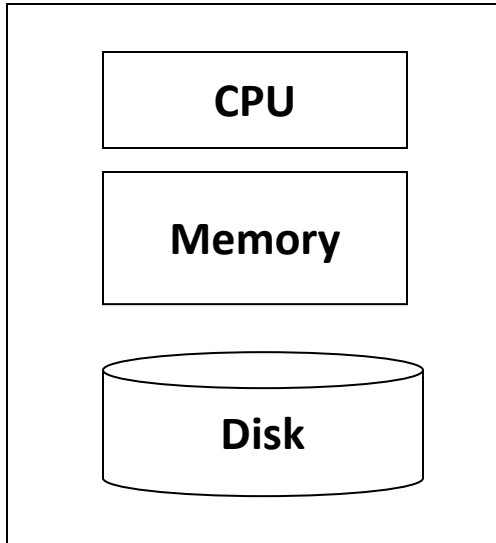


Big Data Systems

Hadoop and beyond

Classical Data Mining



Machine Learning, Statistics

Data Mining

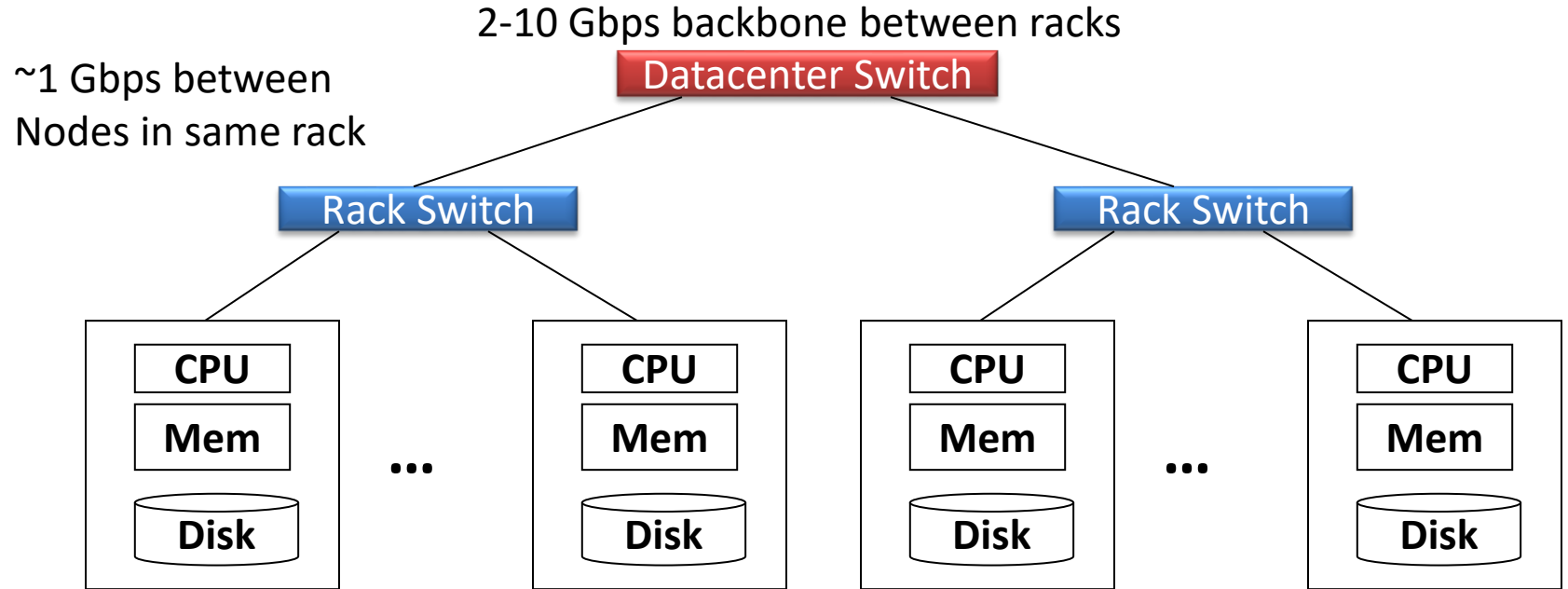
Web Crawl Example

- Google crawls about **10,000,000,000** web pages
 - Average size of a web page is **20KB**
 - Overall, about **200TB** worth of data
- A modern disk reads data at about **150 MB/sec**
 - Need **>16 days** just to read full web crawl
 - But only **24 minutes** if we were able to use 1000 disks
- Most algorithms require multiple passes over their data

Proposed Architecture (early 2000)

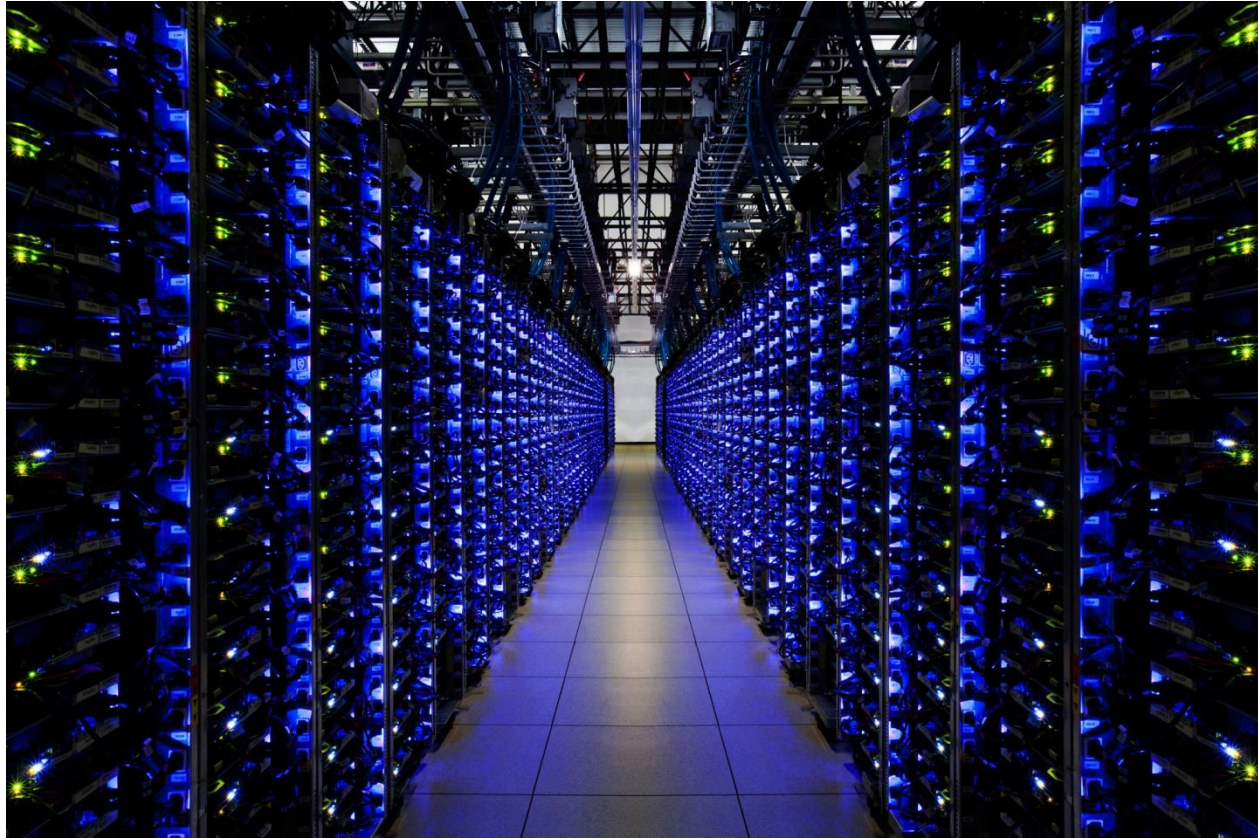
- Cluster of **commodity** Linux nodes
 - Gigabit Ethernet interconnect
- How to organize computations in this architecture?
 - What are the bottlenecks?

Cluster Architecture



Each rack contains 16-80 nodes

<http://www.google.com/about/datacenters>



Check this video

<https://www.youtube.com/watch?v=zDAYZU4A3w0>

Data locality

- About 1Gb/sec between nodes in the same rack
 - This is about as fast as reading from a local disk
 - Thus, even if data lays in the memory of a nearby node, fetching this data over the network incurs a significant overhead
- Solutions?
 - Move processing close to data
 - Divide and conquer (via hashing)

Node failures

- Assume Mean-Time-Between-Failures (MTBF) is 3 years or about 1000 days
- Google has (more than) 1,000,000 nodes (guess)
 - 1 node fails every 1000 days
 - Farm with 100 nodes → One failure every 10 days
 - Farm with 1M nodes → 1000 failures/day!

Stable Storage Requirement

- When nodes fail, how can we manage data persistently?
- **Solution-part 1: Distributed File System**
 - Provides global file namespace
 - Google GFS; Hadoop HDFS; Kosmix KFS

Distributed File Systems (GFS)

- Highly scalable distributed file system for large data-intensive applications.
 - E.g. 10K nodes, 100 million files, 10 PB
- Provides redundant storage of massive amounts of data on cheap and unreliable computers
 - Files are replicated (usually x2 or x3) to handle hardware failure
 - Detects failures and recovers from them
- Support access to files on remote servers
 - Handle dropped connections

Google GFS Motivation

- Google needed a good distributed file system
- Why not use an existing file system?
 - High component failure rates
 - Inexpensive commodity components fail all the time
 - “Modest” number of HUGE files
 - Files are write-once, mostly **appended** to
 - Perhaps concurrently
 - In-place updates are not common
 - Large sequential reads over random access
 - High sustained **throughput** over low **latency**

GFS: Design Decisions

- Files stored as chunks
 - Fixed size (64MB)
- Reliability through replication
 - Each chunk replicated across 3+ chunk servers
- Single master to coordinate access, keep metadata
 - Simple centralized management

GFS Metadata

- Global metadata is stored on the master
 - File and chunk namespaces
 - Mapping from files to chunks
 - Locations of each chunk's replicas
 - All in memory (64 bytes / chunk)
 - Fast
 - Easily accessible
- Master has an operation log for persistent logging of critical metadata updates
 - Persistent on local disk
 - Replicated
 - Checkpoints for faster recovery



Hadoop

- GFS paper published on 2003.
 - But GFS was never made open source.
- Doug Cutting and Yahoo! reverse engineered the GFS and called it Hadoop Distributed File System (HDFS).
- The software framework that supports **HDFS**, **MapReduce** and other related entities is called the **project Hadoop** or simply Hadoop.
 - This is open source and distributed by Apache.

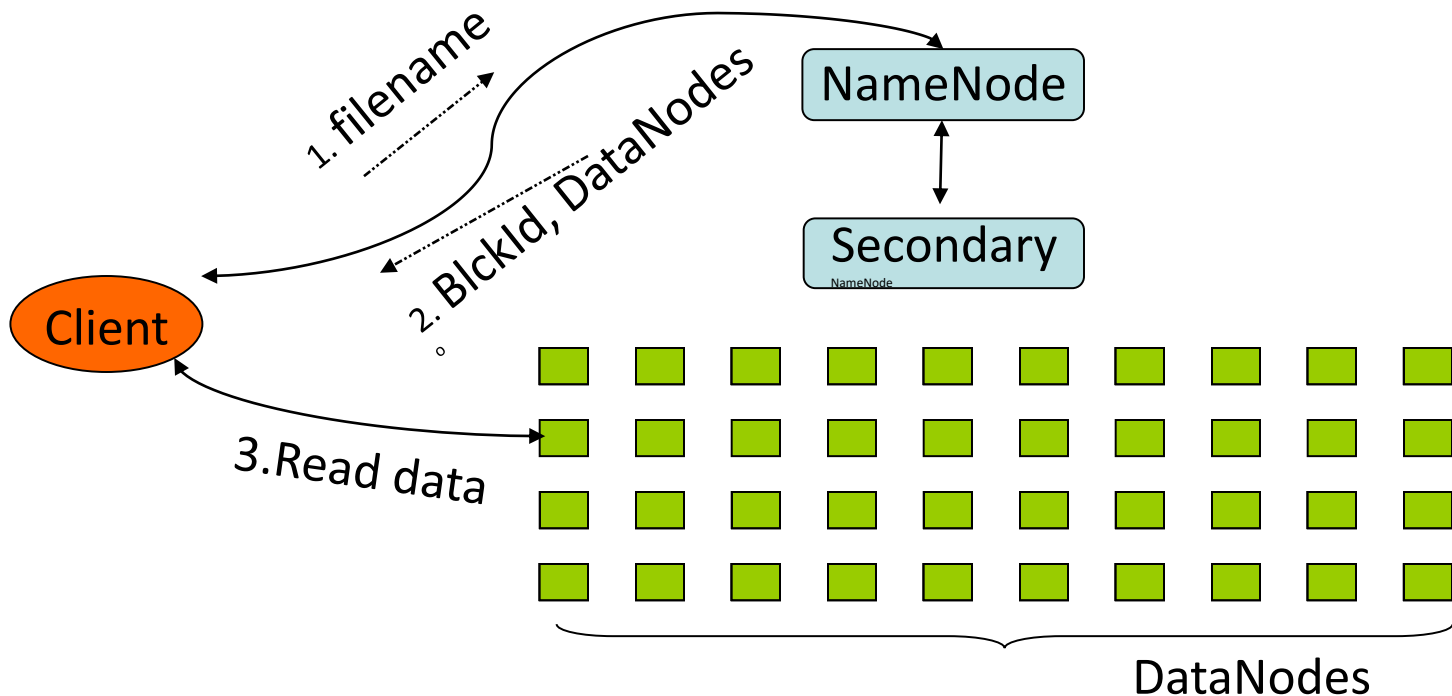
HDFS: Motivation

- Based on Google's GFS
- Data is organized into files and directories
- Files are divided into uniform sized **blocks** and distributed across cluster nodes
 - Typical block size 128 MB
 - Blocks are **replicated** to handle hardware failure
 - The block size and replication factor are configurable per file
 - Filesystem keeps checksums of data for corruption detection and recovery

HDFS Architecture

- Master-Slave architecture
- DFS Master “NameNode”
 - Manages the filesystem namespace
 - Maintains file name to list blocks + location mapping
 - Manages block allocation/replication
 - Checkpoints namespace and journals namespace changes for reliability
 - Controls access to namespace
- DFS Slaves “DataNodes” handle block storage
 - Store blocks using the underlying OS’s files
 - Clients access the blocks directly from DataNodes
 - Periodically send block reports to NameNode
 - Periodically check block integrity

HDFS Architecture



NameNode : Maps a file to a file-id and list of DataNodes

DataNode : Maps a block-id to a physical location on disk

HDFS basic examples

- Create an HDFS folder `/user/test`
 - `hdfs dfs -mkdir /user/test`
- List the content of that folder
 - `hdfs dfs -ls /user/test`
- Copy a file from the local file system to that folder
 - `hdfs dfs -put /home/kotidis/Desktop/myfile /user/test`
- Copy a file from HDFS back to the local fs
 - `hdfs dfs -copyToLocal /user/test/somefile /home/kotidis/Desktop`

MapReduce

- When nodes fail, how can we **manage data persistently** and **run distributed programs** on them?
- **Solution-part 1:** Distributed File System (HDFS)
- **Solution-part 2:** MapReduce
 - Simple programming abstraction that hides complexity of running (and managing) parallel computations in a cluster

Why MapReduce?

- Large scale data processing was difficult (early 2000)!
 - Want to use 1000s of CPUs
 - But don't want hassle of managing things
- MapReduce provides all of these, easily!
 - Managing hundreds or thousands of processors
 - Managing parallelization and distribution
 - I/O Scheduling
 - Status and monitoring
 - Fault/crash tolerance

Also its free! (unlike Parallel DBMSs)

What is MapReduce?

- MapReduce is a concept and method for typically batch-based large scale parallelization. It is inspired by [functional programming's](#) `map()` and `reduce()` functions
- MapReduce is highly scalable and can be used across many computers
- Many small machines can be used to process jobs that normally could not be processed by a large machine
- Abstracts issues of distributed and parallel environment from programmer
- Runs over distributed file systems (GFS,HDFS)

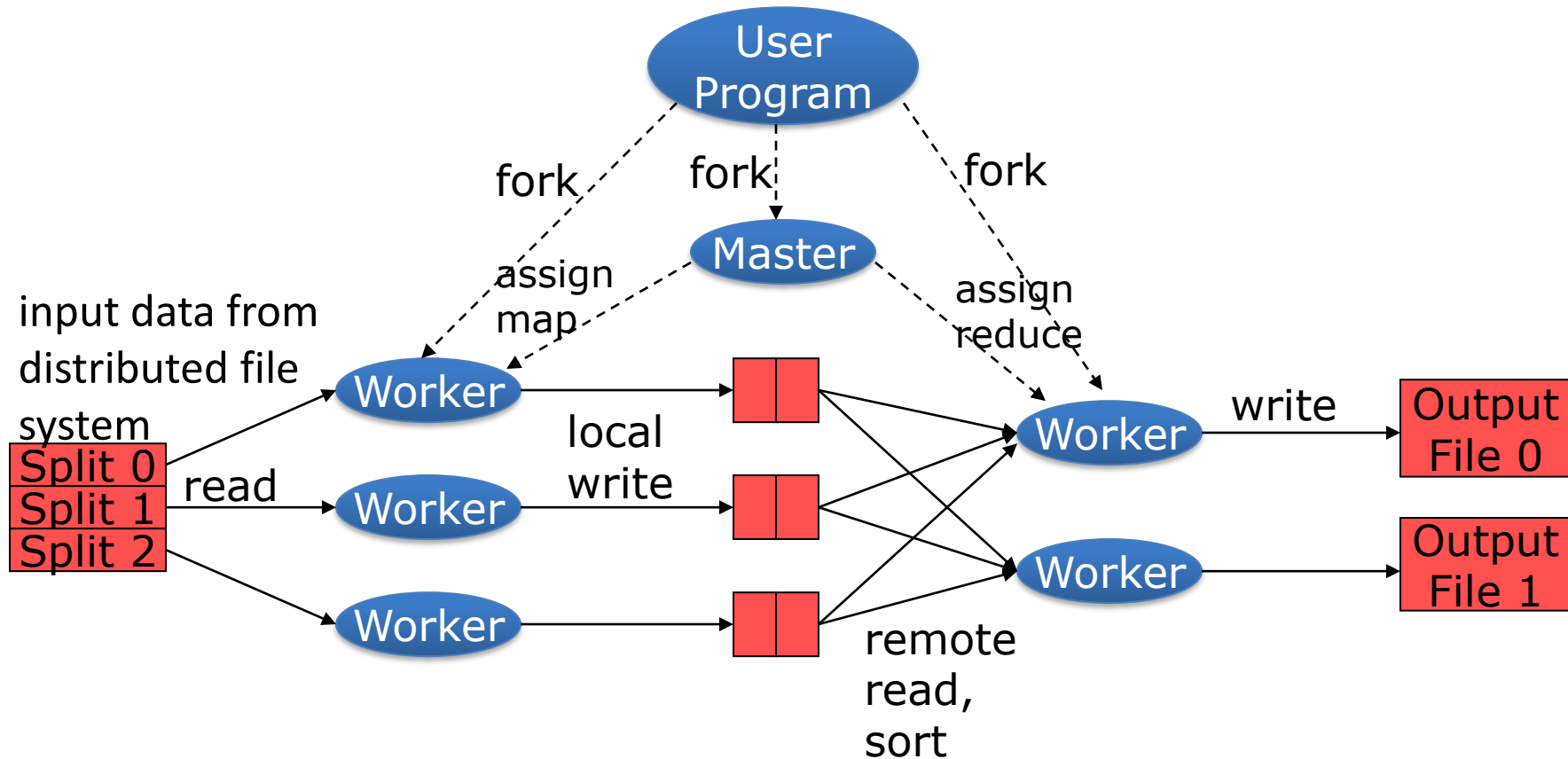
MapReduce

- Created by Google in 2004
- Inspired by LISP
 - Map(function, set of values)
 - Applies function to each value in the set
 - `(map 'length '((a c) (a) (a b) (a b c))) ⇒ (2 1 2 3)`
 - Reduce(function, set of values)
 - Combines all the values using a binary function (e.g., +)
 - `(reduce #'+ '(2 1 2 3)) ⇒ 8`

Map & Reduce

- **Map: (input data) → intermediate(key/value pairs)**
 - Map calls are distributed across machines by automatically partitioning the input data into M “chunks”.
 - MapReduce library groups together all intermediate values associated with the same intermediate key & passes them to the *Reduce function*
- **Reduce: intermediate(key/value pairs) → result files**
 - Accepts an intermediate key & a set of values for the key
 - It merges these values together to form the desired output
 - Reduce calls are distributed by partitioning the intermediate key space into R pieces using a partitioning function (e.g., $hash(key) \bmod R$). *The user specifies the # of partitions (R) and the partitioning function.*

Distributed Execution Overview



Step 1: Split input files into M chunks

- Number M is derived from the following formula:

$$M = \text{total data size} / \text{input split size}$$

where input split size is defined in Hadoop configuration.

Step 2: Fork processes

- Start up many copies of the program on a cluster of machines
 - 1 master: scheduler & coordinator
 - Lots of workers
- Idle workers are assigned either:
 - map tasks (each works on a chunk) – there are M map tasks
 - reduce tasks (each works on intermediate files) – there are R reduce tasks
 - $R = \#$ partitions, defined by the user

Step 3: Map Task

- Reads contents of its assigned input chunk
 - Parses key/value pairs out of the input data
 - Passes each pair to a user-defined map function
- Produces intermediate key/value pairs

Step 4: Create intermediate files

- Intermediate key/value pairs produced by the user's map function are buffered in memory and are periodically written to the local disk
 - Partitioned into R regions by a partitioning function
 - Notifies master when complete
 - Passes locations of intermediate data to the master
 - Master forwards these locations to the reduce worker

Step 5: Reduce Task: sorting

- Reduce worker gets notified by the master about the location of intermediate files for its partition
- Reads the data from the local disks of the map workers
- When the reduce worker reads intermediate data for its partition
 - It sorts the data by the intermediate keys
 - All occurrences of the same key are grouped together

Step 6: Reduce Task: Reduce

- The sort phase grouped data with a unique intermediate key
- User's Reduce function is given the key and the set of intermediate values for that key
 - $\langle \text{key}, (\text{value1}, \text{value2}, \text{value3}, \text{value4}, \dots) \rangle$
 - Performs a user-defined computation on these values
 - The output of the Reduce function is appended to an output file

Step 7: Return to user

- When all map and reduce tasks have completed, the master wakes up the user program
- The MapReduce call in the user program returns and the program can resume execution.
- Output of MapReduce is available in R output files (one per reducer)

Example

- **Count # occurrences of each word in a collection of documents**
 - Map:
 - Parse data; output each word and a count (1)
 - Reduce:
 - Sort: sort by keys (words)
 - Reduce: Sum together counts each key (word)

map(String key, String value):

// key: document name, value: document contents

for each word w in value:
EmitIntermediate(w, "1");

reduce(String key, Iterator values):

// key: a word; values: a list of counts
int result = 0;
for each v in values:
result += ParseInt(v);
Emit(AsString(result));

More details: the MAP Function

Document

a bb ccc dddd
bb ccc a a a ccc
bb ccc bb a a a
bb ccc dddd bb

a bb a bb a
dddd bb ccc a
bb ccc dddd bb
a a bb ccc bb a

dddd ccc bb a
bb dddd a a ccc
bb a bb ccc
dddd a
bb dddd

Chunk 1

a bb ccc dddd
bb ccc a a a
ccc bb ccc bb
a a a bb ccc
dddd bb

Chunk 2

a bb a bb a
dddd bb ccc a
bb ccc dddd
bb a a bb ccc
bb a

Chunk 3

dddd ccc bb a
bb dddd a a
ccc bb a bb
ccc dddd a
bb dddd

Mapper 1

Mapper 2

Mapper 3

Partition 1 Partition 2

a 1	ccc 1
bb 1	dddd 1
bb 1	ccc 1
a 1	ccc 1
a 1	ccc 1
bb 1	ccc 1
bb 1	ccc 1
a 1	ccc 1
a 1	ccc 1
a 1	ccc 1
bb 1	dddd 1
bb 1	

Partition 1 Partition 2

a 1	dddd 1
bb 1	ccc 1
a 1	ccc 1
bb 1	dddd 1
a 1	ccc 1
bb 1	
bb 1	
a 1	
a 1	
bb 1	
bb 1	
a 1	

Partition 1 Partition 2

bb 1	dddd 1
a 1	ccc 1
bb 1	dddd 1
a 1	ccc 1
a 1	ccc 1
bb 1	ccc 1
a 1	ccc 1
bb 1	dddd 1
a 1	dddd 1
bb 1	

Reducer 1

Reducer 2

The MAP Combiner

Document

a bb ccc dddd
bb ccc a a a ccc
bb ccc bb a a a
bb ccc dddd bb
a bb a bb a
dddd bb ccc a
bb ccc dddd bb
a a bb ccc bb a
dddd ccc bb a
bb dddd a a ccc
bb a bb ccc
dddd a
bb dddd

Chunk 1

a bb ccc dddd
bb ccc a a a
ccc bb ccc bb
a a a bb ccc
dddd bb

Chunk 2

a bb a bb a
dddd bb ccc a
bb ccc dddd bb
bb a a bb ccc
bb a

Chunk 3

dddd ccc bb a
bb dddd a a
ccc bb a bb
ccc dddd a
bb dddd

Mapper 1

Partition 1	Partition 2
a 7 bb 6	ccc 5 dddd 2

Mapper 2

Partition 1	Partition 2
a 7 bb 6	ccc 3 dddd 2

Mapper 3

Partition 1	Partition 2
a 5 bb 5	ccc 3 dddd 4

Reducer 1

Reducer 2

The REDUCE Function

Document

a bb ccc dddd
bb ccc a a a ccc
bb ccc bb a a a
bb ccc dddd bb
a bb a bb a
dddd bb ccc a
bb ccc dddd bb
a a bb ccc bb a
dddd ccc bb a
bb dddd a a ccc
bb a bb ccc
dddd a
bb dddd

Chunk 1

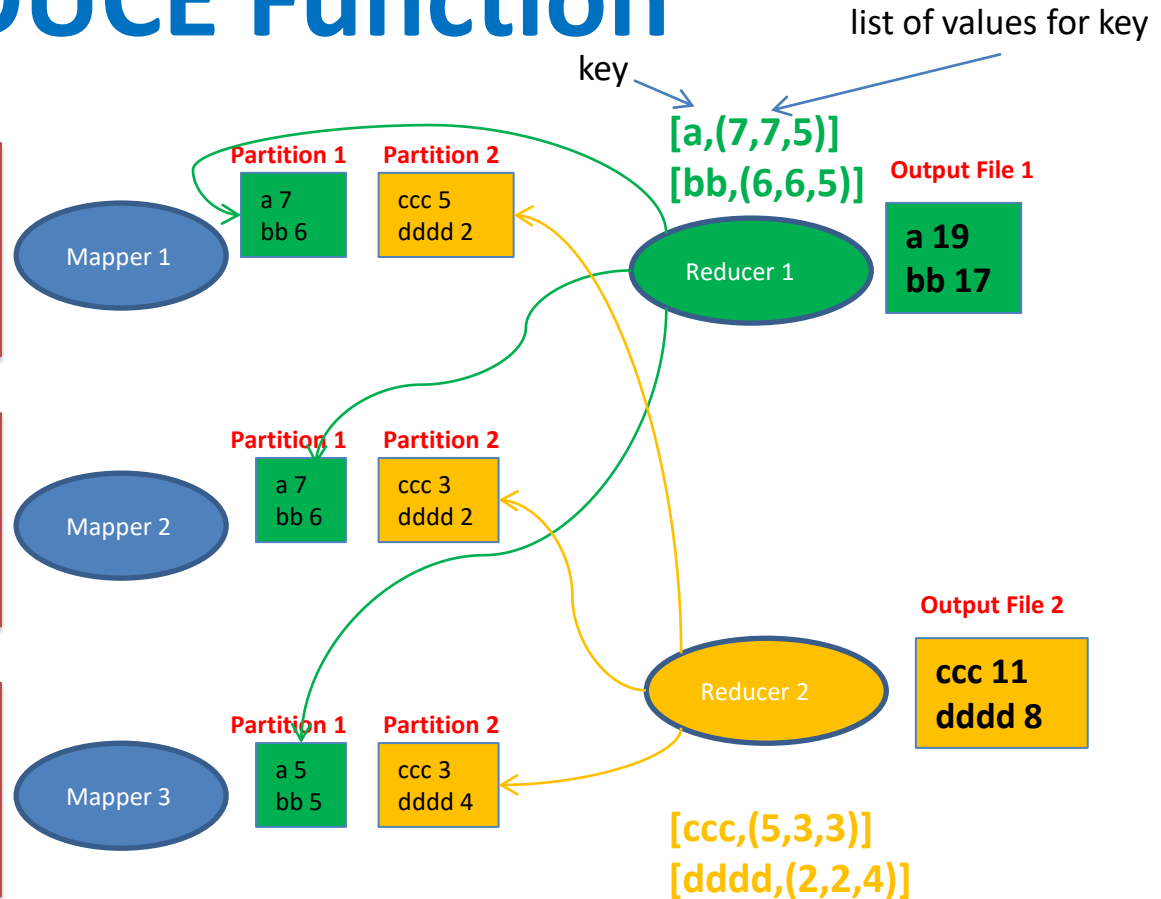
a bb ccc dddd
bb ccc a a a
ccc bb ccc bb
a a a bb ccc
dddd bb

Chunk 2

a bb a bb a
dddd bb ccc a
bb ccc dddd bb
bb a a bb ccc
bb a

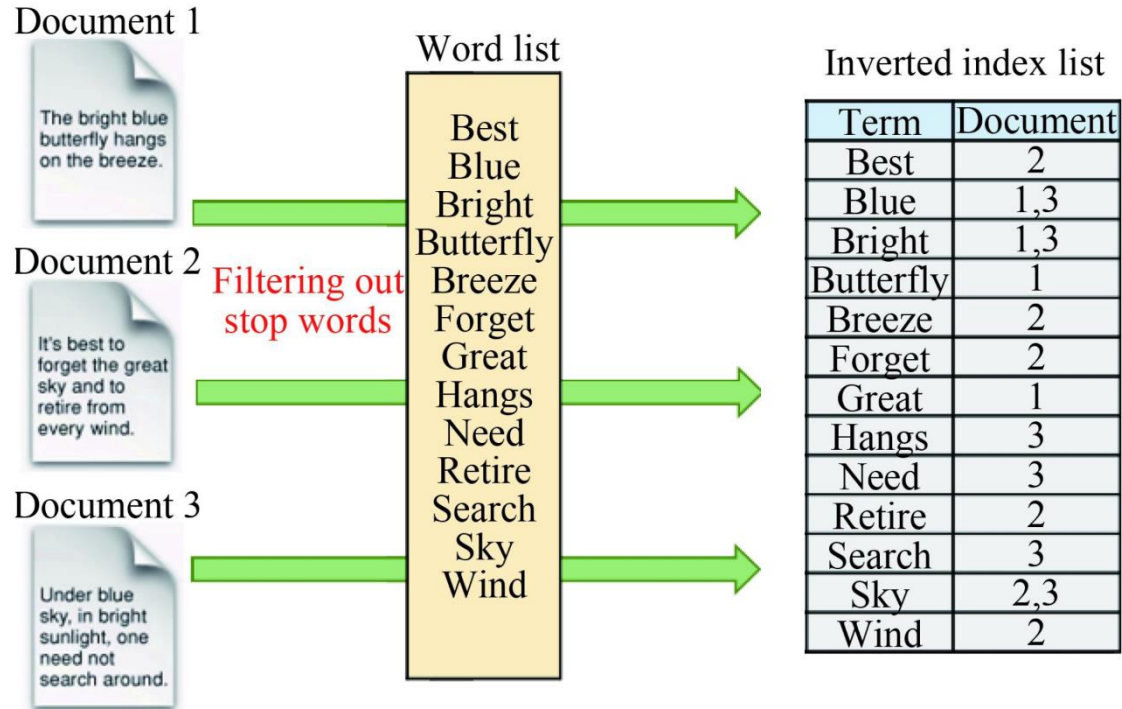
Chunk 3

dddd ccc bb a
bb dddd a a
ccc bb a bb
ccc dddd a
bb dddd



Inverted Index Example

Image adapted from <http://www.xml-data.org/DZKJDXXBYWB/html/20170208.htm>



Inverted index

- *Find what documents contain a specific word*
 - **Map**: parse document, omit stop words
 - Emit $\langle \text{word}, \text{document-ID} \rangle$ pairs
 - **Reduce**: for each word, sort the corresponding document IDs
 - Emit a $\langle \text{word}, \text{list}(\text{document-ID}) \rangle$ pair
 - The set of all output pairs is an inverted index

Count URL access frequency in web logs

- *Find the frequency of each URL in web logs*
 - **Map**: process logs of web page access; output $\langle \text{URL}, 1 \rangle$
 - **Reduce**: add all values for the same URL

Reverse web-link graph

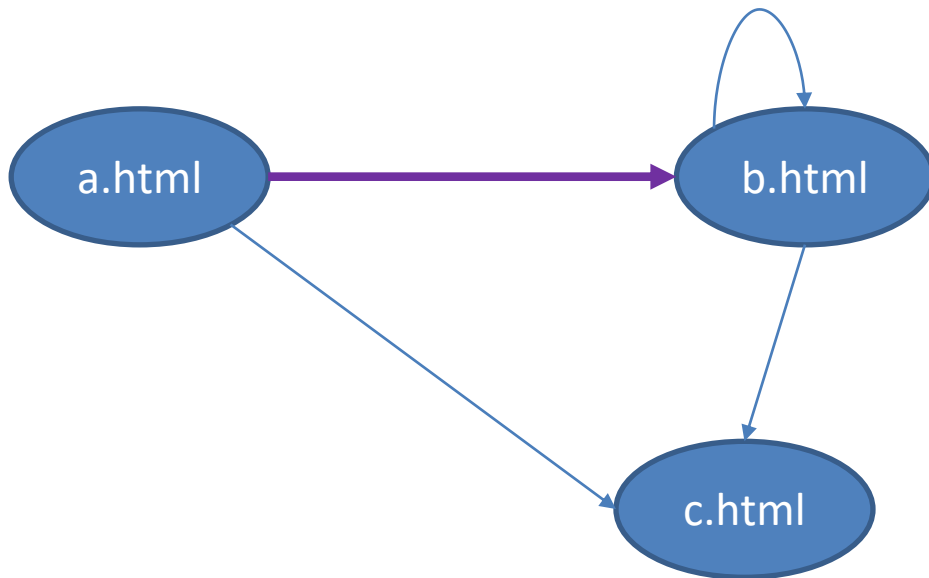
(find where page links come from)

a.html

```
<html>  
  <a href="b.html">...</a>  
  ...  
  <a href="c.html">...</a>
```

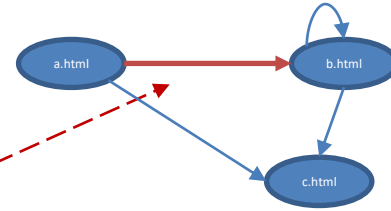
b.html

```
<html>  
  <a href="b.html">...</a>  
  ...  
  <a href="c.html">...</a>
```



Reverse web-link graph

b got a link from a



- Use href as a key, source url as value!

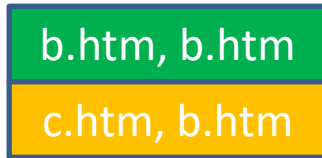
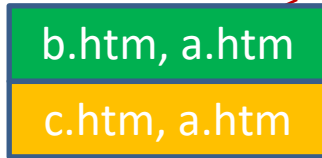
a.htm

```
<html>
  <a href="b.htm">...</a>
  ...
  <a href="c.html">...</a>
```

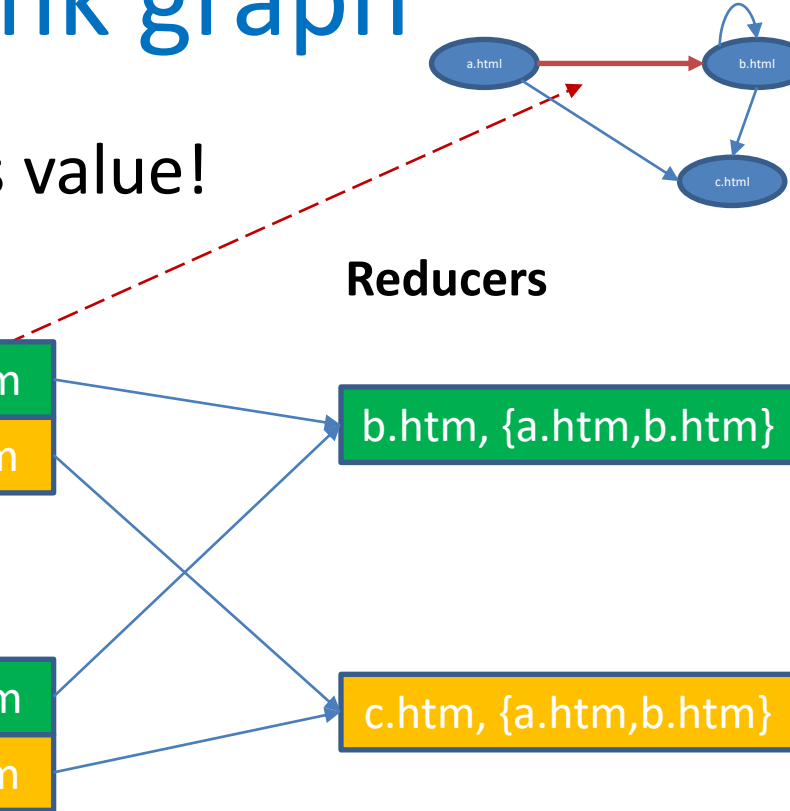
b.htm

```
<html>
  <a href="b.htm">...</a>
  ...
  <a href="c.htm">...</a>
```

Mappers



Reducers



Reverse web-link graph Recap

- Find where page links come from
 - **Map**: output $\langle \text{target}, \text{source} \rangle$ for each link to target in a page source
 - **Reduce**: concatenate the list of all source URLs associated with a target
 - Output $\langle \text{target}, \text{list}(\text{source}) \rangle$

Computing Natural Join by Map-Reduce

- Look at the specific case of joining $R(A,B)$ with $S(B,C)$.
- We must find records that agree on their B attributes
 - Thus, we shall use the **B-value** of tuples from either relation as **the key**
- **The value** will be the other attribute(s) and the name of the relation
 - So that the reduce() function knows where each record came from

The Map Function

- R and S are the relations we are joining
- For each record (a, b) of R, produce the key-value pair
 - $[b, (R, a)]$
- For each record (b, c) of S, produce the key-value pair
 - $[b, (S, c)]$

The Reduce Function:

- Each key value **b** will be associated with a list of tuples (=records) from R or S
- Produces all pairs (result records) consisting of one with first component from R and the other with first component from S

Table R

A	B
a1	b1
a2	b2
a2	b1
a4	b4

Table S

B	C
b1	c1
b1	c1
b2	c2
b4	c4

chunk 1

chunk 2

Assume 2 mappers / 2 reducers

- 1st mapper reads chunk-1 of R,S
- 2nd mapper reads chunk-2 of R,S
- First reducer receives odd key values (b1), second reducer receives even values (b2,b4)

Table R

A	B
a1	b1
a2	b2
a2	b1
a4	b4

Table S

B	C
b1	c1
b1	c1
b2	c2
b4	c4

Chunk 1

Chunk 2

1st mapper file 1: [b1,(R,a1)], [b1,(S,c1)], [b1,(S,c1)]
file 2: [b2,(R,a2)]

2nd mapper file 1: [b1,(R,a2)]
file 2: [b4,(R,a4)], [b2,(S,c2)], [b4,(S,c4)]

key

value

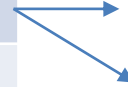
} Recall mapper creates one local file per reducer

Table R

A	B
a1	b1
a2	b2
a2	b1
a4	b4

Table S

B	C
b1	c1
b1	c1
b2	c2
b4	c4



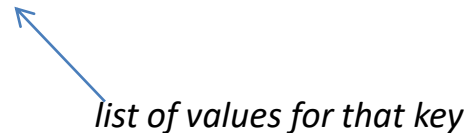
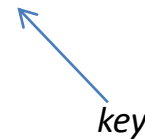
Reducer 1 receives: [(b1, ((R,a1),(S,c1),(S,c1),(R,a2)))]

output for b1: (a1,b1,c1), (a1,b1,c1), (a2,b1,c1), (a2,b1,c1)

Reducer 2 receives: [b2,((R,a2),(S,c2))] , [b4,((R,a4),(S,c4))]

output for b2: (a2,b2,c2)

output for b4: (a4,b4,c4)



Discussion

Table R

A	B
a1	b1
a2	b2
a2	b1
a4	b4

Table S

B	C
b1	c1
b1	c1
b2	c2
b4	c4



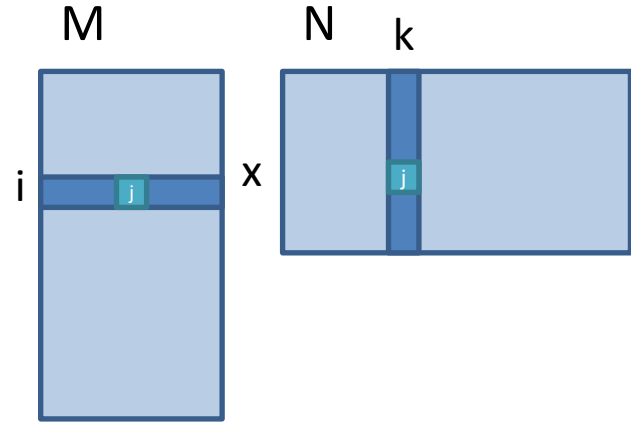
- Map-reduces essentially implements a distributed **hash-based** join algorithm
- All tuples with the same key value (e.g. **b1**) are sent (hashed) to the same “bucket” using B values as keys
- The contents of each “bucket” are all grouped by hadoop and passed to a reducer:
“Bucket” for b1: [(b1, ((R,a1),(S,c1),(S,c1),(R,a2)))]
- The reducer computes the join output for the received tuples

Matrix Multiplication

- Suppose two matrices:
 - Matrix M with element m_{ij} in row i and column j ,
 - Matrix N with element n_{jk} in row j and column k
- We want to compute the matrix P, where
 $P=MN$

Think relational!

- Consider each Matrix as a Relation
 - $M(I, J, V)$, with tuples (i, j, m_{ij})
 - $N(J, K, W)$, with tuples (j, k, n_{jk})
- Matrix multiplication can be seen as a **natural join** (based on **index j**) of the two “relations” N, M followed by **aggregation** (so as to sum-up all $m_{ij} * n_{jk}$ values, for different values of j)



$$p_{ik} = \sum_j m_{ij} * n_{jk}$$

The MAP Function

- For each matrix M element m_{ij} emit the key value pair $(j, (M, i, m_{ij}))$
- For each matrix N element n_{jk} emit the key value pair $(j, (N, k, n_{jk}))$
- as in a join

What each reducer receives

- For each j , a list containing all corresponding (M, i, m_{ij}) and (N, k, n_{jk}) values:

$(j, ((M, i_1, m_{i_1j}), (M, i_2, m_{i_2j}), \dots, (N, k_1, n_{jk_1}), \dots))$

*

$m_{i_1j} * n_{jk_1}$



Contributes to cell $p_{i_1k_1}$

The REDUCE Function

- For each key j produce the tuple $((i, k), m_{ij} * n_{jk})$
 - m_{ij} = each value comes from M for that j
 - (M, i, m_{ij})
 - n_{jk} = each value comes from N for that j
 - (N, k, n_{jk})

The Second MAP Function

- Map function is applied to the pairs that are output from the previous Reduce function
- It reads $((i,k),v)$ and outputs the same tuple

The Second REDUCE Function

- For each key (i, k) , produce the sum of the list of values associated with this key.
- The result is a pair $((i, k), v')$, where v' is the value of the element in row i and column k of the matrix $P = MN$.

All is not perfect

- MapReduce
 - Batch-oriented
 - Not suited for near-real-time processes, streaming data
 - Heavy use of disks (I/O)
- Fixed dependencies (synchronization)
 - Cannot start a new phase until the previous has completed
 - Multi-pass algorithms may have independent sub-tasks
 - Reduce cannot start until all Map workers have completed
 - Suffers from “stragglers” – workers that take too long (or fail)

Beyond MapReduce

- Map-reduce programming model requires developers to write custom programs, which are hard to maintain and reuse
- Newer systems hide low-level MapReduce programming from end-user via the use of a **declarative language** interface (pig, hive)

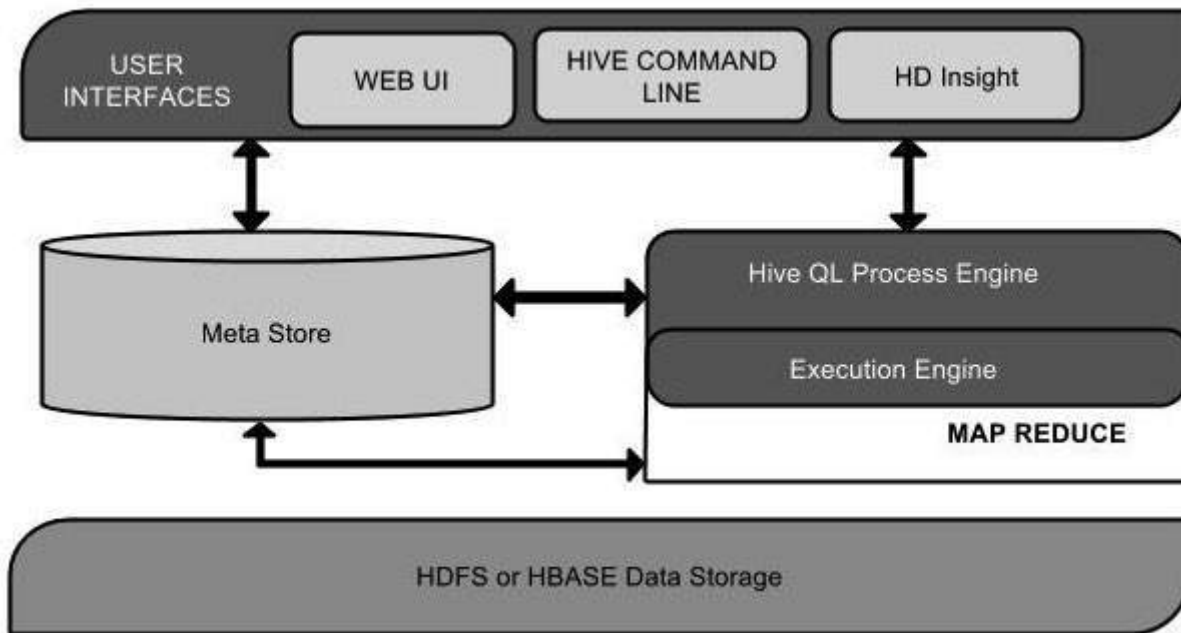
Hive: A Warehousing Solution over MapReduce

- Data warehouse infrastructure tool to process structured data in Hadoop
- Used by facebook for more than 300PB of data (2014)
- Makes querying and analyzing Big Data easy via an SQL-like language called HiveQL
- HiveQL queries are compiled into map-reduce jobs executed on Hadoop



Hive Architecture

<https://hive.apache.org/>

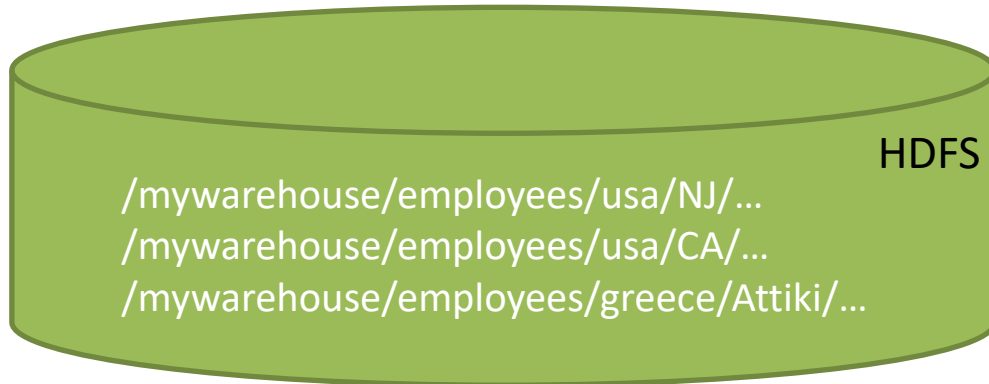


Hive overview

- Hive Database Schema is stored in a relational database (Meta Store)
 - Mysql, MS SQL Server, Oracle, Postgres, Apache derby,...
 - Stores metadata of tables, databases, columns in a table, their data types, and HDFS mapping.
- Hive Tables are stored in HDFS directories
 - or other data storage systems such as Apache HBase™
- Each table can have one or more partitions stored within sub-directories of the table directory
 - E.g. partition sales table by date /mywarehouse/sales/date=14-12-2015/
- Data in each partition is split into buckets (randomly or hashed on a column value)
 - Each bucket is stored as a file in the partition directory
 - Helps when sampling the data
 - May be used for fine-grained data partitioning

Example: Create table

```
CREATE TABLE employees (  
  name STRING,  
  salary FLOAT,  
  address STRUCT<street:STRING, city:STRING, state:STRING, zip:INT, country:STRING> )  
PARTITIONED BY (country STRING, state STRING);
```



Indexes

- Table partitions can be used to expedite queries
- Hive supports indexes on table columns for faster query execution
- Indexes are stored as separate tables providing pointers to data
- These indexes have to be manually updated when the data changes

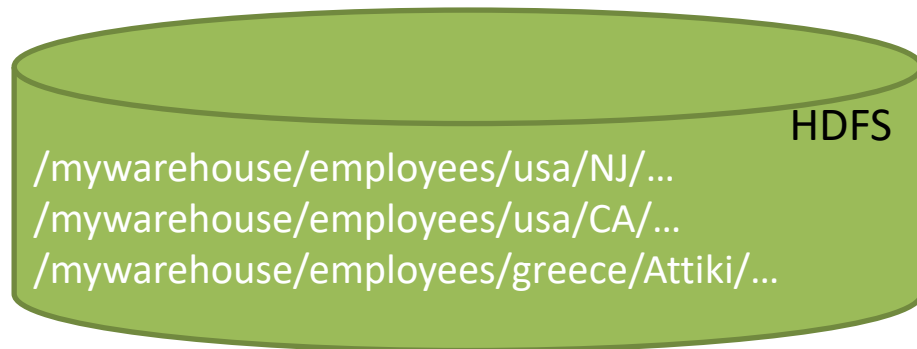
HiveQL

- Inspired by SQL
- Data definition statements used to create/alter tables with specific formats
 - Partitioning, bucketing, indexing constructs
- Supports select, project, join, aggregate, union all and sub-queries
- Can load and insert data into Hive tables, **but can not update or delete rows in existing tables**
 - **Hive is designed as a data warehouse for batch processing rather than an OLTP database**
- Supports User Defined Functions (UDFs) implemented in Java

Processing

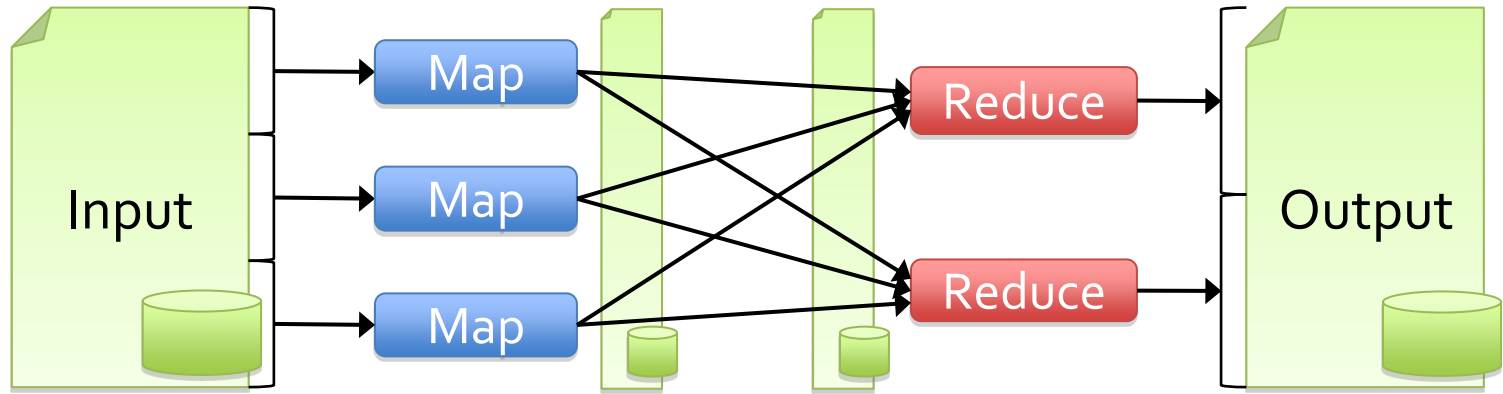
```
CREATE TABLE employees (  
  name STRING,  
  salary FLOAT,  
  address STRUCT<street:STRING, city:STRING, state:STRING, zip:INT, country:STRING> )  
PARTITIONED BY (country STRING, state STRING);
```

- What to find number of Greek employees with salary >20000
- HiveQL ?
- How to process this query?



Disk I/O in Hadoop

- MapReduce processing transforms data flowing from stable storage to stable storage



Apache Spark

(<https://spark.apache.org/>)

- Utilizes cluster memory to cache intermediate results
 - Unlike MapReduce that always stores intermediate results on disks
- Extremely useful for **repetitive** tasks
 - E.g. machine learning computations, pageRank, multi-pass algorithms
- Multiple APIs (Java, Scala, Python, R, SQL) and Libraries (MLlib, GraphX, Streaming)

RDD abstraction

- Resilient Distributed Datasets:
 - read-only collection of records
 - partitioned across the cluster
 - can be operated on in parallel
- May be cached in (distributed) main-memory
 - fallback to disk possible (but slow)

RDD Creation

(examples from <https://spark.apache.org/docs/latest/rdd-programming-guide.html>)

- RDDs can be created from any storage source supported by Hadoop, including your local file system (see wordcount example later)
- RDDs can also be created programmatically:

```
val data = Array(1, 2, 3, 4, 5)  
val distData = sc.parallelize(data)
```



- `parallelize()` cuts the dataset into a number of partitions that are dispersed among the cluster nodes
- Spark will run one task for each partition of the cluster
- You can specify the number of partitions (... if you know what you are doing)

RDD operations

- **Transformations** to build RDDs through deterministic operations on other RDDs
 - transformations include *map*, *filter*, *reduceByKey*, *join*
 - *lazy evaluation*
- **Actions** to return value or export data
 - actions include *count*, *collect*, *reduce*
 - actions trigger *execution*

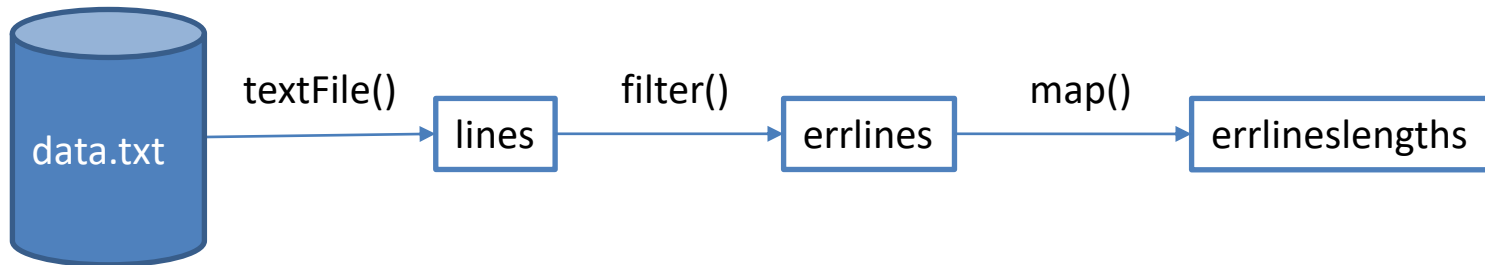
Lazy evaluation of transformations

```
val lines = sc.textFile("weblogs.txt")  
val errlines = lines.filter(_.contains("error"))  
val errlineslengths = errlines.map(s => s.length)
```

Read file from local disk

Filter lines that contain string «error»

Count number of characters per such line

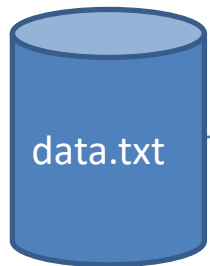


NOTHING IS COMPUTED YET !

Action triggers evaluation

```
val lines = sc.textFile("weblogs.txt")
val errlines = lines.filter(_.contains("error"))
val errlineslengths = errlines.map(s => s.length)
val totalLength = errlineslengths.reduce((a, b) => a + b)
```

Sum counters via reduce()



textFile()

lines

filter()

errlines

map()

errlineslengths

reduce()

totalLength

reduce() triggers execution and returns result

RDD Operations

Transformations (define a new RDD)

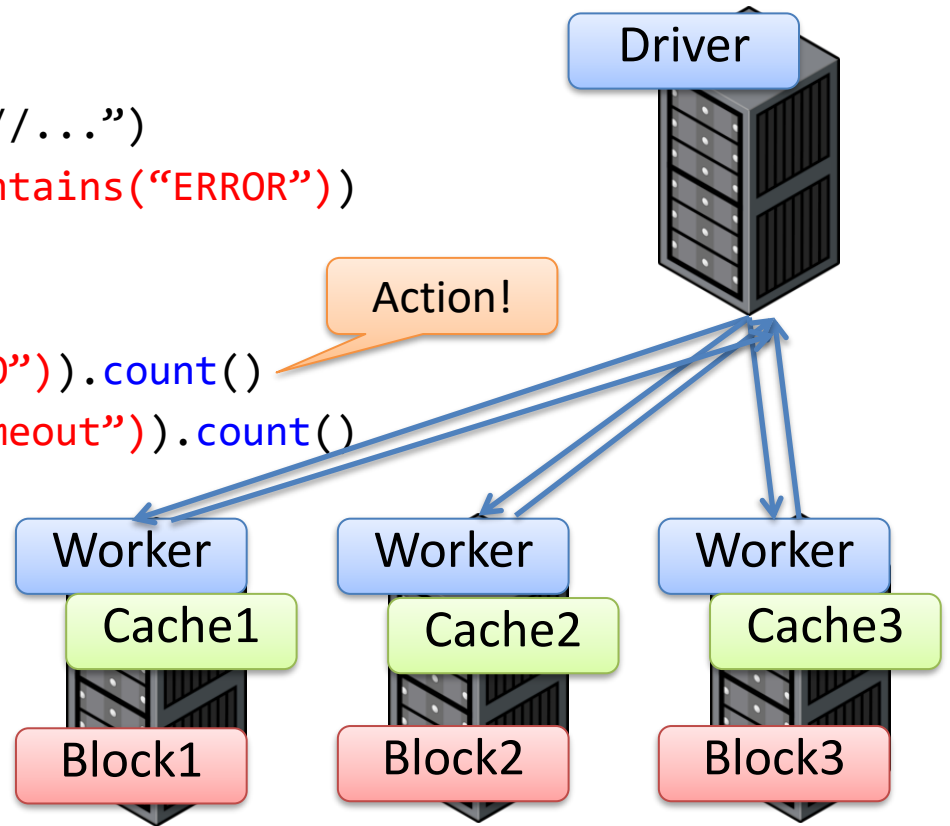
map
filter
sample
union
groupByKey
join
cache
...

Parallel operations (return a result to driver)

reduce
collect
count
...

Another Job example

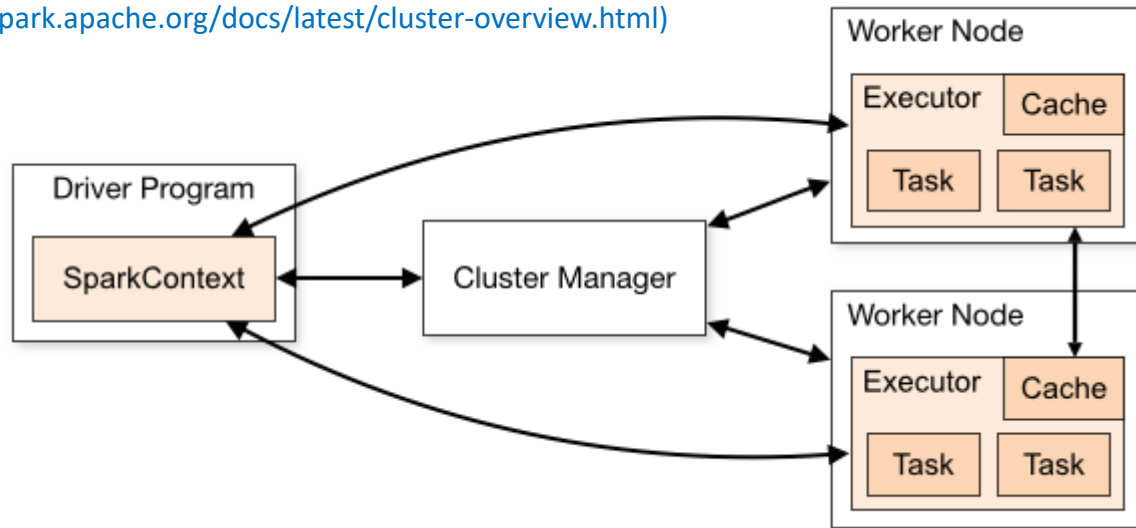
```
val log = sc.textFile("hdfs://...")  
val errors = log.filter(_.contains("ERROR"))  
errors.cache()  
  
errors.filter(_.contains("I/O")).count()  
errors.filter(_.contains("timeout")).count()
```



Note: `persist()` can be used instead of `cache()` in order to store a result to user-defined storage (default is memory)

Application Execution

(<https://spark.apache.org/docs/latest/cluster-overview.html>)

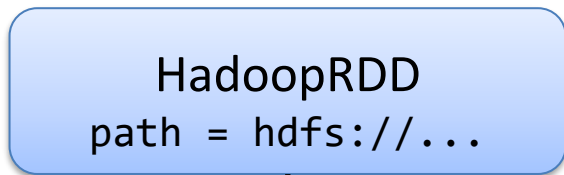


- A Spark application is initiated from your main program (called the driver program) via the SparkContext object which
 - acquires *executors* on nodes in the cluster (via the cluster manager)
 - executors are processes that run computations and store data for your application
 - sends *tasks* to the executors to run
- **Tip: each application gets its own web UI starting from <http://<driver-node>:4040>**

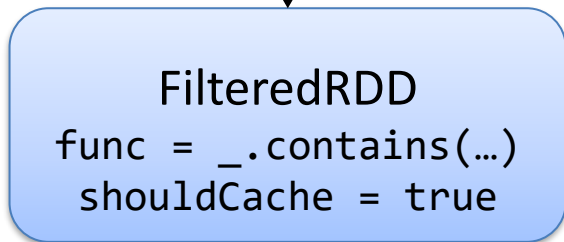
RDD partition-level view

Dataset-level view:

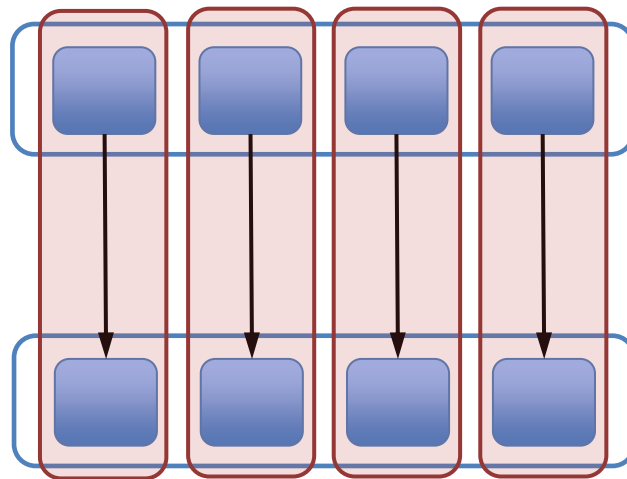
log:



errors:



Partition-level view:



Task 1 Task 2 ...

Word-count example

<http://spark.apache.org/docs/latest/quick-start.html>

```
val wordCounts = textFile.flatMap(line => line.split("
")).map(word => (word, 1)).reduceByKey((a, b) => a + b)
```

- `map(func)`: Return a new distributed dataset formed by passing each element of the source through a function *func*.
- `flatMap(func)`: each input item can be mapped to 0 or more output items (so *func* should return a Seq rather than a single item).
- `reduceByKey`: When called on a dataset of (K, V) pairs, returns a dataset of (K, V) pairs where the values for each key are aggregated using the given reduce function *func*, which must be of type (V,V) => V. Like in `groupByKey`, the number of reduce tasks is configurable through an optional second argument.

Top-10 most frequent words

```
val file = sc.textFile("/home/yannisk/wordfile")
val words = file.flatMap(line => line.split(" "))
val pairs = words.map(word => (word, 1))
val wordCounts = pairs.reduceByKey(_ + _)

val invCounts = wordCounts.map(x => (x._2, x._1)).sortByKey(false)

invCounts.take(10).foreach(println)
```

Sample text file

Panathinaikos B.C. also known simply as Panathinaikos, or by its full name, Panathinaikos BSA Athens, is the professional basketball team of the major Athens-based multi-sport club Panathinaikos A.O. It is owned by the billionaire Giannakopoulos family.

The parent athletic club was founded in 1908, while the basketball team was created in 1919, being one of the oldest in Greece. Alongside Aris, they are the only un-relegated teams with participation in every Greek First Division Championship until today. Panathinaikos has developed into the most successful basketball club in Greek basketball's history, and among the best in Europe, creating its own dynasty. They have won thirty-seven Greek Basket League Championships, eighteen Greek Cups, ten Doubles (all records), six EuroLeague Championships, one Intercontinental Cup and two Triple Crowns. They also hold the record for most consecutive Greek League titles, as they are the only team to have won nine consecutive championships (2003 - 2011), as well as for the most consecutive Greek Basketball Cup titles (six in a row 2012 to 2017). The team plays in the Olympic Indoor Hall, which has a capacity of 18,989 for basketball games.

Output

```
val file = sc.textFile("/home/yannisk/wordfile")
```

```
val words = file.flatMap(line => line.split(" "))
```

```
val pairs=words.map(word => (word, 1))
```

```
val wordCounts = pairs.reduceByKey(_ + _)
```

```
val invCounts = wordCounts.map(x => (x._2,x._1)).sortByKey(false)
```

```
invCounts.take(10).foreach(println)
```



(12,the)

(8,in)

(6,Greek)

(4,Panathinaikos)

(4,as)

(4,team)

(4,basketball)

(3,consecutive)

(3,club)

(3,for)

Issues with RDDs

- RDDs enable low-level transformation and actions better suited for unstructured data
- **DataFrames** (DF) permit developers to impose a structure onto a distributed collection of data, allowing higher-level abstraction
 - A DF is conceptually equivalent to a table in a relational database (or to a DataFrame in Python 😊)
 - Combined with SQL, DFs enable declarative programming

Top-10 most freq. words (RDDs+DFs)

```
val file = sc.textFile("/home/yannisk/wordfile")
val words = file.flatMap(line => line.split(" "))
val pairs=words.map(word => (word, 1))
val wordCounts = pairs.reduceByKey(_ + _)

case class Record(word: String, count: Int)

val df=wordCounts.map(rec=>Record(rec._1,rec._2)).toDF

df.orderBy(desc("count")).show(10)
```

Declare schema & convert to DF

```
val file = sc.textFile("/home/yannisk/wordfile")
val words = file.flatMap(line => line.split(" "))
val pairs=words.map(word => (word, 1))
val wordCounts = pairs.reduceByKey(_ + _)
```

```
case class Record(word: String, count: Int)
```

```
val df=wordCounts.map(rec=>Record(rec._1,rec._2)).toDF
```

```
df.orderBy(desc("count")).show(10)
```

Exploit new DFs API


```
val file = sc.textFile("/home/yannisk/wordfile")
val words = file.flatMap(line => line.split(" "))
val pairs=words.map(word => (word, 1))
val wordCounts = pairs.reduceByKey(_ + _)

case class Record(word: String, count: Int)

val df=wordCounts.map(rec=>Record(rec._1,rec._2)).toDF

df.orderBy(desc("count")).show(10)
```

DF schema



word	count
the	12
in	8
Greek	6
team	4
as	4
Panathinaikos	4
basketball	4
club	3
of	3
most	3

only showing top 10 rows

DF+SQL

- Now that we have tabular data, why not express our computations using SQL?
- Notice that SQL is **declarative** permitting run-time optimizations unlike hardcoded programs

Top-10 most freq. words (DFs+SQL)

```
import org.apache.spark.sql.Session
import spark.implicits._
val spark = SparkSession.builder.config(sc.getConf).getOrCreate()

val file = sc.textFile("/home/yannisk/wordfile")
val words = file.flatMap(line => line.split(" "))

val wordsDF = words.toDF(colNames="word")
wordsDF.createOrReplaceTempView("R")

val result = spark.sql("Select word,count(*) as counter from R
                        group by word order by counter DESC limit 10")
result.show()
```

Create a DF containing all words in doc

```
import org.apache.spark.sql.SparkSession
import spark.implicits._
val spark = SparkSession.builder.config(sc.getConf).getOrCreate()

val file = sc.textFile("/home/yannisk/wordfile")
val words = file.flatMap(line => line.split(" "))

val wordsDF = words.toDF(colNames="word")
wordsDF.createOrReplaceTempView("R")

val result = spark.sql("Select word,count(*) as counter from R
                        group by word order by counter DESC limit 10")
result.show()
```

```
scala> wordsDF.show(10)
+-----+
|      word|
+-----+
|Panathinaikos|
|      B.C.|
|      also|
|      known|
|      simply|
|       as|
|Panathinaikos,|
|       or|
|       by|
|      its|
+-----+
```

Create a View in Hive

```
import org.apache.spark.sql.Session
import spark.implicits._
val spark = SparkSession.builder.config(sc.getConf).getOrCreate()

val file = sc.textFile("/home/yannisk/wordfile")
val words = file.flatMap(line => line.split(" "))

val wordsDF = words.toDF(colNames="word")
wordsDF.createOrReplaceTempView("R")

val result = spark.sql("Select word,count(*) as counter from R
                        group by word order by counter DESC limit 10")
result.show()
```


Query View R using SQL

```
import org.apache.spark.sql.Session
import spark.implicits._
val spark = SparkSession.builder.config(sc.getConf).getOrCreate()

val file = sc.textFile("/home/yannisk/wordfile")
val words = file.flatMap(line => line.split(" "))

val wordsDF = words.toDF(colNames="word")
wordsDF.createOrReplaceTempView("R")

val result = spark.sql("Select word,count(*) as counter from R
                        group by word order by counter DESC limit 10")
result.show()
```


Top-10 most freq. words (DFs+SQL)

```
import org.apache.spark.sql.SparkSession
import spark.implicits._
val spark = SparkSession.builder.config(sc.getConf).getOrCreate()

val file = sc.textFile("/home/yannisk/wordfile")
val words = file.flatMap(line => line.split(" "))

val wordsDF = words.toDF(colNames="word")
wordsDF.createOrReplaceTempView("R")

val result = spark.sql("Select word,count(*) as counter from R
                        group by word order by counter DESC limit 10")
result.show()
```



word	counter
the	12
in	8
Greek	6
Panathinaikos	4
team	4
basketball	4
as	4
for	3
of	3
club	3

Spark Streaming engine



Streaming Word-count example

- Streaming text (e.g. tweets)
- Count #words at a per-minute interval

Set up Configuration

```
// Create a configuration with two threads and batch interval of 1 minute
```

```
// The master requires 2 cores to prevent from a starvation scenario
```

```
val conf = new
```

```
    SparkConf().setMaster("local[2]").setAppName("NetworkWordCount")
```

```
val ssc = new StreamingContext(conf, Seconds(60))
```

```
// DStream stands for Discretized Stream
```

```
// Create a DStream that will connect to hostname:port, like localhost:9999
```

```
val lines = ssc.socketTextStream("localhost", 9999)
```

Map-reduce style computation

```
// Split each line into words
```

```
val words = lines.flatMap(_.split(" "))
```

```
// Count each word in each batch
```

```
val pairs = words.map(word => (word, 1))
```

```
val wordCounts = pairs.reduceByKey(_ + _)
```

```
// Print the first ten elements of each RDD generated in this DStream to the  
console
```

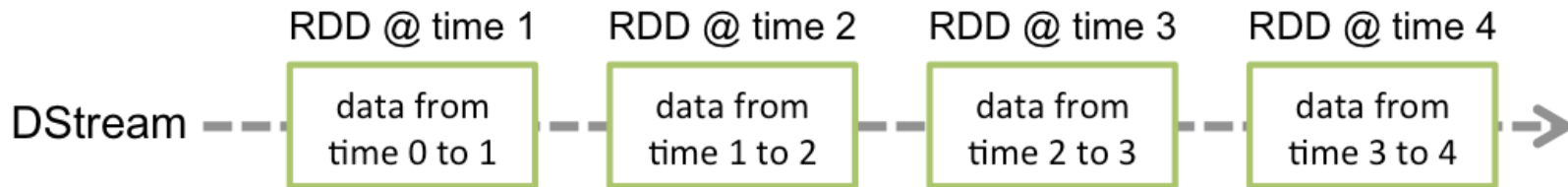
```
wordCounts.print()
```

Start stream processing

- `ssc.start()` // Start the computation
`ssc.awaitTermination()` // Until you stop it

From Streams to Batch Processing

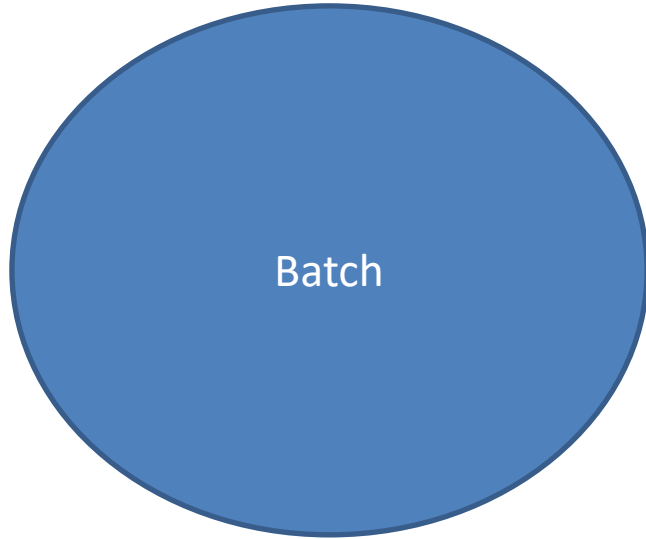
- Treat streaming computations as a series of deterministic micro-batch computations on small time intervals
- Batches are executed using Spark's distributed data processing framework



Integration with other platforms

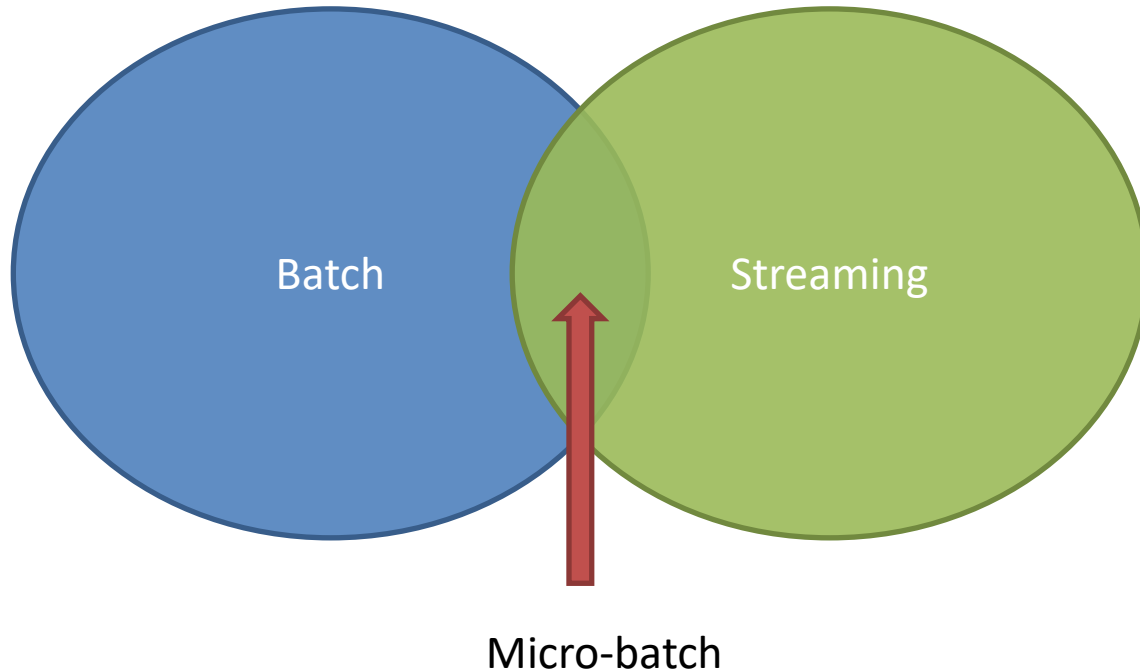


Batch vs Streaming



Micro-Batching (Spark Streaming)

*P. Taylor Goetz



Hash tag-count example

- Build a stream engine that processes incoming tweets and reports the most frequent hash-tags in them

Connect to Twitter API via tweepy

```
from tweepy import Stream
from tweepy import OAuthHandler
from tweepy.streaming import StreamListener
import json
import sys

#consumer key, consumer secret, access token, access secret.
ckey= "...get your own keys!....."
csecret= "...."
atoken= "....."
asecret= "....."
```

Connect to Twitter API via tweepy

```
class listener(StreamListener):
    def on_data(self, data):
        all_data = json.loads(data)
        try:
            tweet = all_data["text"]
        except:
            return
        for hashtag in all_data["entities"]["hashtags"]:
            print("#"+hashtag["text"].upper())
        return(True)
    def on_error(self, status):
        print(status)
```

Connect to Twitter API via tweepy

```
auth = OAuthHandler(ckey, csecret)
auth.set_access_token(accessToken, asecret)
```

```
twitterStream = Stream(auth, listener())
#retrieve tweets related to nba!!!
twitterStream.filter(track=["nba"])
```

Lets try it

```
yannisk@ubuntuserver:~$ unbuffer python3 mytwitter.py
```

```
#NBAMEDIADAY
```

```
#NBA
```

```
#NBA
```

```
#GOSPURSGO
```

```
#NBAMEDIADAY
```

```
#DUBNATION
```

```
#NBA
```

```
#NBAMEDIADAY
```

```
#NBAMEDIADAY
```

```
#NBAMEDIADAY
```

```
#CELTICS
```

```
#NBA
```

```
#NBAMEDIADAY
```

```
#NBAMEDIADAY
```


Redirect to a local port

```
yannisk@ubuntuuser:~$ ubnuffer python3 mytwiter.py | nc -lk 9999
```

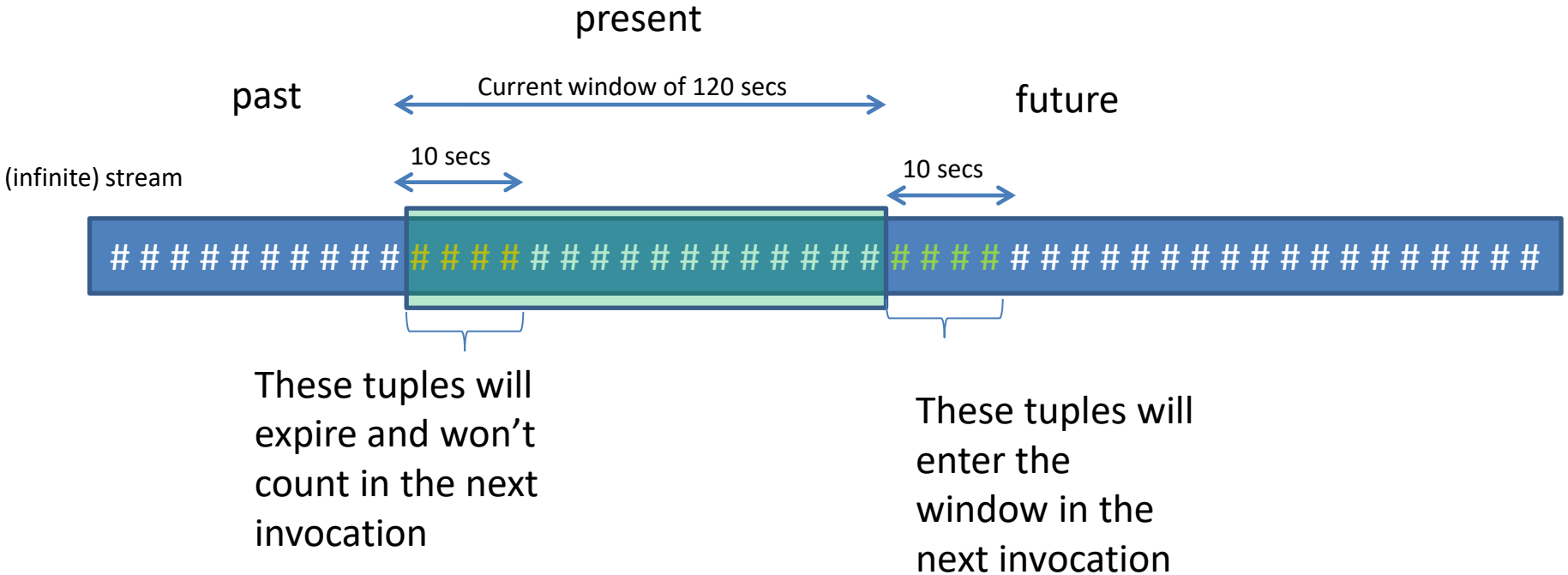
Stream logic

- Collect all hashtags in the past 120 seconds
 - This defines the width of the **window** that is used to keep fresh hashtags
 - Alternatively think that each hashtag has a Time-to-Live (TTL) of 120 seconds
 - Expired hashtags are thrown out of the window and are not used in subsequent computations

Window shift (slide)

- This is not a one-time computation but rather a **continuous query**
- How often would you like to perform the computation?
 - Let's assume a default value of 10 seconds
- Thus, every 10 second we will compute the counts for all hashtags in the past 120 seconds

Sliding windows



Main logic

(full code available @eclass)

```
val lines = ssc.socketTextStream("localhost", 9999)
val words = lines.flatMap(_.split(" "))
//filter hashtags only
val hashtags = words.filter(w=>w.contains("#"))
//count all hashtags in the last 120 seconds
val winh = hashtags.window(Seconds(120))
//iterate over accumulated hashtags
winh.foreachRDD { (rdd: RDD[String], time: Time) =>
  val spark = SparkSession.builder.config(rdd.sparkContext.getConf).getOrCreate()
  // Convert RDD[String] to RDD[case class] to DataFrame
  val wordsDataFrame = rdd.map(w => Record(w)).toDF()
  // Creates a temporary view using the DataFrame
  wordsDataFrame.createOrReplaceTempView("words")
  // Do word count on table using SQL and print it
  val wordCountsDataFrame =
    spark.sql("select word, count(*) as total from words group by word order by total DESC")
  println(s"===== $time =====")
  wordCountsDataFrame.show(20,false)
```



This computation is repeated
for all RDD data accumulated
within a window

Sample Output

===== 153781615000 ms =====

word	total
#NBAMEDIADAY	77
#DUBNATION	6
#NBA	4
#TRAININGCAMPTIPOFF	2
#CLIPPERS	2
#PLAYGROUNDS2	2
#GOSPURSGO	2
#TRUETOATLANTA	1
#GSW	1
#JOINTHEREVOLUTION	1
#SANANTONIOSPURS	1
#MEMORABILIA	1
#STACKED	1
#TORONTORAPTORS	1
#COLT45	1
#SPORTS	1
#CLAMPCITY	1
#1ON1	1
#STEPHENCURRY	1
#NBADRAFTROOM	1

only showing top 20 rows



===== 1537816160000 ms =====

word	total
#NBAMEDIADAY	79
#NBA	6
#DUBNATION	5
#TRAININGCAMPTIPOFF	2
#CLIPPERS	2
#GOSPURSGO	2
#GSW	1
#JOINTHEREVOLUTION	1
#TORONTORAPTORS	1
#MEMORABILIA	1
#1ON1	1
#STEPHENCURRY	1
#SANANTONIOSPURS	1
#COLT45	1
#NBADRAFTROOM	1
#KAWHILEONARD	1
#STACKED	1
#PLAYGROUNDS2	1
#SPORTS	1
#CLAMPCITY	1

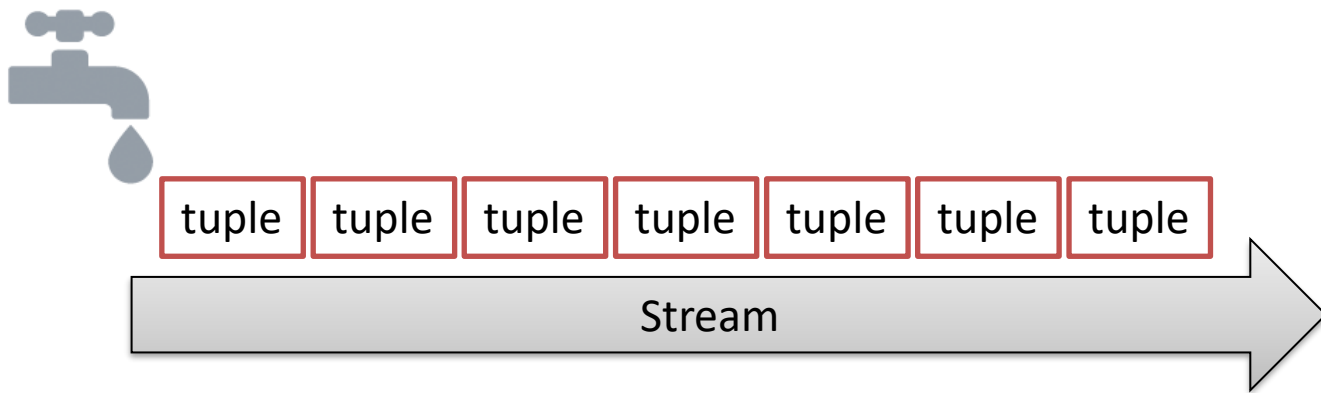
only showing top 20 rows

Apache Storm

- Distributed realtime computation system
- Operates on tuple streams
- Tuple-at-a-time operation
- Lower latency than spark (claimed)

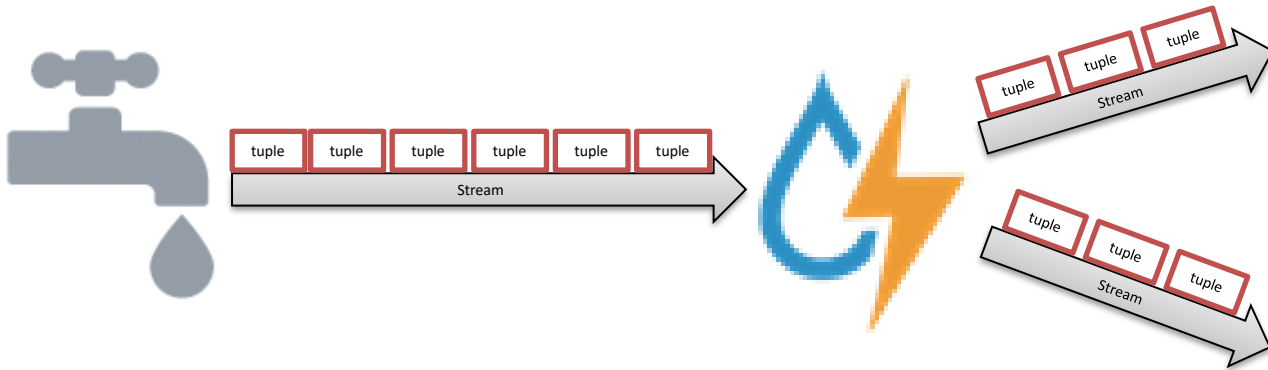
Spouts

- Spouts are sources of data streams
 - API calls (e.g. twitter), event data (e.g. RFID reader), web logs, etc

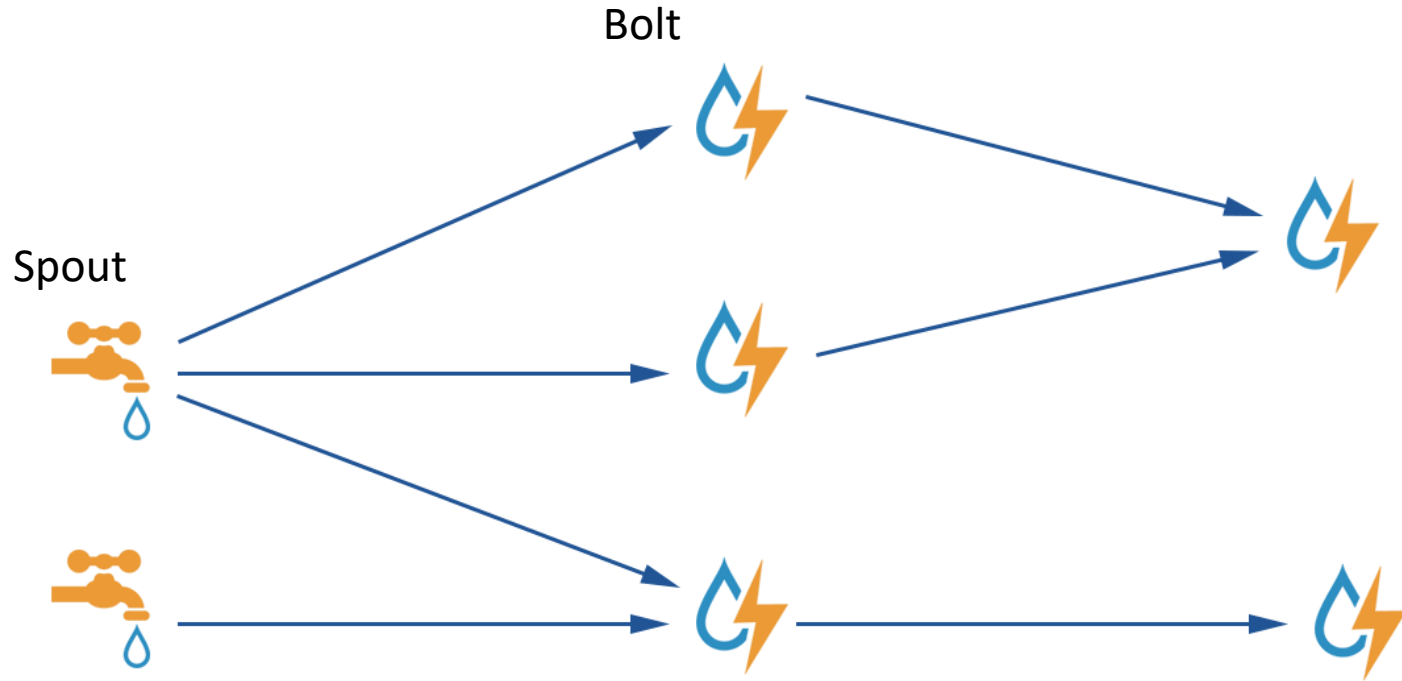


Bolts

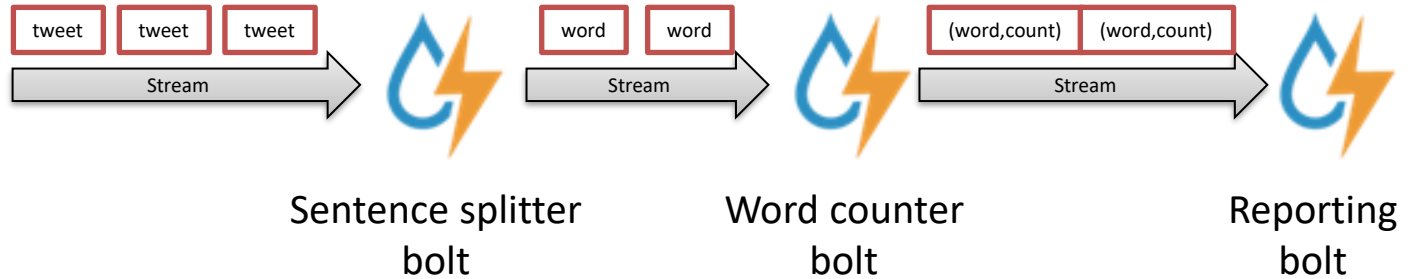
- Process stream tuples
 - Apply functions, filters, transformations, aggregation, etc
 - May create new streams



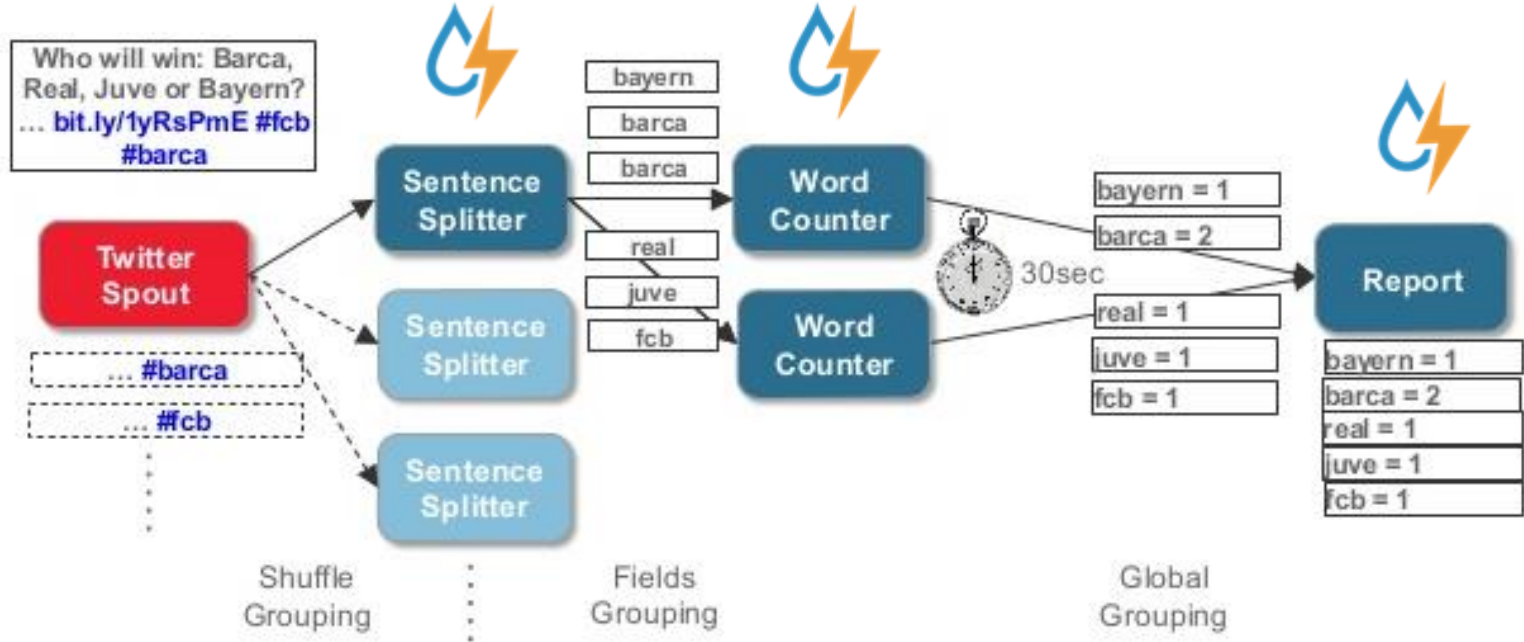
Storm topology



Word-count in Storm



Word-count execution*



*Image from Guido Schmutz (<http://www.slideshare.net/gschmutz/apache-storm-vs-spark-streaming-two-stream-processing-platforms-compared>)

Storm Architecture

