CSC 369

Introduction to Distributed Computing

MapReduce in a nutshell

Distributed File System.

Data files are stored in a distributed way across cluster

Distributed Execution

Map: process/transform key-value pairs one at a time

Reduce: process/aggregate values with the same key

Automated Shuffle

Shuffle operation turns output of **Map** into input of **Reduce**

Several Enhancements

Combiners, Distributed Cache, ….

Limitations of MapReduce

Only **Map** and **Reduce** operations

- Joins are awkward at best

Data Sharing Between **MapReduce** Jobs

- Only through writing output to disk

Jobs run in batch mode

- Good for “overnight” analysis
- Not so much for exploration of data

Fails then can’t look at the output

But the greatest of them all…

MapReduce is an eager evaluation system

Can’t look at partial results

Eager Evaluation vs. Lazy Evaluation

```
x = 12
y = 10


z = x+y


w = z*(x-y)

d = z *(x+y)

print(d)
```

Eager Evaluation vs. Lazy Evaluation

Eager Evaluation

x	y
12	10

```
x = 12
y = 10

z = x+y

w = z*(x-y)

d = z *(x+y)

print(d)
```

Lazy Evaluation

Eager Evaluation vs. Lazy Evaluation

Eager Evaluation

x	y
12	10

```
x = 12
y = 10

z = x+y

w = z*(x-y)

d = z *(x+y)

print(d)
```

Lazy Evaluation

x	y
12	10

Eager Evaluation vs. Lazy Evaluation

Eager Evaluation

x	y
12	10

x	y	z
12	10	22

```
x = 12
y = 10

z = x+y

w = z*(x-y)

d = z *(x+y)

print(d)
```

Lazy Evaluation

x	y
12	10

Eager Evaluation vs. Lazy Evaluation

Eager Evaluation

x	y
12	10

x	y	z
12	10	22

x	y	z	w
12	10	22	44

```
x = 12
y = 10

z = x+y

w = z*(x-y)

d = z *(x+y)

print(d)
```

Lazy Evaluation

x	y
12	10

Eager Evaluation vs. Lazy Evaluation

Eager Evaluation

x	y
12	10

x	y	z
12	10	22

x	y	z	w
12	10	22	44

x	y	z	w	d
12	10	22	44	484

```
x = 12
y = 10

z = x+y

w = z*(x-y)

d = z *(x+y)

print(d)
```

Lazy Evaluation

x	y
12	10

Eager Evaluation vs. Lazy Evaluation

Eager Evaluation

x	y
12	10

x	y	z
12	10	22

x	y	z	w
12	10	22	44

x	y	z	w	d
12	10	22	44	484

x	y	z	w	d
12	10	22	44	484

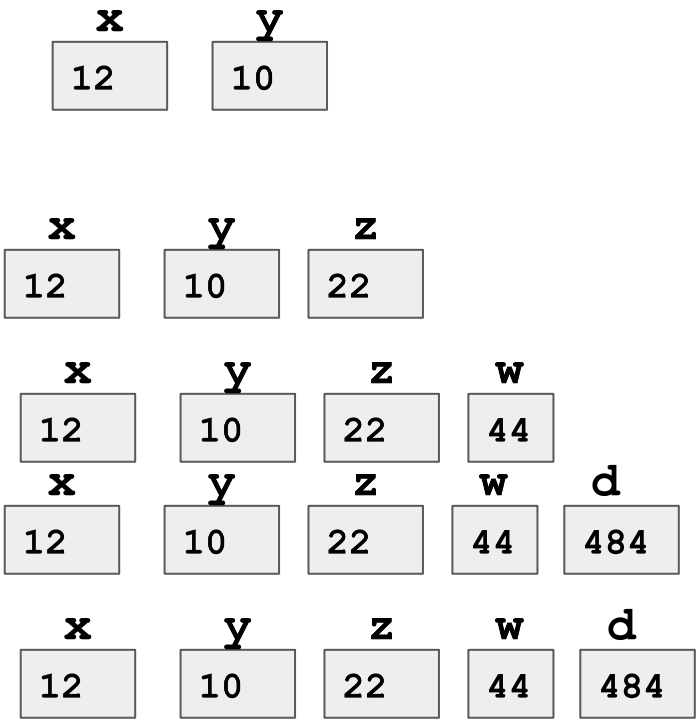
```
x = 12
y = 10

z = x+y

w = z*(x-y)

d = z *(x+y)

print(d)
```

Lazy Evaluation

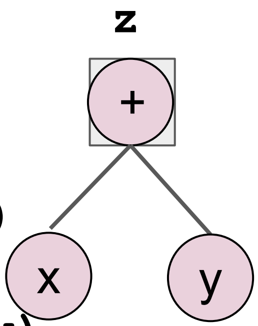
x	y
12	10

Eager Evaluation vs. Lazy Evaluation

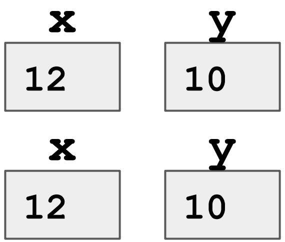
Eager Evaluation




```
x = 12
y = 10

z = x+y

w = z*(x-y)

d = z *(x+y)

print(d)
```

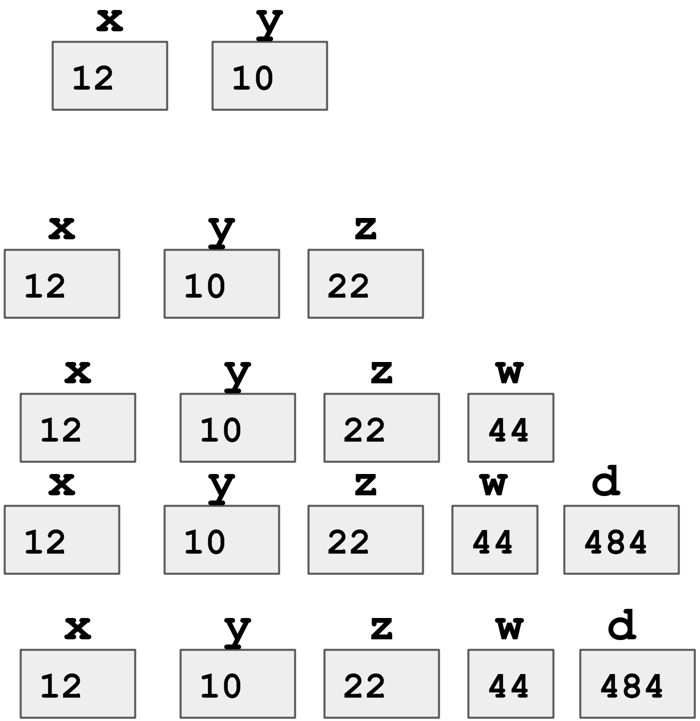


Lazy Evaluation



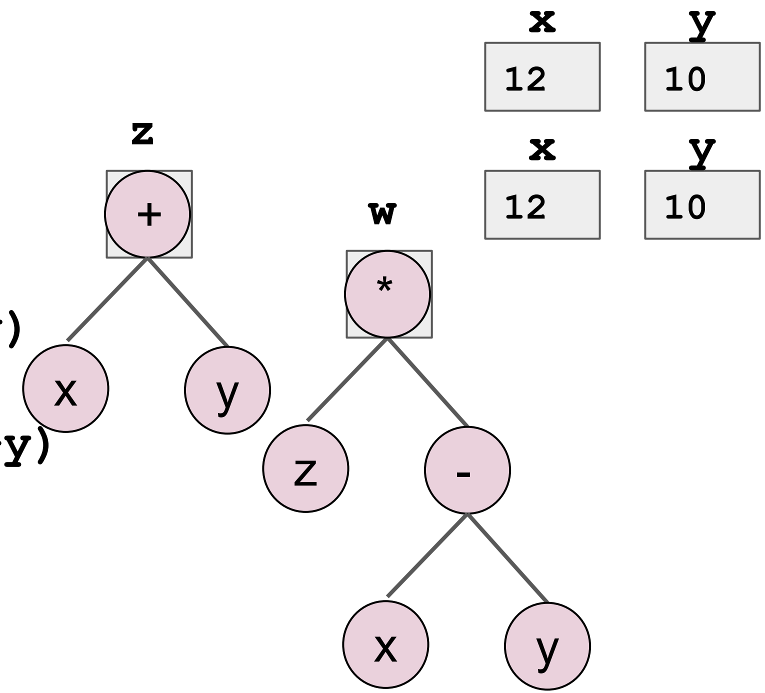
Eager Evaluation vs. Lazy Evaluation

Eager Evaluation



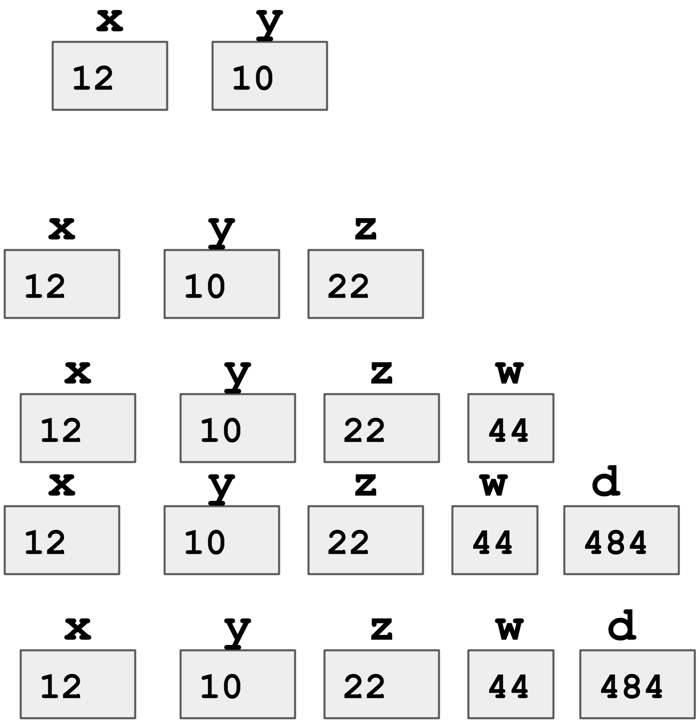

```
x = 12
y = 10

z = x+y

w = z*(x-y)

d = z *(x+y)

print(d)
```

Lazy Evaluation



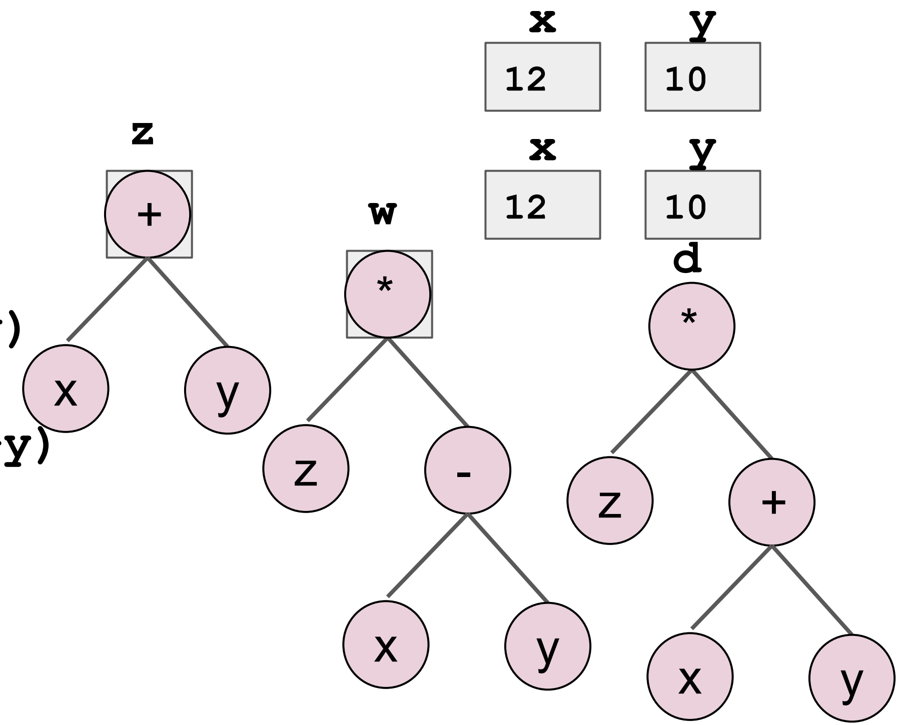
Eager Evaluation vs. Lazy Evaluation

Eager Evaluation



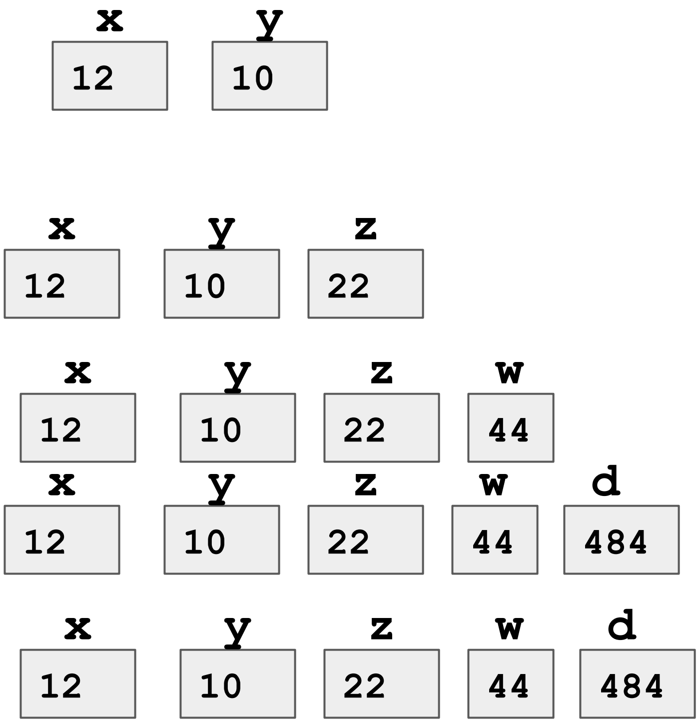

```
x = 12
y = 10

z = x+y

w = z*(x-y)

d = z *(x+y)

print(d)
```

Lazy Evaluation



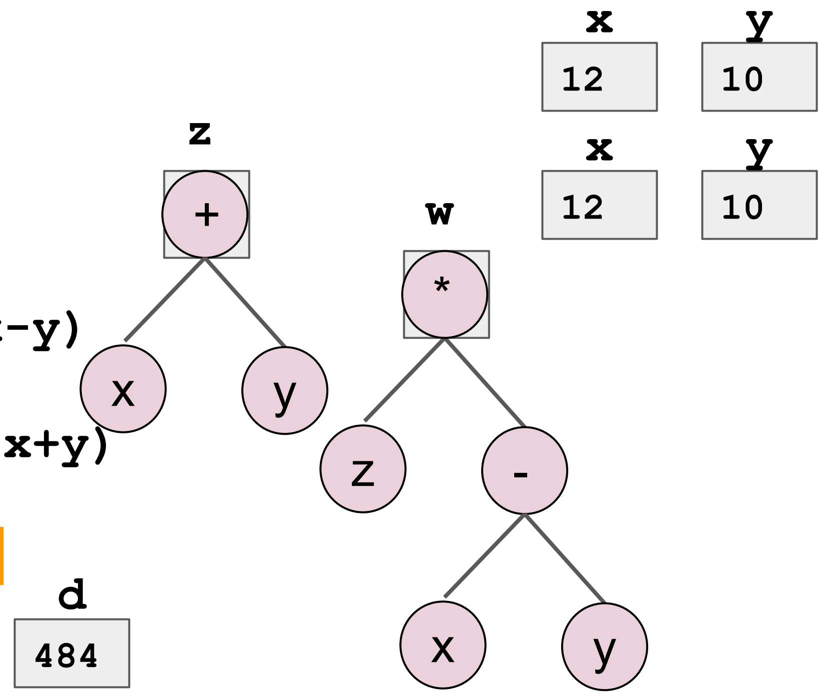
Eager Evaluation vs. Lazy Evaluation

Eager Evaluation




```
x = 12
y = 10

z = x+y

w = z*(x-y)

d = z *(x+y)

print(d)
```

Lazy Evaluation



Benefits of Lazy Evaluation

Lazy: no computation unless necessary

No temporary storage

Interactive computing-friendly

Next Steps - Beyond MapReduce

- Dataflow engines (such as Spark) perform less materialization of intermediate state.

- Underlying assumption of batch processing: input data is **bounded** (known, fixed size). A job knows when it has finished reading the entire input.

- Stream processing involves **unbounded**, never-ending streams of data. Some batch processing concepts carry over, but many assumptions change.

Enter Spark

Spark runs on **Resilient Distributed Datasets (RDDs)**

Resilient Distributed Datasets

Volatile storage: stored **both** in RAM and (when necessary) on disk

Resilience: can be rebuilt in case of cluster partitioning event

Data Transformations: extend **map** and **reduce** with **other** operations

Actions: operations that trigger materialization of RDDs

Lazy Evaluation: RDD content is computed ONLY in response to actions

Interactivity: RDDs as variables that can be extended

Resilient Distributed Dataset

Read-only collections of objects partitioned across a set of machines

Transformations

map - “python” map()
flatMap - MapReduce map() (emits multiple outputs)
mapValues - key-preserving map()
filter - selection
sample - deterministic sample
groupByKey - grouping (w/o aggregation)
reduceByKey - reduce()/aggregation as a transformation
sort - sort
partitionBy - partition the data into splits by a criterion

union - union of two RDDs
join - equijoin by key
cogroup - differently structured “join”
crossproduct - cartesian product

Actions

collect() - materialize/output RDD
reduce() - action version of reduce() (materializes results)
lookup() - report all data for a given key value
collectAsMap() - materialize RDD as a map (dictionary)
count() - return number of objects in RDD
countByKey() - count #occurrences for each key
countByValue() - count #occurrence for each value
first() - first element of RDD
max(), min(), mean(), stdev() - individual statistics
stats() - statistics in a single "report"

countApprox() - approximate (time-limited) count
countApproxDistinct() - approximate (time limited) count of uniques

Transformations

f: *T* => *U*

T,*U* are object "types"

map(*f: T=>U*)

Input: RDD containing objects of type T
Output: RDD containing objects of type U

```
for r in R do
   return f(r)
end for
```

Like map() in Python

Transformations

$$f: T \Rightarrow Sequence(U)$$

flatMap(*f: T=>U*)

Input: RDD containing objects of type T
Output: RDD containing objects of type U

Like MapReduce map()

Transformations

f: *T* => *U*

T,*U* are object "types"

map(*f: T=>U*)

Input: RDD containing objects of type T
Output: RDD containing objects of type U

```
for r in R do
   return f(r)
end for
```

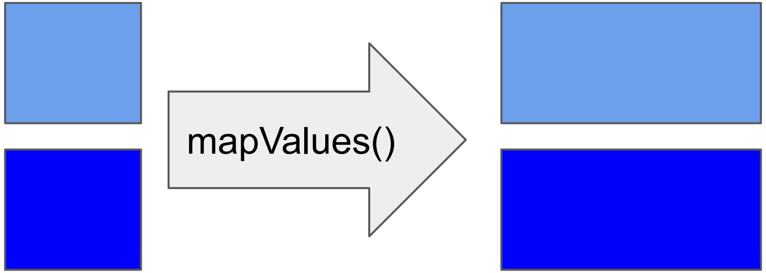
Like map() in Python

Transformations

$f\colon (K,V) \Rightarrow (K,W)$

mapValues$(f\colon (K,V) \Rightarrow (K,W))$

map that keeps the keys



Keeps partitions!

Transformations

map - “python” map()
flatMap - MapReduce map() (emits multiple outputs)
mapValues - key-preserving map()
filter - selection
sample - deterministic sample
groupByKey - grouping (w/o aggregation)
reduceByKey - reduce()/aggregation as a transformation
sort **-** sort
partitionBy - partition the data into splits by a criterion

union - union of two RDDs
join - equijoin by key
cogroup - differently structured “join”
crossporduct - cartesian product