read_sql_query('select * from experience limit 5000',
conn)
skills_df = pd.read_sql_query('select * from skills limit 8000', conn)

temp = merge(people_df, experience_df, '_id', 'person')  # 2228 rows
mktg_df = skills_df[skills_df['value'] == 'Marketing'][['person']] # 23 rows
```



https://tinyurl.com/cis545-lecture-02-09-22

Do We Have to Choose Orders Manually?

Order of evaluation matters because intermediate result sizes matter

SQL (and query optimization) decide for us!

```
conn.execute('create view people500 as select * from people limit 500')
conn.execute('create view experience5000 as select * from experience limit
5000')
conn.execute('create view skills8000 as select * from skills limit 8000')

pd.read_sql_query('select * from (people500 join experience5000 on _id=person)
pe join ' + \
                  'skills8000 sk on pe._id=sk.person and sk.value="Marketing"',
conn)
```

```
CPU times: user 0 ns, sys: 15.6 ms, total: 15.6 ms
Wall time: 20.9 ms
```

https://tinyurl.com/cis545-lecture-02-09-22

Do We Have to Do this Manually?

Order of evaluation matters because intermediate result sizes matter

SQL (and query optimization) decide for us!

```
conn.execute('create view people500 as select * from people limit 500')
conn.execute('create view experience5000 as select * from experience limit
5000')
conn.execute('create view skills8000 as select * from skills limit 8000')

pd.read_sql_query('select * from (people500 join skills8000 on _id=person) ps
join ' + \
                  'experience5000 ex on ps._id=ex.person and value="Marketing"',
conn)
```

```
CPU times: user 15.6 ms, sys: 0 ns, total: 15.6 ms
Wall time: 18 ms
```

https://tinyurl.com/cis545-lecture-02-09-22

Recap

- Joins are expensive, requiring a quadratic (in the number of inputs) number of comparisons
- Cleverly ordering our joins to reduce intermediate result sizes makes a huge performance difference
- SQL databases can choose this automatically for us

https://tinyurl.com/cis545-lecture-02-09-22

Brief Review

https://canvas.upenn.edu/courses/1636888/quizzes/2771532 (07D)

If we join R (1,000 tuples) and S (1,000) tuples, our join result will be:

a. anywhere from 0 to 1 million tuples
b. 1,000 tuples
c. 10,000 tuples
d. 25,000 tuples

If we are joining three tables, our best strategy is to

e. choose a join order that minimizes the intermediate result
f. choose a join order by random sampling
g. join the first table with the second, then the third
h. choose a join order that maximizes the intermediate result

https://tinyurl.com/cis545-lecture-02-09-22

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-02-09-22

Two Key Ideas

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

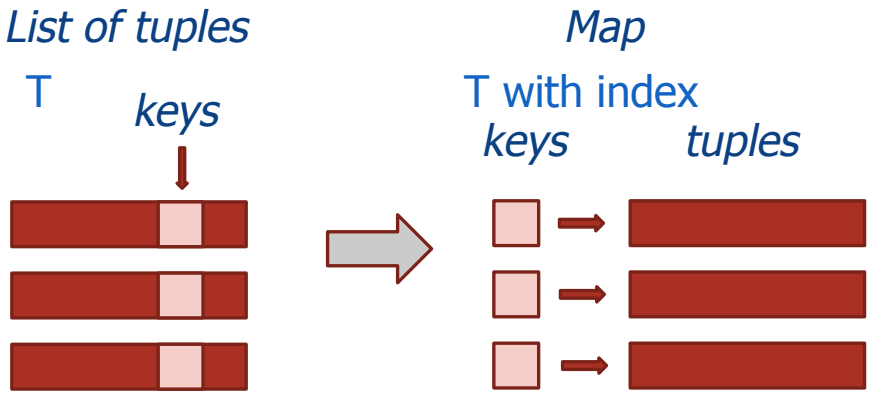
- Now: two ideas for improving performance by changing how we access data
  - Dictionaries / in-memory indices / maps
  - Buffering / blocks

https://tinyurl.com/cis545-lecture-02-09-22

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)?

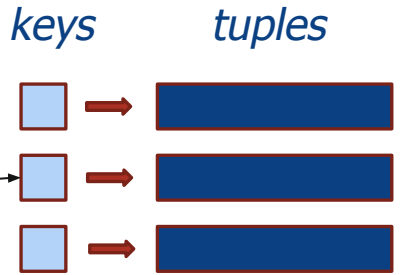
We had a way of doing fast lookups: maps from keys to values (i.e., dicts, indices)



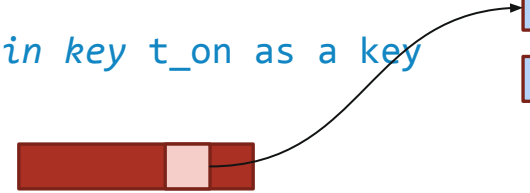
https://tinyurl.com/cis545-lecture-02-09-22

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-02-09-22

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-02-09-22

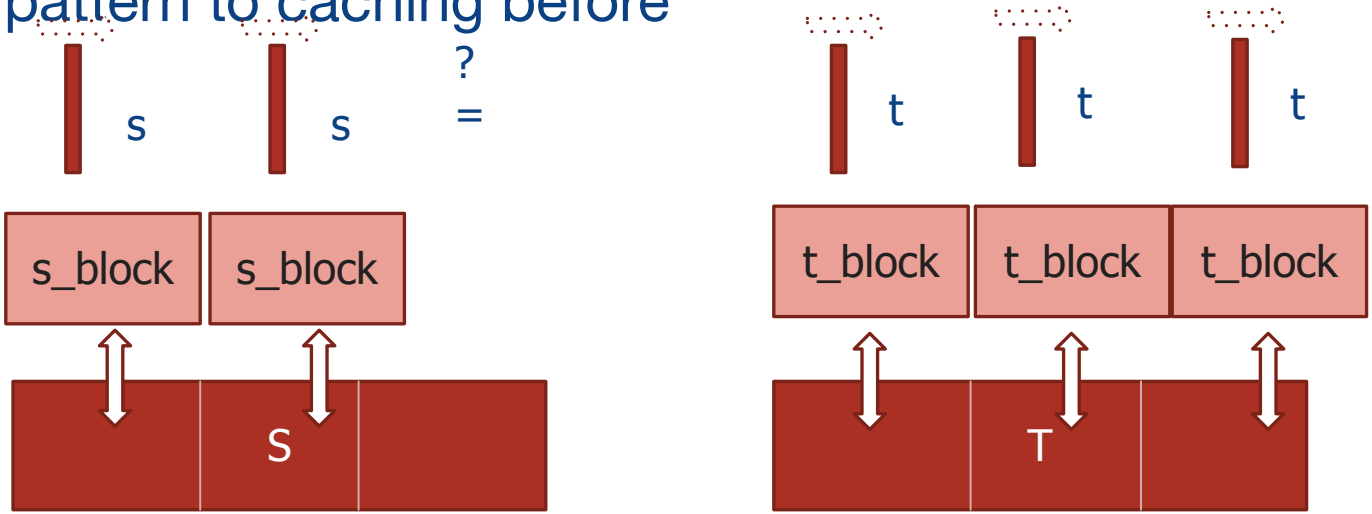
For Further Thought

How does this work if you are joining on more than one field? (first + last name)?

https://tinyurl.com/cis545-lecture-02-09-22

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-02-09-22

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-02-09-22

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-02-09-22

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-02-09-22

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-02-09-22

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-02-09-22

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-02-09-22

Multiple Processors Are Important!

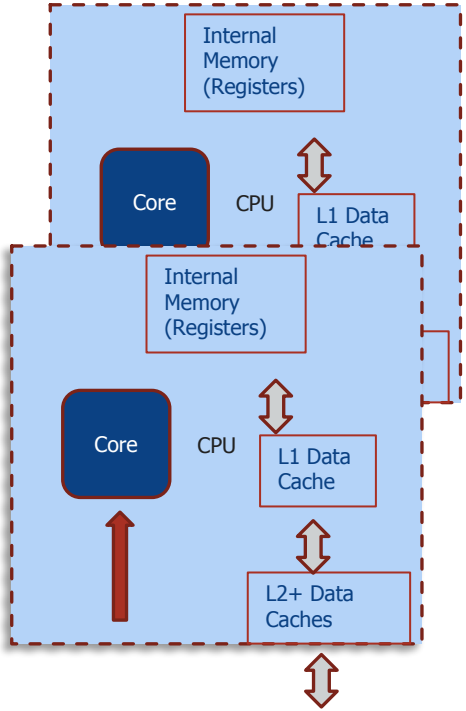
For decades, CPU designers made cores dramatically faster each generation:

- 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-02-09-22



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-02-09-22

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-02-09-22

Multicore Processing of Dataframes

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

https://tinyurl.com/cis545-lecture-02-09-22

Zachary G. Ives

University of Pennsylvania

CIS 545 – Big Data Analytics

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

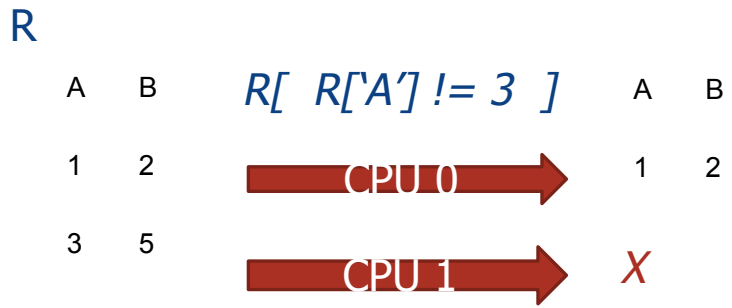
- Not done by Pandas, but there are alternatives

https://tinyurl.com/cis545-lecture-02-09-22

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-02-09-22

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-02-09-22

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-02-09-22

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-02-09-22

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-02-09-22

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-02-09-22

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-02-09-22

Brief Review

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

How does Dask speed up common dataframe operations?

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

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

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

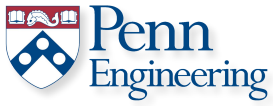
https://tinyurl.com/cis545-lecture-02-09-22

Beyond a Single Machine: Clusters and Data Centers

Susan B. Davidson and Zachary G. Ives

University of Pennsylvania

CIS 545 – Big Data Analytics

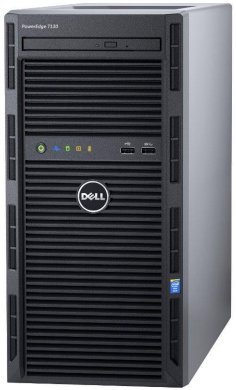


Portions of this lecture have been contributed to the OpenDS4All project, piloted by Penn, IBM, and the Linux Foundation

https://tinyurl.com/cis545-lecture-02-09-22

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

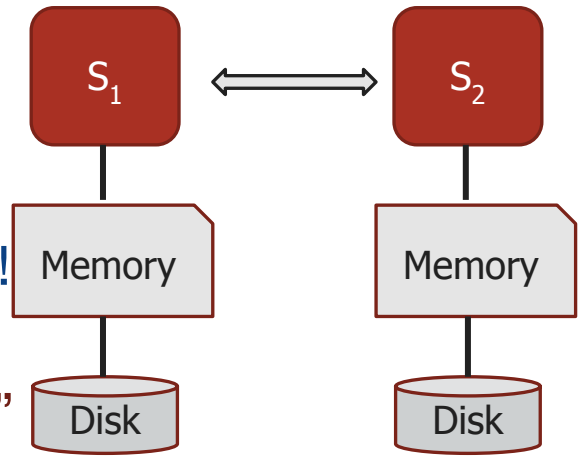
https://tinyurl.com/cis545-lecture-02-09-22

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)

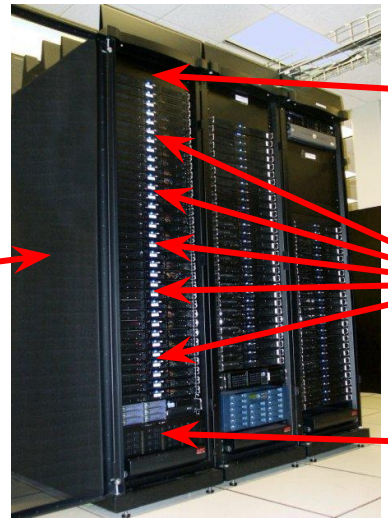
- Same is true if we have multiple computers!



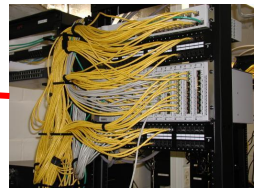
Each processor should work "independently" and "locally" as much as possible

https://tinyurl.com/cis545-lecture-02-09-22

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-02-09-22

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-02-09-22

"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-02-09-22

Google data centers

https://tinyurl.com/cis545-lecture-02-09-22

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

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-02-09-22

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-02-09-22