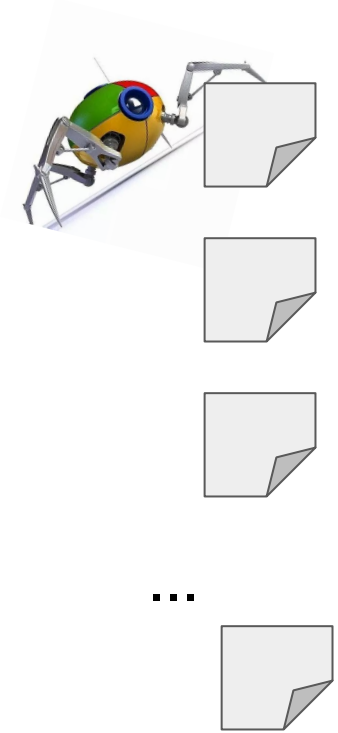
CSC 369

Introduction to Distributed Computing

Motivation: The Google Example

word occurrence example

The World Wide Web:

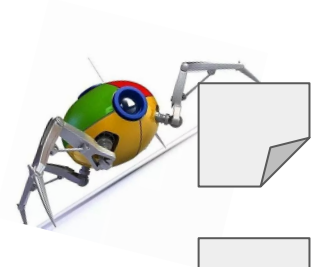


```
Map
(k, v) -> ? -> List((k2, v2))

1 split string on spaces, remove
2 emit(('the', 1))
.
.
.
emit(('the', 1))

shuffle ????
```

```
Reduce
(k2, list(v2) -> ? -> list(v2)

reducer receive
('the', [1,1,1,1…])
.
.
.
1 sum all values
2 print k2
3 emit total
```

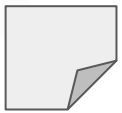
Motivation: The Google Example

The World Wide Web:

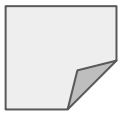


{“Cal”, “Poly”, “San”, “Luis”, “Obispo”,.... }

‘Cal Poly’ [https//…, http//…, …]
‘Map Reduce’ [https//,…]

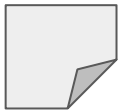


```
input: (k, v) is https://, <html> … <html>
1 parse the page (remove marked up), extract key words
2 for each key word(
emit(‘<word>’, url)
…
```



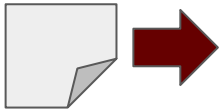
```
(‘Cal Poly’, [url, url2, url3,…])
1 emit input as our inverted index
```

…

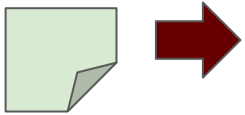


Motivation: The Google Example

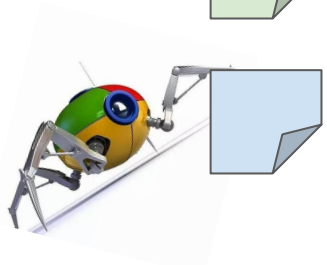
The World Wide Web



{“Cal”, “Poly”, “San”, “Luis”, “Obispo”, “university”.... }



{“Covid-19”, “San”, “Luis”, “Obispo”, “positive”...}



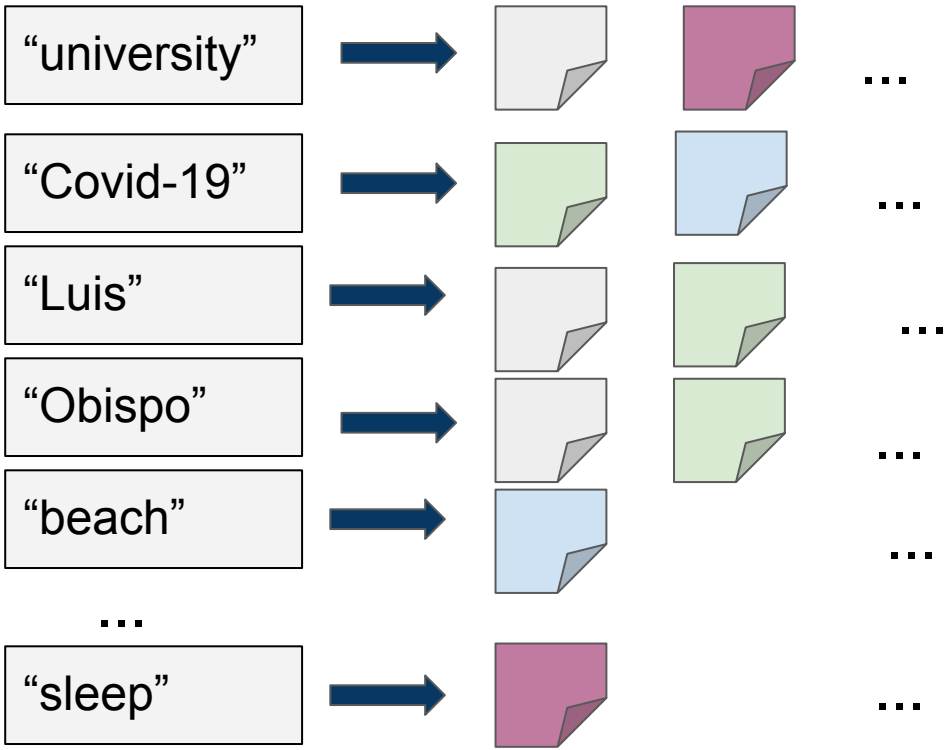
{“Covid-19”, “Newsom”, “beach”, “stay-at-home”...}

...

{“students”, “university”, “on-line”, “classes”, “sleep”

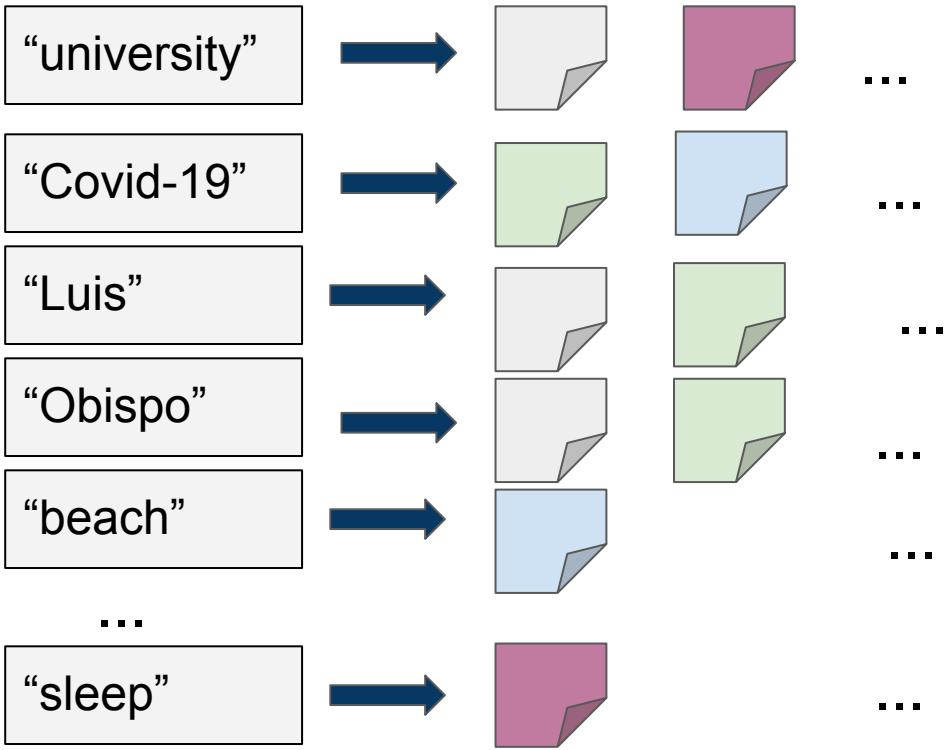
Motivation: The Google Example

Create an **inverted index** that lists each word along with the document(s) in which the word appears



Motivation: The Google Example

The Inverted Index



Challenges

Distributed
(Petabyte scale)

Fast

Simple

MapReduce

Jeffrey Dean, Sanjay Ghemawat, *MapReduce: Simplified Data Processing on Large Clusters*

Noticed that a lot of code of distributed computing kept doing same “types” of things.

Writing correct, efficient, fault-tolerant distributed code is difficult

Proposed a radically simple level of abstraction

Data

<key,value> pairs

Data Processing

<key,value> pairs

Just three types of operations:

Map: <key, value> → <key1, value1>

Shuffle: collect/sort keys

Reduce: <key1, [value1,value2,..,valueN] → <key2, value2>

MapReduce

Write a Map() and Reduce() transformations of data

- Simple code

Build a distributed computing framework that does the rest

MapReduce: Inverted Index

```
Map(key, value):    //key=url, value= bag of words
  for word in value do
    emit(word, key)
  end for
```

```
Reduce(key, values)://key=word, values= [url1,...,urln]
    return(key, values)
```

More Formally: Map()

$$Map: \; K \times V \rightarrow \{K' \times V'\}$$

K, K' -- universes of keys

V, V' -- universes of values

Transformation

More Formally: Map()

$$Map: \; K \times V \rightarrow \{K' \times V'\}$$

K, K' -- universes of keys

V, V' -- universes of values

emit() instead of return()

Transformation

More Formally: Map()

$$Map: K \times V \rightarrow \{K' \times V'\}$$

```
map(key, value):     //value - bag of words
    for word in value:
        emit(word,1)
    end for
```

More Formally: Reduce()

$Reduce: K \times (V)^* \rightarrow (V)^*$

$Reduce\ K \times (V)^* \rightarrow K \times (V)^*$

Aggregation

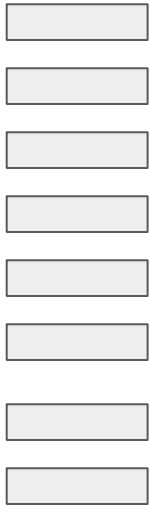
More Formally: Reduce()

Map: K x (V) → (V)**

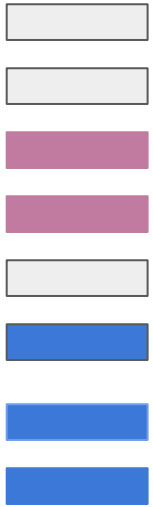
Map: K x (V) → K x (V)**

```
reduce(key, value):     //value - [1,1,1,...,1]
    count := 0
    for x in value:
        count := count+x
    end for
    emit(key, count)
```

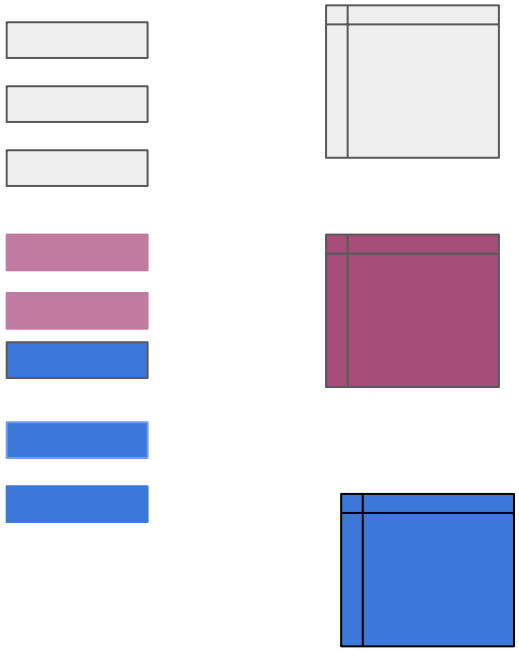
Map-Shuffle-Reduce



Map-Shuffle-Reduce

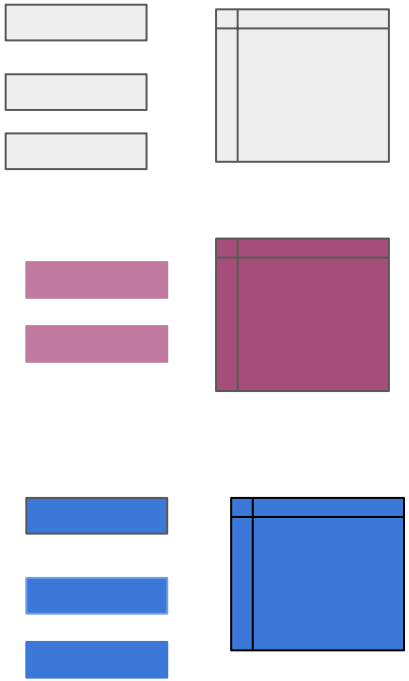


Map-Shuffle-Reduce



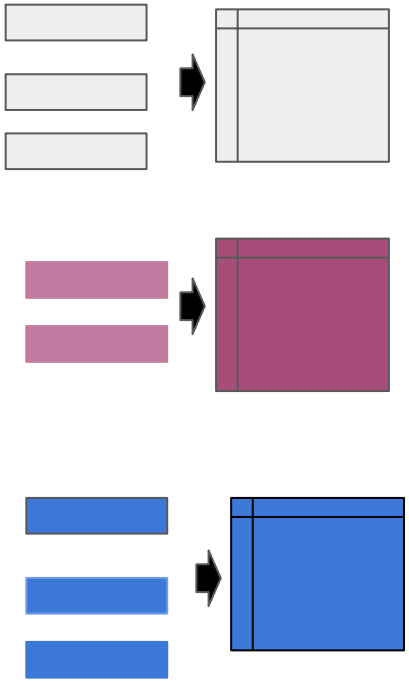
Mappers

Map-Shuffle-Reduce



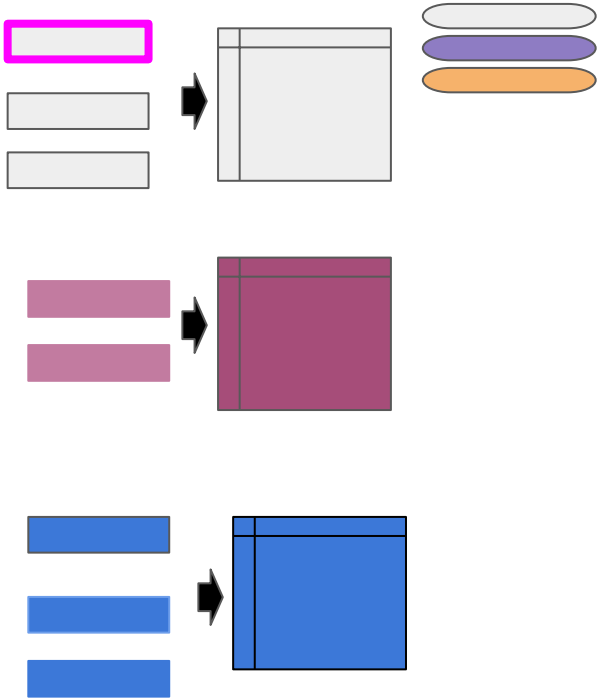
Mappers

Map-Shuffle-Reduce

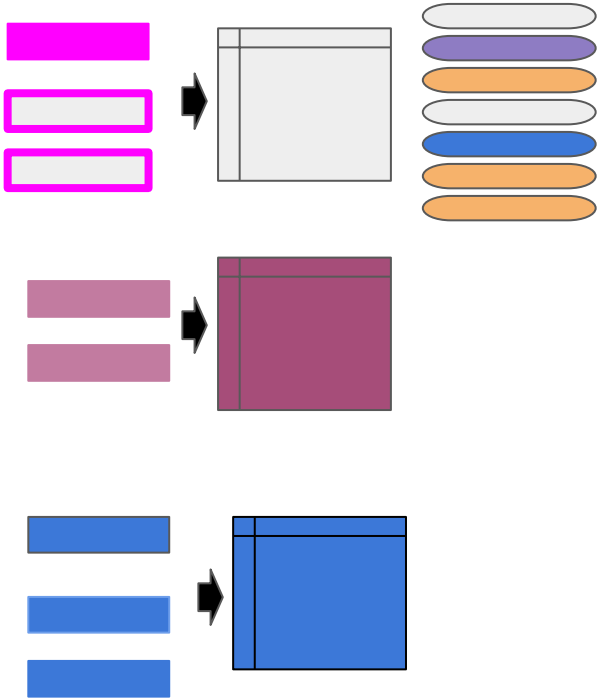


Mappers

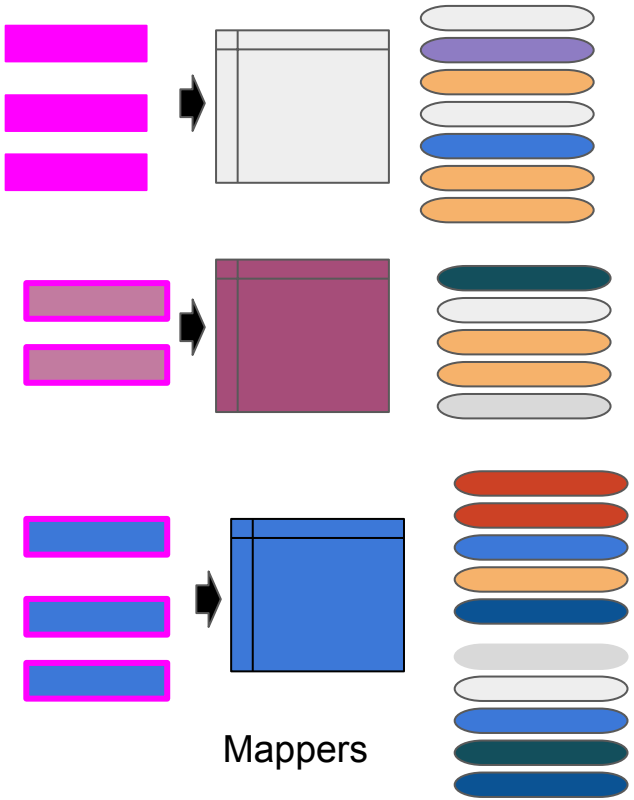
Map-Shuffle-Reduce



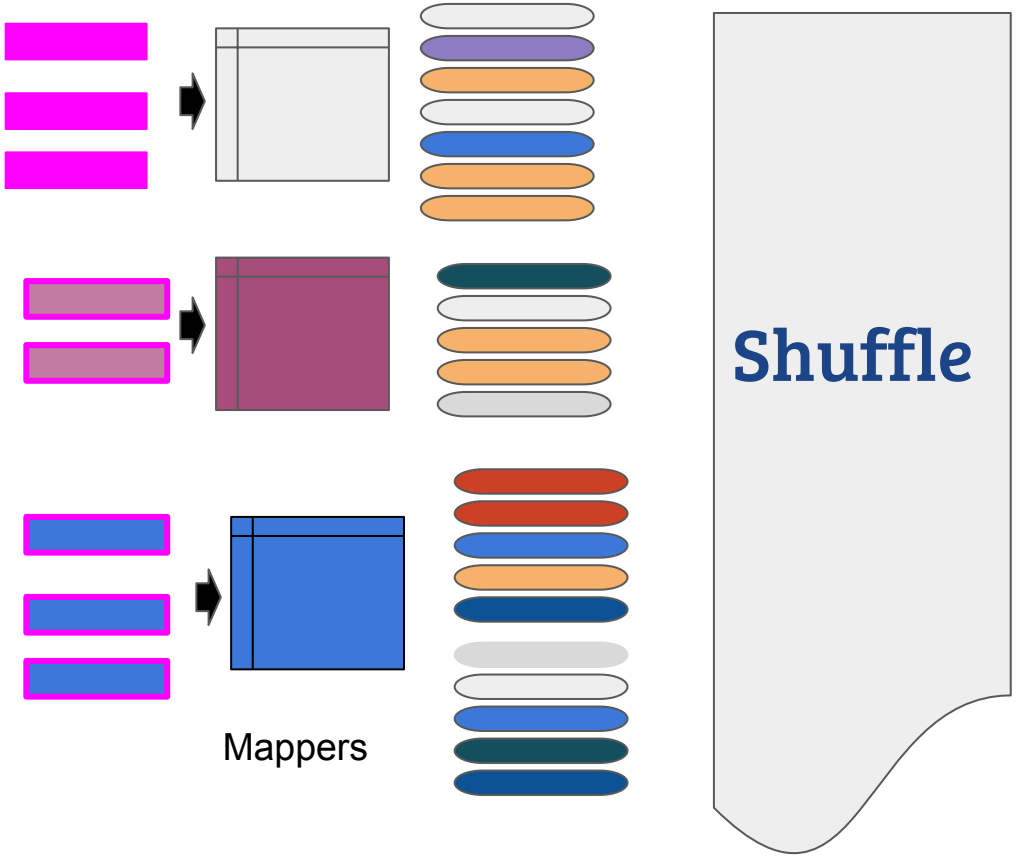
Map-Shuffle-Reduce



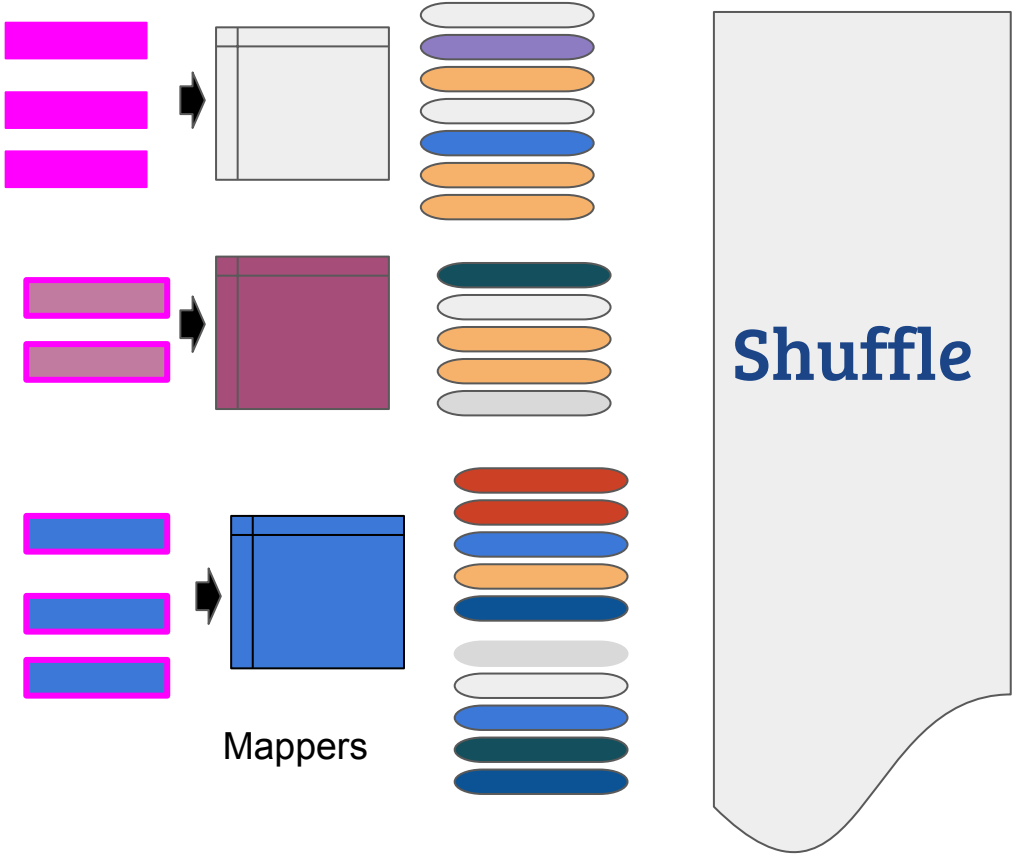
Map-Shuffle-Reduce



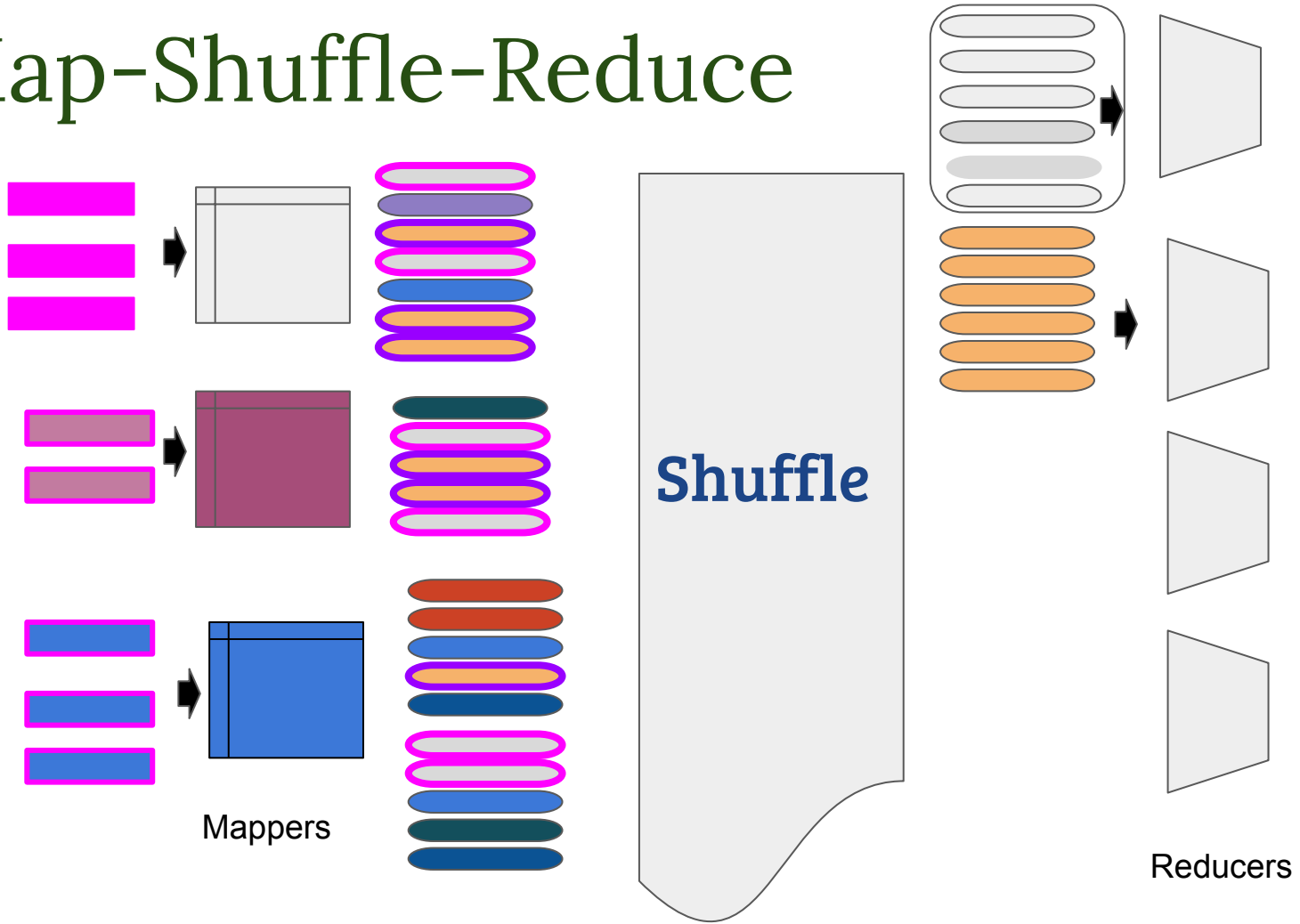
Map-Shuffle-Reduce



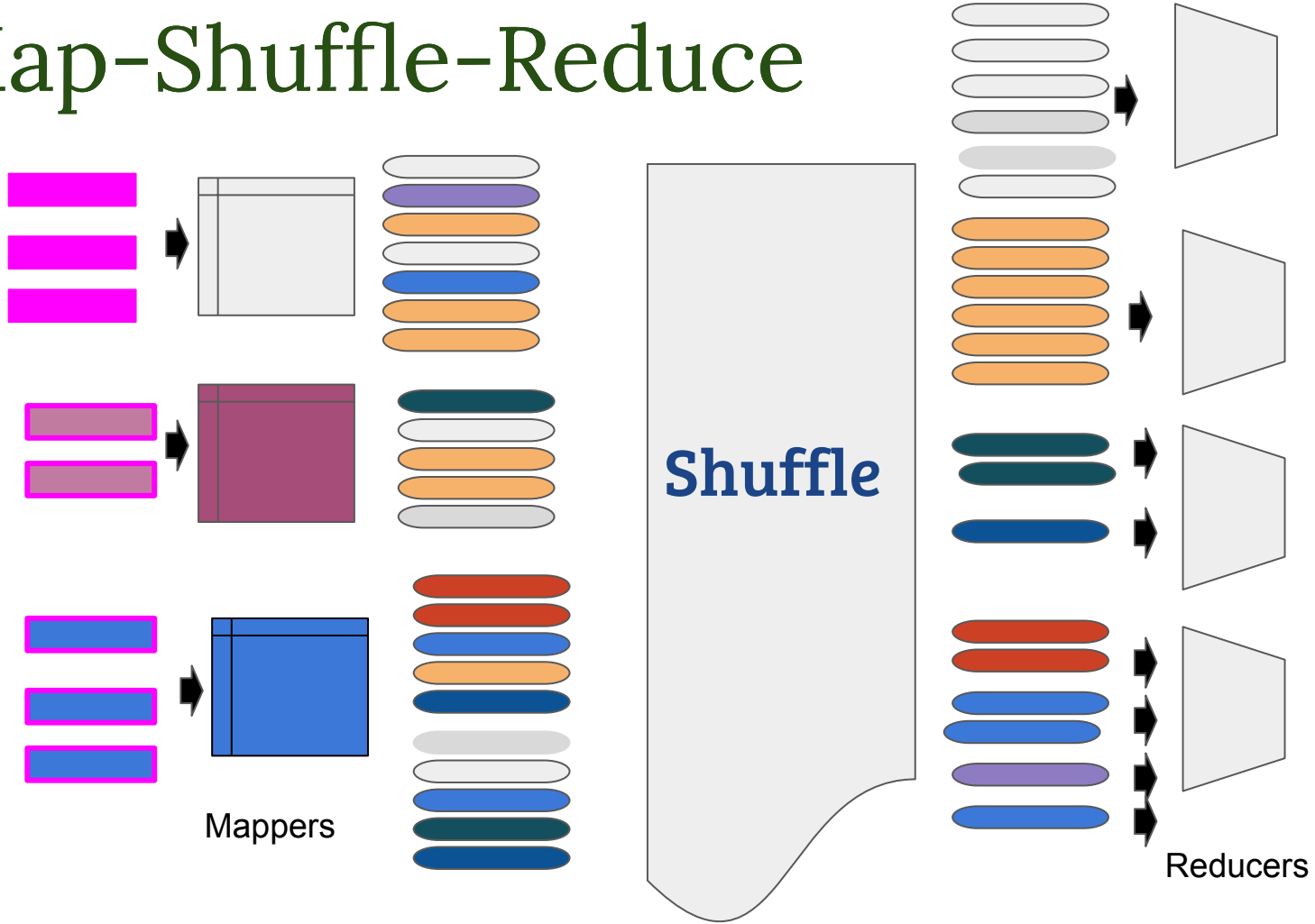
Map-Shuffle-Reduce



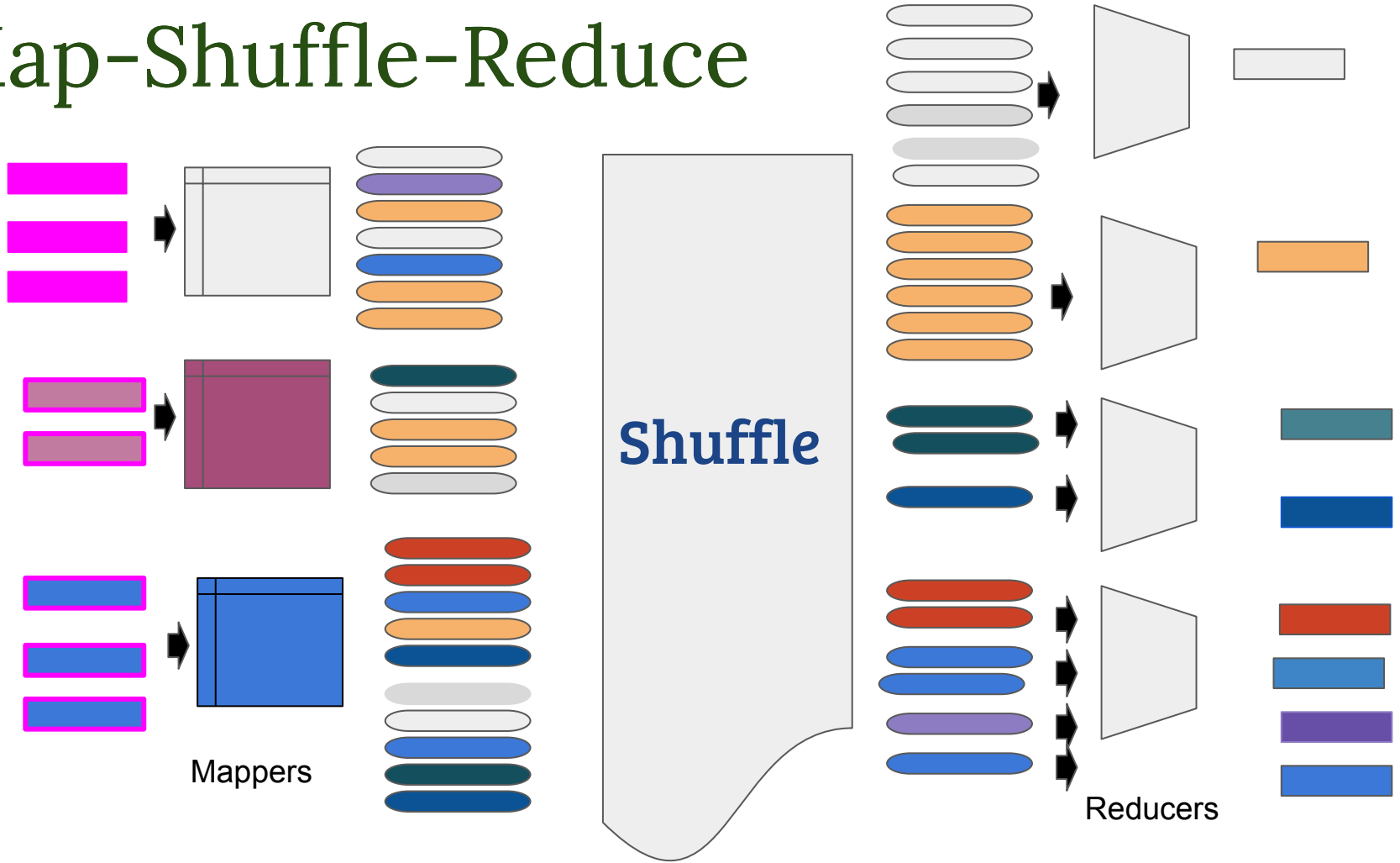
Map-Shuffle-Reduce



Map-Shuffle-Reduce

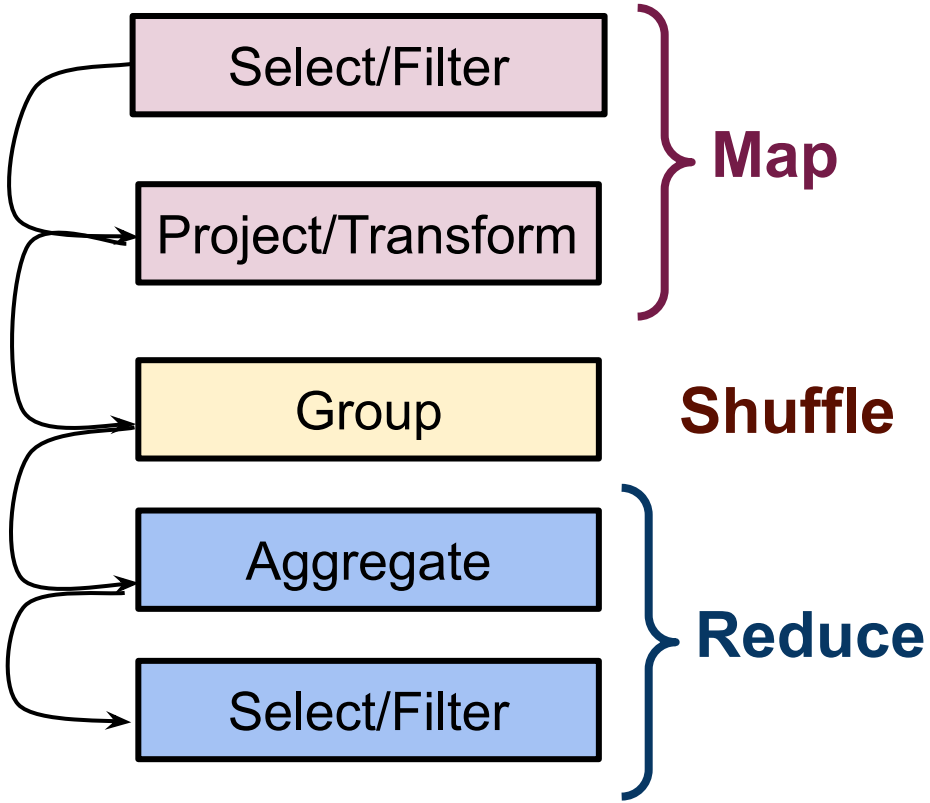


Map-Shuffle-Reduce



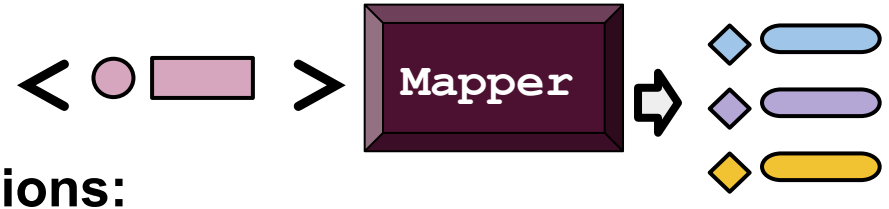
Batch Processing in MapReduce

Data Processing Pipeline



Map and Reduce

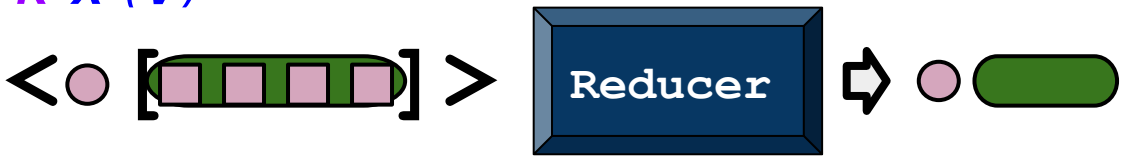
Map: K x V → {K' x V'}



Record-by-record operations:

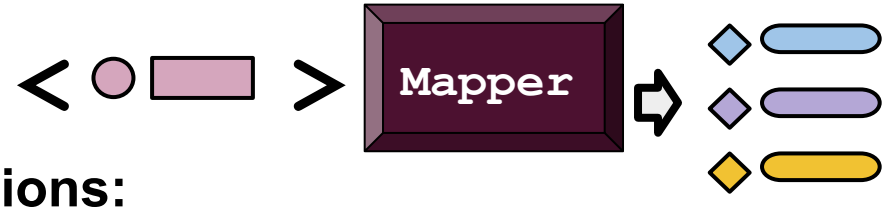
Filter/Selection
Projection

Reduce K x (V) → K x (V)**



Map and Reduce

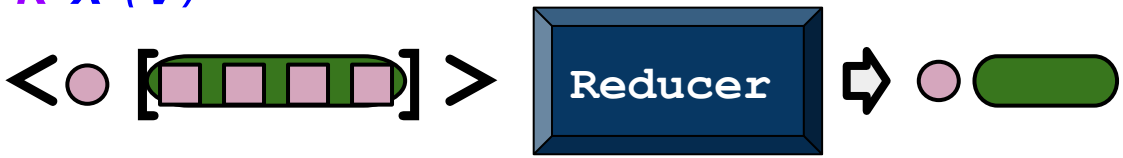
Map: K x V → {K' x V'}



Record-by-record operations:

Filter/Selection
Projection

Reduce K x (V) → K x (V)**



Aggregation

Map and Reduce

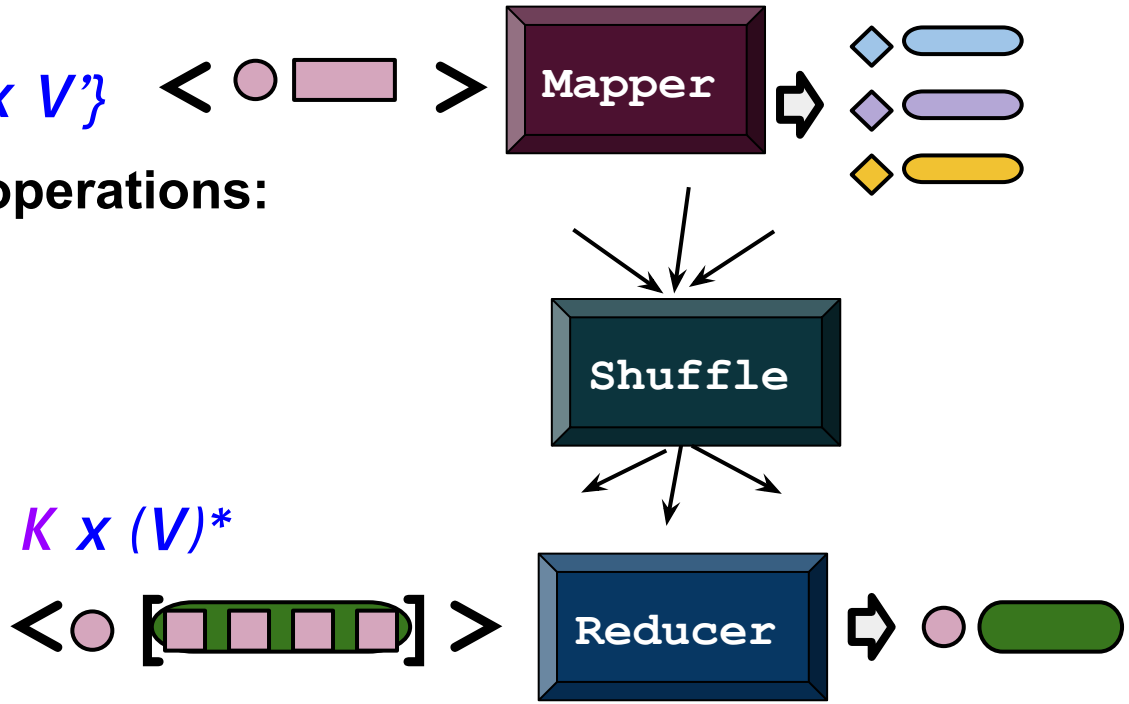
Map: K x V → {K' x V'}

Record-by-record operations:

Filter/Selection
Projection

Reduce K x (V) → K x (V)**

Aggregation



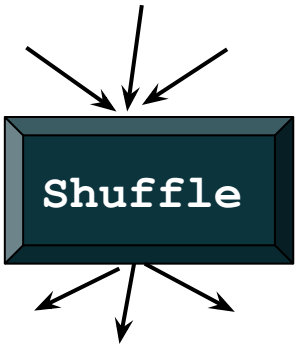
Map and Reduce

Map: K x V → {K' x V'}

Record-by-record operations:

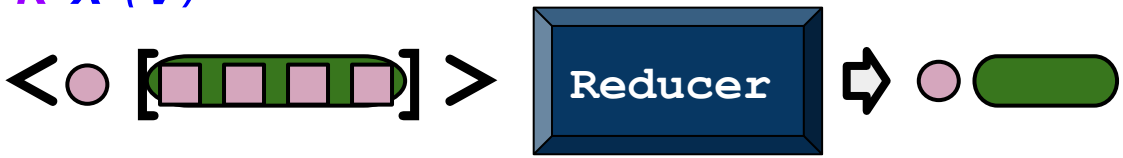
Filter/Selection
Projection

Grouping



Reduce K x (V) → K x (V)**

Aggregation



Resource Manager (YARN)

Distributed Data Store (HDFS)

MapReduce

Other distributed frameworks

Resource Manager (YARN)

Distributed Data Store (HDFS)

Hadoop v1.0

MapReduce
Data Processing
& Resource Management

HDFS
Distributed File Storage

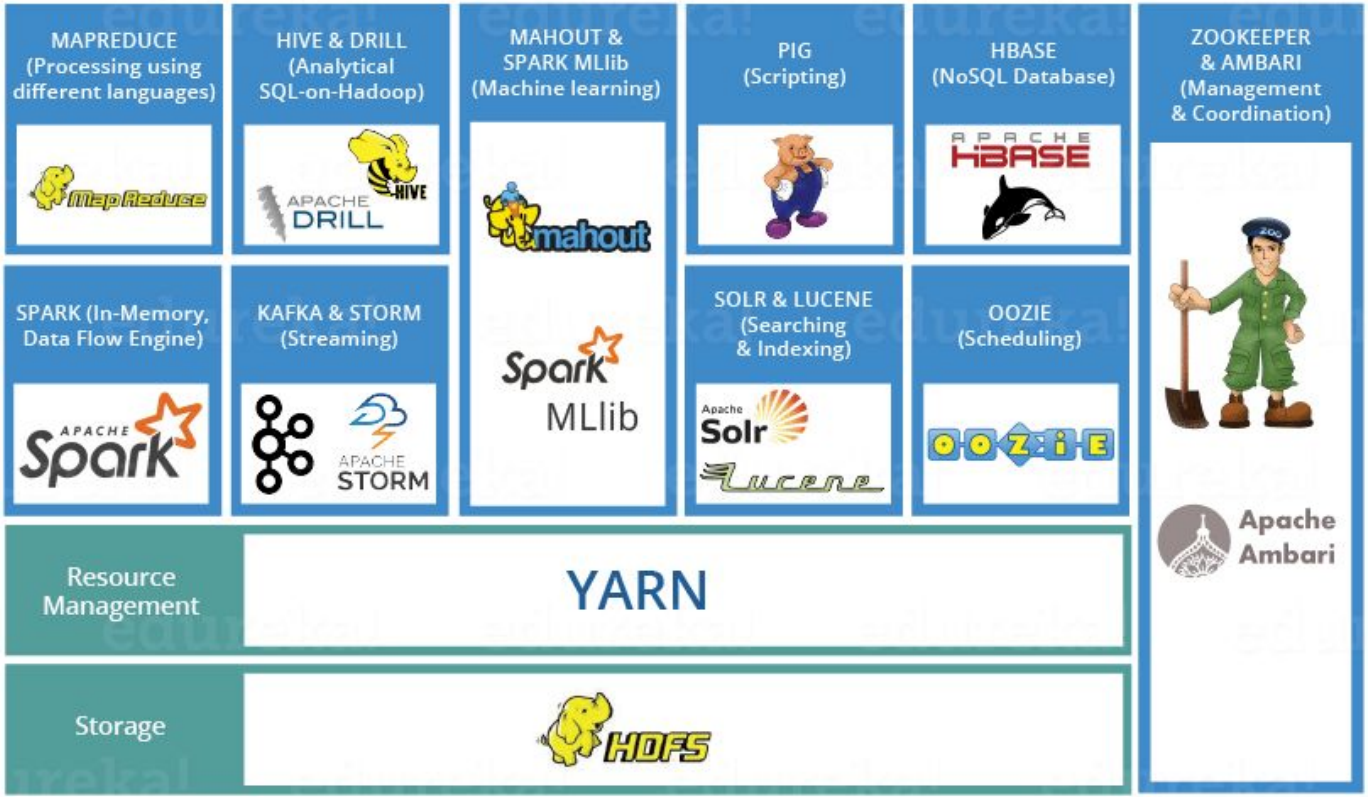
Hadoop v2.0

MapReduce

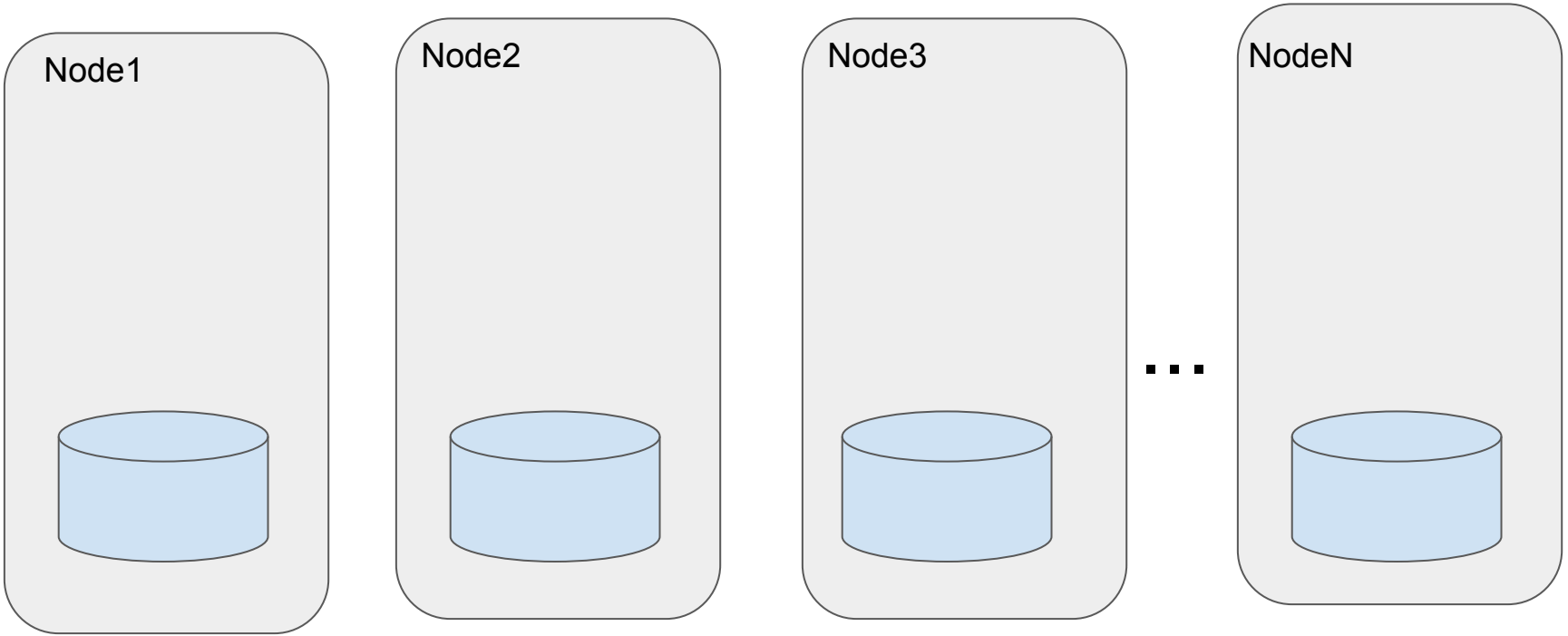
Other Data Processing Frameworks

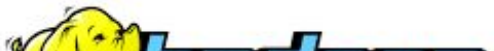
YARN
Resource Management

HDFS
Distributed File Storage

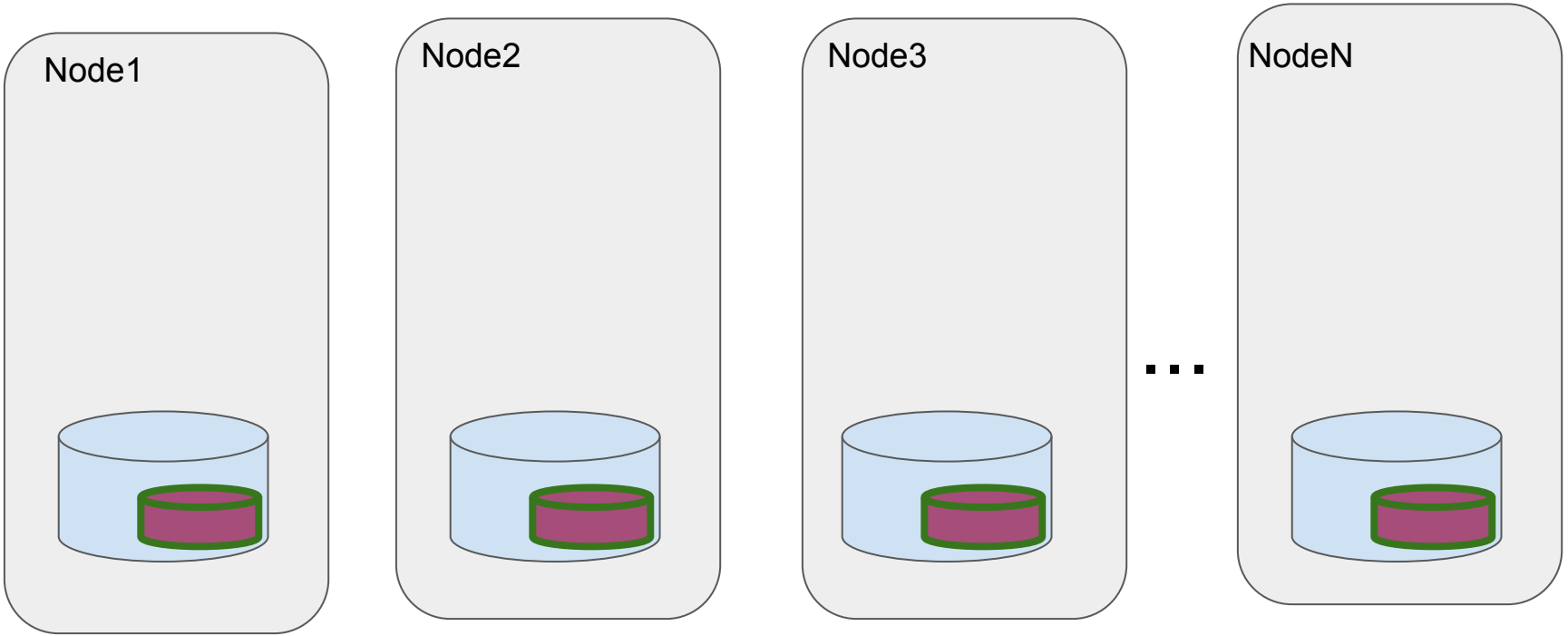

MAPREDUCE (Processing using different languages)
HIVE & DRILL (Analytical SQL-on-Hadoop)
MAHOUT & SPARK MLlib (Machine learning)
PIG (Scripting)
HBASE (NoSQL Database)
ZOOKEEPER & AMBARI (Management & Coordination)
SPARK (In-Memory, Data Flow Engine)
KAFKA & STORM (Streaming)
SOLR & LUCENE (Searching & Indexing)
OOZIE (Scheduling)
Resource Management
YARN
Storage
HDFS
Flume
Unstructured/
Sqoop
Structured Data
Apache Ambari

Distributed Data Store (HDFS)

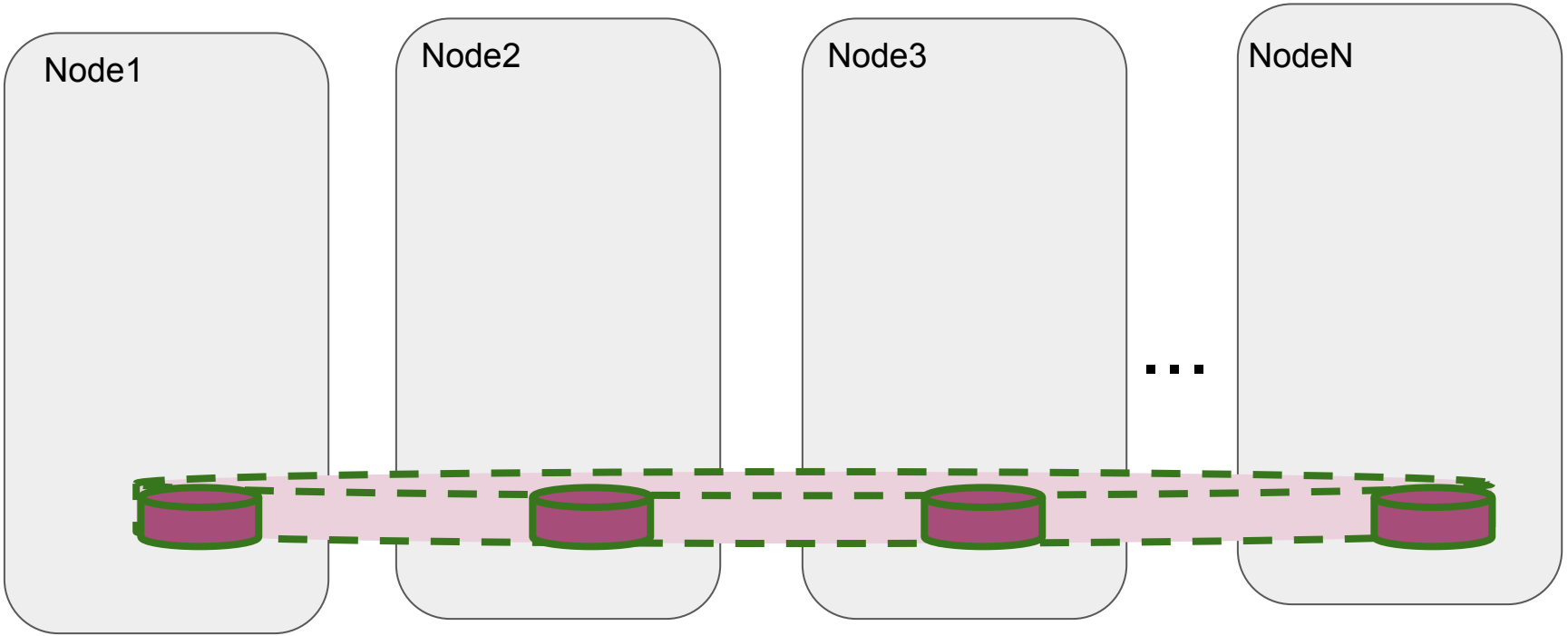




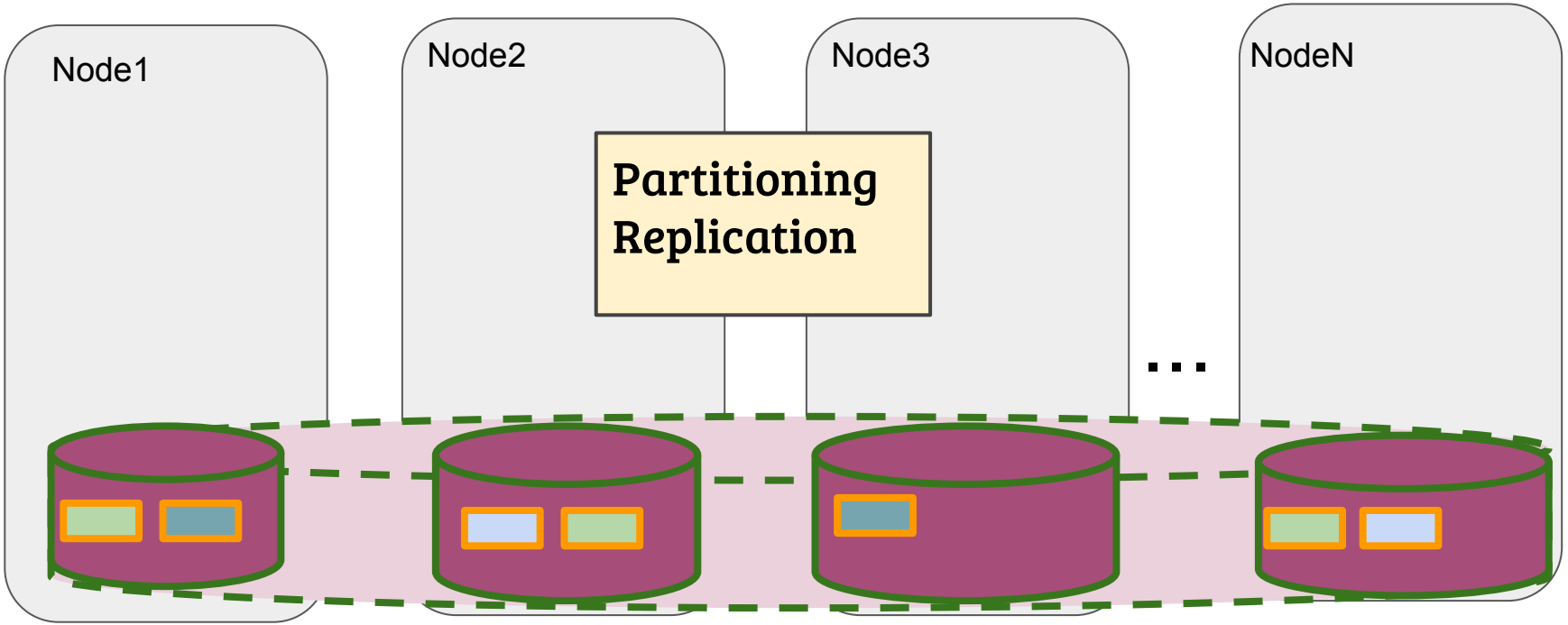
Distributed Data Store (HDFS)



Distributed Data Store (HDFS)



Distributed Data Store (HDFS)



MapReduce Demo

```
Job runner / driver
Mapper
Reducer
```

Find Top Visitors: UNIX / MapReduce / SQL

```
awk '{print $1}' access.log | # Split by whitespace, extract client address (field #1)
  sort     | # Occurrences of the same client address will appear consecutively
  uniq -c  | # Collapse repeated client address into a count
  sort -nr | # Sort (numeric), in reverse (descending) order
  head -n 10 # Show top 10 client addresses (preceded by count of log lines)
```

Map

Shuffle

Reduce

```
SELECT COUNT(*) AS c, client_address
FROM access_log
GROUP BY client_address
ORDER BY c DESC
LIMIT 10
```

Access Log Queries

1. Most popular URL paths
2. Requests count for each HTTP response code, sorted by response code
3. Request count for each calendar month and year, sorted chronologically
4. Total bytes sent to the client with a specified hostname or IPv4 address (you may hard code an address)
5. Based on a given URL (hard coded), compute a request count for each client (hostname or IPv4) who accessed that URL, sorted by request count, highest to lowest