

Map-reduce and Spark

Introduction to Databases

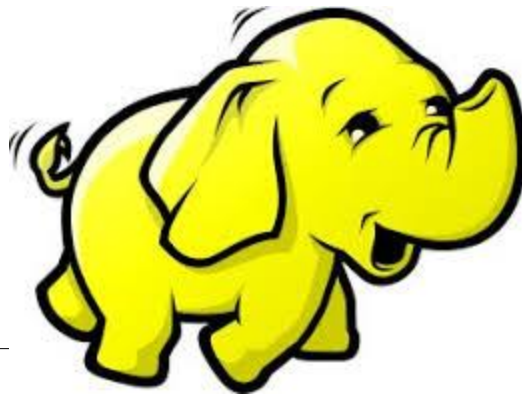
CompSci 316 Spring 2019



DUKE
COMPUTER SCIENCE

Announcements (Tue., Apr. 16)

- **Project demos**—sign-up instructions to be emailed soon
- **Homework #4** final due dates
 - Problem 3: today 04/16
 - Problems 4, 5, 6 : next Monday 04/22
 - Problem X1: next Wednesday 04/24



MapReduce: motivation

- Many problems can be processed in this pattern:
 - Given a lot of unsorted data
 - **Map**: extract something of interest from each record
 - **Shuffle**: group the intermediate results in some way
 - **Reduce**: further process (e.g., aggregate, summarize, analyze, transform) each group and write final results
(Customize map and reduce for problem at hand)
- ☞ Make this pattern easy to program and efficient to run
 - Original Google paper in *OSDI* 2004
 - Hadoop has been the most popular open-source implementation
 - Spark still supports it

M/R programming model

- Input/output: each a collection of key/value pairs
- Programmer specifies two functions
 - $\text{map}(k_1, v_1) \rightarrow \text{list}(k_2, v_2)$
 - Processes each input key/value pair, and produces a list of intermediate key/value pairs
 - $\text{reduce}(k_2, \text{list}(v_2)) \rightarrow \text{list}(v_3)$
 - Processes all intermediate values associated with the same key, and produces a list of result values (usually just one for the key)

Simple Example: Map-Reduce

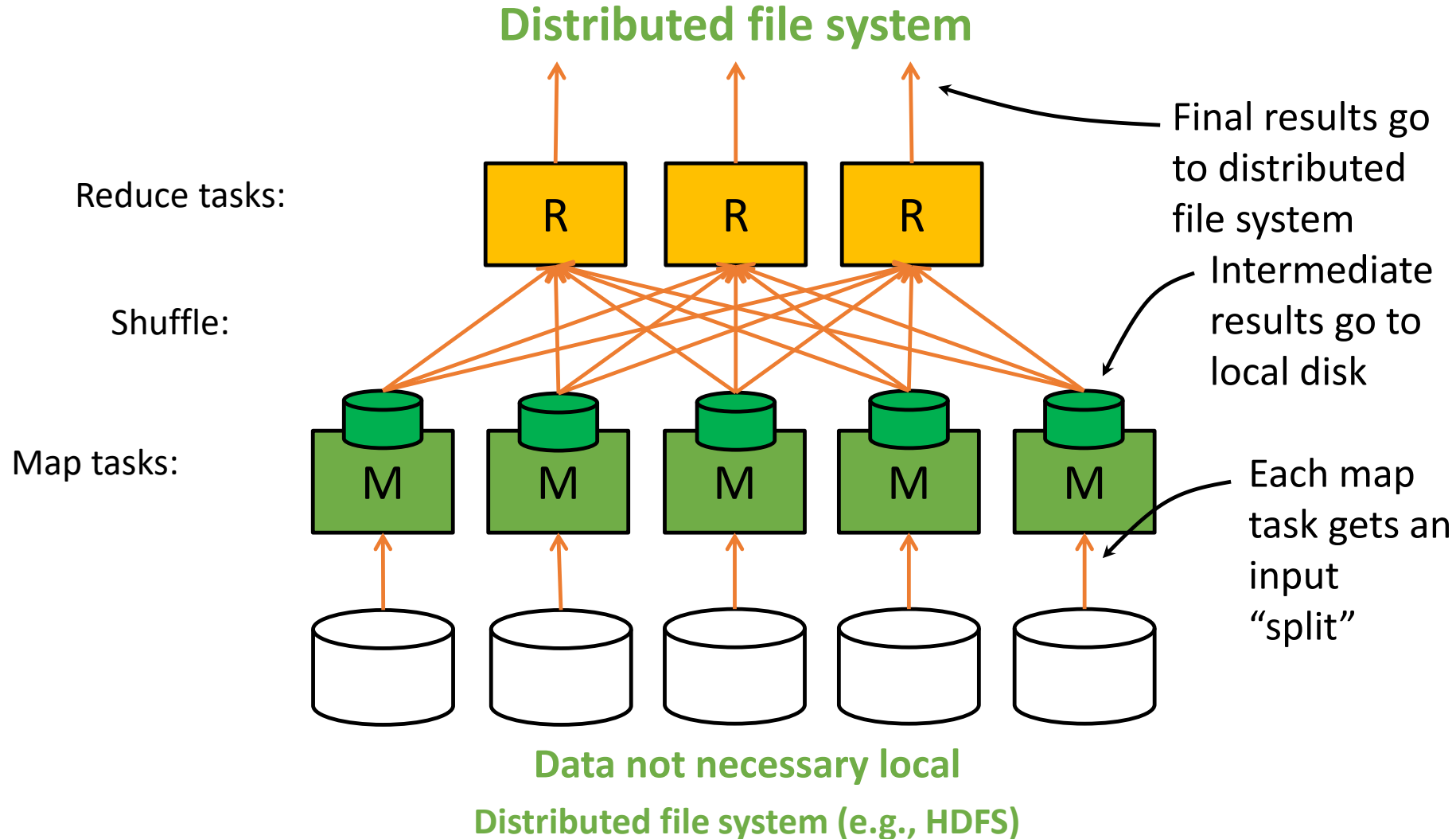
- Word counting
- Inverted indexes

Ack:
Slide by Prof. Shivnath Babu

M/R example: word count

- Expected input: a huge file (or collection of many files) with millions of lines of English text
- Expected output: list of (word, count) pairs
- Implementation
 - $\text{map}(_, \text{line}) \rightarrow \text{list}(\text{word}, \text{count})$
 - Given a line, split it into words, and output $(w, 1)$ for each word w in the line
 - $\text{reduce}(\text{word}, \text{list}(\text{count})) \rightarrow (\text{word}, \text{count})$
 - Given a word w and list L of counts associated with it, compute $s = \sum_{\text{count} \in L} \text{count}$ and output (w, s)
 - Optimization: before shuffling, map can pre-aggregate word counts locally so there is less data to be shuffled
 - This optimization can be implemented in Hadoop as a “combiner”

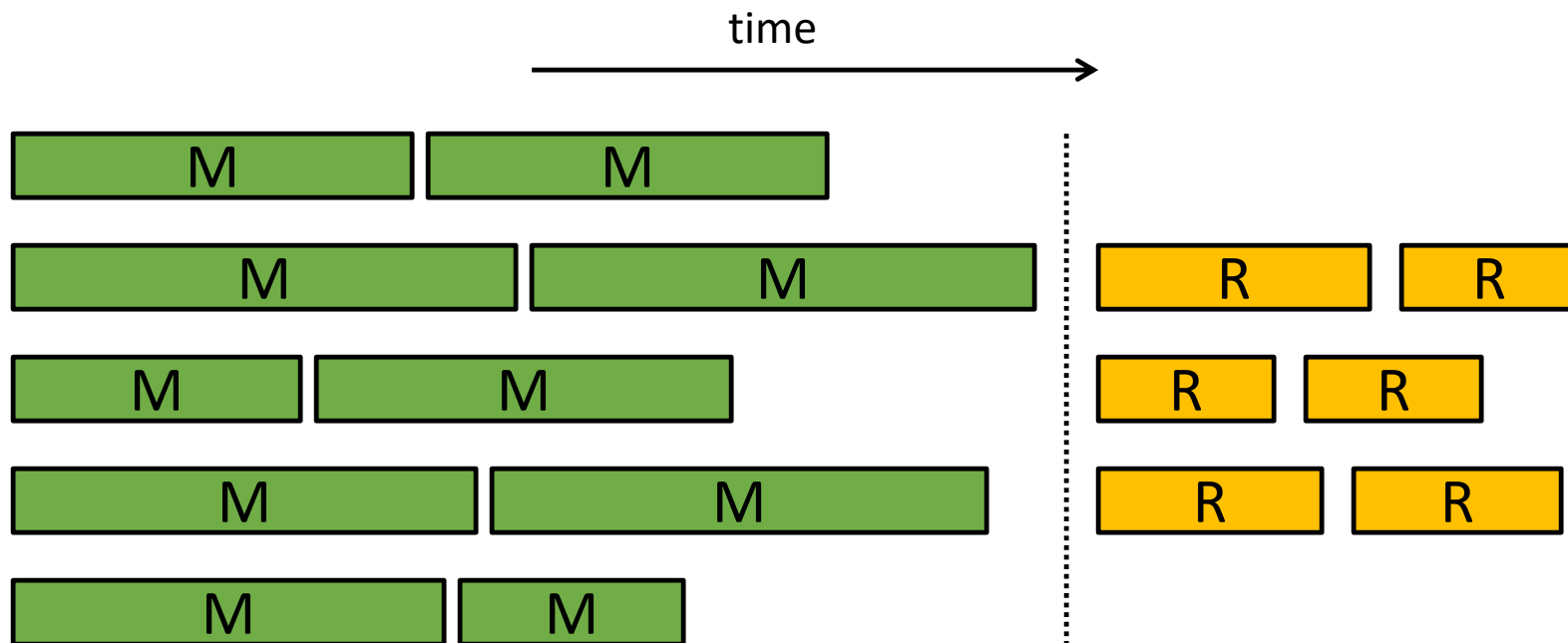
M/R execution



- $$w = 3$$

$$\gamma = 4$$

M/R execution timeline



- When there are more tasks than workers, tasks execute in “waves”
 - Boundaries between waves are usually blurred
- Reduce tasks can't start until all map tasks are done

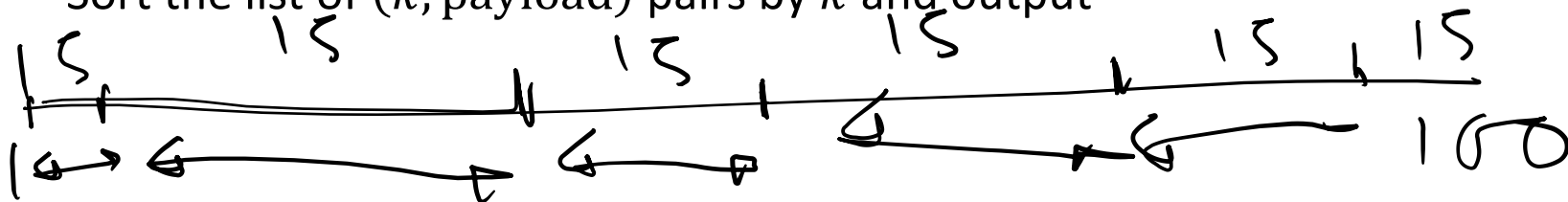
More implementation details

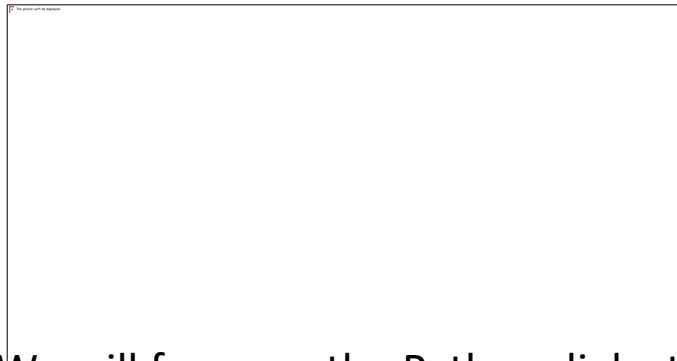
- Numbers of map and reduce tasks
 - Larger is better for load balancing
 - But more tasks add overhead and communication
- Worker failure
 - Master pings workers periodically
 - If one is down, reassign its split/region to another worker
- “Straggler”: a machine that is exceptionally slow
 - Pre-emptively run the last few remaining tasks redundantly as backup

M/R example: Hadoop TeraSort

- Expected input: a collection of (key, payload) pairs
- Expected output: sorted (key, payload) pairs
- Implementation
 - Using a small sample of input, find $r - 1$ key values that divides the key range into r subranges where # pairs is roughly equal across them
 - $\text{map}(k, \text{payload}) \rightarrow (j, \langle k, \text{payload} \rangle)$
 - If k falls within the j -th subrange
 - $\text{reduce}(j, \text{list}(\langle k, \text{payload} \rangle)) \rightarrow \text{list}(k, \text{payload})$
 - Sort the list of $(k, \text{payload})$ pairs by k and output

10^m



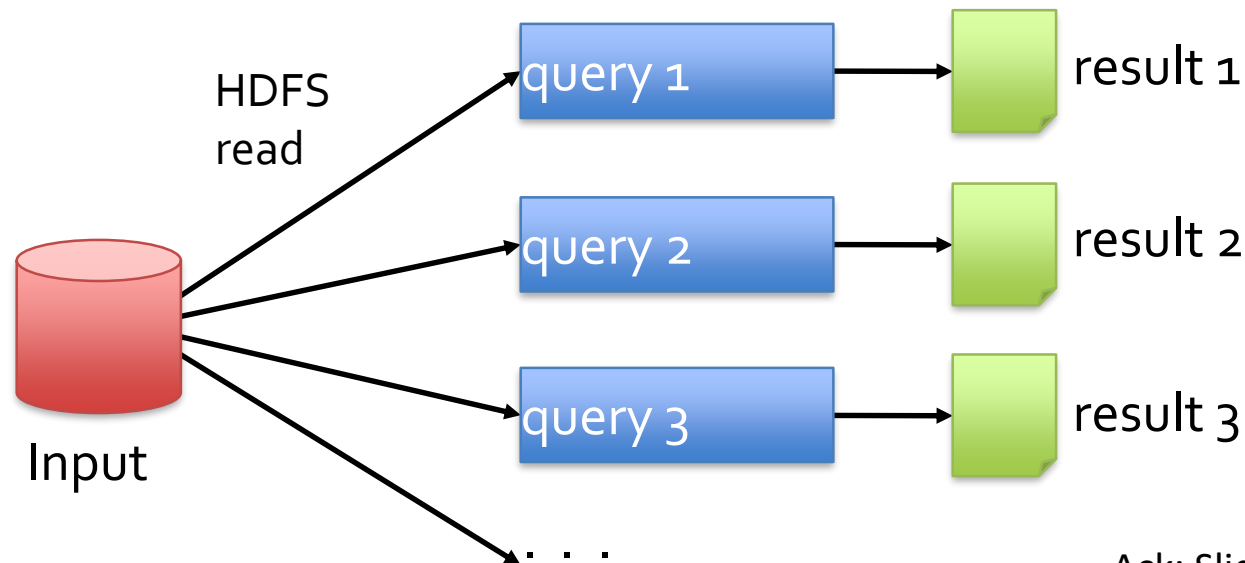
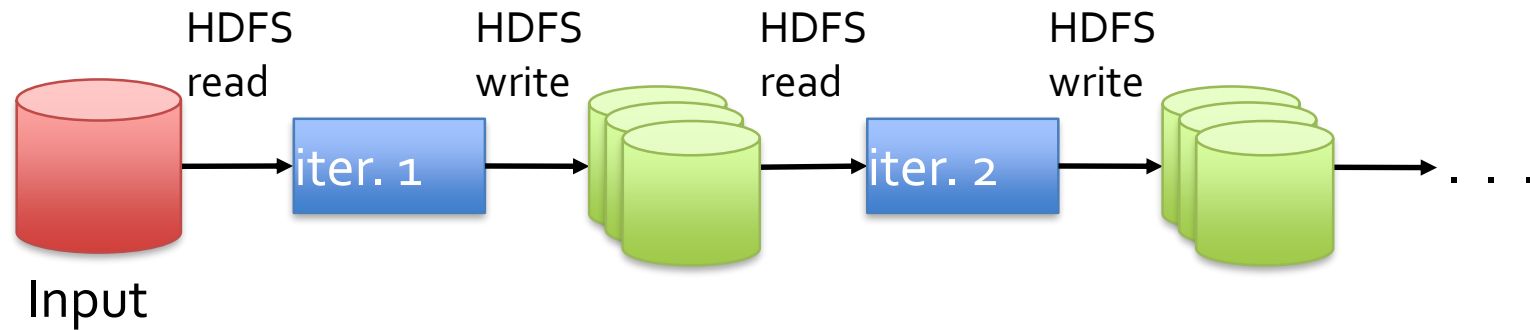


We will focus on the Python dialect,
although Spark supports multiple languages

Why a New Programming Model?

- MapReduce greatly simplified big data analysis
- But as soon as it got popular, users wanted more:
 - More complex, multi-stage **iterative** applications (graph algorithms, machine learning)
 - More **interactive** ad-hoc queries
 - More **real-time** online processing
- All three of these apps require **fast data sharing** across parallel jobs

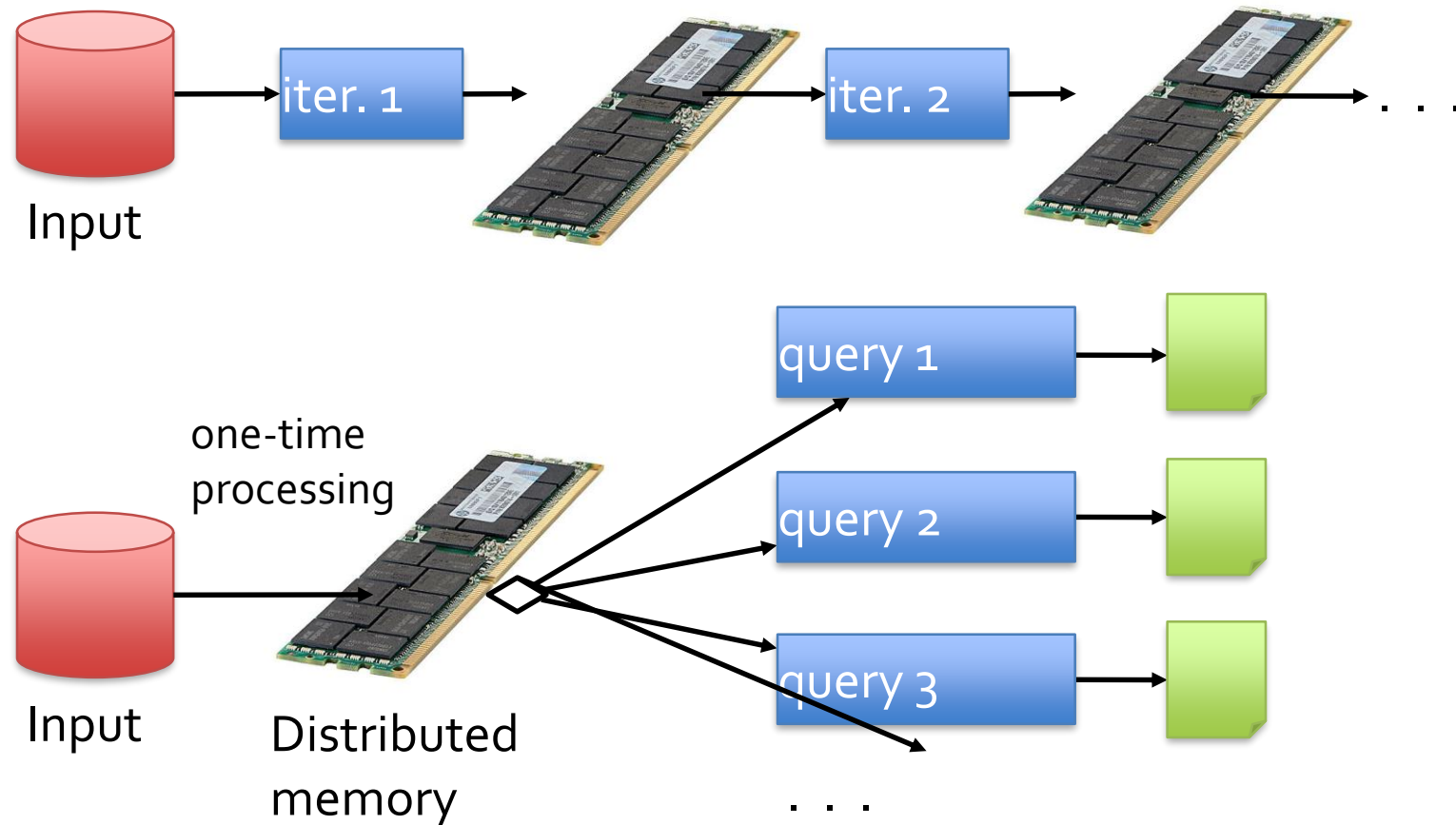
Data Sharing in MapReduce



Ack: Slide by Prajakta Kalmegh

Slow due to replication, serialization, and disk IO

Data Sharing in Spark



10-100× faster than network and disk

Ack: Slide by Prajakta Kalmegh

Addressing inefficiencies in Hadoop

- Hadoop: no automatic optimization

☞ Spark

- Allow program to be a DAG of DB-like operators, with less reliance on black-box code
 - Delay evaluation as much as possible
 - Fuse operators into stages and compile each stage
- Hadoop: too many I/Os
 - E.g., output of each M/R job is always written to disk
 - But such checkpointing simplifies failure recovery

☞ Spark

- Keep intermediate results in memory
- Instead of checkpointing, use “lineage” for recovery

RDDs

- Spark stores all intermediate results as **Resilient Distributed Datasets (RDDs)**
 - Immutable, memory-resident, and distributed across multiple nodes
- Spark also tracks the “lineage” of RDDs, i.e., what expressions computed them
 - Can be done at the partition level

What happens to a RDD if a node crashes?

- The partition of this RDD on this node will be lost
- But with lineage, the master simply recomputes the a lost partition when needed
 - Requires recursive recomputation if input RDD partitions are also missing

Example: votes & explanations

- Raw data reside in lots of JSON files obtained from ProPublica API
- Each vote: URI (id), question, description, date, time, result
- Each explanation: member id, name, state, party, vote URI, date, text, category
 - E.g., “P000523”, “David E. Price”, “NC”, “D”,
“<https://api.propublica.org/congress/v1/115/house/sessions/2/votes/269.json>”, “2018-06-20”, “Mr. Speaker, due to adverse weather and numerous flight delays and cancellations in North Carolina, I was unable to vote yesterday during Roll Call 269, the motion...”, “Travel difficulties”

Basic M/R with Spark RDD


```
explain_fields = ('member_id', 'name', 'state', 'party', 'vote_api_uri',
                  'date', 'text', 'category')
```

Map function: $\text{map}(k_1, v_1) \rightarrow \text{list}(k_2, v_2)$

```
def rdd_count_by_category_map(record):
    if len(record) == len(explain_fields):
        return [(record[explain_fields.index('category')], 1)]
    else:
        return []
```

Reduce function: $\text{reduce}(k_2, \text{list}(v_2)) \rightarrow \text{list}(v_3)$

```
def rdd_count_by_category_reduce(record):
    key, vals = record
    return [(key, len(vals))]
```



Basic M/R with Spark RDD

setting up one RDD that contains all the input:

rdd = sc. ...

count number of explanations by category; order by

number (descending) and then category (ascending):

result = rdd\

```
.flatMap(rdd_count_by_category_map)\
  .groupByKey()\
  .flatMap(rdd_count_by_category_reduce)\
  .sortBy(lambda x: (-x[1], x[0]))
```

for row in result collect():

print(' '.join(str(field) for field in row))

!-many flatmap
1-1 map.

Be lazy: build up a DAG of "transformations," but no evaluation yet!

Optimize & evaluate the whole DAG only when needed, e.g., triggered by "actions" like collect()

Be careful: Spark RDDs support map() and reduce() too, but they are not the same as those in MapReduce

Moving “BD” to “DB”

Each element in a RDD is an opaque object—hard to program

- Why don’t we make each element a “row” with named columns—easier to refer to in processing
 - RDD becomes a *DataFrame* (name from the *R* language)
 - Still immutable, memory-resident, and distributed
- Then why don’t we have database-like operators instead of just MapReduce?
 - Knowing their semantics allows more optimization
- Spark in fact pushed the idea further
 - Spark *Dataset* = DataFrame with type-checking
 - And just run SQL over Datasets using *SparkSQL*!

Spark DataFrame

```
from pyspark.sql import functions as F
```

```
explain_fields = ('member_id', 'name', 'state', 'party', 'vote_api_uri',  
                  'date', 'text', 'category')
```

```
# setting up a DataFrame of explanations:
```

```
df_explain = sc. ...
```

```
# count number of explanations by category; order by
```

```
# number (descending) and then category (ascending):
```

```
df_explain.groupBy('category')\  
    .agg(F.count('name'))\  
    .withColumnRenamed('count(name)', 'count')\  
    .sort(['count', 'category'], ascending=[0, 1])\  
    .show(10000, truncate=False)
```


Another Spark DataFrame Example

```
from pyspark.sql import functions as F

vote_fields = ('vote_uri', 'question', 'description', 'date', 'time', 'result')

explain_fields = ('member_id', 'name', 'state', 'party', 'vote_api_uri',
                  'date', 'text', 'category')
```

Check yourself

setting up DataFrames for each type of data:

```
df_votes = sc. ...
```

```
df_explain = sc. ...
```

what does the following do?

For each vote, find out which legislators provided explanations; order by the number of such legislators (descending), then date and time (descending)

```
df_votes.join(df_explain.select('vote_api_uri', 'name'),
              df_votes.vote_uri == df_explain.vote_api_uri, 'left_outer')\
.groupBy('vote_uri', 'date', 'time', 'question', 'description', 'result')\
.agg(F.count('name'), F.collect_list('name'))\
.withColumnRenamed('count(name)', 'count')\
.withColumnRenamed('collect_list(name)', 'names')\
.sort(['count', 'date', 'time'], ascending=[0, 0, 0])\
.select('vote_uri', 'date', 'time', 'question', 'description', 'result',
        'count', 'names')\
.show(20, truncate=False)
```