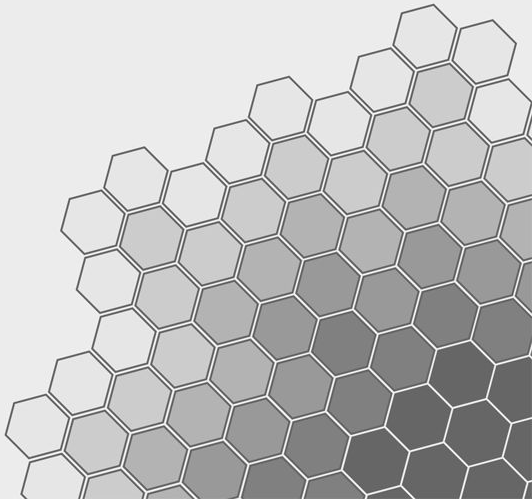
Data Analytics with Hadoop and Spark

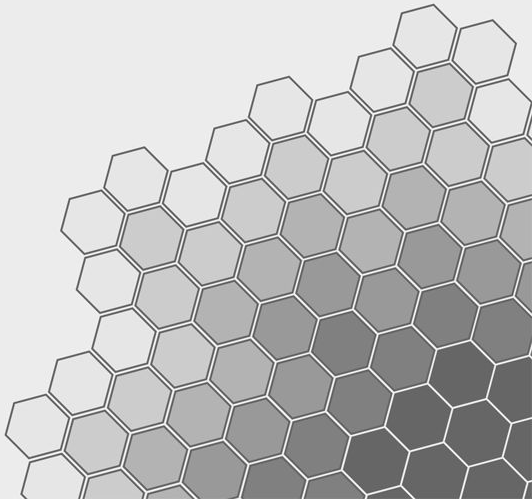
Navy Federal Credit Union

September 21, 2018



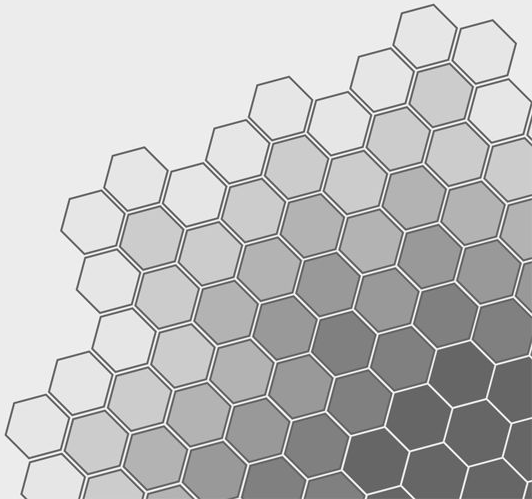
Agenda

- A Distributed Computing Environment
  - Hadoop as an Operating System for Big Data
  - HDFS for Distributed Data Storage
  - MapReduce for Distributed Computing
- A Functional Programming Model
- Fast Analytics with Spark
- Toward Last Mile Computing

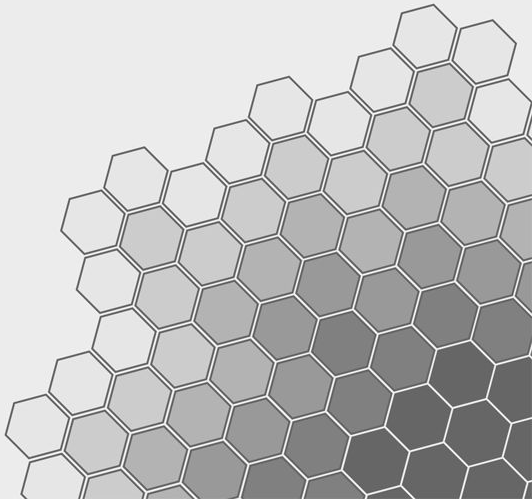


Primary Goal:

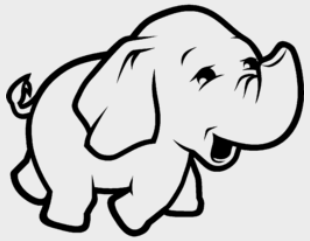
Understand Analytics in a Distributed Computing Context.



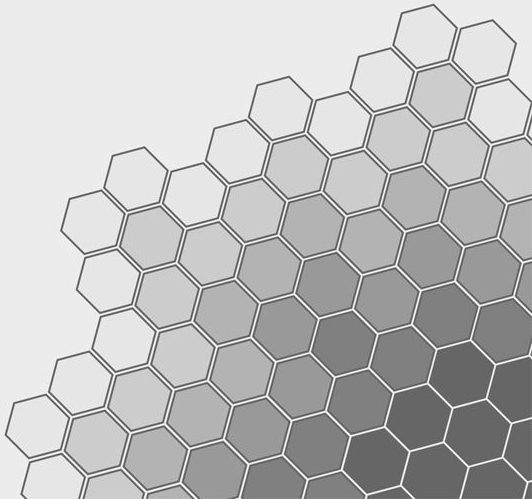
Big Data



The Motivation for Hadoop



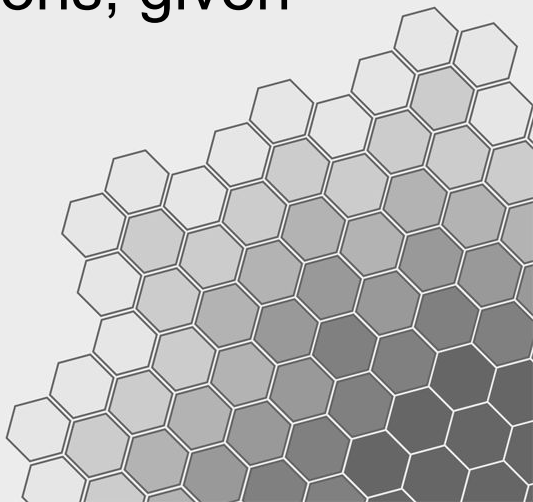
- Traditional data analyses involve complex processing (regressions, etc.) upon small data sets, usually representative samples of a larger population of data.
- Computations of this type are dependent on the size and performance of a processor or of a computer's main memory.
- To improve computational speed or the amount of data a computer is able to process, faster processors and more RAM is required



Supercomputers are Expensive

One of the world's most powerful supercomputers, Titan at Oak Ridge National Laboratory, cost approximately USD $100M and will cost the U.S. Department of Energy (DOE) USD $9M in electricity bills alone to operate.

The economic impracticality is underscored by the fact that supercomputing architecture cannot keep pace with data velocity in modern machine learning applications, given Moore's law.

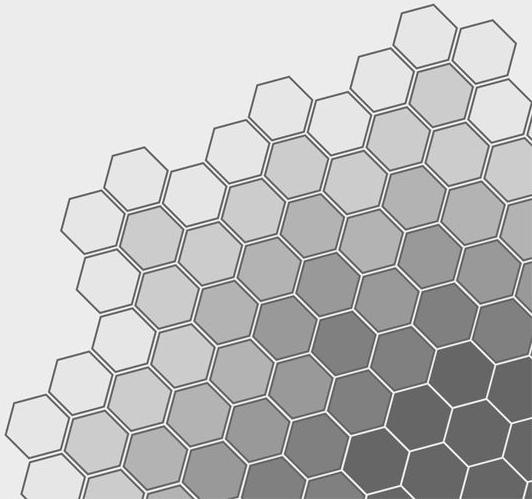


Clusters of COTS Hardware

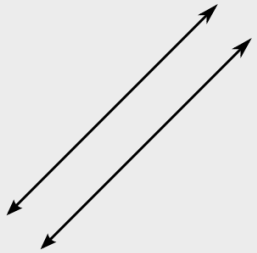
Use many, cost-effective machines that can be easily replaced to do computations over Big Data.

Startups can afford a 10-node cluster of $300 servers from warehouse off-lease sites like ServerMonkey.com or the use of Elastic MapReduce on Amazon's cloud.

It's much harder to justify the purchase of a similarly beefy server (Dell starts at $4500 for a comparable PowerEdge).



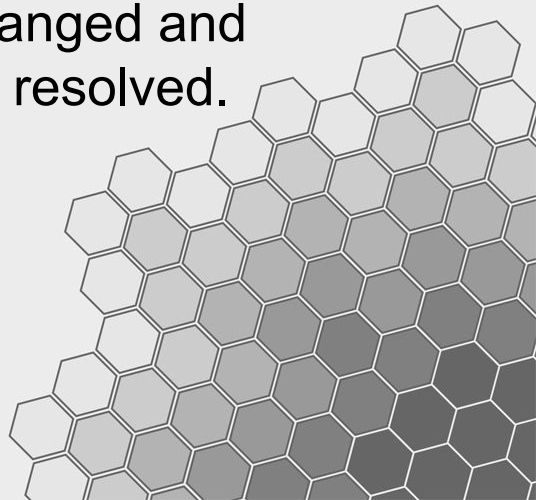
Parallelism



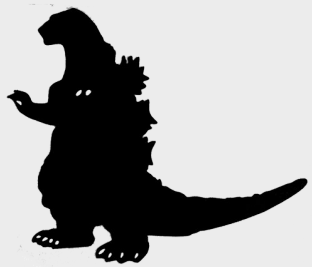
The solution to data velocity in computing is parallelism - the use of many computers to simultaneously perform many intermediate computations that aggregate to a larger whole.

Parallelism exists in many places in programming, including at the hardware layer (parallel processors) and the software layer (threading or multiprocessing)

Development of distributed applications is still difficult. Distributed algorithms need to be devised, data needs to be exchanged and synchronized, and temporal dependencies need to be resolved.



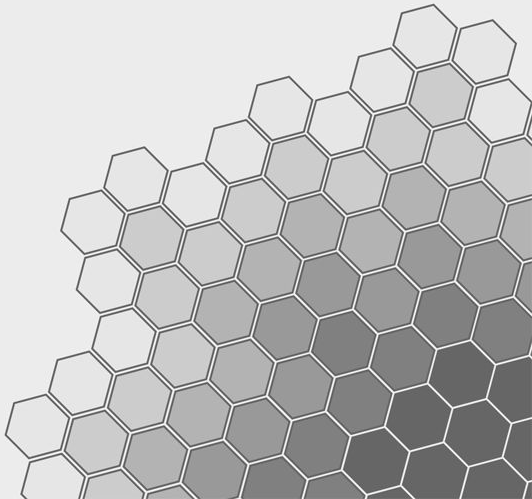
The Big Data Bottleneck



Data storage is cheap - a 4TB Western Digital Enterprise Hard Drive is only about $369.99 and getting cheaper every day, that's less than USD $0.10 per GB!

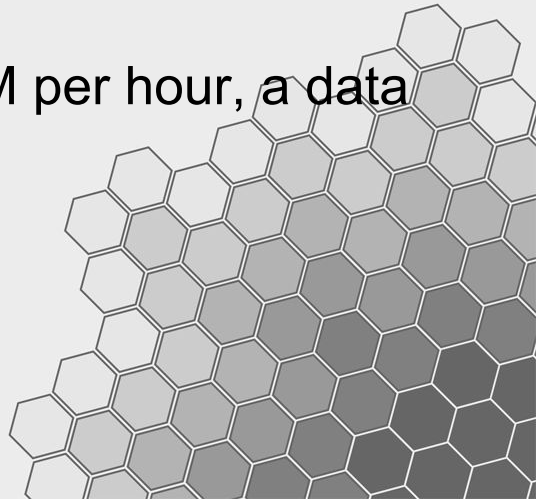
The problem is not the storage, however, it's getting the data off the disk and to the processor. This drive delivers a sustained transfer rate of 182 MB/s for sequential data - but in order to read the entire contents of this disk to the processor, it would take 6 hours, 6 minutes, and 18.02 seconds.

Moore's law isn't the bottleneck!



The V’s of Big Data

- **Volume**
  - Pearson OpenClass program is 10 TB of graph data with 6.24B vertices and 121B edges.
- **Velocity**
  - Twitter generates 1.39 MB per second, 41.83 TB per year. (for only 160 bytes per post!)
- **Variety**
  - Facebook is working on a query engine to analyze 250 PB of data - the Presto Engine.
- **Veracity**
  - Walmart transactions happen at a rate of 1.5M per hour, a data volume of 1.43 GB per hour.

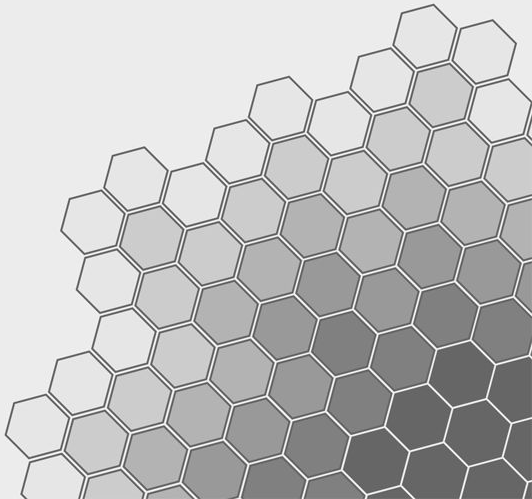


Handling Failure

The biggest threat to a distributed computation is **job failure** - for any number of reasons a part of a computation may fail; a distributed computing framework must be able to gracefully recover in the middle of a computing process.

In short, distributed applications must solve:

- Input and output synchronization and bandwidth
- Handling task failure and evaluating output quality
- Algorithms for asynchronous computation
- Data must be stored in a central location

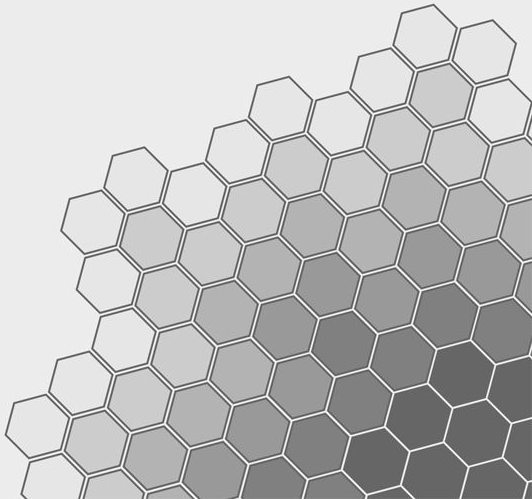


Requirements

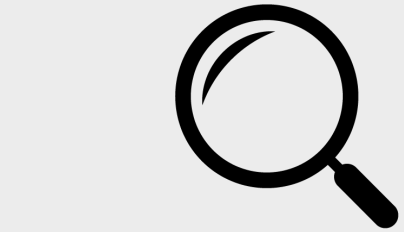
Based on these motivations, the Hadoop architecture provides:

- **Support for failure**: systems should gracefully degrade into lower performing states. If a failed component recovers, it should be able to rejoin the cluster.
- **Recoverability**: in the event of failure, no data should be lost.
- **Consistency**: The failure of one task should not affect the result of the entire job.
- **Scalability**: adding load leads to a decline of performance, not failure. Increasing resources results in proportional increase in capacity.

Brief History



Born of Search

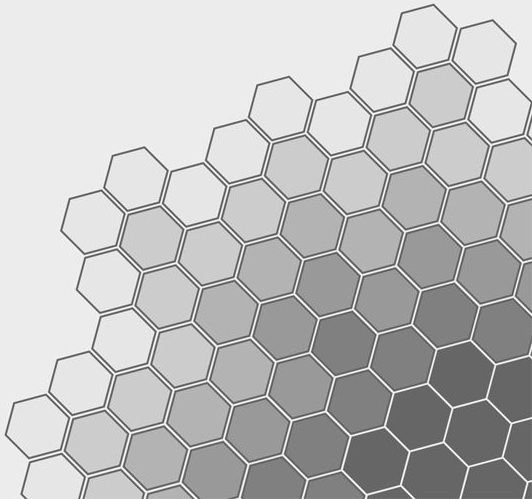


The Apache Nutch project

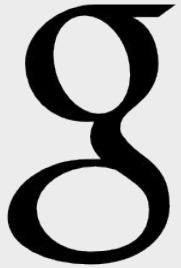
- Open source web crawler and search engine
- Created by Doug Cutting and Mike Cafarella

Searching and Indexing the Web is Expensive!

- Estimated that a billion page index = $500K in hardware
- Operating cost of $30K/month



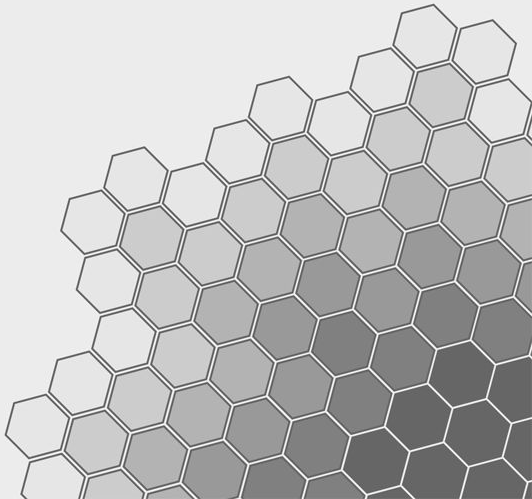
Google File System



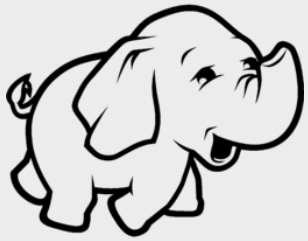
2003: *The Google File System* by Sanjay Ghemawat et al

Google File System:

- stores data computation locally
- distributed file system
- management with single master server
- low-cost, commodity storage nodes



MapReduce and Hadoop



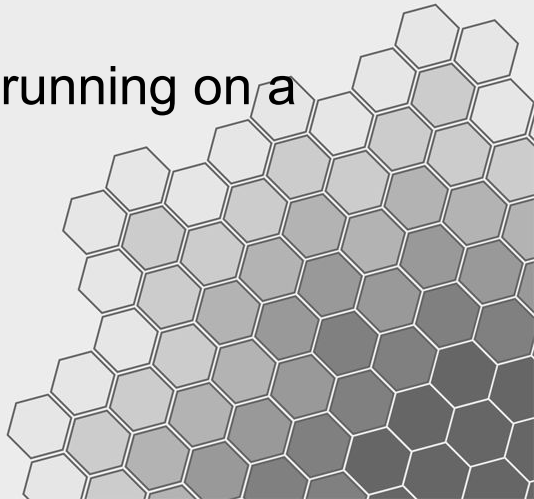
2004: *MapReduce: Simplified Data Processing on Large Clusters by* Dean & Ghemawat

MapReduce is the computational framework that enables the parallel, distributed processing off the GFS cluster.

2005: Doug and Mike reimplement MapReduce and GFS (HDFS) in open-source to power Nutch.

2008:

- Yahoo! announces that production search engine running on a 10,000 node Hadoop cluster
- Apache elevates Hadoop to a top-level project



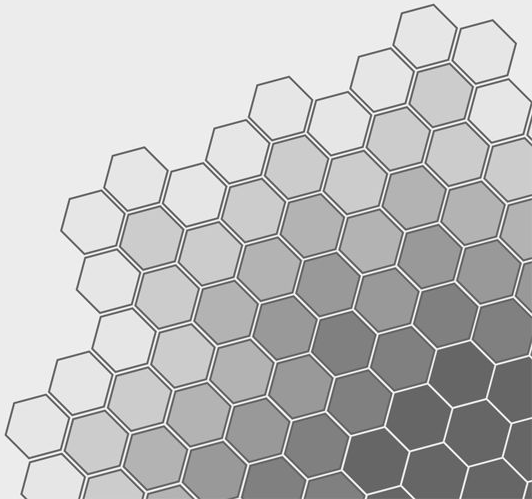
Performance

Although the primary motivator for a system like Hadoop is the ability to use COTS technology as a cost-effective solution to big data processing, Hadoop is also blazingly fast.

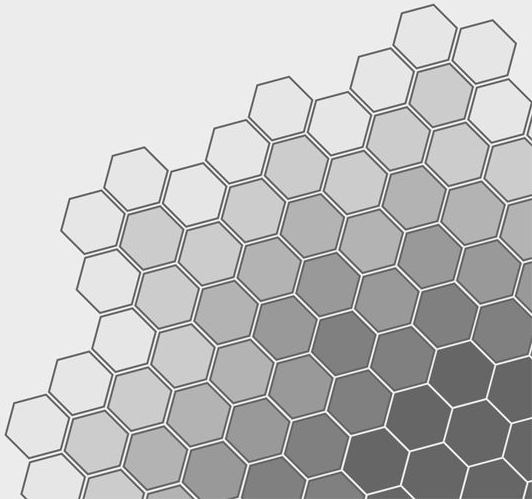
In 2013, Hadoop broke the terabyte sort record - able to sort 1.42 TB per minute!

Google reports being able to sort a petabyte of data in six hours on it’s Hadoop-like cluster.

Now even faster with Spark and YARN



An Operating System for Big Data



Core Concepts

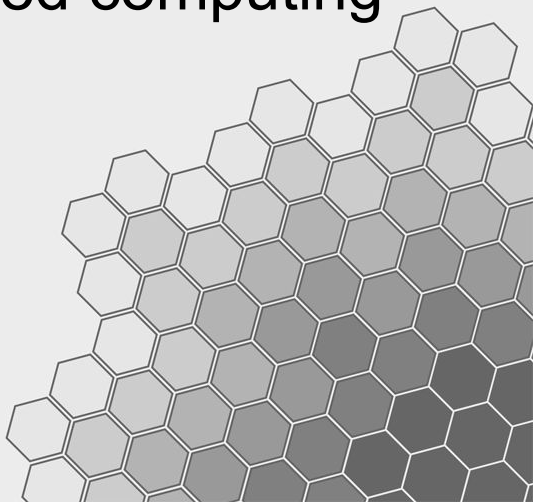
- Data is distributed and stored on nodes, and those nodes attempt to perform work on the data they have rather than transfer it across the network
- Applications are written at a high level and are not concerned with network programming or time
- Minimize the amount of network traffic
- Duplicate data to provide data safety
- Master programs allocate work and manage jobs
- Each task only operates on a single block of data
- Jobs, nodes, and disks are failure tolerant via redundancy

Two Primary Components

HDFS, a distributed file system manages data across the cluster and makes it available for processing.

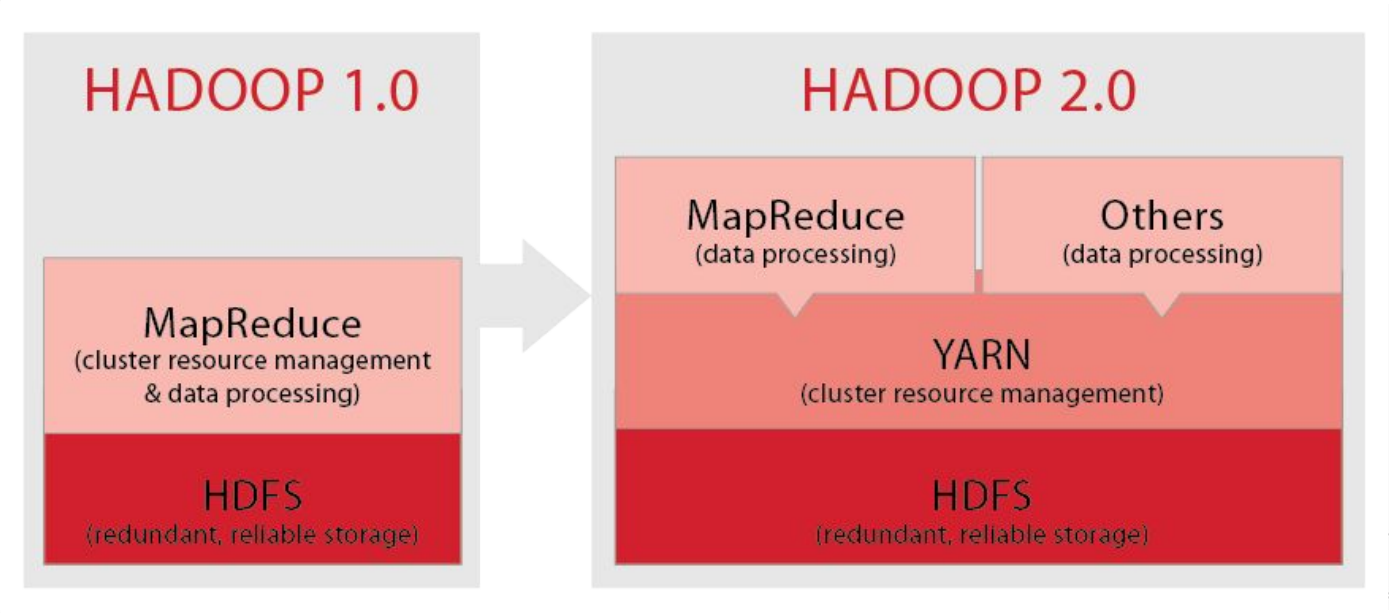
YARN, a resource manager that manages and schedules computational assets and an application deployment framework for running processing jobs.

Together, HDFS and YARN abstract distributed computing away from the developer or data scientist.



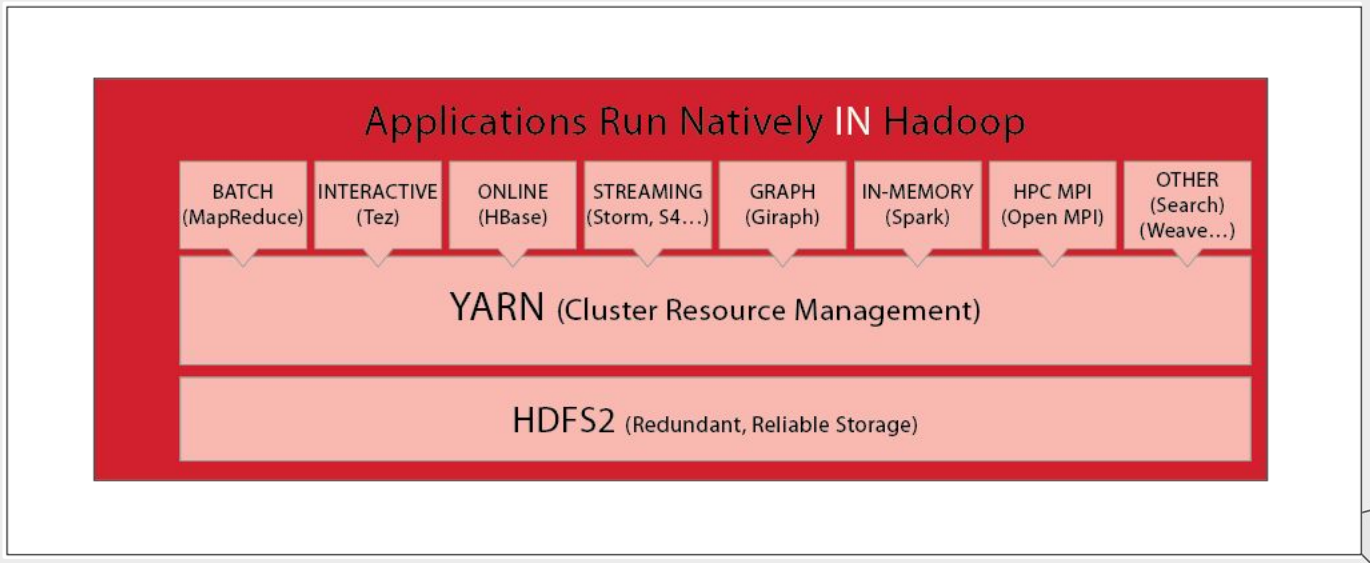
Hadoop 2 and YARN

YARN is the resource management and computation framework that is new as of Hadoop 2, which was released late in 2013.



Hadoop 2 and YARN

YARN supports multiple processing models in addition to MapReduce. All share common resource management service.



Daemon Services

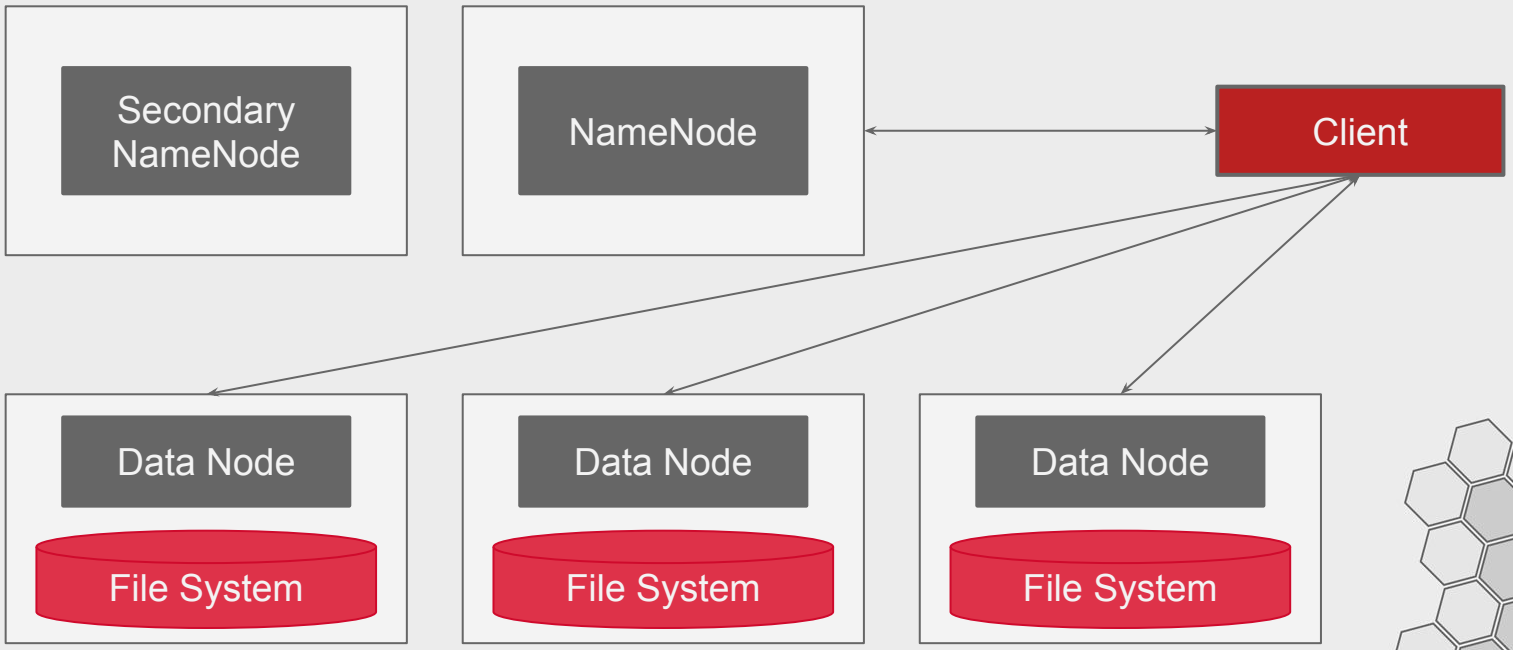
A set of machines that are running HDFS and YARN is called a cluster. Each individual machine is referred to as a node.

A cluster can have a single node or many thousands, the cluster scales linearly - every node you add, you get that much more capacity and performance.

There are two particular kinds of nodes that are differentiated by which processes run on them. Master nodes run the global management processes. Worker nodes run local data and application processes.

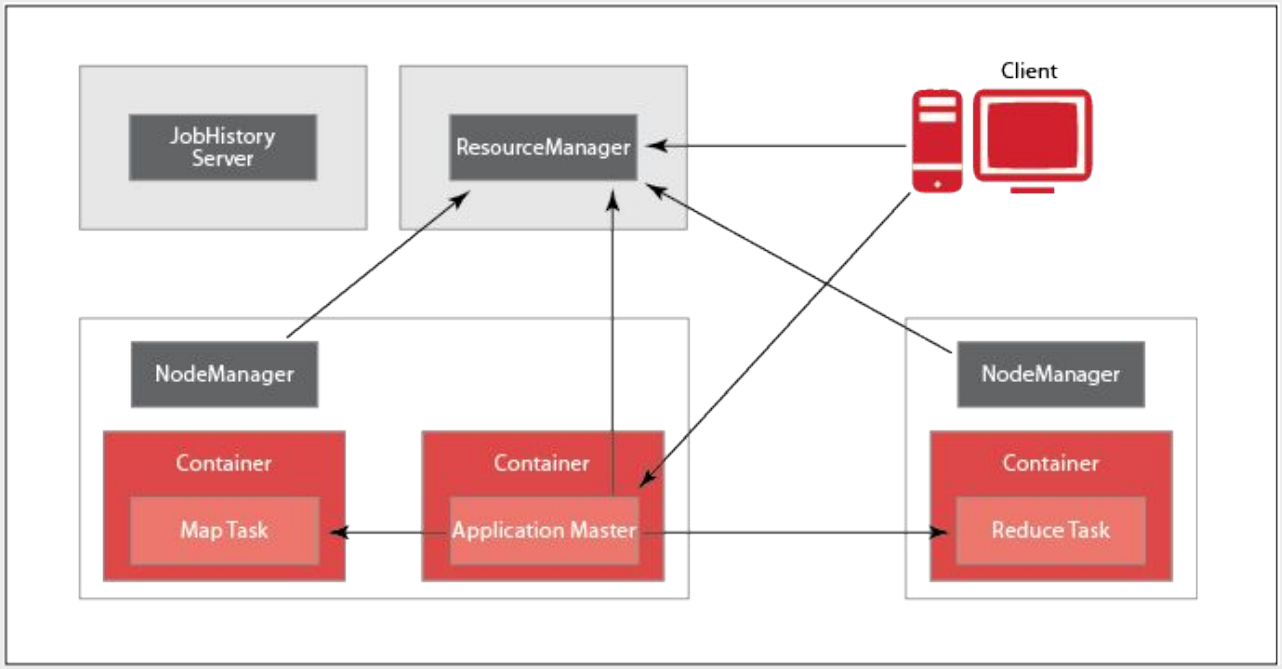
HDFS Daemons

Active Name Node (NN) - coordinates file system operations by clients and specifies the locations of files and blocks. **Data Node (DN)** - per node agent that responds to data requests and manages storage. **Secondary Name Node (SN)** - performs housekeeping and reporting tasks (not a backup).



YARN Daemons

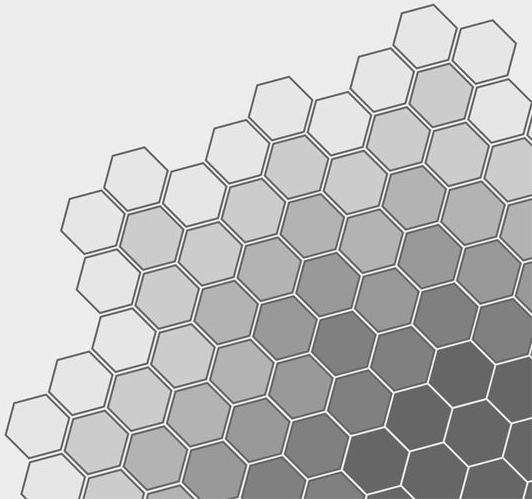
Resource Manager (RM) - serves as the central agent for managing and allocating cluster resources. **Node Manager (NM)** - per node agent that manages and enforces node resources. **Application Master (AM)** - per application manager that manages lifecycle and task scheduling



HDFS

HDFS provides redundant storage of extremely large data sets by keeping data in a cluster of cheap, unreliable, but cost effective computers.

HDFS ensures that disks can be added to linearly increase storage space, but especially that nodes can be added to increase both redundancy and storage capacity.



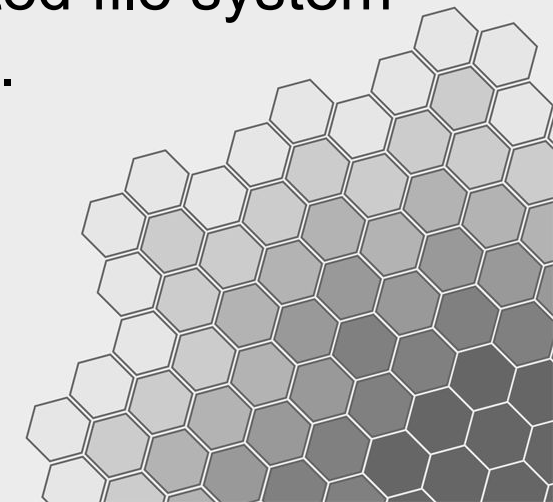
Data Local Computing

HDFS

By spreading data across a cluster, HDFS ensures that data is made local to computational processes.

Using multiple computers for redundancy also provides increased file storage and faster disk reads, increasing the number of available processors to surface disk space.

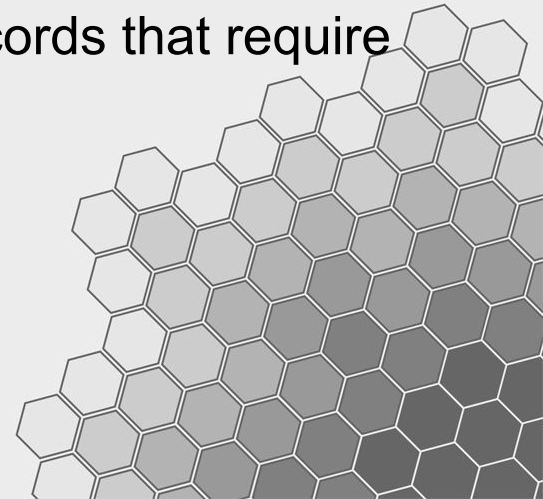
However, the networked nature of a distributed file system makes it more complex than traditional ones.



WORM

HDFS

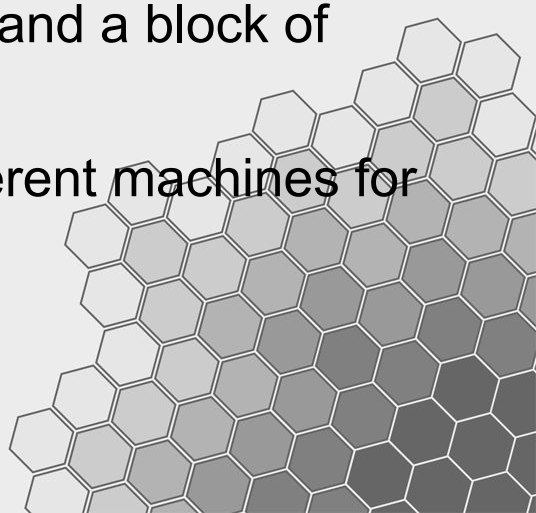
- HDFS performs best with a modest number of large files e.g. millions of files that are 100MB or more, not billions of files that are only a single MB.
- HDFS is a WORM storage, write once, read many. No random writes or appending to files are allowed.
- HDFS is optimized for large, streaming reads of files, not random access or selection.
- Use HDFS for storing input data for computations and files needed for intermediary computation. Do not use it for records that require lookups in real time.



Blocks

HDFS

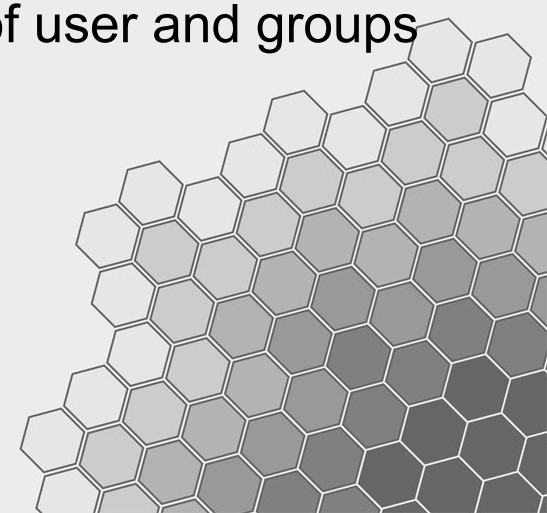
- All files are split into blocks, usually either 64 or 128 MB, but this is configurable. Block size is the minimum amount of data that can be read or written to in HDFS, similar to blocks on a disk based file system.
- Blocks allow big files to be split across and distributed to many machines at run time. Different blocks containing the same file will be stored on multiple machines to provide more efficient processing.
- There is a one-to-one connection between a task and a block of data (each task gets a block to work on).
- Blocks are replicated, usually stored on three different machines for durability and to minimize network transfer.



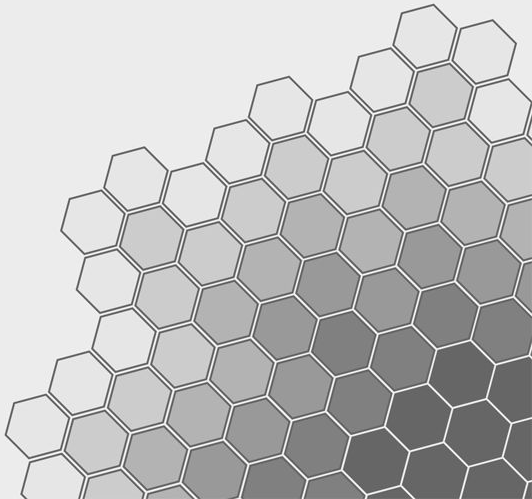
Permissions and FS

HDFS

- File permissions are handled similarly to posix systems by assigning an owner and group to every file and directory, as well as read and write permissions. (There is no concept of executable files in HDFS)
- Most HDFS deployments have a `/user/` directory at the root, which acts similarly to `/home/` on Linux. When users execute HDFS commands with a relative path, it is assumed to be from their “home” directory.
- Every process has a two part identity consisting of user and groups on which HDFS performs checks.



MapReduce



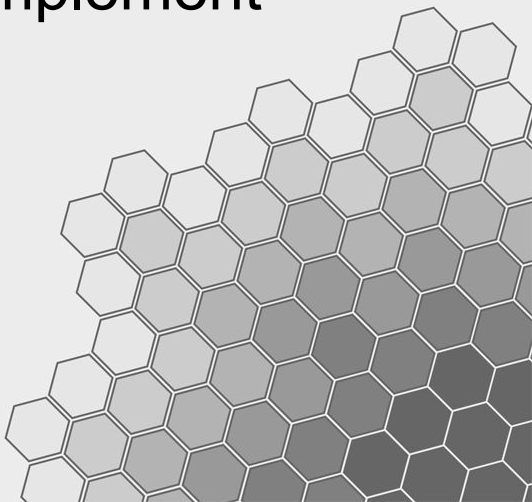
Functional Programming

MapReduce is a functional programming paradigm:

Map ⇒ Reduce

On Hadoop, MapReduce is the first and primary methodology for cluster computing.

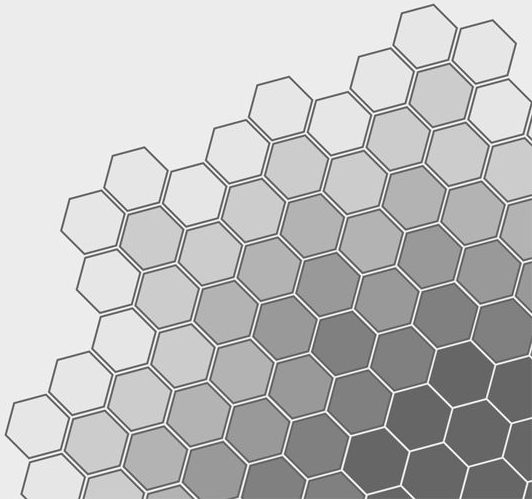
Simple, but powerful and flexible enough to implement many analytical algorithms in parallel.



Parallelizable Functions

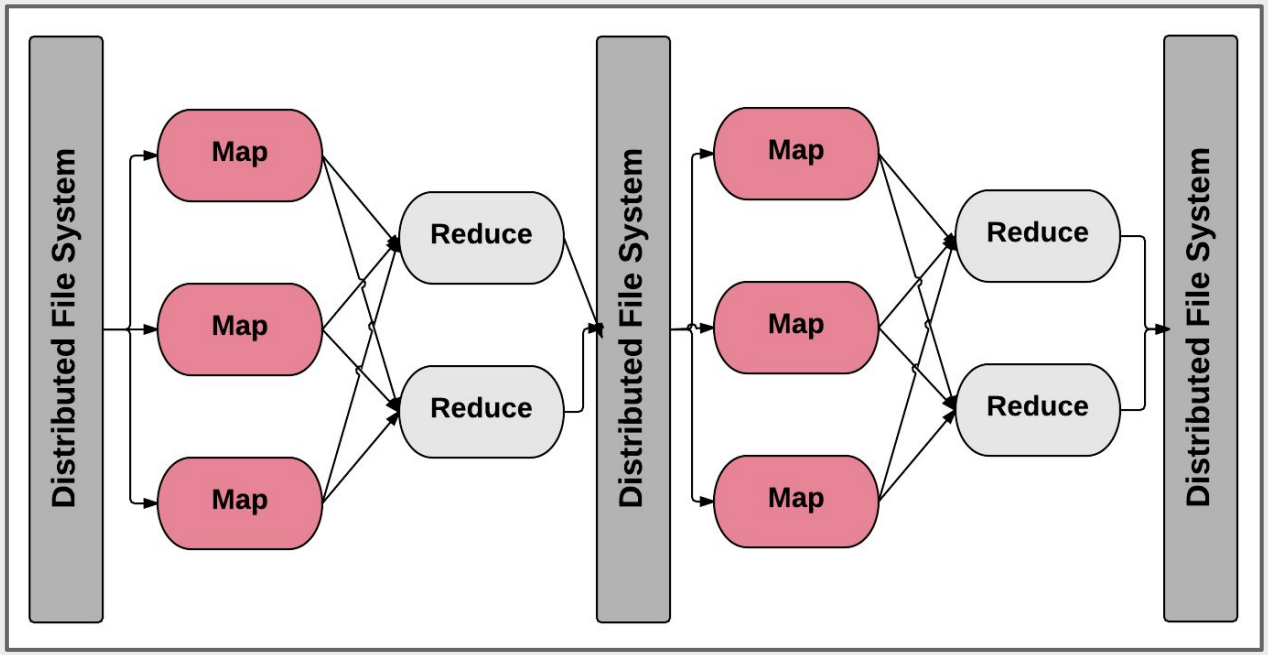
MapReduce is parallel by design, hence its selection for use in a distributed computing framework!

```
def map(key, value):
    ...
    return (intermed_key, intermed_value)

def reduce(key, value):
    ...
    return output
```



MapReduce Execution

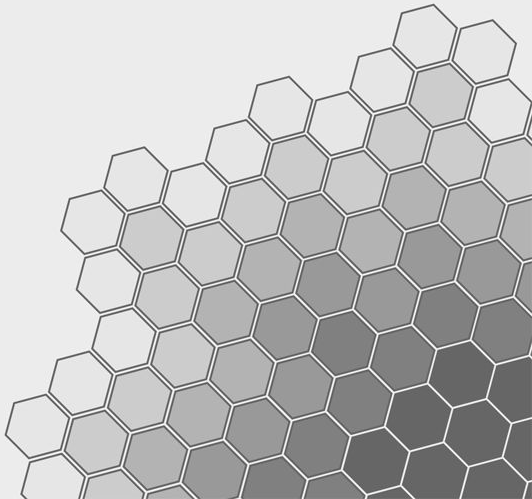
A MapReduce job is composed of many Map and Reduce tasks that operate on data that is stored locally, thus minimizing network traffic.



MapReduce on Hadoop

Hadoop takes care of the low-level distributed computing details:

- automatic parallelization
- job distribution
- job management
- fault tolerance
- monitoring

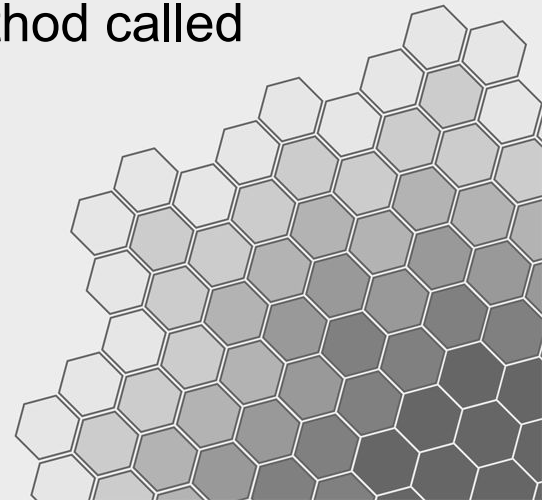


Jobs and Speculative Execution

A *Job* is a full program, the complete execution of Map and Reduce functions across all input data.

A Job is composed of many *tasks*, the execution of a single attempt at Map or Reduce on a block of data. Tasks are presided over by the NodeManager.

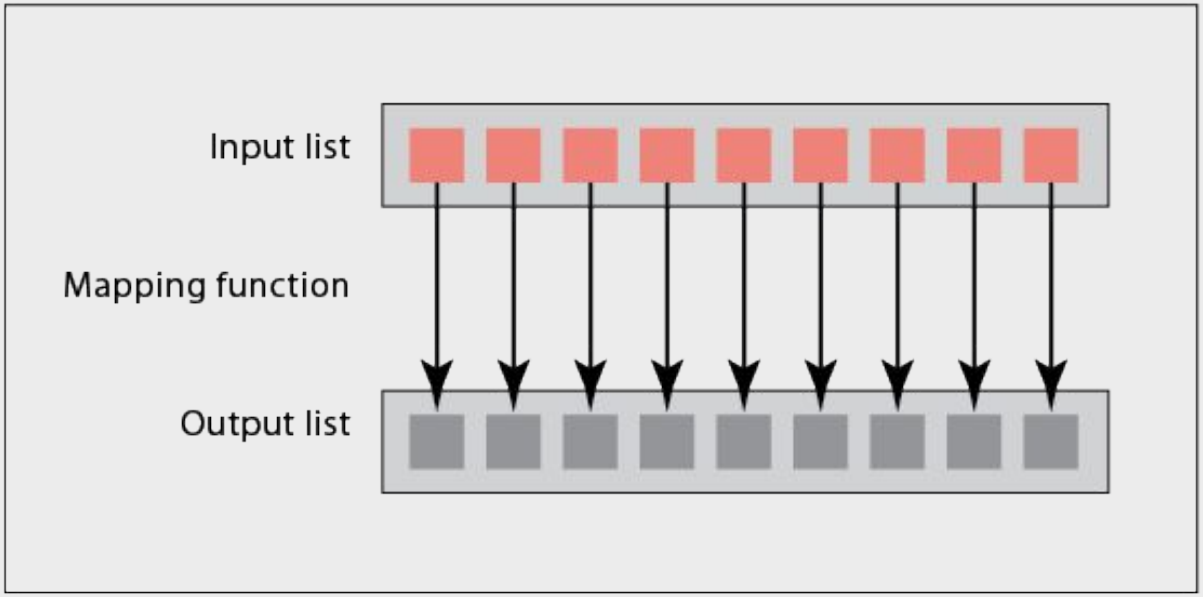
A *task attempt* is a particular instance of a task. There are at least as many attempts as tasks - but if attempts become slow or unresponsive, Hadoop ensures consistency in computation by a method called *speculative execution*.

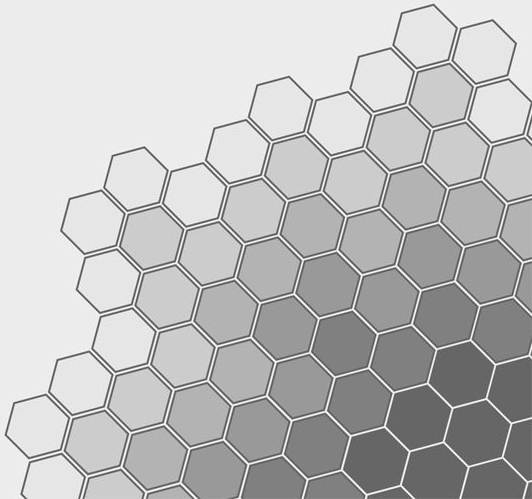


Map Functions

A Map function takes as input a list and operates singly upon each individual element in the list, e.g. mapping a function to each item in a list.

In the case of Hadoop, the mapper receives a list of key value pairs and is expected to output zero or more key/values.

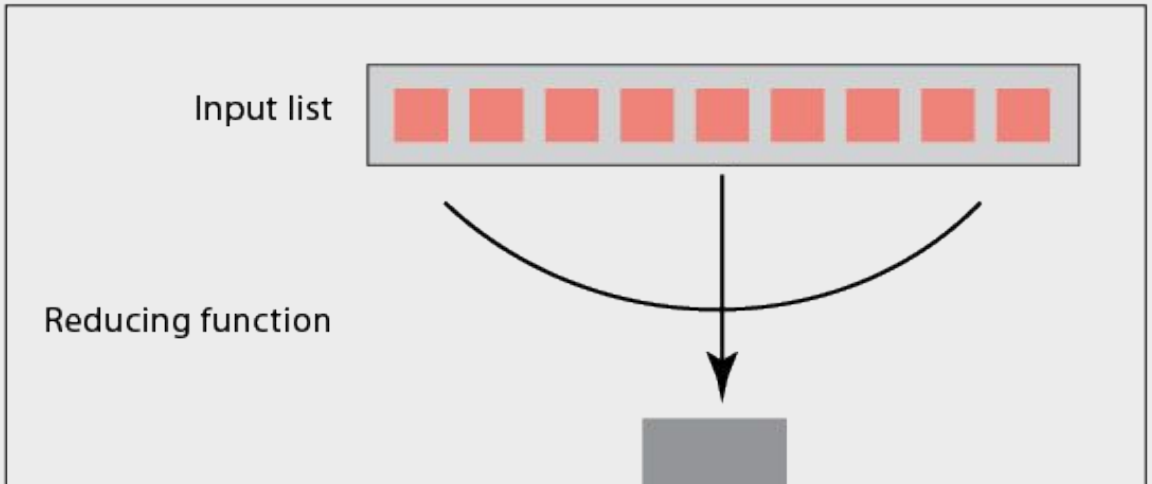




Reduce Functions

A Reducer also takes a list as input, but then combines the values in the list to a single output value.

Hadoop Reducers receive that list of values on a per-key basis via the key/value pairs the Mapper output.

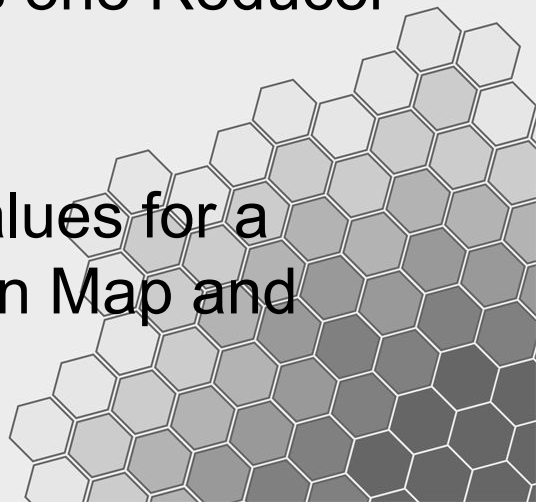


Parallelization

Because Mappers apply the same function on any list of items, they are uniquely suited to being distributed across an entire computational cluster. The output of a Map function is not dependent on anything but the incoming value.

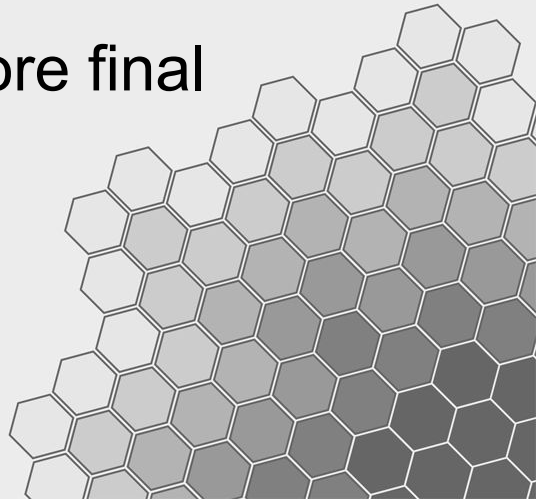
Because Reducers receive all values from Mappers on a per key basis, they can also be distributed as one Reducer per one key.

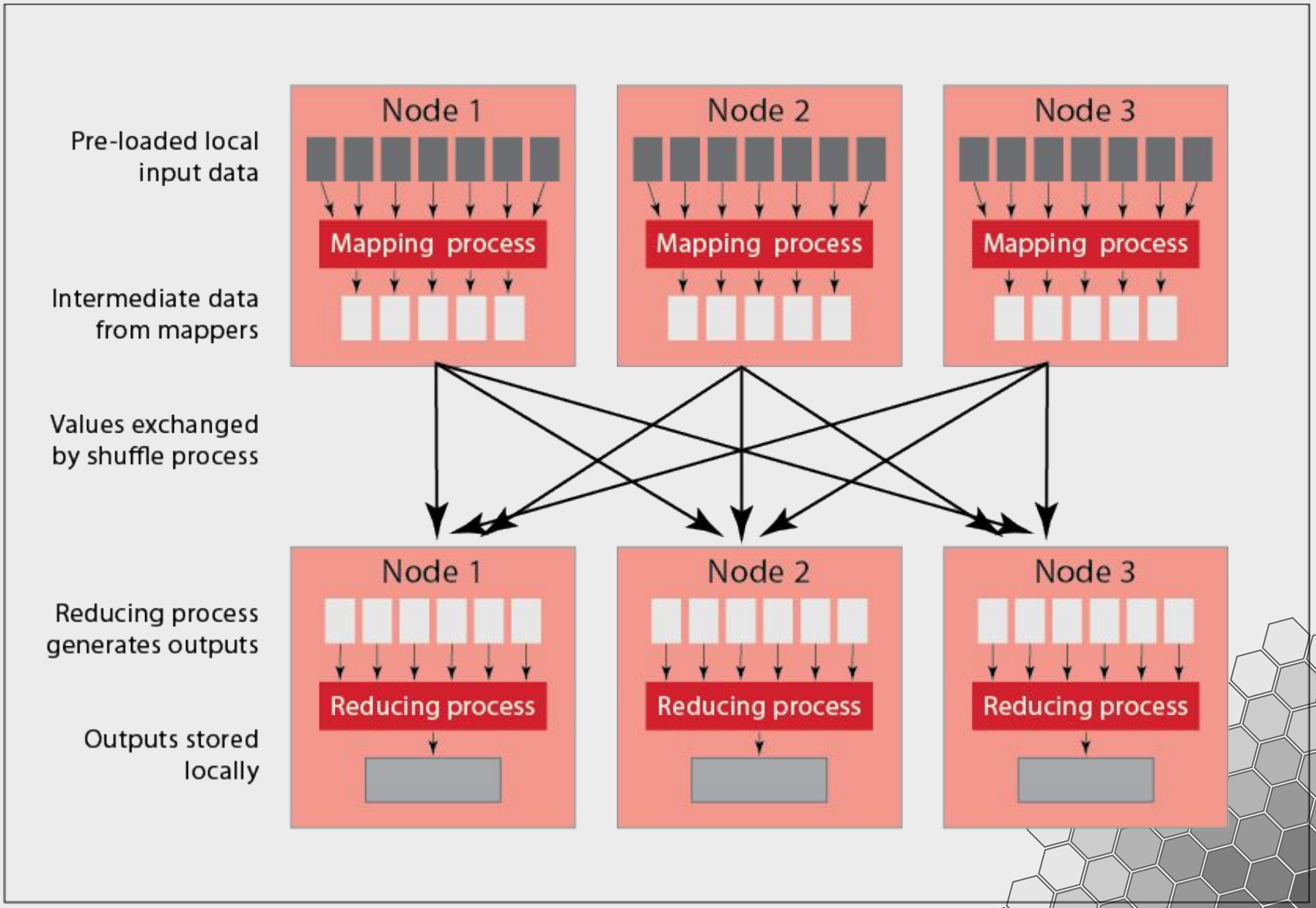
The guarantee that a Reducer receives all values for a single key requires a shuffle and sort between Map and Reduce.



MapReduce Phases

1. Local data is loaded into a mapping process as key/value pairs from HDFS.
2. The Mapper outputs zero or more key value pairs, mapping computed values to a particular key.
3. These pairs are then sorted and shuffled based on the key and are then passed to a reducer such that all values for a key are available to it.
4. The Reducer then must output zero or more final key/value pairs which are the output.
5. Output is then written back to HDFS.




Pre-loaded local input data
Intermediate data from mappers
Values exchanged by shuffle process
Reducing process generates outputs
Outputs stored locally
Node 1
Mapping process
Node 2
Mapping process
Node 3
Mapping process
Node 1
Reducing process
Node 2
Reducing process
Node 3
Reducing process

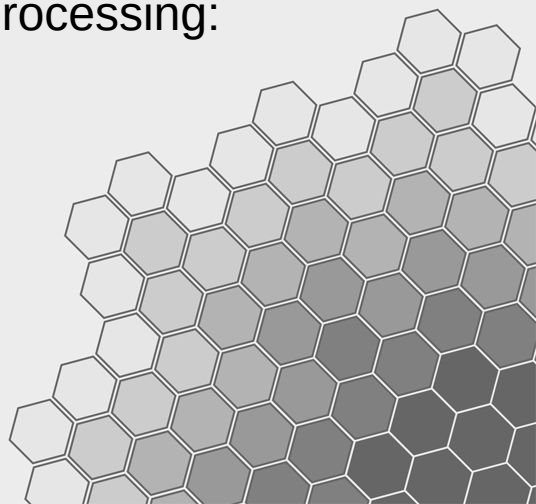
Word Frequency

count how often a word appears in a document or collection of documents (corpus).

Is the “canary” of Big Data/Distributed computing because a distributed computing framework that can run WordCount efficiently in parallel at scale can likely handle much larger and more interesting compute problems - Paco Nathan

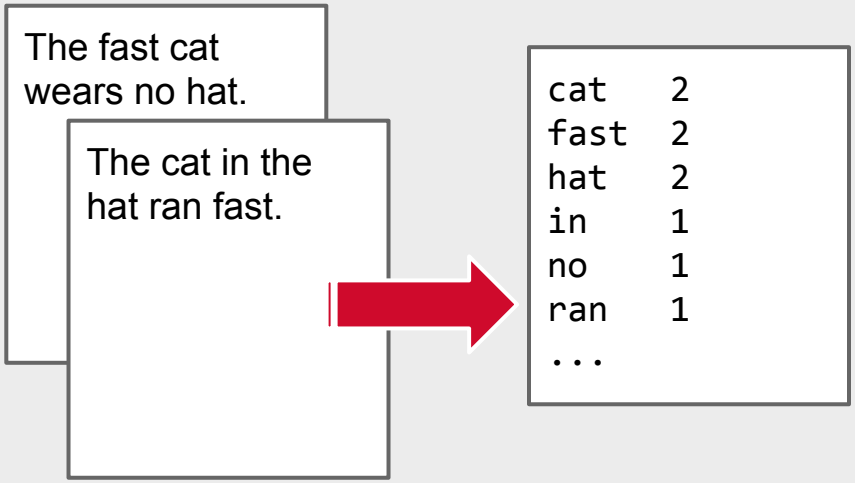
This simple program provides a good test case for parallel processing:

- requires a minimal amount of code
- demonstrates use of both symbolic and numeric values
- isn’t many steps away from search indexing/statistics



Word Frequency

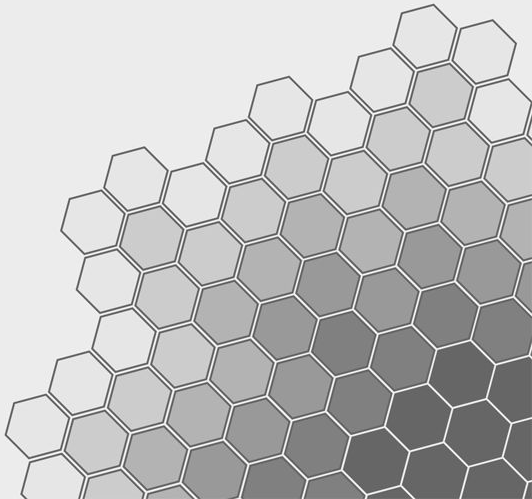
```
def map(key, value):
    for word in value.split():
        emit(word, 1)

def reduce(key, values):
    count = 0
    for val in values:
        count += val
    emit(key, count)

# emit is a function that performs distributed I/O
```



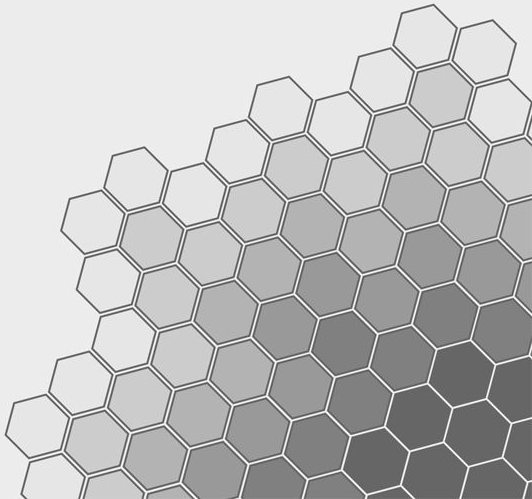
Each document is passed to a mapper, which does the tokenization. The output of the mapper is reduced by key (word) and then counted.

What is the data flow for word count?

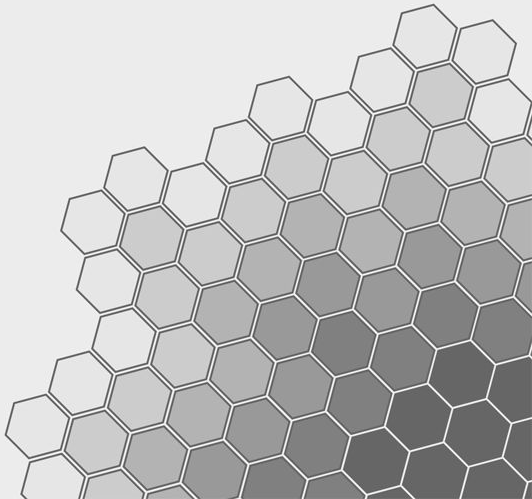
```
# Input to WordCount Mappers

(31416, “the cat in the hat ran fast”)
(27183, “the fast cat wears no hat”)

# Output Mapper 1

(“the”, 1), (“cat”, 1), (“in”, 1), (“the”, 1),
(“hat”, 1), (“ran”, 1),(“fast”, 1)

# Output Mapper 2

(“the”, 1), (“fast”, 1), (“cat”, 1),
(“wears”, 1), (“no”, 1), (“hat”, 1)
```

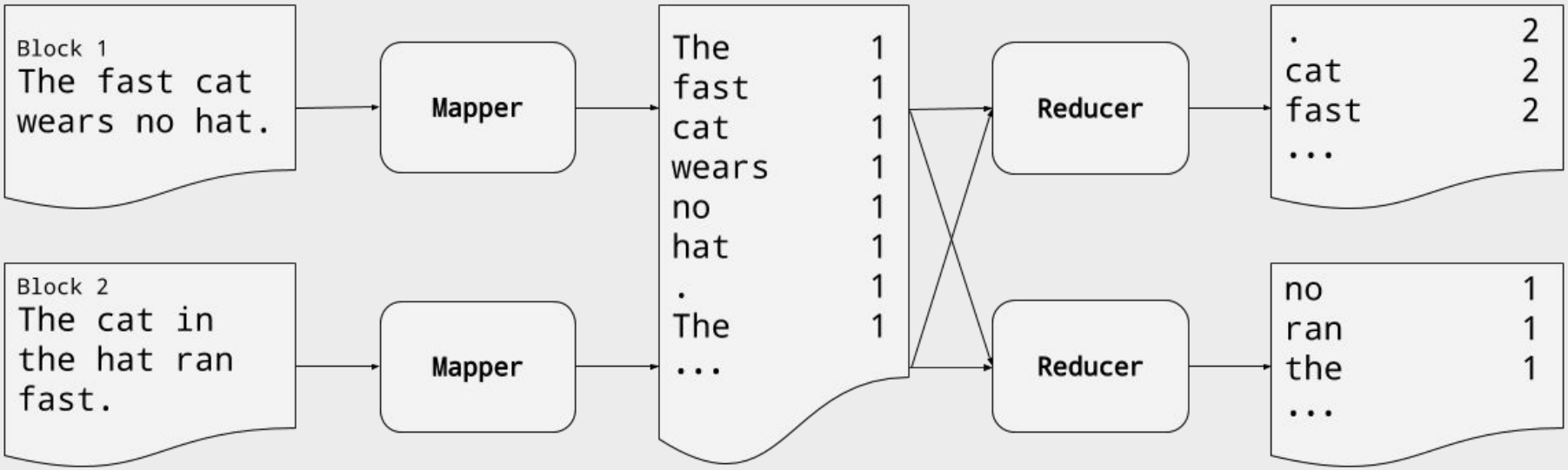


```
# Input to WordCount Reducers
# This data was computed by shuffle/sort

(“cat”, [1, 1])
(“fast”, [1, 1])
(“hat”, [1, 1])
(“in”, [1])
(“no”, [1])
(“ran”, [1])
(“the”, [1, 1, 1])
(“wears”, [1])
```



```
# Output by all WordCount Reducers

(“cat”, 2)
(“fast”, 2)
(“hat”, 2)
(“in”, 1)
(“no”, 1)
(“ran”, 1)
(“the”, 3)
(“wears”, 1)
```



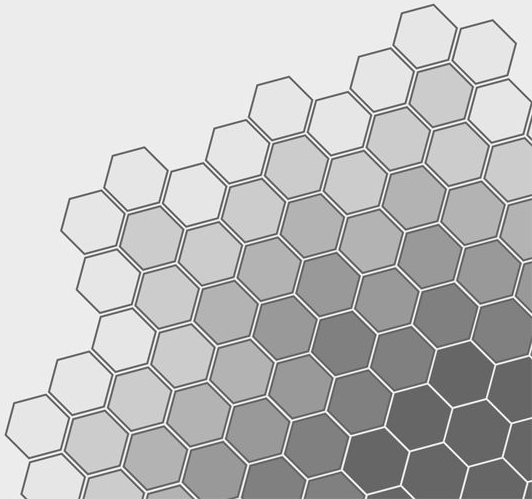


Execution of Word Count

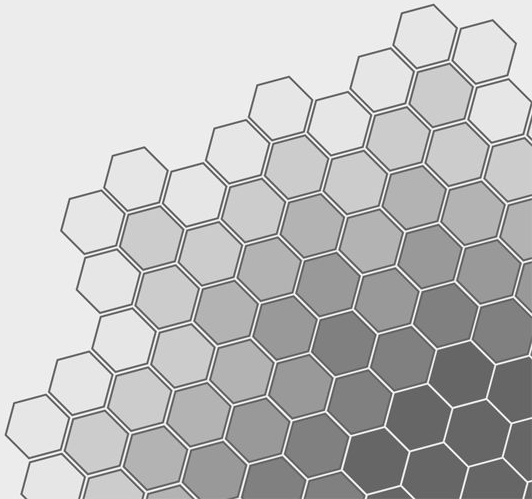
Shared Friendships

For all pairs of friends, find the set of friends that they have in common.

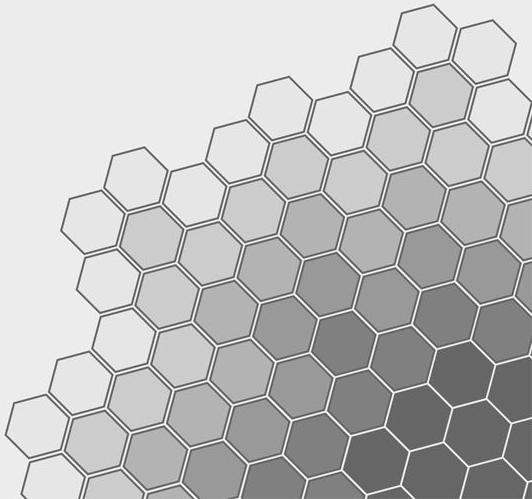
Analyze a social network to see which friend relationships users have in common. First step in social recommendations, e.g. “you might also know”. But also a good example of multi-key joins.



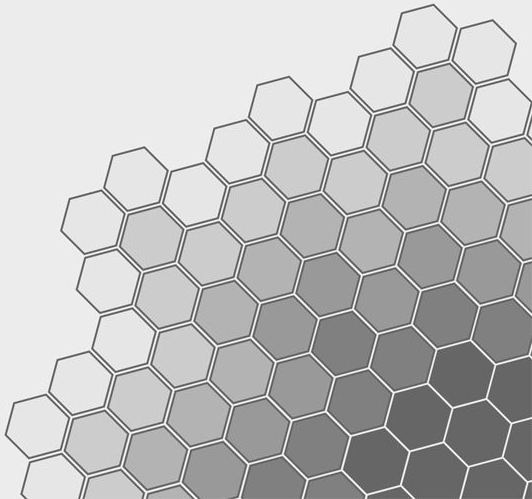
```
# Input (Key → Value)
Allen → Betty, Chris, David
Betty → Allen, Chris, David, Ellen
Chris → Allen, Betty, David, Ellen
David → Allen, Betty, Chris, Ellen
Ellen → Betty, Chris, David
```



```
def map(person, friends):
    for friend in friends:
        pair = sort([person, friend])
        emit(pair, friends)

def reduce(pair, friends):
    shared = set(friends[0])
    shared = shared.intersection(friends[1])
    emit(pair, shared)
```

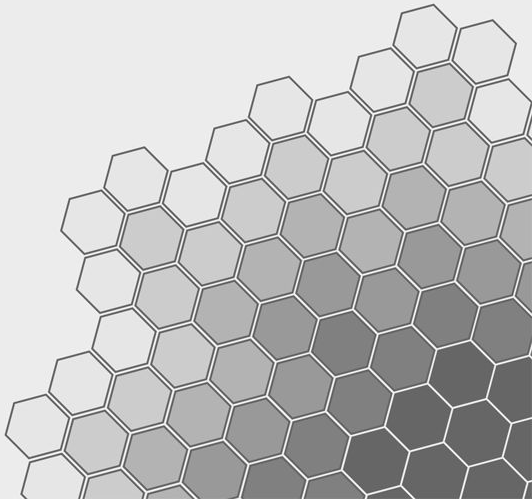


```
# Mapper 1 Output
(Allen, Betty), (Betty, Chris, David)
(Allen, Chris), (Betty, Chris, David)
(Allen, David), (Betty, Chris, Daivd)


# Mapper 2 Output
(Allen, Betty), (Allen, Chris, David, Ellen)
(Betty, Chris), (Allen, Chris, David, Ellen)
(Betty, David), (Allen, Chris, David, Ellen)
(Betty, Ellen), (Allen, Chris, David, Ellen)

# Mapper 3 Output
(Allen, David), (Allen, Chris, David, Ellen)
(Betty, David), (Allen, Chris, David, Ellen)
(Chris, David), (Allen, Chris, David, Ellen)
(David, Ellen), (Allen, Chris, David, Ellen)

# Mapper 4 output
(Betty, Ellen), (Betty, Chris, David)
(Chris, Ellen), (Betty, Chris, David)
(David, Ellen), (Betty, Chris, David)
```



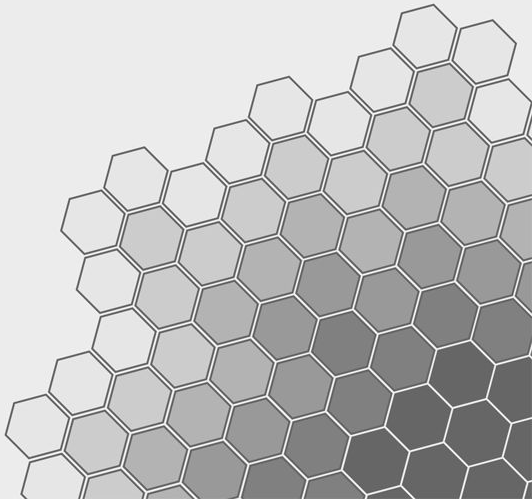
After Sort Reducers see:

(Allen, Betty) → (A C D E) (B C D)
(Allen, Chris) → (A B D E) (B C D)
(Allen, David) → (A B C E) (B C D)
(Betty, Chris) → (A B D E) (A C D E)
(Betty, David) → (A B C E) (A C D E)
(Betty, Ellen) → (A C D E) (B C D)
(Chris, David) → (A B C E) (A B D E)
(Chris, Ellen) → (A B D E) (B C D)
(David, Ellen) → (A B C E) (B C D)



After Reduction:

(Allen, Betty) → (Chris, David)
(Allen, Chris) → (Betty, David)
(Allen, David) → (Betty, Chris)
(Betty, Chris) → (Allen, David, Ellen)
(Betty, David) → (Allen, Chris, Ellen)
(Betty, Ellen) → (Chris, David)
(Chris, David) → (Allen, Betty, Ellen)
(Chris, Ellen) → (Betty, David)
(David, Ellen) → (Betty, Chris)

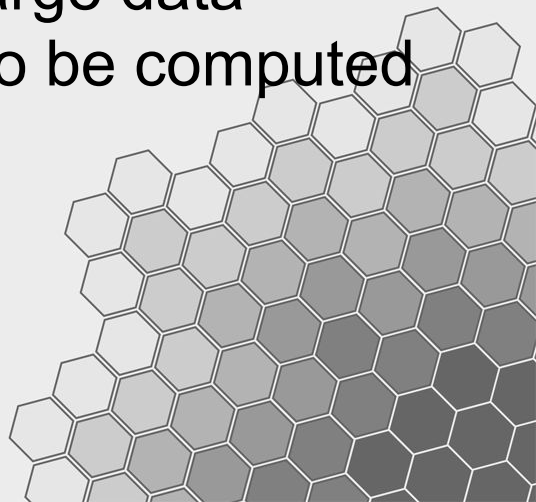


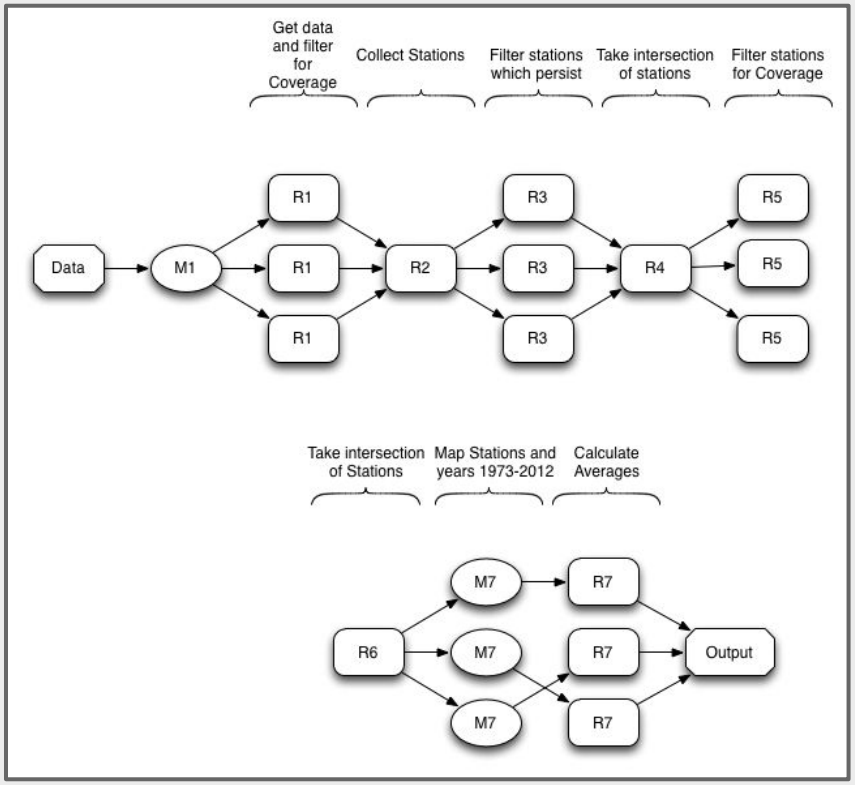
Job Chaining

MapReduce jobs can then be chained together to produce more complex algorithms.

The job of the analyst or developer on Hadoop is to devise algorithms that implement Map and Reduce in order to come to a single analytical conclusion.

Often MapReduce is used to decompose a large data space into a smaller data space that is able to be computed upon in memory.





Map 1: Map list of Files from 1973 to be downloaded

Reduce 1: Filter list to only include stations with good coverage

Reduce 2: Take union of sets from Reduce 1

Reduce 3: Filter list to find stations which persist until the present

Reduce 4: Take intersection of sets from Reduce 3

Reduce 5: Filter list to find stations which have good coverage for each year

Reduce 6: Take intersection of sets from Reduce 5

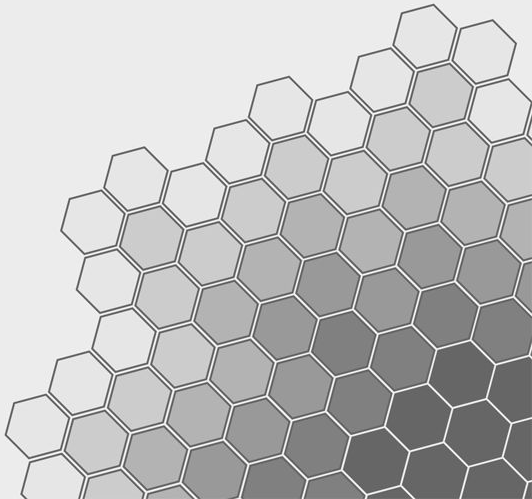
Map 7: Map list of stations for year. Key = Year, Value = Station ID

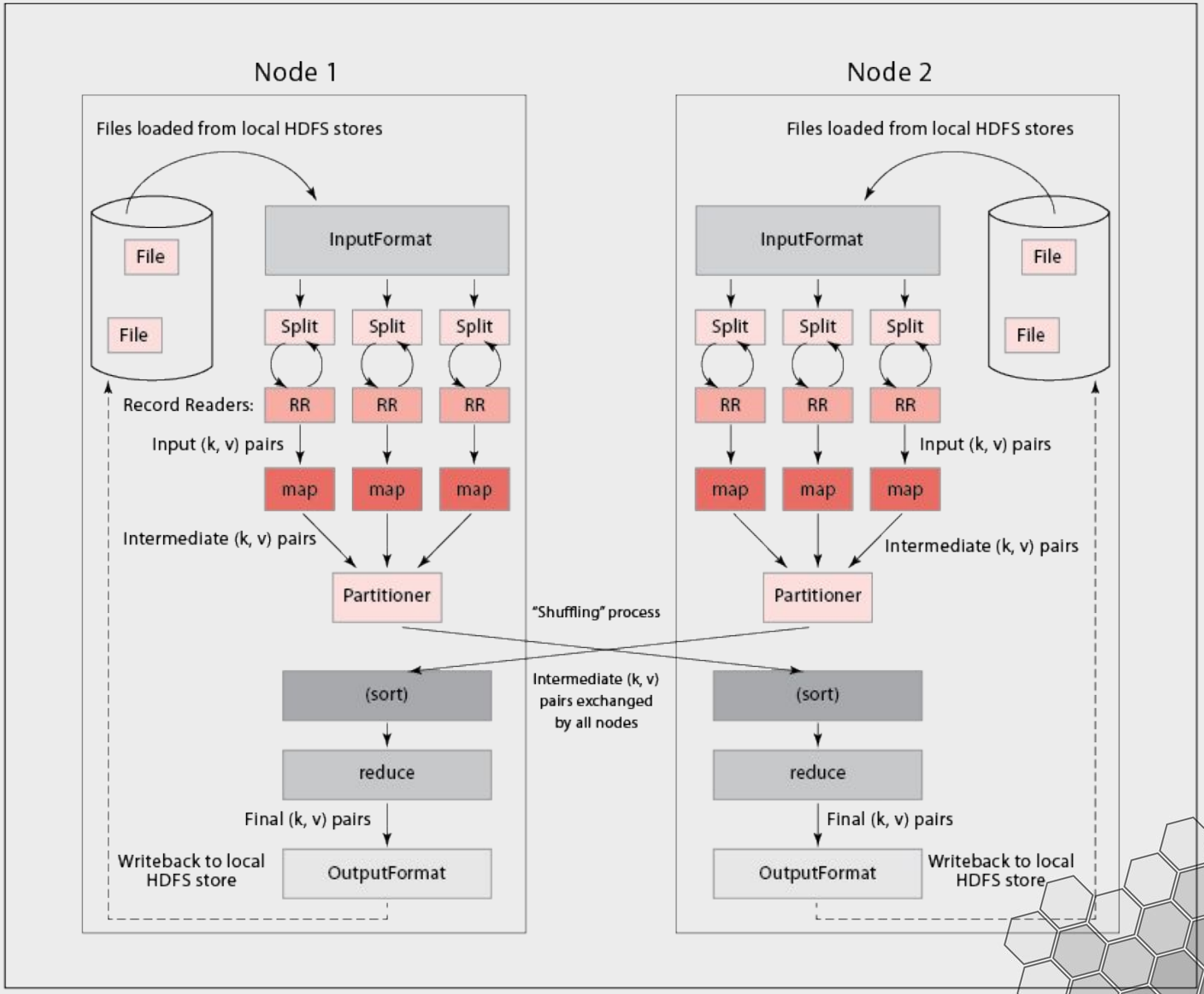
Reduce 7: Calculate Average and Standard Deviation for all stations in a year

Hello NOAA, by Benjamin Zaitlen: http://continuum.io/blog/hello-noaa

MapReduce in Detail

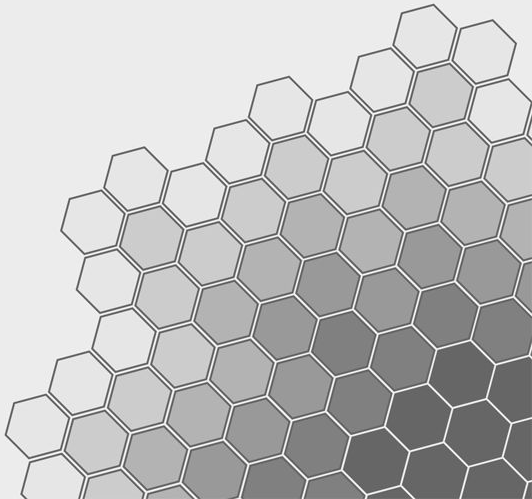
- InputFormat
- Counters/Reporters
- MapTask
- Combiners
- Shuffle and Sort
- Partitioners
- ReduceTask
- OutputFormat





Java-Focused Class!

Common MapReduce Tasks



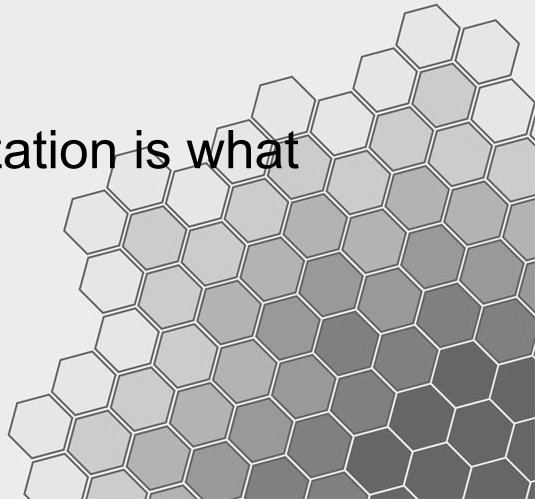
Data Flows

When thinking about constructing MapReduce jobs, the idea is to think about *data flows* - how does your data flow from mapper to reducer to mapper, etc?

One way to think about these tasks is in broad classes:

- Summarization
- Filtering
- Aggregation
- Organization

A series of filters, summaries, aggregates and organization is what makes up most MapReduce jobs.



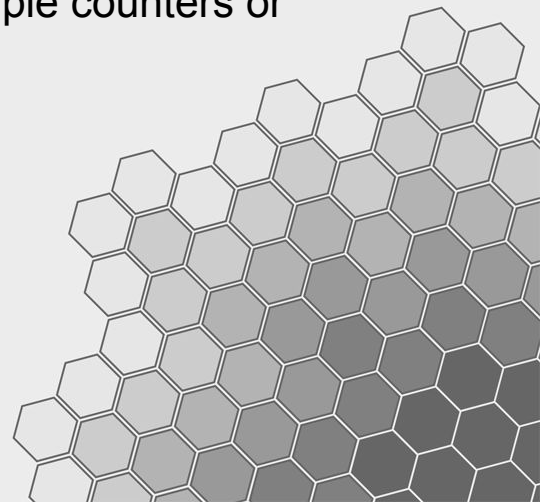
Common Mappers

The Explode Mapper
Divide a value into its constituent parts

The Filter Mapper
Output values only if they meet a particular criteria

Keyspace Changes
Invert the key or value or change the space of the key to influence how the reducer interacts with the value.

Valuespace Changes
Create values that do not necessarily depend on the key - for example counters or hashes that are used later in computations.



Common Reducers

The Aggregate Reducer

Compute some aggregate of the values (Sum or Average commonly)

Superlative Reducers

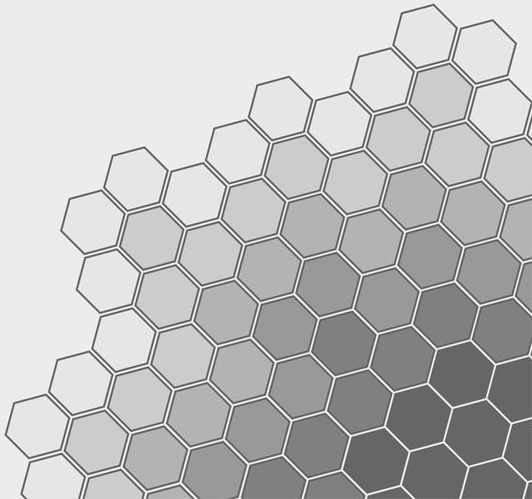
Find a distinct value that meets some criteria (Minimum or Maximum)

Keyspace Changes

Change the key for a grouped set of values so that they're all emitted to the next phase of mappers based on their sorted value.

Filtering Reducers

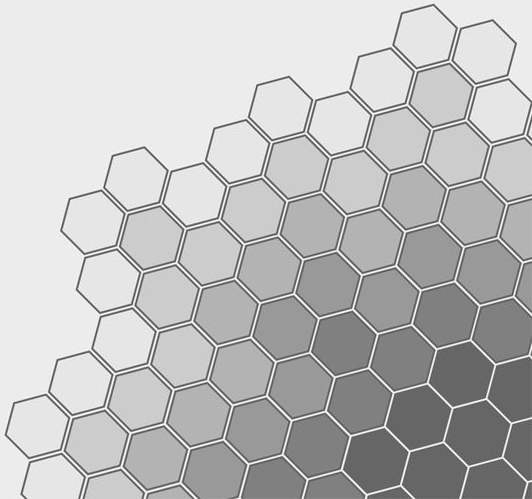
Output only a subset of the values for a particular key.



The Identity Pattern

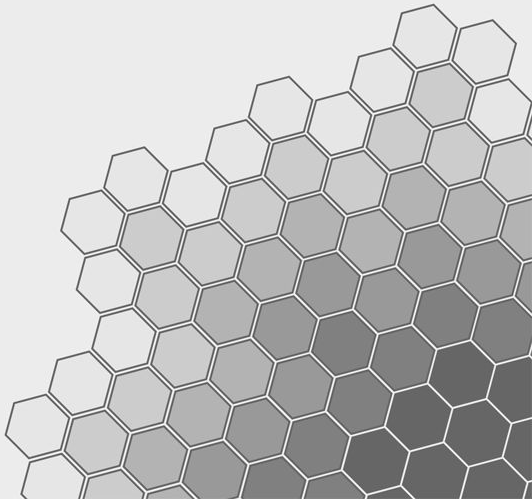
A pass through pattern that is typically implemented on Reducers. In functional terms, the identity relationship returns as output the same key and value as input.

```
def identityMapper(key, value):
   emit(key, value)

def identityReducer(key, values):
   for value in values:
      emit(key, value)
```



Search Based Filtering (grep)

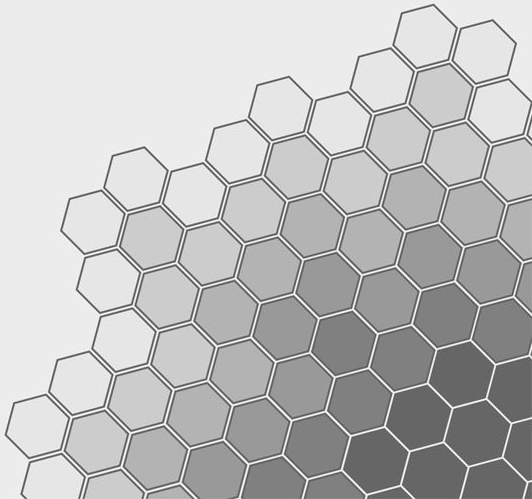
```
pattern = re.compile(input_pattern)
#pattern = re.compile(r'[-\w.+_]+@[-\w.+_]+\.[a-zA-Z]{2,4}')

def map((filename, lineno), line):
    if pattern.search(value):
        emit(filename, None)


def combine(filename, values):
    emit(filename, None)


def identityReducer(key, values):
    pass
```



Count Distinct

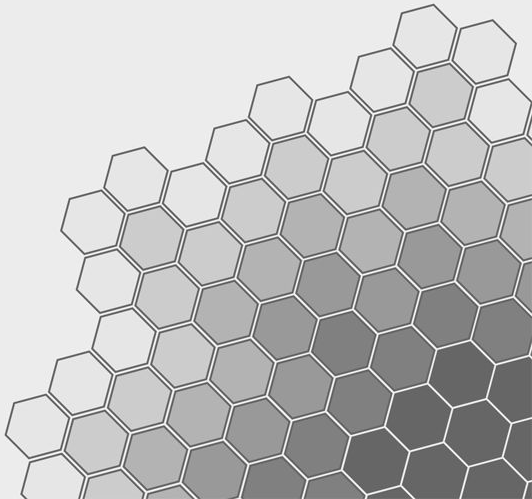
Given a set of records and an arbitrary list of categories, count the total number of unique values for each subset of records for each category.

```
1,“Washington DC”,”{startup, government, data}”
2,“San Francisco”,”{data, startup, tech}”
3,“Washington DC”,”{government}”
4,“New York",”{data, tech}”

data:       3 (rows 1,2,4)
startup:    2 (rows 1,2)
tech:       2 (rows 2,4)
government: 1 (rows 1)
```



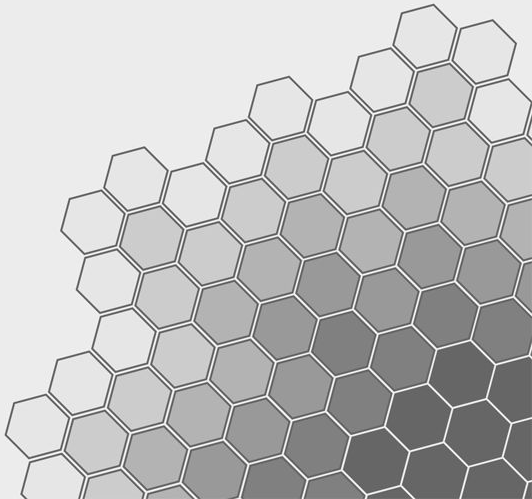
Count Distinct Job 1

```
def mapper(_, value):
    # Parse records (simple implementation)
    record, city, categories = value.split(“,”)
    categories = categories.split(“,”)

    # Emit a “dummy counter”
    for category in categories:
        emit((category, city), 1)


def reduce((category, city), counts):
    emit((category, city), None)
```



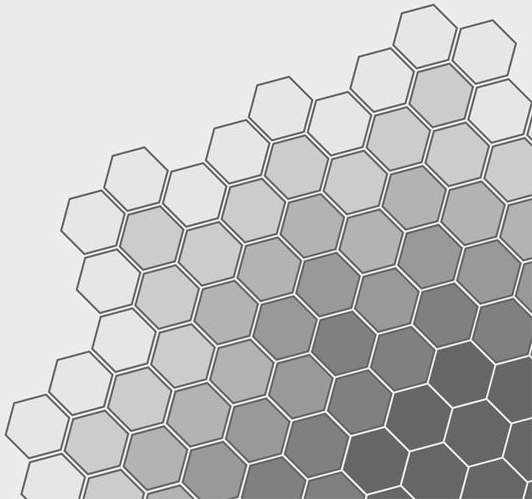
Count Distinct Job 2

```
def mapper((category, city), _):
    emit(category, 1)


def reduce(category, counts):
    emit(category, sum(counts))
```



Descriptive Statistics

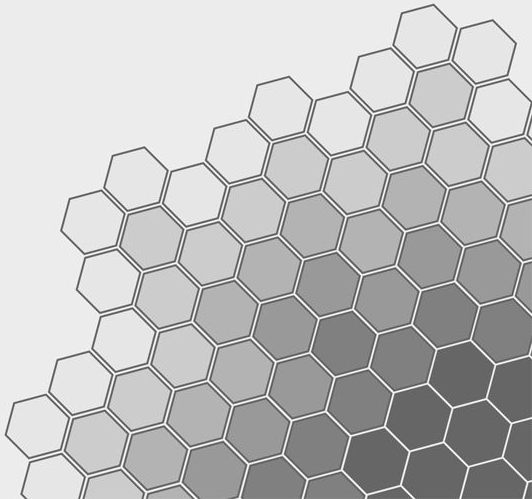
```
def map(key, value):
    emit(key, (1, value, value*value, value, value))


def combine(key, values):
    total, count, square = 0,0,0
    minimum, maximum = None, None
    for num, val, sqr, minval, maxval in values:
        total  += val
        count  += num
        square += sqr
        if minimum is None or minval < minimum:
            minimum = minval
        if maximum is None or maxval > maximum:
            maximum = maxval
    emit (key, (count, total, square, minimum, maximum))
```



Cross Correlation (Co-Occurrence)

Given a set of items in transactions or records, find out which items occur together and how often, creating a matrix containing the frequency of the relationship of some set of items.

Co-occurrence is the basis for many algorithms, especially recommendation systems, graph construction, community analysis - as well as many machine learning algorithms.

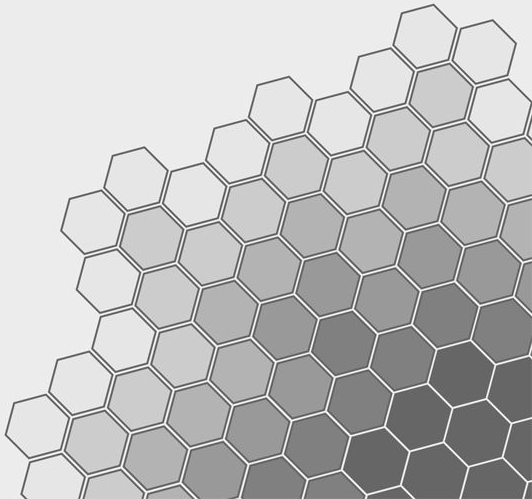


Pairs vs. Stripes

Particularly in matrix computations some approach is required to parallelize multiple related values (rows or columns). Two approaches are pairs and stripes.

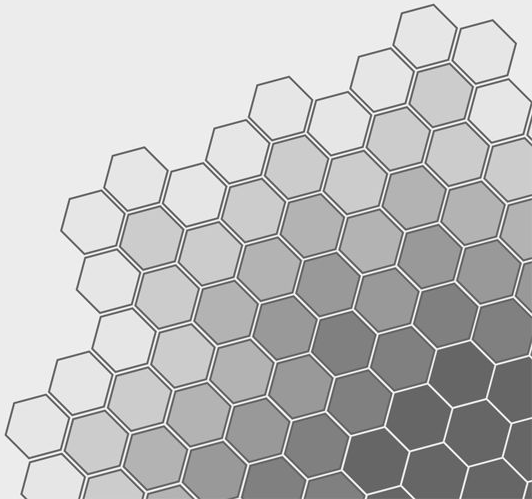
Pairs: Emit the related values as a combined key. For example for rows/cols - the key would be (i,j) val.

Stripes: Leverage the fact that every reducer gets the same values for a key. Collect per-term vectors as an associative array for each key.



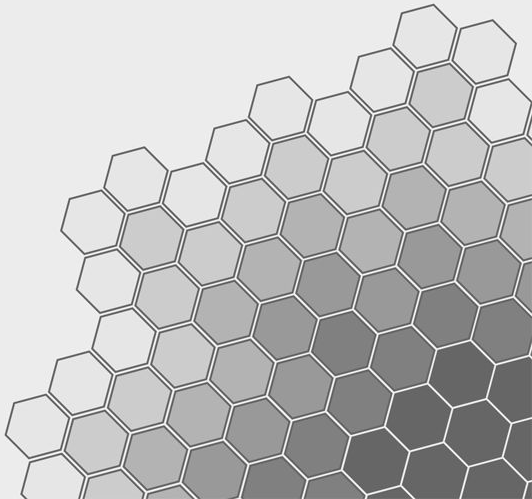
Generic Co-Occurrence

```
order_id, customer_id, product_id, quant

def mapper(docid, line):
   emit(order_id, product_id)

def reducer(order_id, products):
   emit(order_id, “,”.join(products))

order_1, prod,prod,prod,prod …
order_2, prod,prod,prod …
```



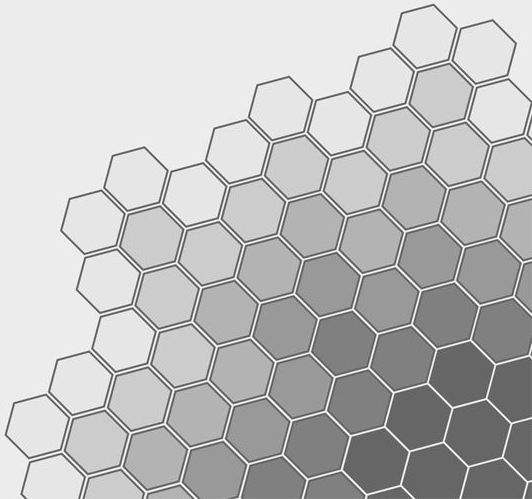
Pairs Method

```
def mapper(key, value):
   products = value.split(“,”)
   for proda in products:
      for prodb in products:
         emit((proda,prodb), 1)

def reducer(key, values):
   emit(key, sum(values))
```



Stripes Method

```
from collections import defaultdict

def mapper(key, value):
    products = value.split(“,”)
    for product in products:
        H = defaultdict(int)
        for item in products:
            H[item] += 1
        stripe = “,”.join(str(t) for t in H.items())
        emit(product, stripe)


def reduce(key, values):
    H = defaultdict(int)
    for stripe in values:
        for k,v in stripe:
            H[k] += v
    for item in H:
        emit((key, item), H[item])
```



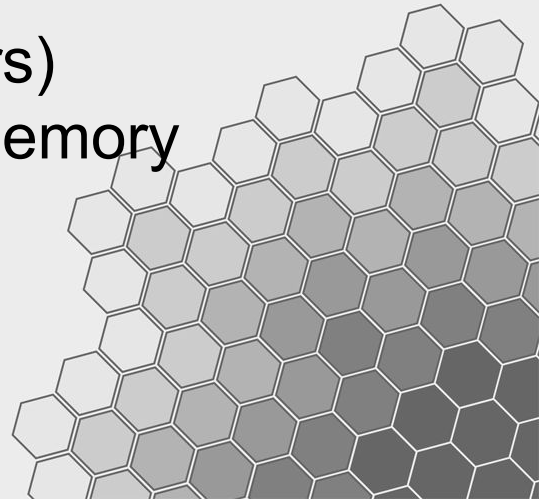
Pairs vs. Stripes

Pairs

- Memory safe generation of data structures
- Combiners don’t help; pairs are distinct

Stripes

- Fewer intermediate keys (less sorting)
- Generally faster than pairs (use combiners)
- Associative Arrays (maps) can blow up memory

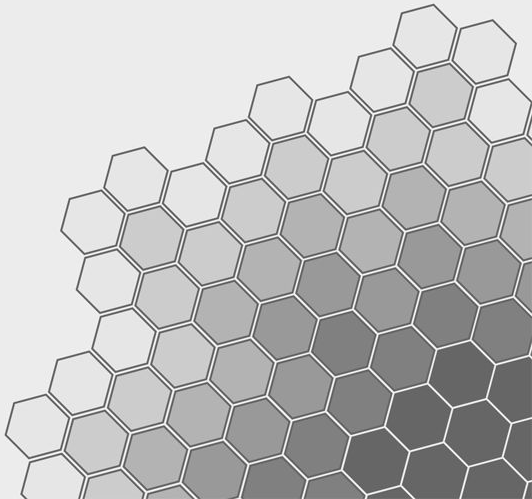


Summarizing Documents with TF-IDF

Term Frequency - Inverse Document Frequency is a technique to determine the significance of words to documents in a larger corpus.

Used as a feature in many machine learning algorithms and in search. Can also augment larger processes like social network analysis, web analysis and topic modeling.

This is also typically the first example of a chained MapReduce job since three MapReduce computations are required.

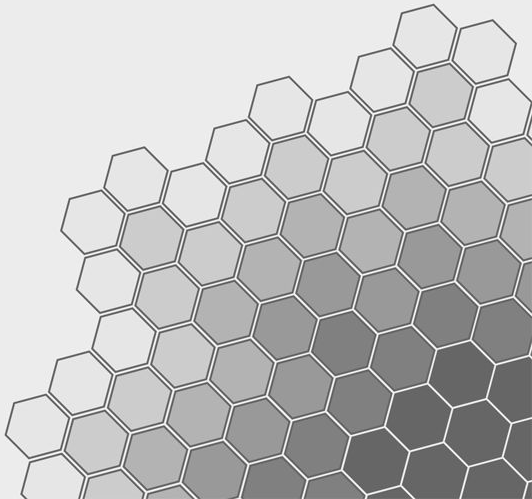


Summarizing Documents with TF-IDF

The term frequency is the relationship between the number of times a term appears in a document and the number of terms in the document.

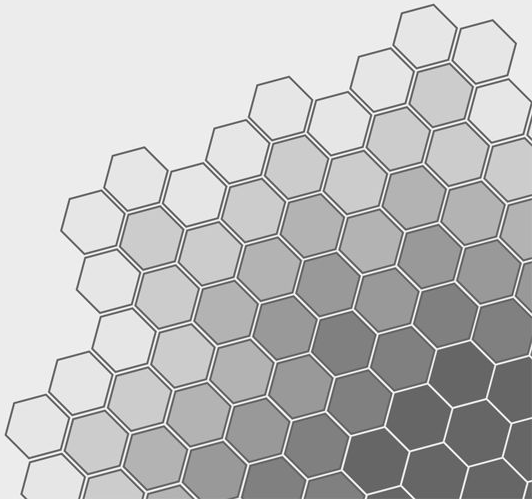
The inverse document frequency is the logarithmic relationship between the total number of documents and the number of documents the term appears in.

TF-IDF is the combined term frequency and inverse document frequency scores.



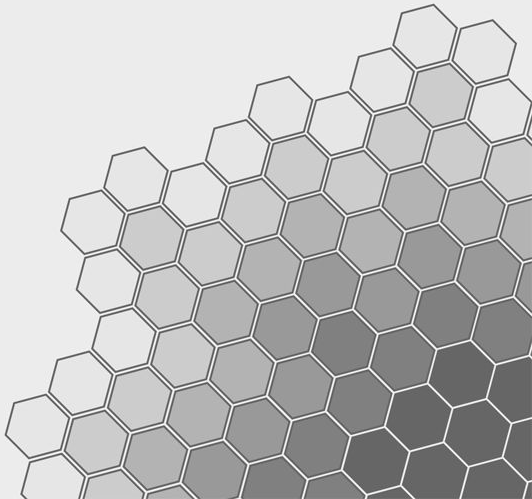
TF-IDF Job 1

```
# Word Frequency Per Document

import re
tokenize = re.compile(r'\W+')


def mapper(docid, line):
    for word in re.split(tokenize, line):
        emit((word, docid), 1)


def reducer((word, docid), counts):
    emit((word, docid), sum(counts))


combiner = reducer
```



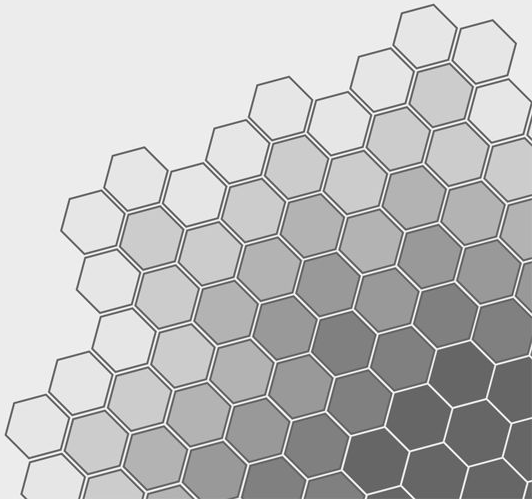
TF-IDF Job 2

```
# Words Per Document

def mapper((word, docid), tf):
    emit(word, (docid, tf, 1))


def combiner(word, values):
    terms = sum(num for (docid, tf, num) in values)
    emit(word, (docid, tf, terms))


def reducer(word, values):
    terms = sum(num for (docid, tf, num) in values)
    for docid, tf, num in values:
        emit((word, docid), (tf, terms))
```

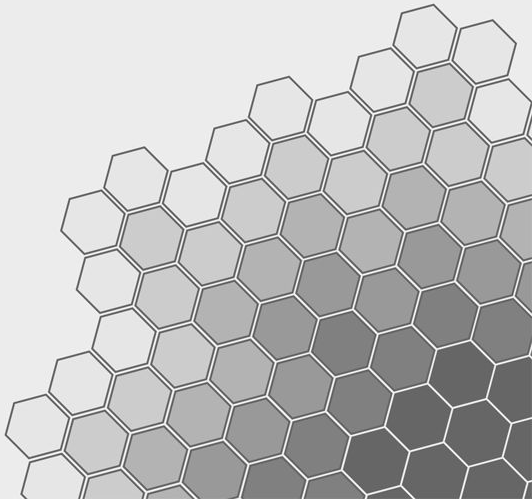


TF-IDF Job 3

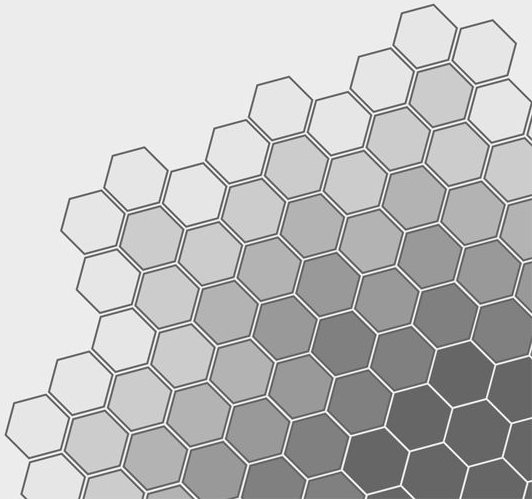
```
# Compute IDF

import math

def mapper((word, docid), (tf, n)):
    # Assume the number of documents is known
    # N is the number of documents in the corpus
    idf = math.log(N/n)
    emit((word, docid), idf*tf)


def identityReducer(key, values):
    pass
```



Spark



Spark is a *fast* and *general-purpose* cluster ***computing framework*** (like MapReduce) that has been implemented to run on a resource managed cluster of servers

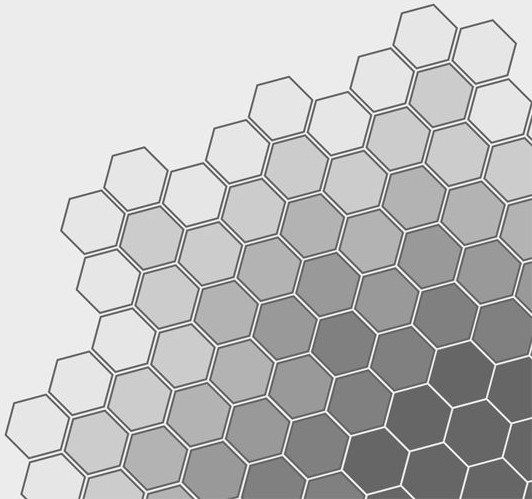


Motivation for Spark

MapReduce has been around as the major framework for distributed computing for 10 years - this is pretty old in technology time! Well known limitations include:

1. Programmability
    a. Requires multiple chained MR steps
    b. *Specialized* systems for applications
2. Performance
    a. Writes to disk between each computational step
    b. Expensive for apps to "reuse" data
        i. Iterative algorithms
        ii. Interactive analysis

Most machine learning algorithms are iterative …



Motivation for Spark

Computation frameworks are becoming *specialized* to solve problems with MapReduce

All of these systems present “data flow” models, which can be represented as a directed acyclic graph.

[The State of Spark and Where We’re Going Next]

Matei Zaharia (Spark Summit 2013, San Francisco)

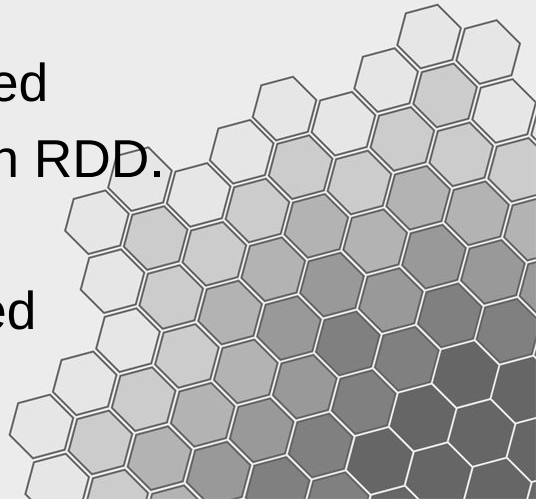
Generalizing Computation

Programming Spark applications takes lessons from other higher order data flow languages learned from Hadoop. Distributed computations are defined in code on a driver machine, then lazily evaluated and executed across the cluster. APIs include:

- Java
- Scala
- Python

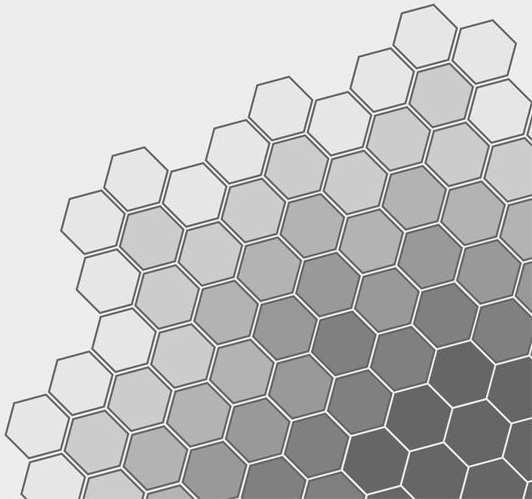
Under the hood, Spark (written in Scala) is an optimized engine that supports general execution graphs over an RDD.

Note, however - that Spark doesn’t deal with distributed storage, it still relies on HDFS, S3, HBase, etc.

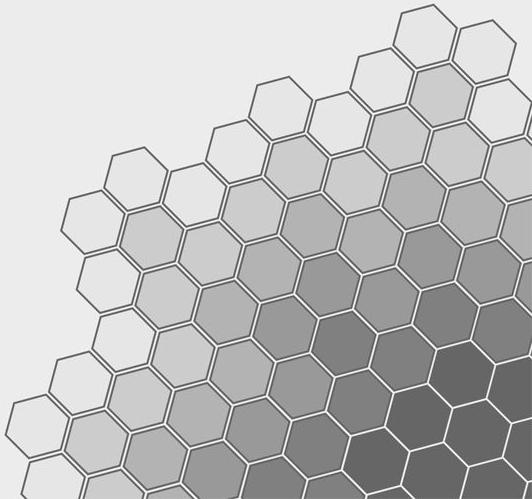


A Complete Spark Program

```
from operator import add

def tokenize(text):
    return text.split()

text = sc.textFile("tolstoy.txt")   # Create RDD

# Transform
wc   = text.flatMap(tokenize)
wc   = wc.map(lambda x: (x,1)).reduceByKey(add)

wc.saveAsTextFile("counts")         # Action
```



Resilient Distributed Datasets



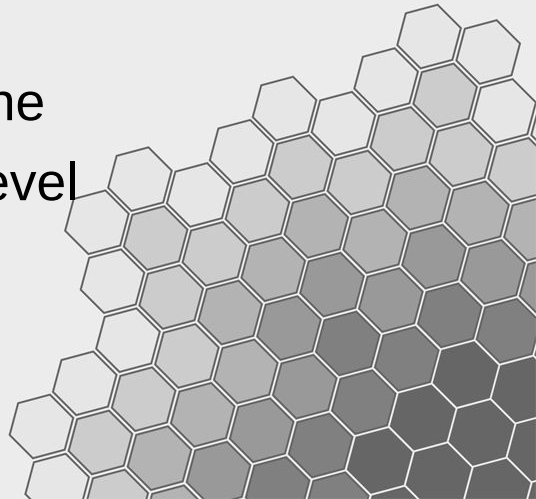
Science (and History)

Like MapReduce + GFS, Spark is based on two important papers authored by Matei Zaharia and the Berkeley AMPLab.

M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica, "*Spark: cluster computing with working sets*," in Proceedings of the 2nd USENIX conference on Hot topics in cloud computing, 2010, pp. 10–10.

M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauley, M. J. Franklin, S. Shenker, and I. Stoica, "*Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing*," in Proceedings of the 9th USENIX conference on Networked Systems Design and Implementation, 2012, pp. 2–2.

Matei is now the CTO and co-founder of Databricks, the corporate sponsor of Spark (which is an Apache top level open source project).

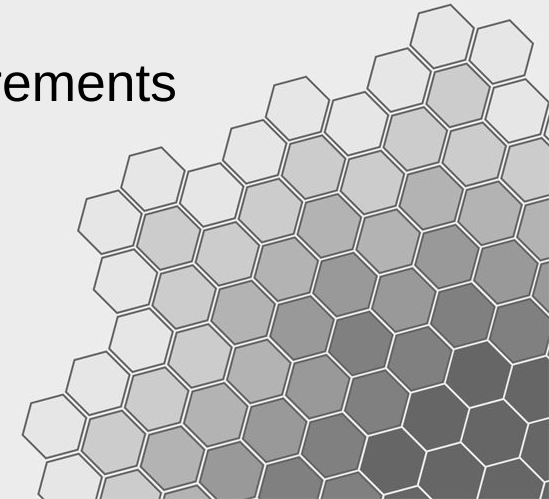


The Key Idea: RDDs

The principle behind Spark's framework is the idea of RDDs - an abstraction that represents a read-only collection of objects that are partitioned across a set of machines. RDDs can be:

1. Rebuilt from lineage (fault tolerance)
2. Accessed via MapReduce-like (functional) parallel operations
3. Cached in memory for immediate reuse
4. Written to distributed storage

These properties of RDDs all meet the Hadoop requirements for a distributed computation framework.



Working with RDDs

Most people focus on the in-memory caching of RDDs, which is great because it allows for:

- batch analyses (like MapReduce)
- interactive analyses (humans exploring Big Data)
- iterative analyses (no expensive Disk I/O)
- real time processing (just “append” to the collection)

However, RDDs also provide a more general interaction with functional constructs at a higher level of abstraction: *not just MapReduce*!

Spark Metrics

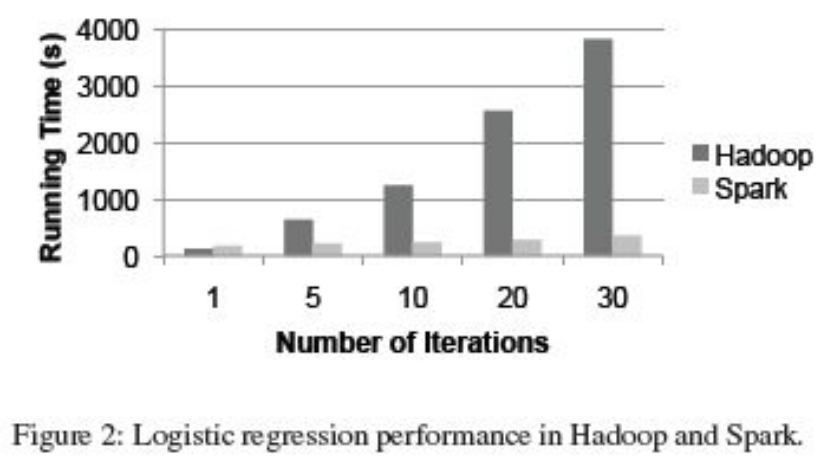
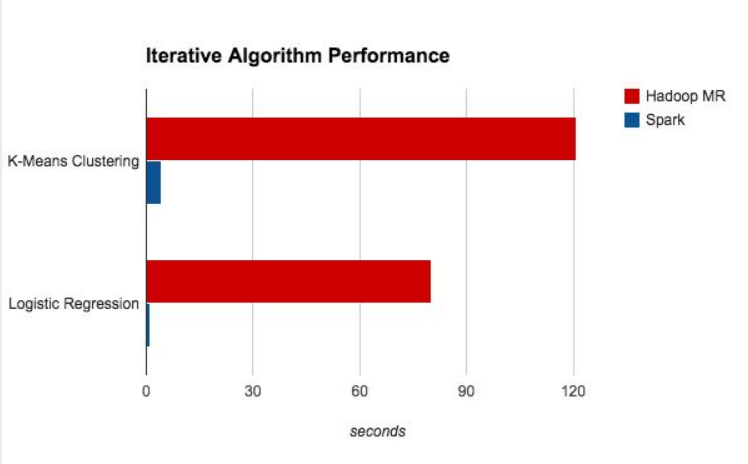
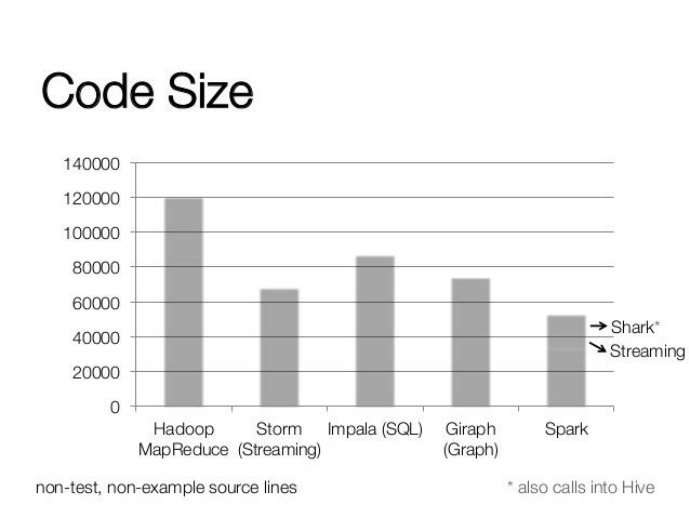


Figure 2: Logistic regression performance in Hadoop and Spark.





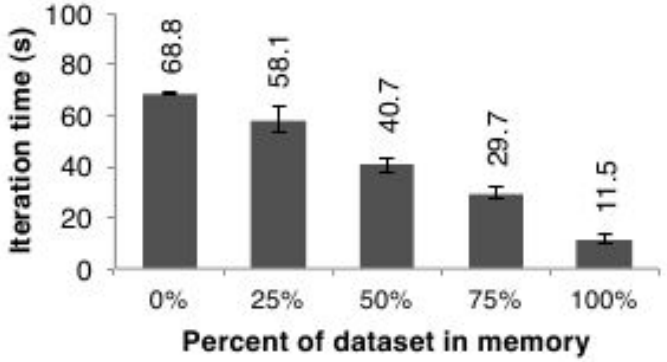


Figure 12: Performance of logistic regression using 100 GB data on 25 machines with varying amounts of data in memory.

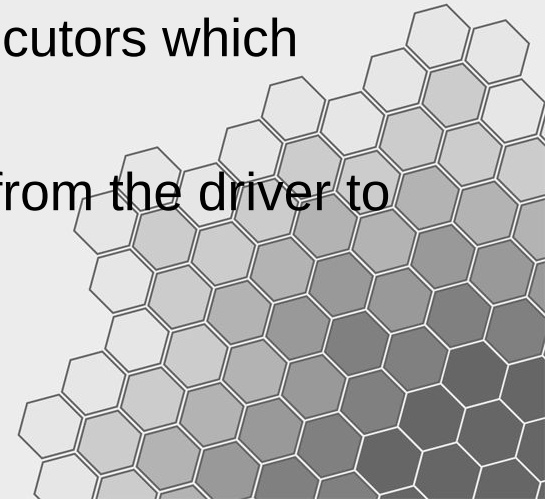
Programming Spark

Create a driver program (app.py) that does the following:

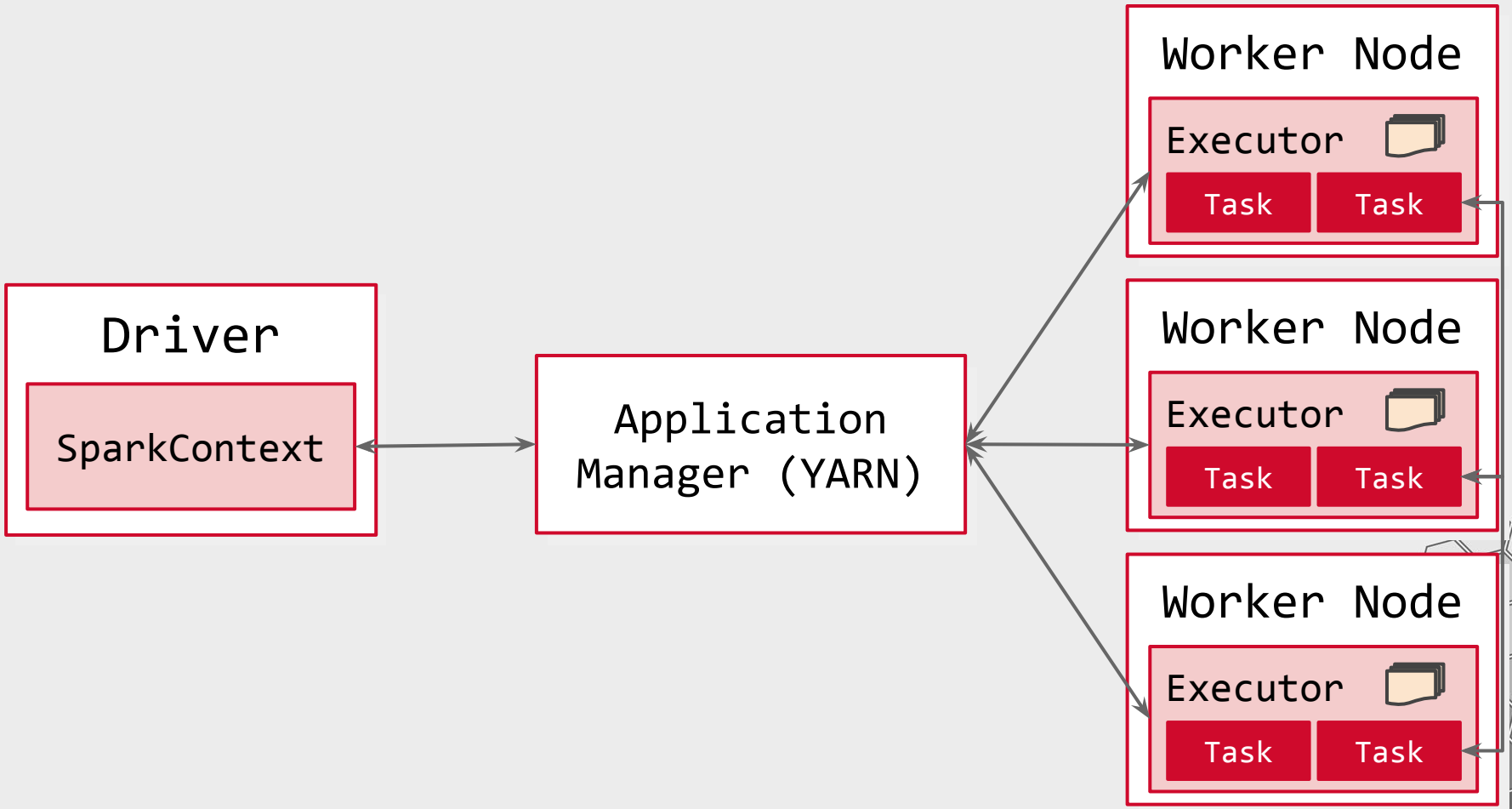
1. Define one or more RDDs either through accessing data stored on disk (HDFS, Cassandra, HBase, Local Disk), parallelizing some collection in memory, *transforming* an existing RDD or by *caching* or *saving*.
2. Invoke *operations* on the RDD by passing *closures* (functions) to each element of the RDD. Spark offers over 80 high level operators beyond Map and Reduce.
3. Use the resulting RDDs with *actions* e.g. count, collect, save, etc. Actions kick off the computing on the cluster, not before.

Spark Execution

- Spark applications are run as independent *sets* of processes
- Coordination is by a `SparkContext` in a *driver program*.
- The context connects to a cluster manager which allocates computational resources.
- Spark then acquires *executors* on individual nodes on the cluster.
- Executors manage individual worker computations as well as manage the storage and caching of data.
- Application code is sent from the driver to the executors which specifies the context and the *tasks* to be run.
- Communication can occur between workers and from the driver to the worker.

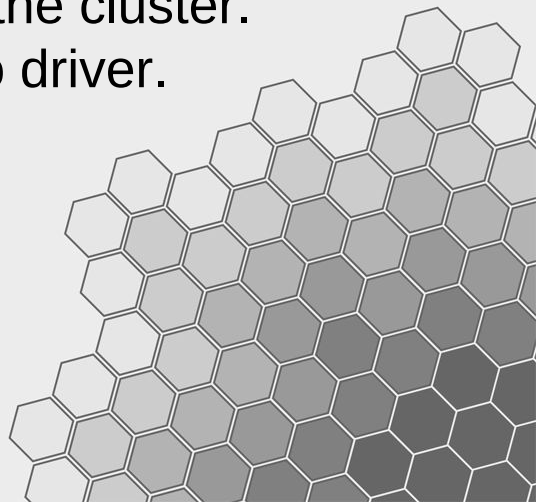


Spark Execution



Key Points regarding Execution

1. Each application gets its own executor for the duration.
2. Tasks run in multiple threads or processes.
3. Data can be shared between executors, but not between different Spark applications without external storage.
4. The Application Manager can be anything - Yarn on Hadoop, Mesos or Spark Standalone. Spark handles most of the resource scheduling.
5. Drivers are key participants in a Spark applications; therefore drivers should be on the same local network with the cluster.
6. Remote cluster access should use RPC access to driver.



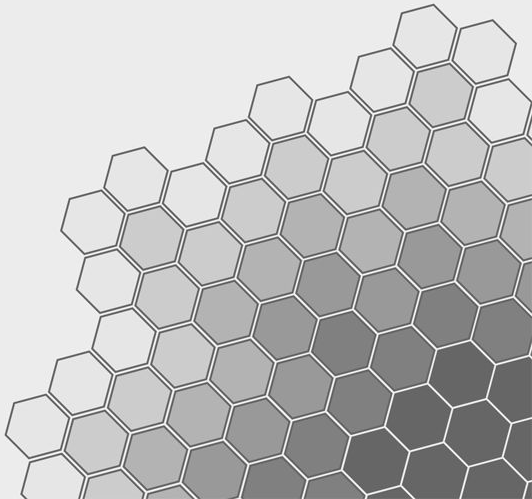
Executing Spark Jobs

Use the `spark-submit` command to send your application to the cluster for execution along with any other Python files and dependencies.

```
# Run on a YARN cluster
export HADOOP_CONF_DIR=XXX
/srv/spark/bin/spark-submit \
  --master yarn-cluster \
  --executor-memory 20G \
  --num-executors 50    \
  --py-files mydeps.egg
  app.py
```

This will cause Spark to allow the driver program to acquire a Context that utilizes the YARN ResourceManager.

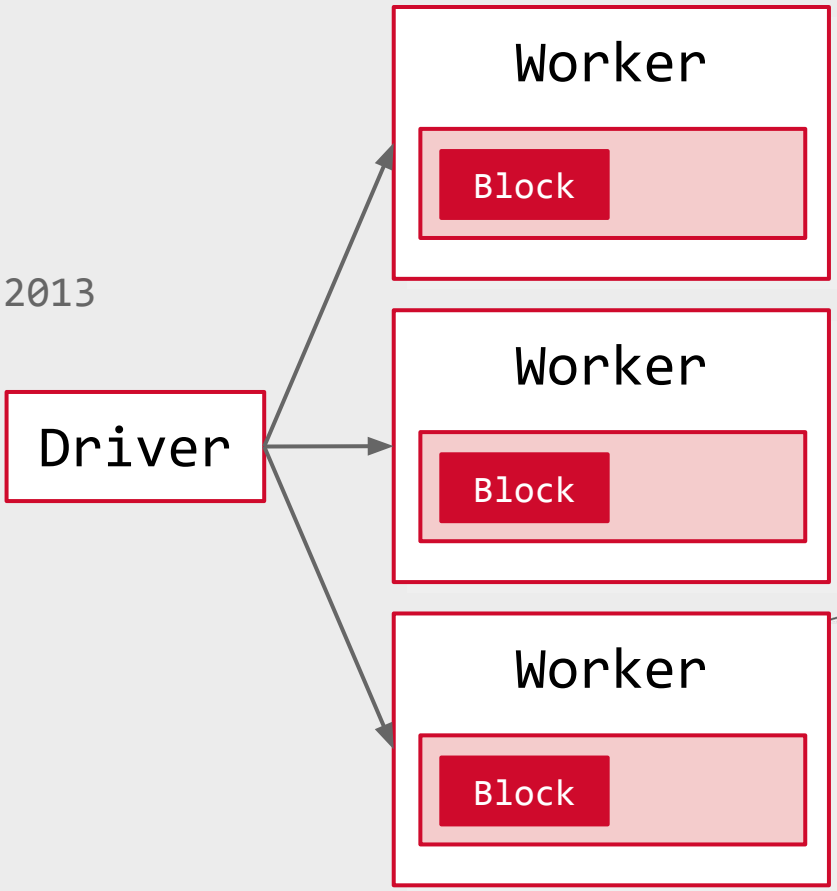
You can also specify many of these arguments in your driver program when constructing a `SparkContext`.



Example Data Flow

```
# Base RDD
orders = sc.textFile("hdfs://...")
orders = orders.map(split).map(parse)
orders = orders.filter(
    lambda order: order.date.year == 2013
)
orders.cache()
```

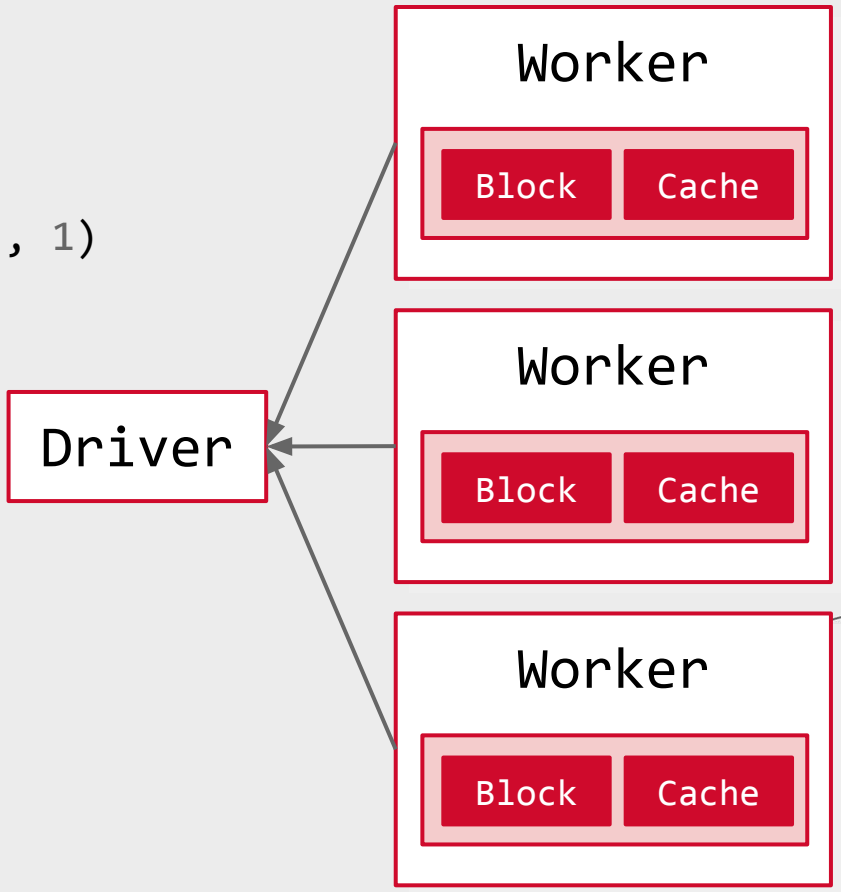
1. Read Block from HDFS
2. Process RDDs
3. Cache the data in memory



Example Data Flow

```
months = orders.map(
    lambda order: ((order.date.year,
                    order.date.month), 1)
    )

months = months.reduceByKey(add)
print months.take(5)
```

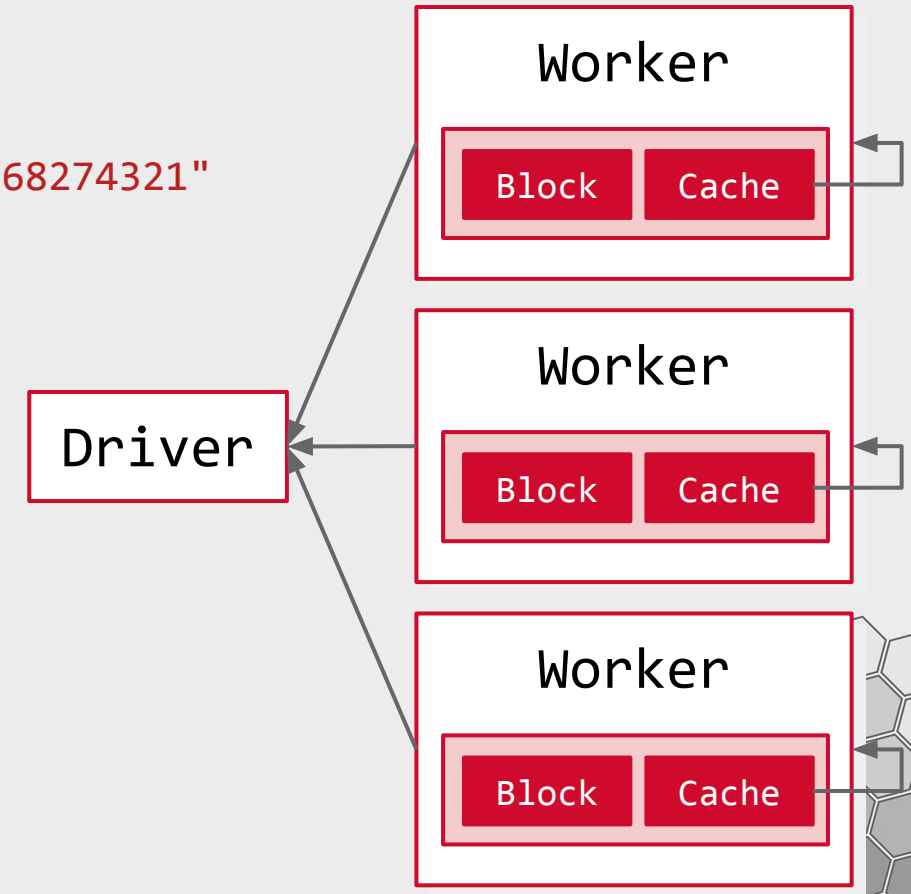
1. Process final RDD
2. On action send result back to driver
3. Driver outputs result (print)



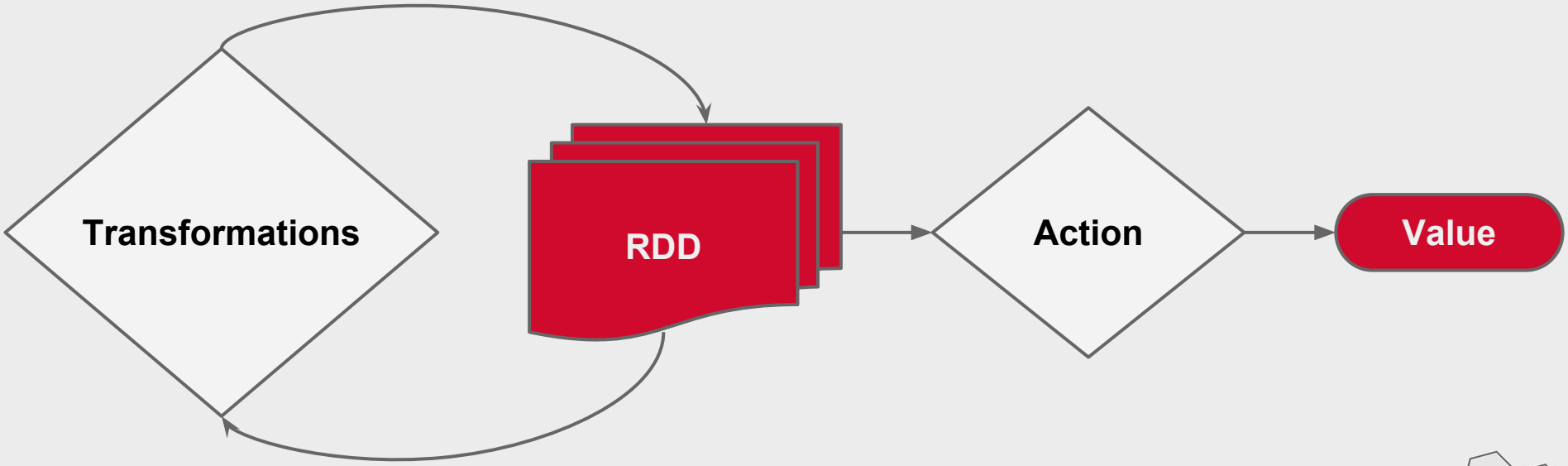
Example Data Flow

```
products = orders.filter(
    lambda order: order.upc == "098668274321"
)

print products.count()
```

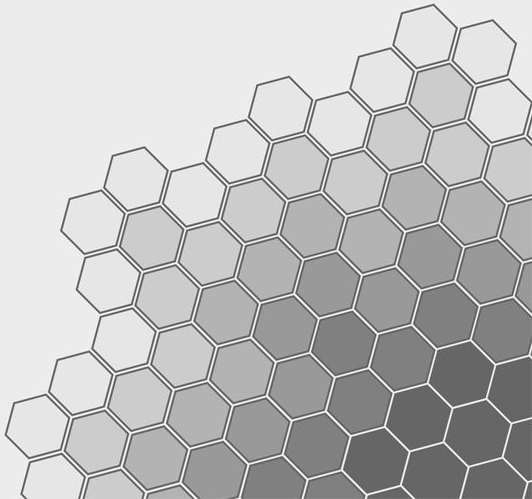
1. Process RDD from cache
2. Send data on action back to driver
3. Driver outputs result (print)



Spark Data Flow



Writing Spark Applications



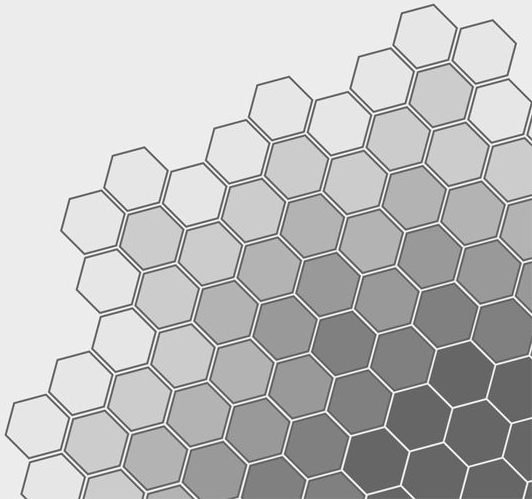
Creating a Spark Application

Writing a Spark application in Java, Scala, or Python is similar to using the interactive console - the API is the same. All you need to do first is to get access to the `SparkContext` that was loaded automatically for you by the interpreter.

```
from pyspark import SparkConf, SparkContext
conf = SparkConf().setAppName("MyApp")
sc = SparkContext(conf=conf)
```

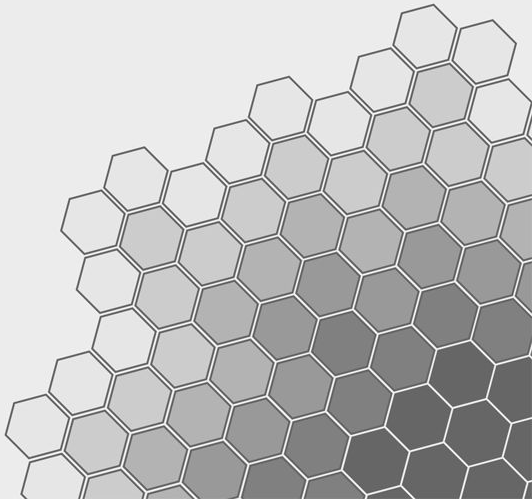
To shut down Spark:

```
sc.stop() or sys.exit(0)
```



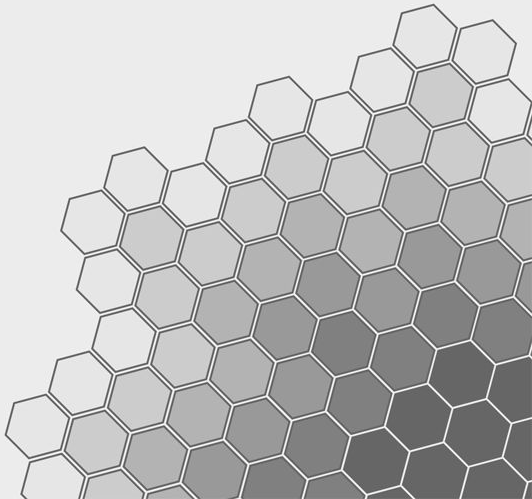
Structure of a Spark Application

- Dependencies (import)
  - third party dependencies can be shipped with app
- Constants and Structures
  - especially `namedtuples` and other constants
- Closures
  - functions that operate on the RDD
- A main method
  - Creates a SparkContext
  - Creates one or more RDDs
  - Applies *transformations* to RDDs
  - Applies *actions* to kick off computation



```
## Spark Application - execute with spark-submit

## Imports
from pyspark import SparkConf, SparkContext

## Module Constants
APP_NAME = "My Spark Application"

## Closure Functions

## Main functionality
def main(sc):
    pass

if __name__ == "__main__":
    # Configure Spark
    conf = SparkConf().setAppName(APP_NAME)
    sc   = SparkContext(conf=conf)

    # Execute Main functionality
    main(sc)
```

A Spark Application Skeleton



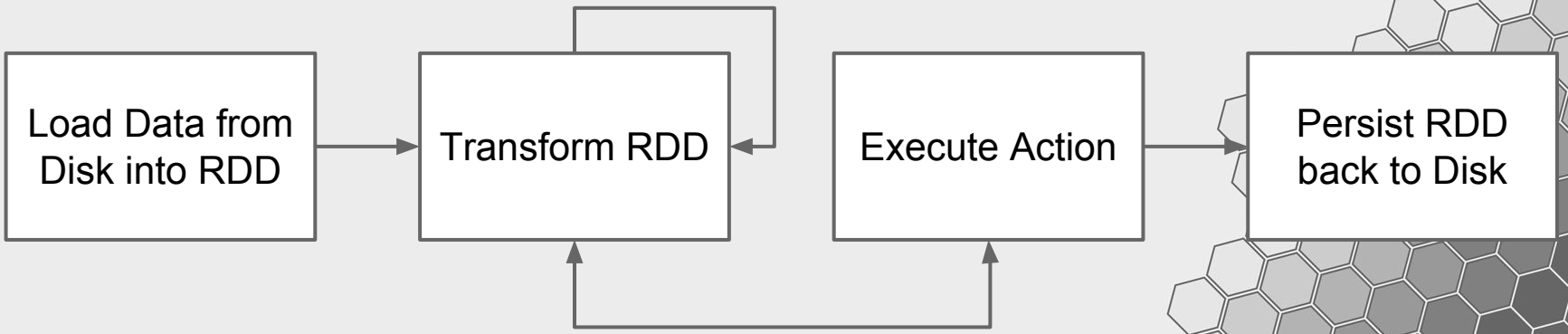
Programming Model

Two types of operations on an RDD:

- *transformations*
- *actions*

Transformations are *lazily* evaluated - they aren't executed when you issue the command.

RDDs are recomputed when an action is executed.



Initializing an RDD

Two types of RDDs:

- *parallelized collections* - take an existing in memory collection (a list or tuple) and run functions upon it in parallel

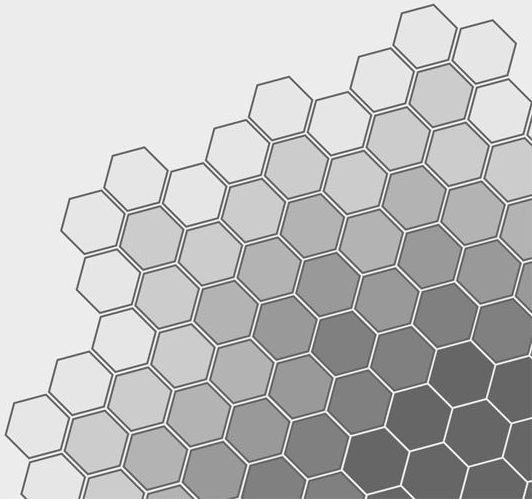
- *Hadoop datasets* - run functions in parallel on any storage system supported by Hadoop (HDFS, S3, HBase, local file system, etc).

Input can be text, `SequenceFiles`, and any other Hadoop `InputFormat` that exists.

Initializing an RDD

```
# Parallelize a list of numbers
distributed_data = sc.parallelize(xrange(100000))

# Load data from a single text file on disk
lines = sc.textFile('tolstoy.txt')

# Load data from all csv files in a directory using glob
files  = sc.wholeTextFiles('dataset/*.csv')

# Load data from S3
data = sc.textFile('s3://databucket/')
```

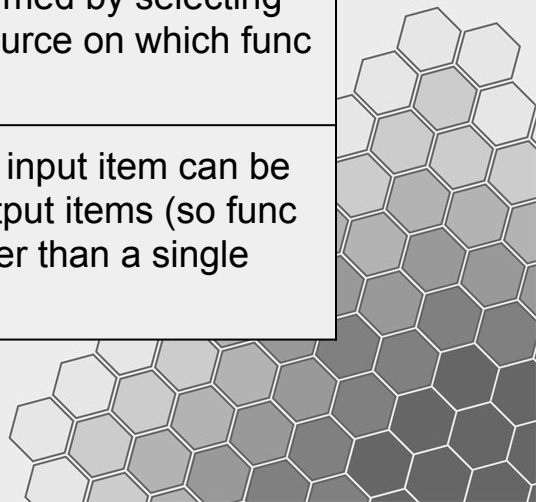
For HBase example, see: [hbase_inputformat.py]



Transformations

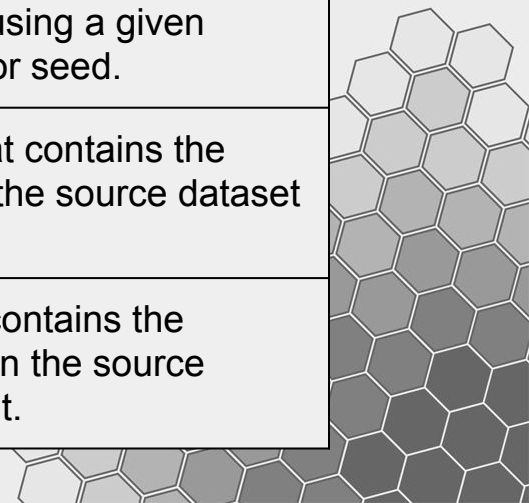
- create a new dataset from an existing one
- evaluated lazily, won’t be executed until action

Transformation	Description
`map(func)`	Return a new distributed dataset formed by passing each element of the source through a function func.
`filter(func)`	Return a new dataset formed by selecting those elements of the source on which func returns true.
`flatMap(func)`	Similar to map, but each input item can be mapped to 0 or more output items (so func should return a Seq rather than a single item).



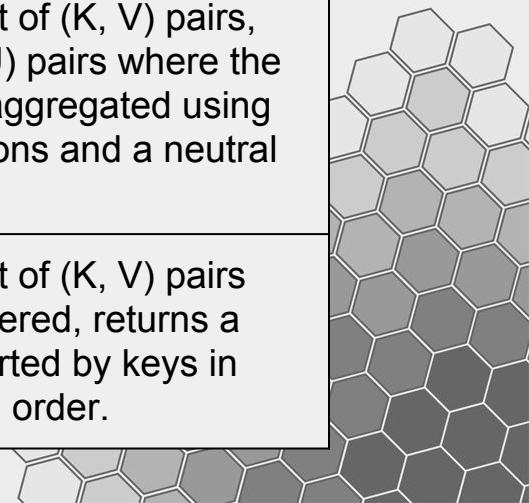
Transformations

Transformation	Description
`mapPartitions(func)`	Similar to map, but runs separately on each partition (block) of the RDD, so func must be of type Iterator<T> => Iterator<U> when running on an RDD of type T.
`mapPartitionsWithIndex(func)`	Similar to mapPartitions, but also provides func with an integer value representing the index of the partition, so func must be of type (Int, Iterator<T>) => Iterator<U> when running on an RDD of type T.
`sample(withReplacement, fraction, seed)`	Sample a fraction fraction of the data, with or without replacement, using a given random number generator seed.
`union(otherDataset)`	Return a new dataset that contains the union of the elements in the source dataset and the argument.
`intersection(otherDataset)`	Return a new RDD that contains the intersection of elements in the source dataset and the argument.



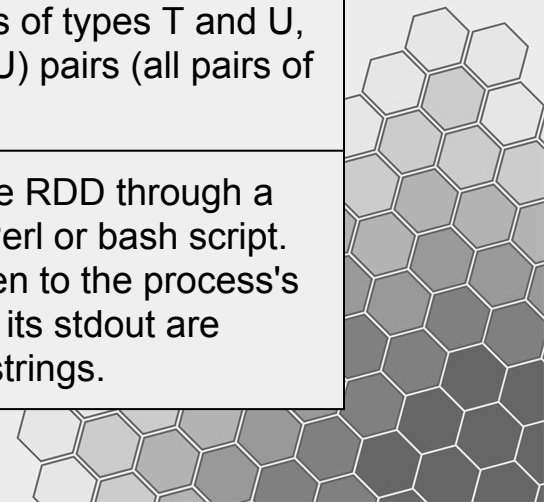
Transformations

Transformation	Description
`distinct([numTasks]))`	Return a new dataset that contains the distinct elements of the source dataset.
`groupByKey([numTasks])`	When called on a dataset of (K, V) pairs, returns a dataset of (K, Iterable<V>) pairs.
`reduceByKey(func, [numTasks])`	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.
`aggregateByKey(zeroValue)(seqOp, combOp, [numTasks])`	When called on a dataset of (K, V) pairs, returns a dataset of (K, U) pairs where the values for each key are aggregated using the given combine functions and a neutral "zero" value.
`sortByKey([ascending], [numTasks])`	When called on a dataset of (K, V) pairs where K implements Ordered, returns a dataset of (K, V) pairs sorted by keys in ascending or descending order.



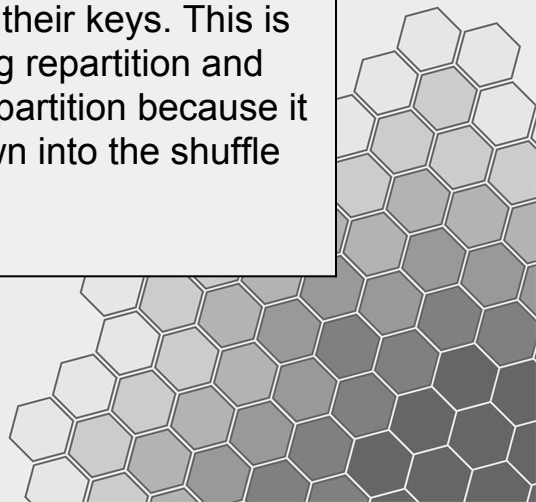
Transformations

Transformation	Description
`join(otherDataset, [numTasks])`	When called on datasets of type (K, V) and (K, W), returns a dataset of (K, (V, W)) pairs with all pairs of elements for each key. Outer joins are supported through leftOuterJoin,rightOuterJoin, and fullOuterJoin.
`cogroup(otherDataset, [numTasks])`	When called on datasets of type (K, V) and (K, W), returns a dataset of (K, Iterable<V>, Iterable<W>) tuples. This operation is also called groupWith.
`cartesian(otherDataset)`	When called on datasets of types T and U, returns a dataset of (T, U) pairs (all pairs of elements).
`pipe(command, [envVars])`	Pipe each partition of the RDD through a shell command, e.g. a Perl or bash script. RDD elements are written to the process's stdin and lines output to its stdout are returned as an RDD of strings.



Transformations

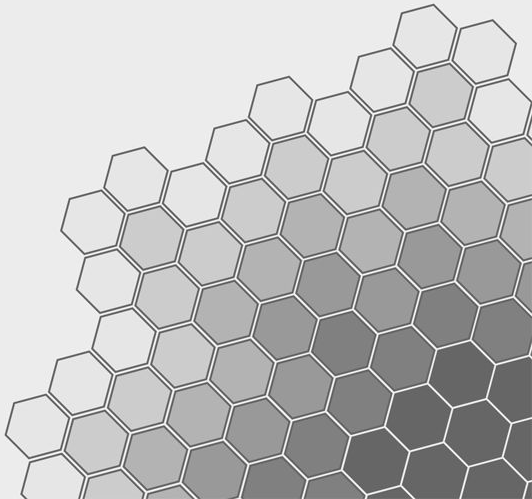
Transformation	Description
`coalesce(numPartitions)`	Decrease the number of partitions in the RDD to numPartitions. Useful for running operations more efficiently after filtering down a large dataset.
`repartition(numPartitions)`	Reshuffle the data in the RDD randomly to create either more or fewer partitions and balance it across them. This always shuffles all data over the network.
`repartitionAndSortWithinPartitions(partitioner)`	Repartition the RDD according to the given partitioner and, within each resulting partition, sort records by their keys. This is more efficient than calling repartition and then sorting within each partition because it can push the sorting down into the shuffle machinery.



Exercise: What is the difference between Map and FlatMap?

```
lines = sc.textFile('fixtures/poem.txt')
lines.map(lambda x: x.split(' ')).collect()
lines.flatMap(lambda x: x.split(' ')).collect()
```

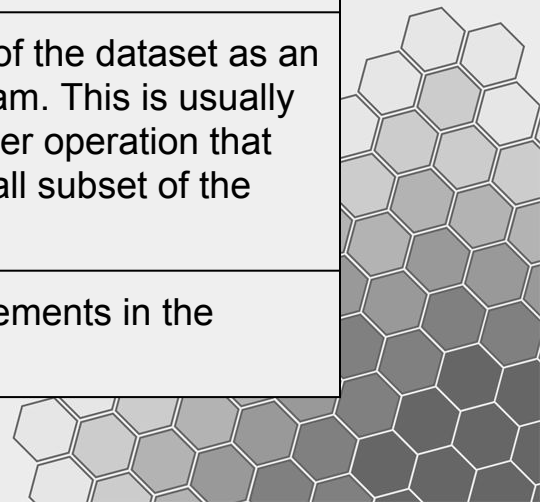
Note the use of closures with the **lambda** keyword



Actions

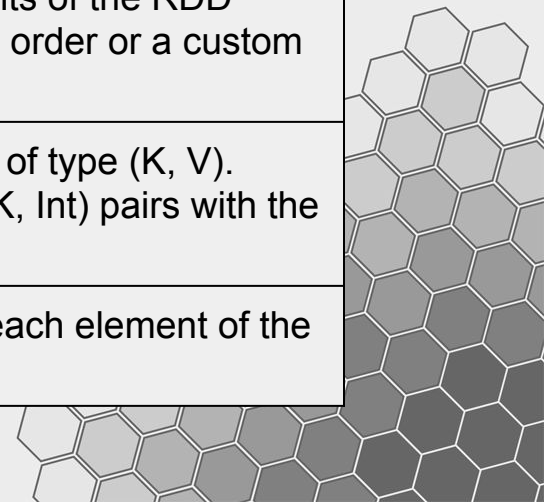
- kick off evaluations and begin computation
- specify the result of an operation or aggregation

Action	Description
`reduce(func)`	Aggregate the elements of the dataset using a function func (which takes two arguments and returns one). The function should be commutative and associative so that it can be computed correctly in parallel.
`collect()`	Return all the elements of the dataset as an array at the driver program. This is usually useful after a filter or other operation that returns a sufficiently small subset of the data.
`count()`	Return the number of elements in the dataset.



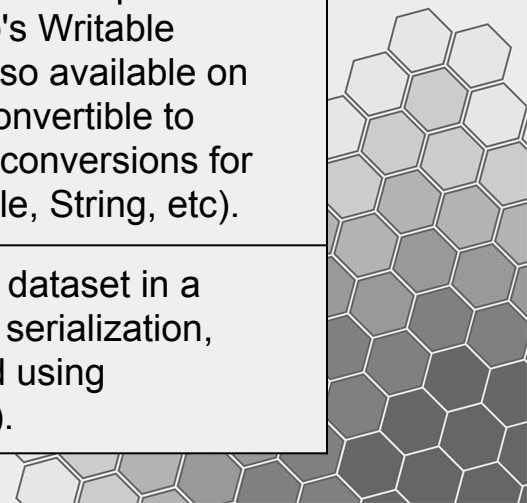
Actions

Action	Description
`first()`	Return the first element of the dataset (similar to take(1)).
`take(n)`	Return an array with the first n elements of the dataset. Note that this is currently not executed in parallel. Instead, the driver program computes all the elements.
`takeSample(withReplacement,num, [seed])`	Return an array with a random sample of num elements of the dataset, with or without replacement.
`takeOrdered(n, [ordering])`	Return the first n elements of the RDD using either their natural order or a custom comparator.
`countByKey()`	Only available on RDDs of type (K, V). Returns a hashmap of (K, Int) pairs with the count of each key.
`foreach(func)`	Run a function func on each element of the dataset.

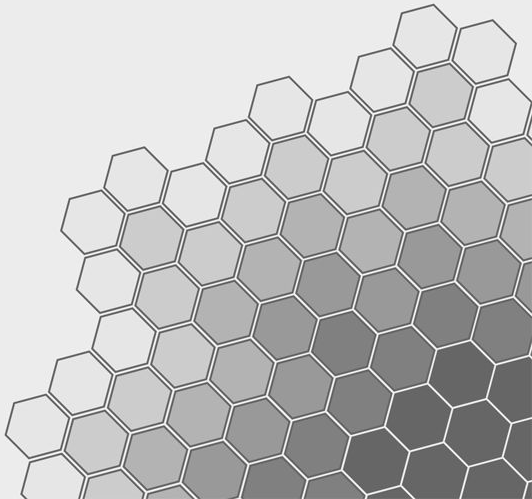


Actions

Action	Description
`saveAsTextFile(path)`	Write the elements of the dataset as a text file (or set of text files) in a given directory in the local filesystem, HDFS or any other Hadoop-supported file system. Spark will call toString on each element to convert it to a line of text in the file.
`saveAsSequenceFile(path)` `(Java and Scala)`	Write the elements of the dataset as a Hadoop SequenceFile in a given path in the local filesystem, HDFS or any other Hadoop-supported file system. This is available on RDDs of key-value pairs that either implement Hadoop's Writable interface. In Scala, it is also available on types that are implicitly convertible to Writable (Spark includes conversions for basic types like Int, Double, String, etc).
`saveAsObjectFile(path)` `(Java and Scala)`	Write the elements of the dataset in a simple format using Java serialization, which can then be loaded using SparkContext.objectFile().



Was that comprehensive list really necessary? Remember in MapReduce you only get two operators - map and reduce; so maybe I’m just excited at the 80+ operations in Spark!



Persistence

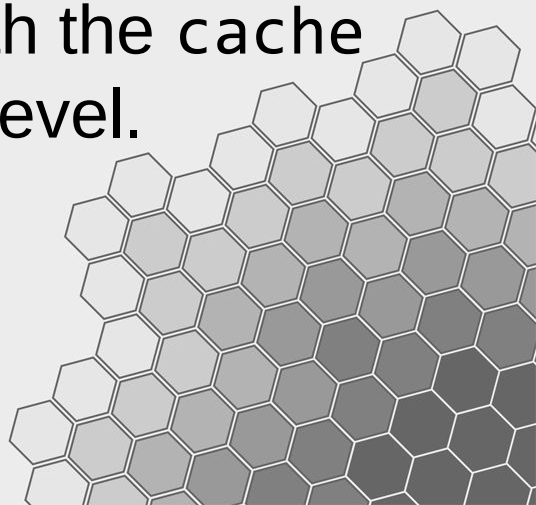
Spark will *persist* or *cache* RDD slices in memory on each node during operations.

Fault tolerant - in case of failure, Spark can rebuild the RDD from the lineage, automatically recreating the slice.

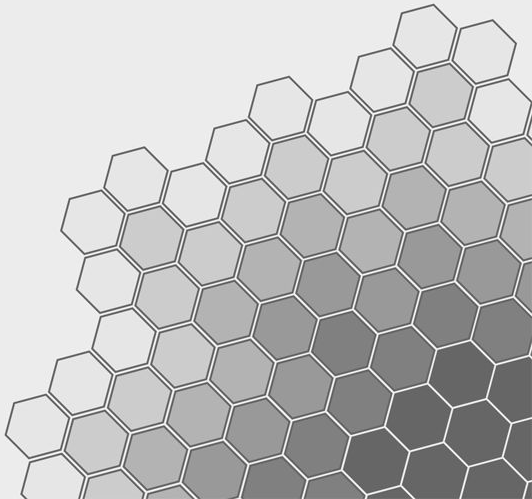
Super fast - will allow multiple operations on the same data set.

You can mark an RDD to be persisted with the `cache` method on an RDD along with a storage level.

Python objects are always pickles.



Beyond RDDs

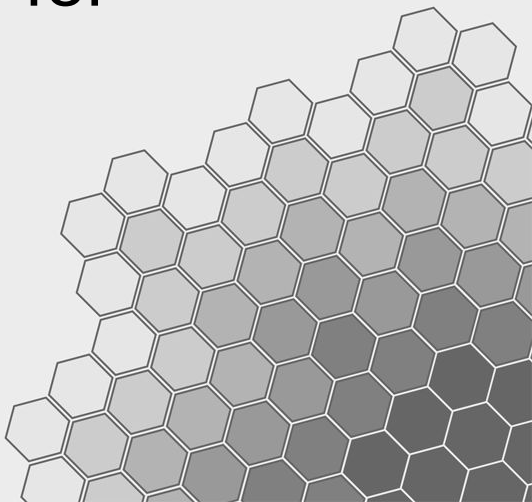


Variables and Memory

Any variable or data within a closure or data structure will be distributed to the worker nodes along with the closure.

Usually these variables are just constants but they cannot be shared across workers. Instead, the following restricted structures are used for inter-process communication:

- *broadcast variables*
- *accumulators*

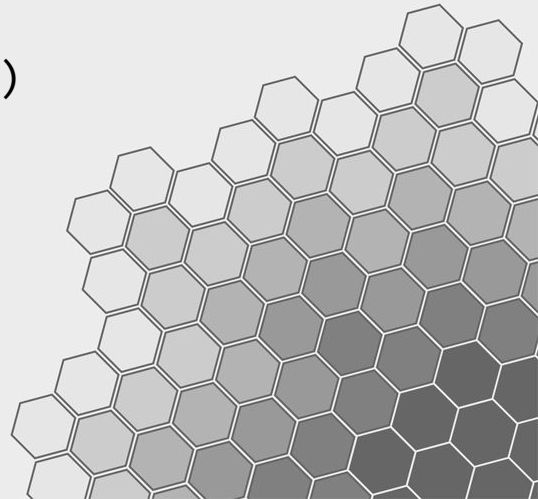


Broadcast Variables

Distribute some large piece of read-only data to all workers only once (e.g. a lookup table or stopwords).

This prevents multiple distribution per task, and efficiently gives nodes a copy of larger data using efficient broadcast algorithms.

```
import nltk

# Initialize the stopwords broadcast variable
stopwords = set(nltk.corpus.stopwords.words('english'))
stopwords = sc.broadcast(stopwords)

# Access the broadcast variable
if word in stopwords.value:
    pass
```

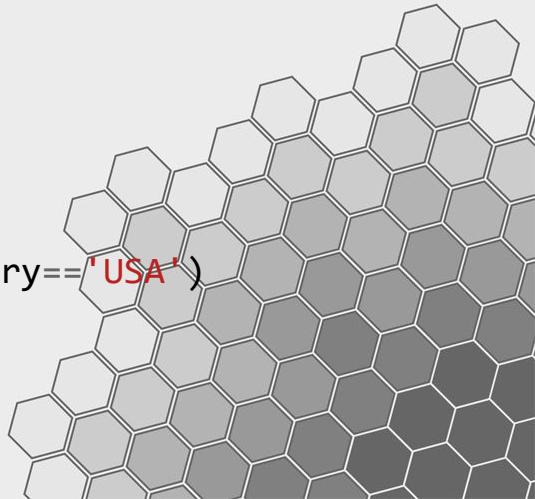


Accumulators

Variables that workers can “add” to using associative operations. These are read-only for the driver, but can be used as counters or summations

- Spark natively supports accumulators of numeric value types and standard mutable collections.
- Accumulators write-only to the workers.

```
gold   = sc.accumulator(0)

def count_medals(events):
    global gold,silver, bronze
    for event in events:
        if event.medal == 'GOLD':
            gold.add(1)

results = sc.textFile('olympics.csv').filter(lambda x: x.country=='USA')
results.foreach(count_medals)

print gold.value
```

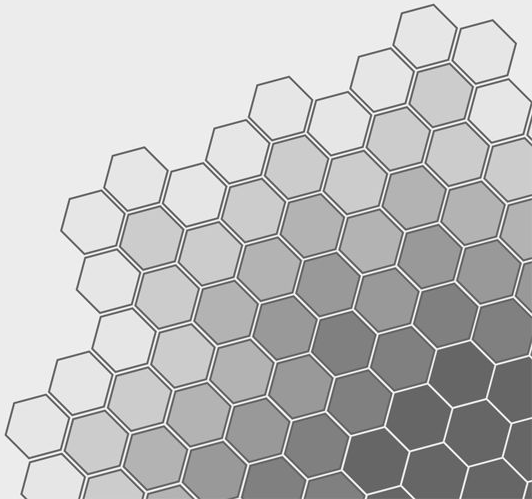


Key Value Pairs

Most Spark operations work on RDDs containing any type of objects, a few special operations are only available on RDDs of key-value pairs (“shuffle” operations, such as grouping or aggregating the elements by a key)

In Python, these operations work on RDDs containing built-in Python tuples such as (1, 2). Simply create such tuples and then call your desired operation.

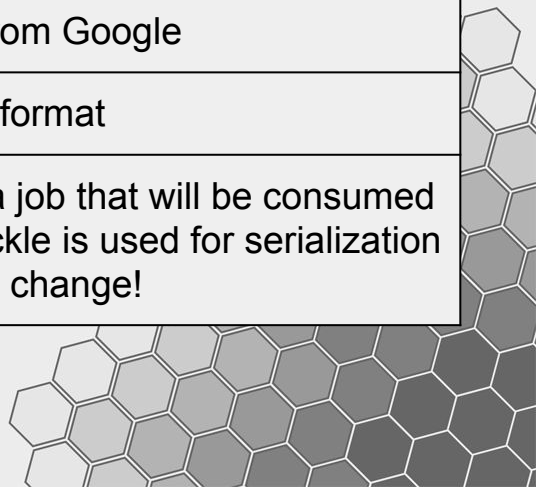
```
lines = sc.textFile("data.txt")
pairs = lines.map(lambda s: (s, 1))
counts = pairs.reduceByKey(lambda a, b: a + b)
```



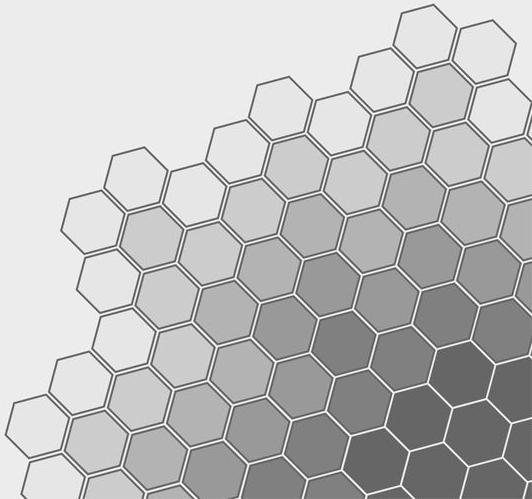
Input and Output Formats

Selecting the best data file format:

Format	Splitable	Structured	Description
text files	yes	no	records are split by line
JSON	yes	semi	log-json one record per line
CSV	yes	yes	very common format, used with databases
Sequence files	yes	yes	common Hadoop format for key-value data
Protocol buffers	yes	yes	fast, space-efficient format from Google
Avro	yes	yes	compact binary serialization format
Object Files	yes	yes	Useful for saving data from a job that will be consumed by shared code; however pickle is used for serialization so your objects in code can’t change!



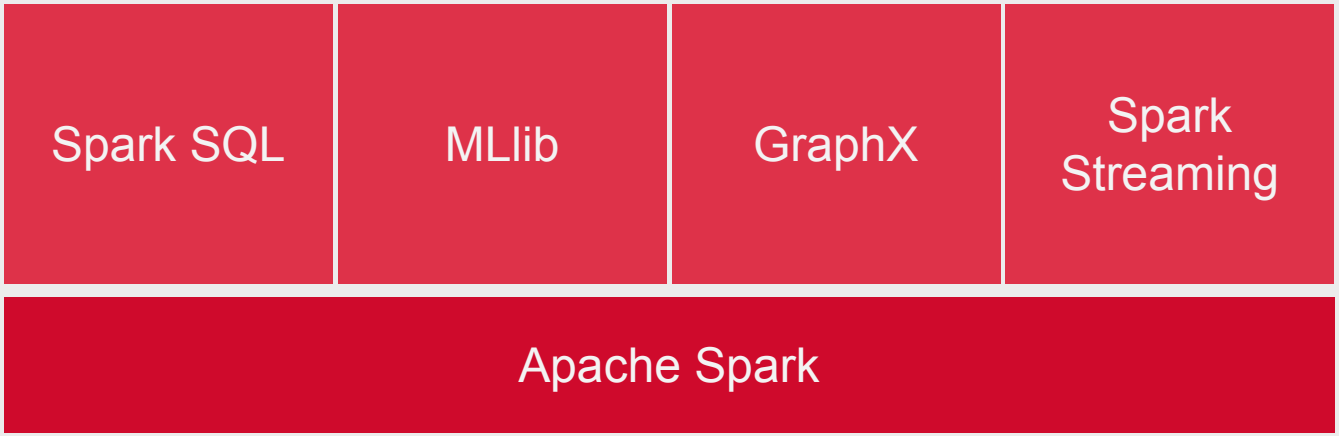
Spark Libraries

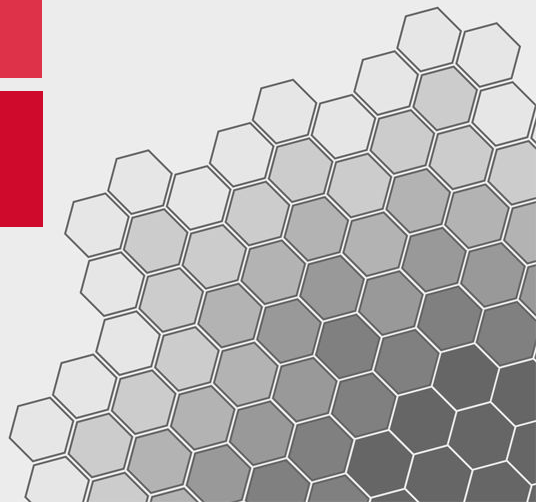


Workflows and Tools

The RDD data model and cached memory computing allow Spark to quickly and easily solve similar workflows and use cases that are part of Hadoop.

Spark has a series of high level tools at it's disposal that are added as component libraries, not integrated into the general computing framework:



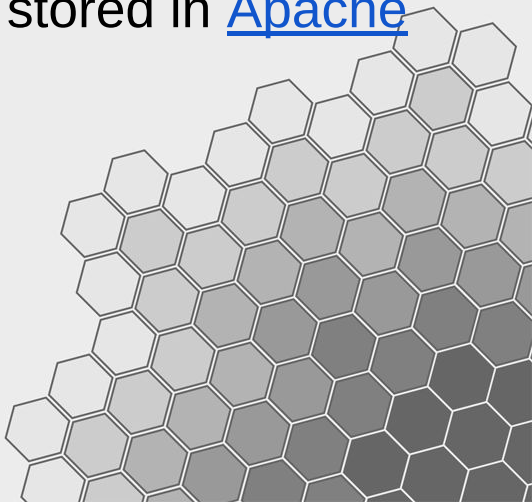


SparkSQL

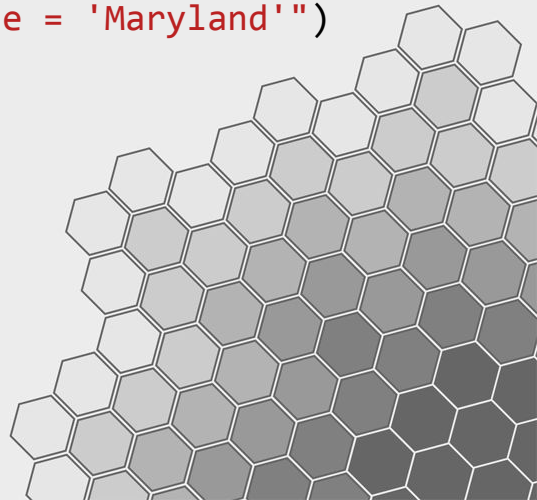
Spark SQL allows relational queries expressed in SQL or HiveQL to be executed using Spark.

- SchemaRDDs are composed of Row objects, along with a schema that describes the data types of each column in the row.
- A SchemaRDD is similar to a table in a traditional relational database and is operated on in a similar fashion.
- SchemaRDDs are created from an existing RDD, a Parquet file, a JSON dataset, or by running HiveQL against data stored in Apache Hive.

Spark SQL is currently an alpha component.



```
import csv

from StringIO import StringIO
from pyspark import SparkConf, SparkContext
from pyspark.sql import SQLContext, Row

def split(line):
    return csv.reader(StringIO(line)).next()

def main(sc, sqlc):
    rows = sc.textFile("fixtures/shopping/customers.csv").map(split)
    customers = rows.map(lambda c: Row(id=int(c[0]), name=c[1], state=c[6]))

    # Infer the schema and register the SchemaRDD
    schema = sqlc.inferSchema(customers).registerTempTable("customers")

    maryland = sqlc.sql("SELECT name FROM customers WHERE state = 'Maryland'")
    print maryland.count()

if __name__ == '__main__':
    conf = SparkConf().setAppName("Query Customers")
    sc   = SparkContext(conf=conf)
    sqlc = SQLContext(sc)
    main(sc, sqlc)
```



Spark Streaming

Spark Streaming is an extension of the core Spark API that enables scalable, high-throughput, fault-tolerant stream processing of live data.

- Data can be ingested from many sources like Kafka, Flume, Twitter, ZeroMQ, Kinesis or TCP sockets
- Can be processed using complex algorithms expressed with high-level functions like map, reduce, join and window.
- Finally, processed data can be pushed out to file systems, databases, and live dashboards; or apply Spark's machine learning and graph processing algorithms on data streams.

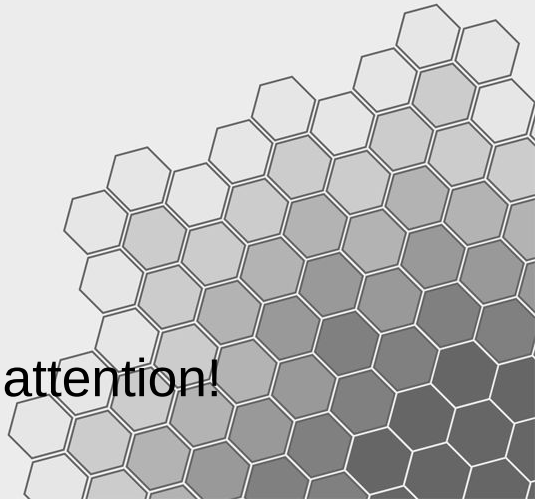
Spark MLLib

Spark’s scalable machine learning library consisting of common learning algorithms and utilities, including classification, regression, clustering, collaborative filtering, dimensionality reduction, as well as underlying optimization primitives.

Highlights include:

- summary statistics and correlation
- hypothesis testing, random data generation
- Linear models of regression (SVMs, logistic and linear regression)
- Naive Bayes and Decision Tree classifiers
- Collaborative Filtering with ALS
- K-Means clustering
- SVD (singular value decomposition) and PCA
- Stochastic gradient descent

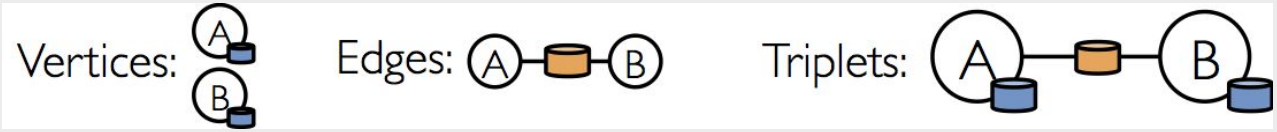
Not fully featured, still experimental - but gets a ton of attention!



Spark GraphX

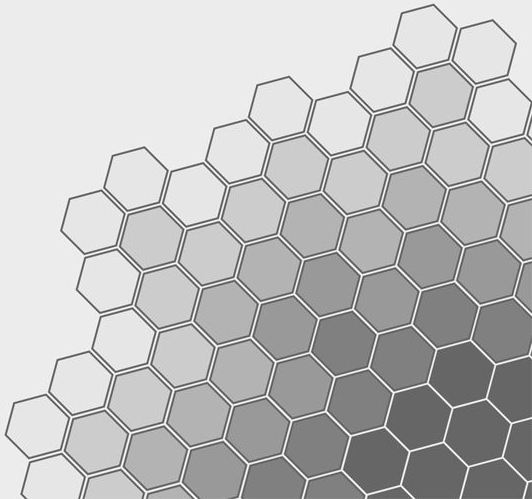
GraphX is the (alpha) Spark API for graphs and graph-parallel computation.

- GraphX extends the Spark RDD by introducing the Resilient Distributed Property Graph: a directed multigraph with properties attached to each vertex and edge.
- GraphX exposes a set of fundamental operators (e.g., `subgraph`, `joinVertices`, and `aggregateMessages`) as well as an optimized variant of the Pregel API.
- GraphX includes a growing collection of graph algorithms and builders to simplify graph analytics tasks.



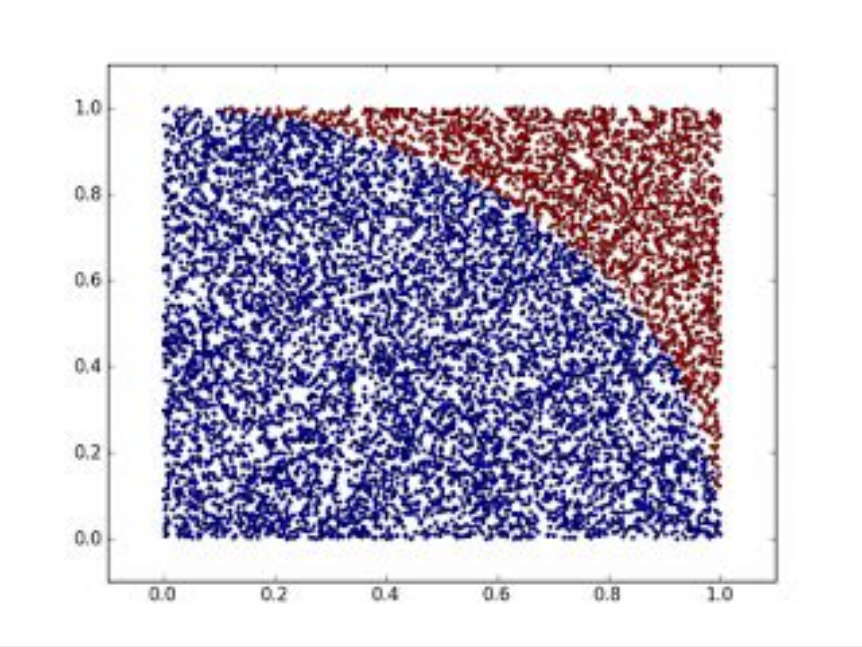
No Python API ...

Spark Example

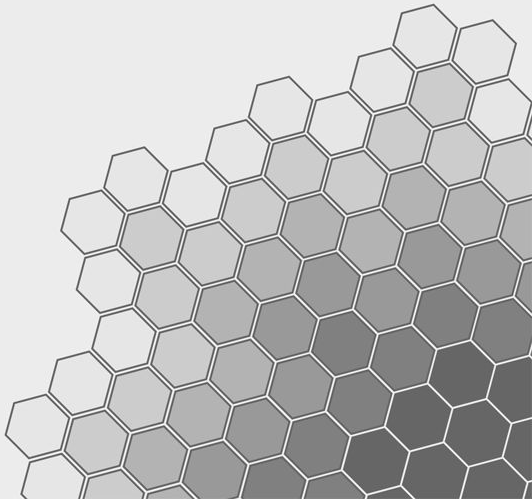


Estimate Pi

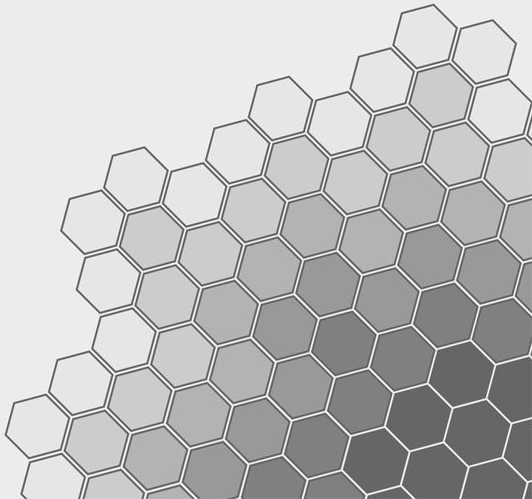
Databricks has a great example where they use the Monte Carlo method to estimate Pi in a distributed fashion.



```
import sys
import random

from operator import add
from pyspark import SparkConf, SparkContext

def estimate(idx):
    x = random.random() * 2 - 1
    y = random.random() * 2 - 1
    return 1 if (x*x + y*y < 1) else 0

def main(sc, *args):
    slices = int(args[0]) if len(args) > 0 else 2
    N = 100000 * slices

    count  = sc.parallelize(xrange(N), slices).map(estimate)
    count  = count.reduce(add)

    print "Pi is roughly %0.5f" % (4.0 * count / N)
    sc.stop()

if __name__ == '__main__':
    conf = SparkConf().setAppName("Estimate Pi")
    sc   = SparkContext(conf=conf)
    main(sc, *sys.argv[1:])
```



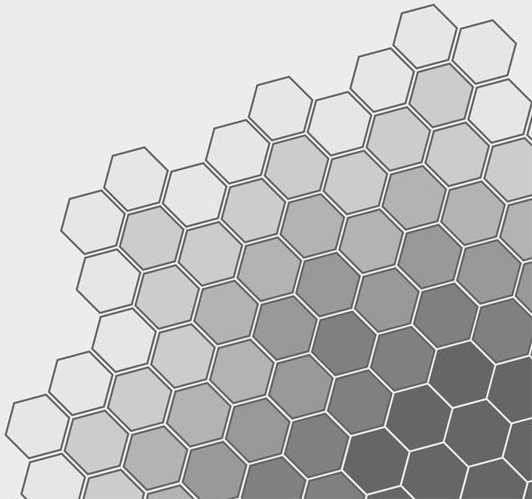
Toward Last Mile Computing



The Last Mile

In telecommunications the “last mile” refers to the final leg of network communication systems that actually goes from the street to the end user.

Typically it is is this last mile that is the bottleneck for both latency and bandwidth and problems that occur in this leg of the network are the most difficult to diagnose and work with since they are more specific than general.



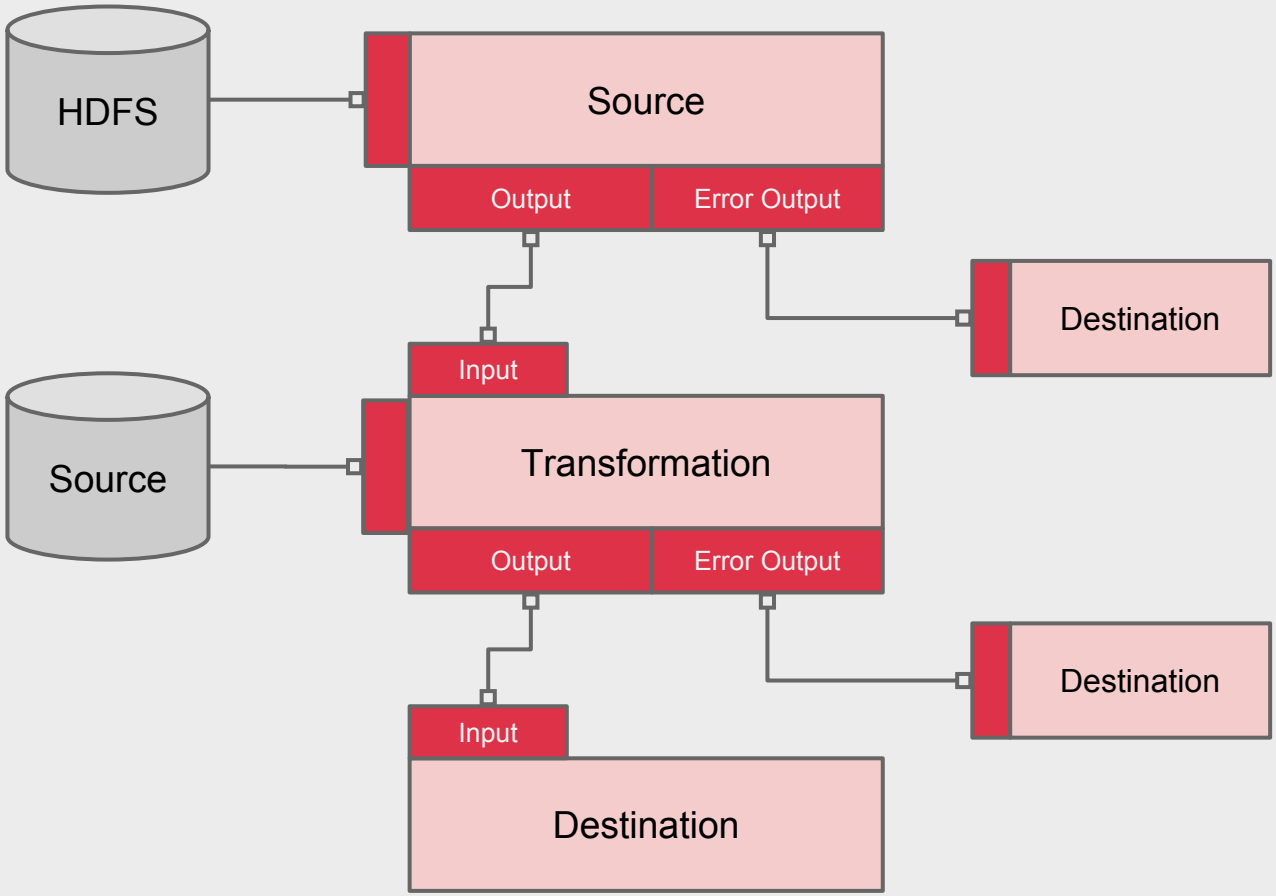
Last Mile Computing

Therefore, the construction of **data flows** in Hadoop should be the decomposition of large data sets into smaller and smaller ones, until you get a data space that can fit into the memory of a single machine.

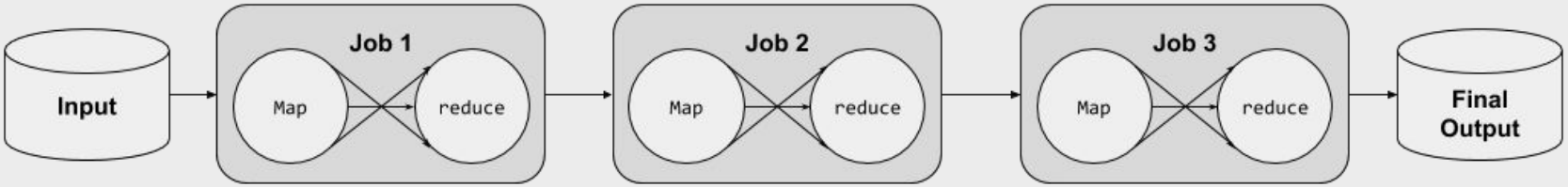
From 10-500TB → 16-64 GB for high performance computing applications.

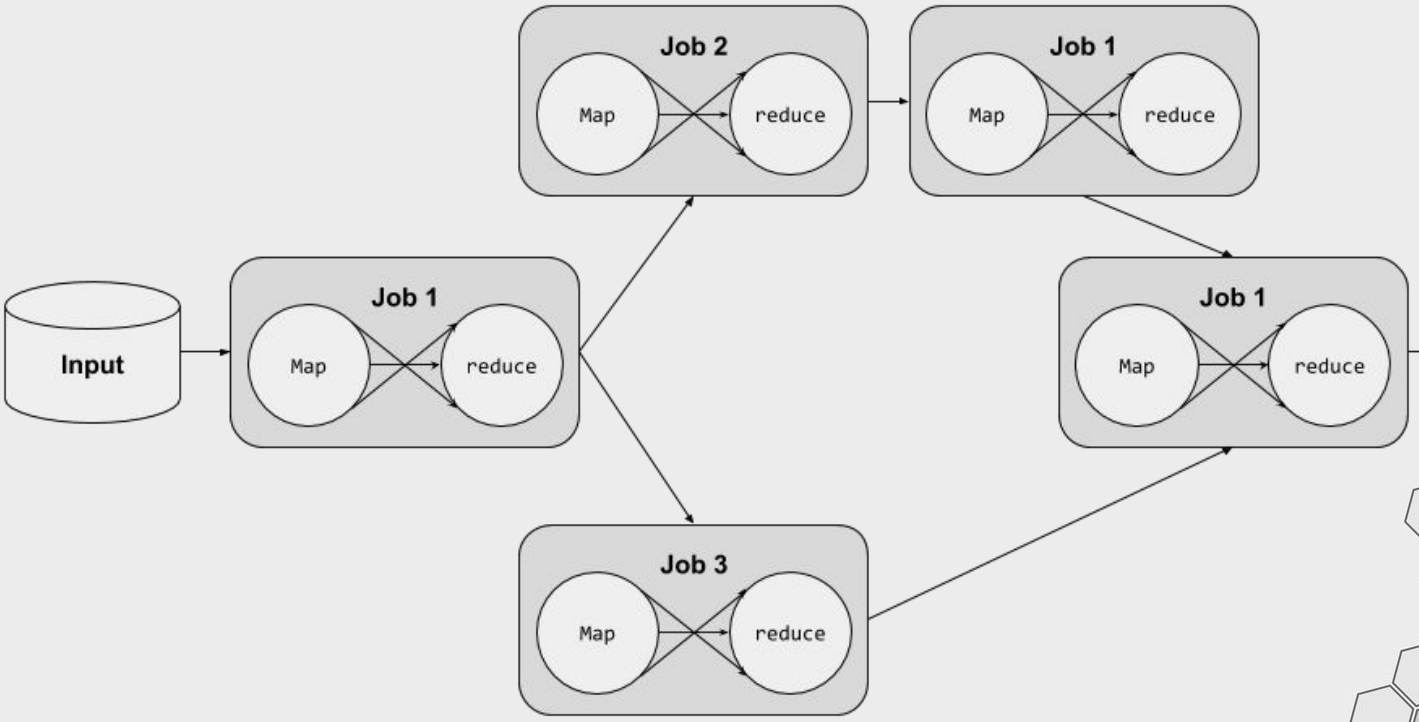
Most Hadoop jobs are not single MapReduce jobs, but are rather chained together Maps and Reduces whose execution can be represented as a **Directed Acyclic Graph**. A programming language that implements this is dataflow programming.

Data Flow

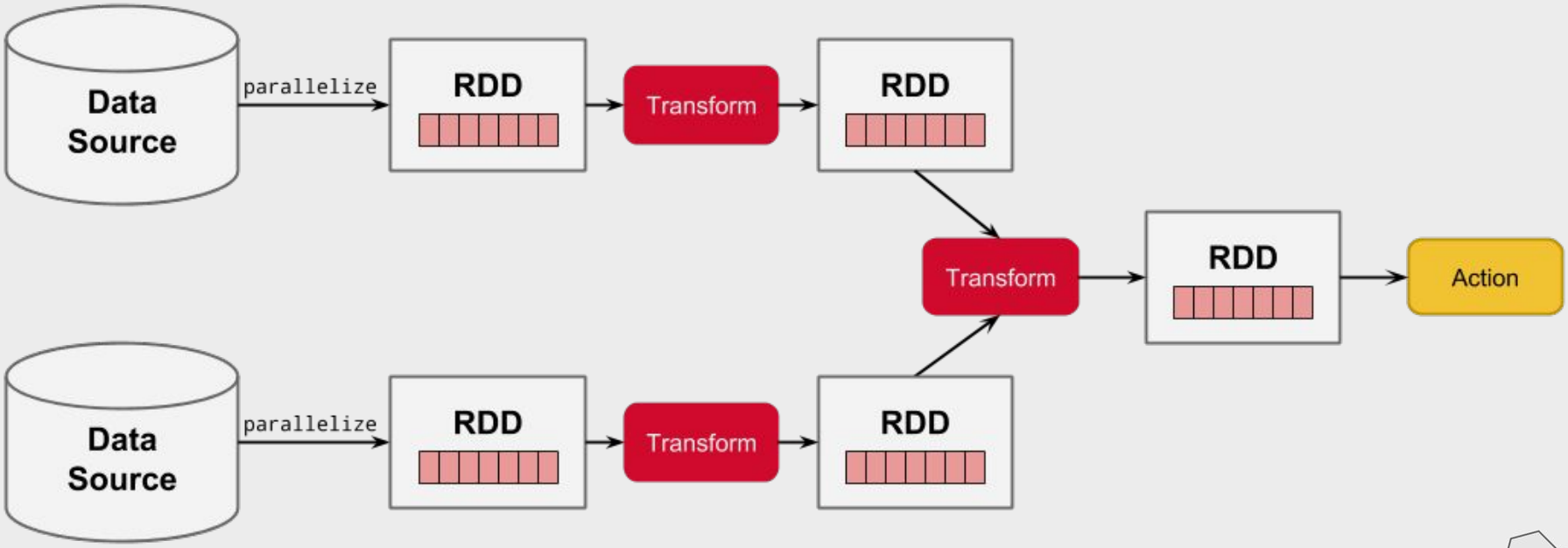

HDFS
Source
Output
Error Output
Destination
Input
Source
Transformation
Output
Error Output
Destination
Input
Destination

Hadoop Data Flow

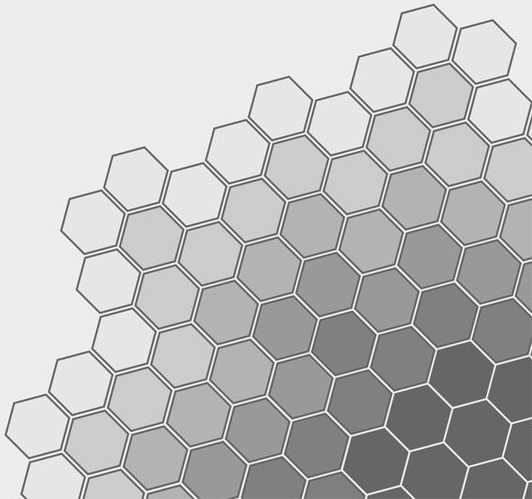




Spark Data Flow



Big Data and Machine Learning



Hypothesis One

More examples means better machine learning.

- See the forest from the trees.
- Noisy data

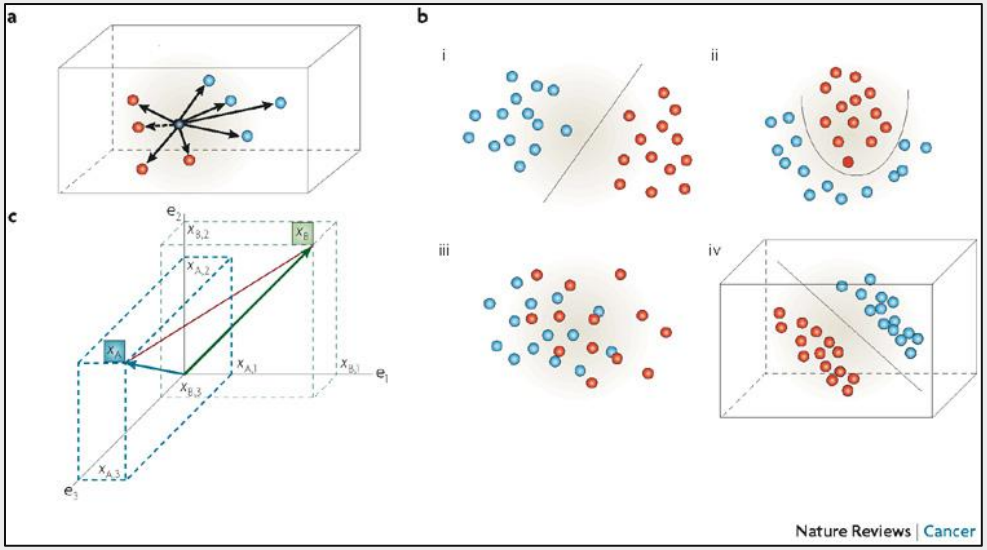
Attempt to capture a complete search space.

Hypothesis Two

Increasing dimensions or finding better properties improves pattern recognition.

- Semantic Embedding
- Separability

Attempt to divide a search space better.

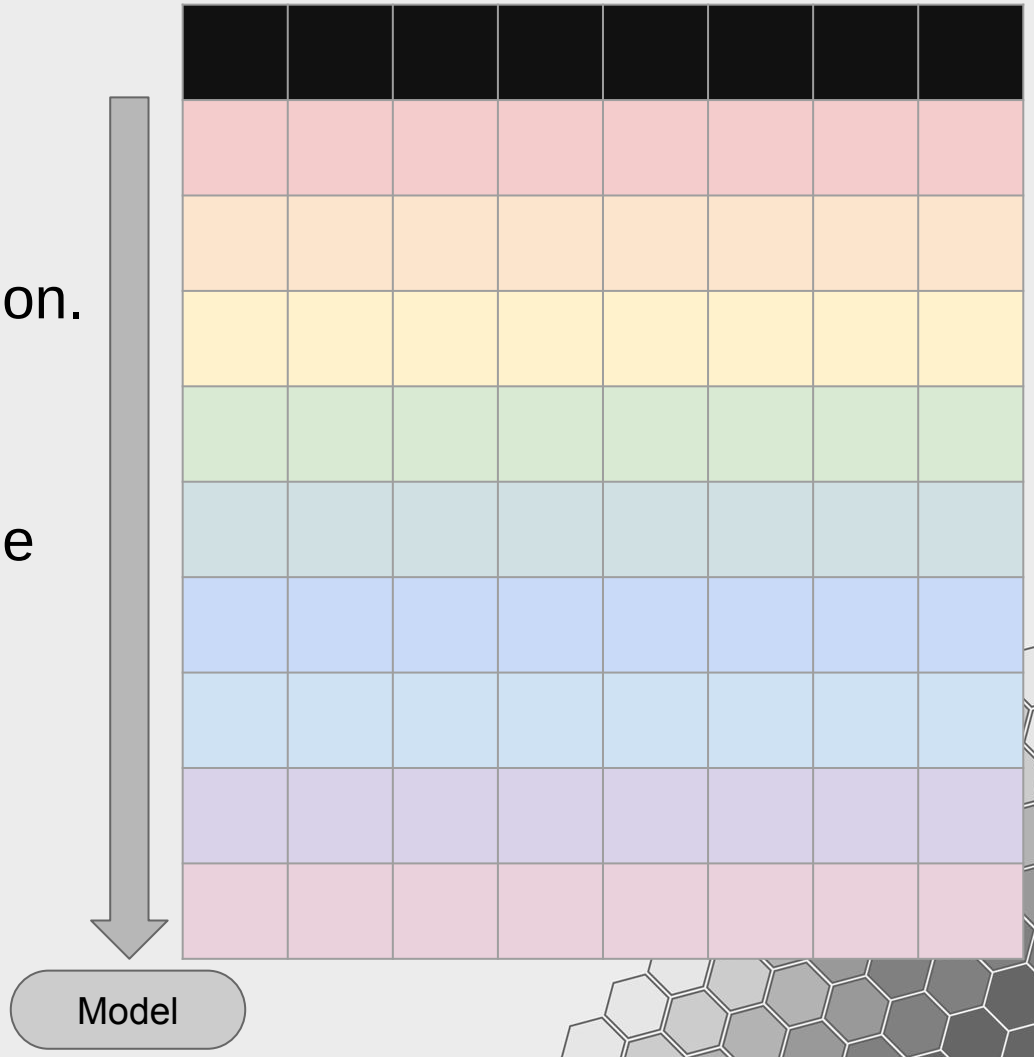


Sequential Machine Learning

Machine learning generally finds the best set of parameters for a model by optimizing some error function.

(This is most apparent in regression)

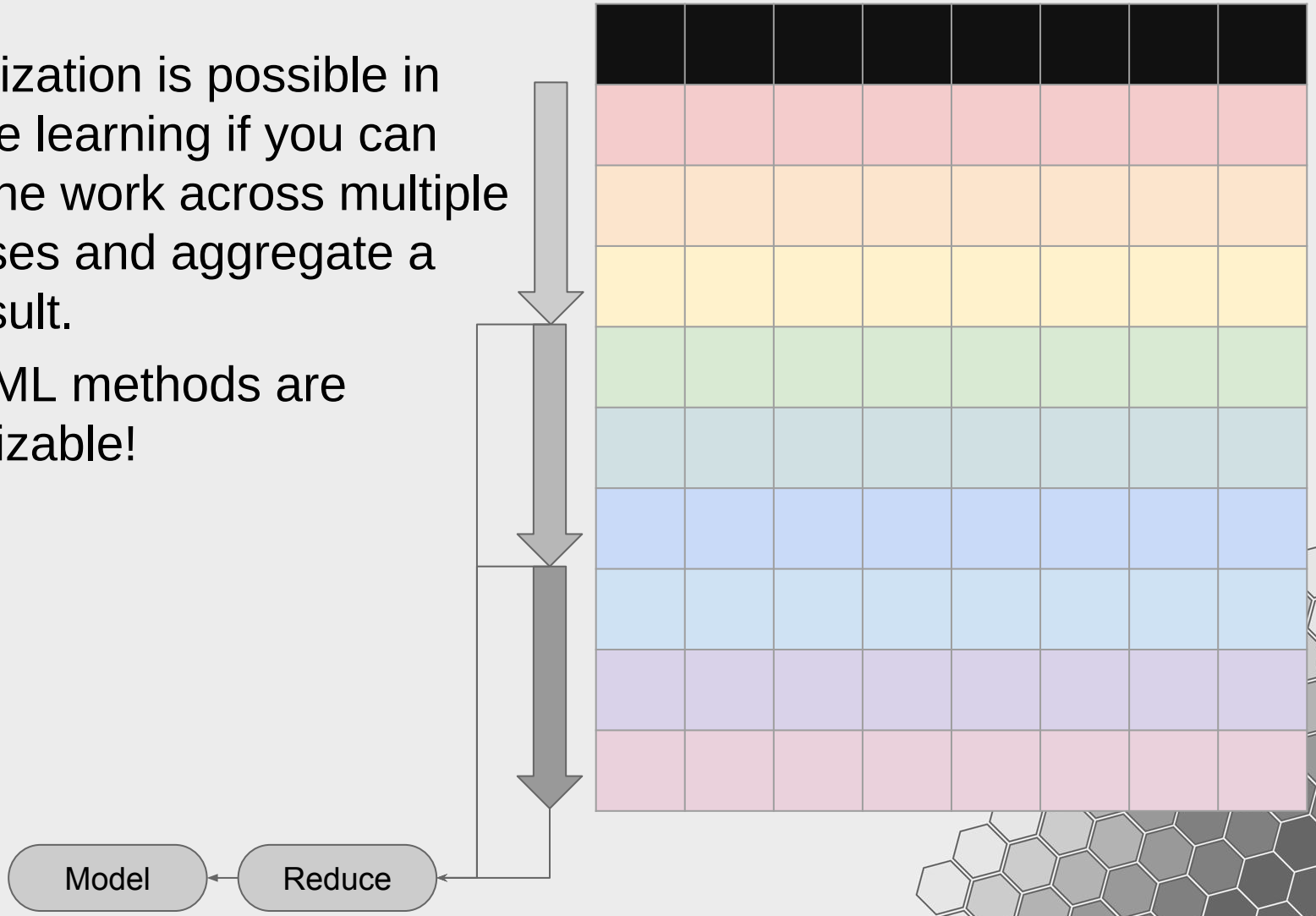
Multiple passes over multiple instances.



Parallel Machine Learning

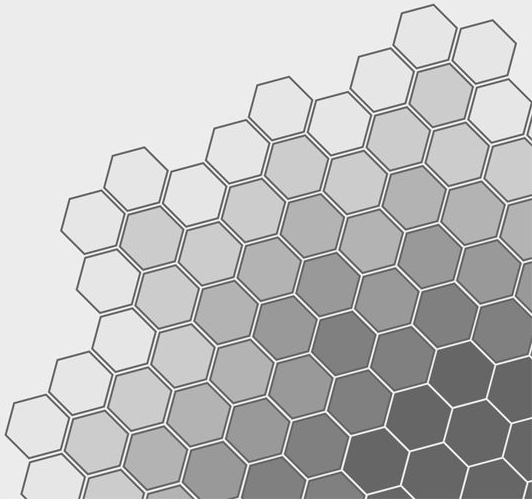
Parallelization is possible in machine learning if you can divide the work across multiple processes and aggregate a final result.

Not all ML methods are parallelizable!



Linear Regression

How would we implement Gradient Descent on a cluster? (Besides using a library for this)

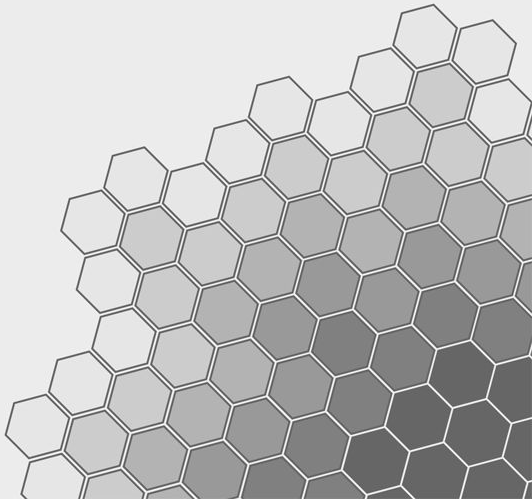


Linear Regression

How would we implement Gradient Descent on a cluster? (Besides using a library for this)

Train on all data:

- initialize some set of parameters, Θ
- broadcast Θ to nodes in a cluster
- map $\rightarrow$ use Θ to compute error
- reduce $\rightarrow$ sum
- modify Θ to reduce errors, repeat until threshold.

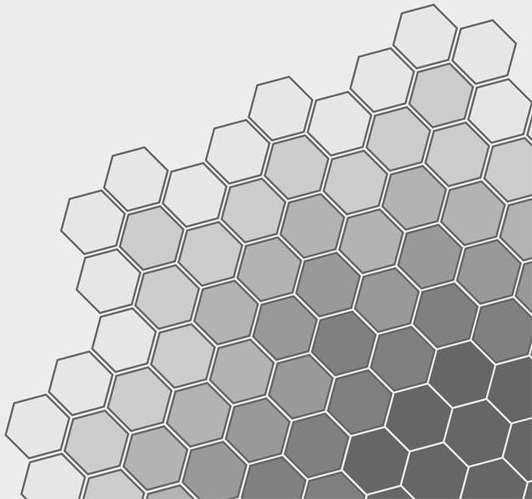


Linear Regression

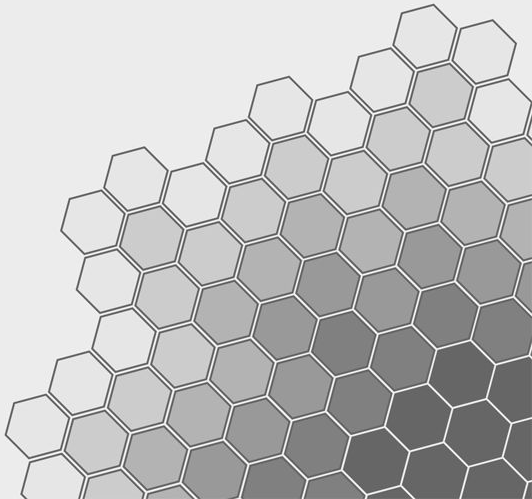
How would we implement Gradient Descent on a cluster? (Besides using a library for this)

Sample and Evaluate:

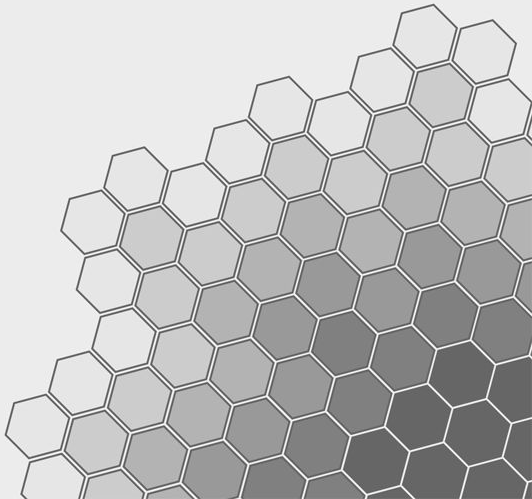
- use reservoir sampling to collect a small dataset
- fit model on small subsample
- broadcast params to entire dataset
- compute error

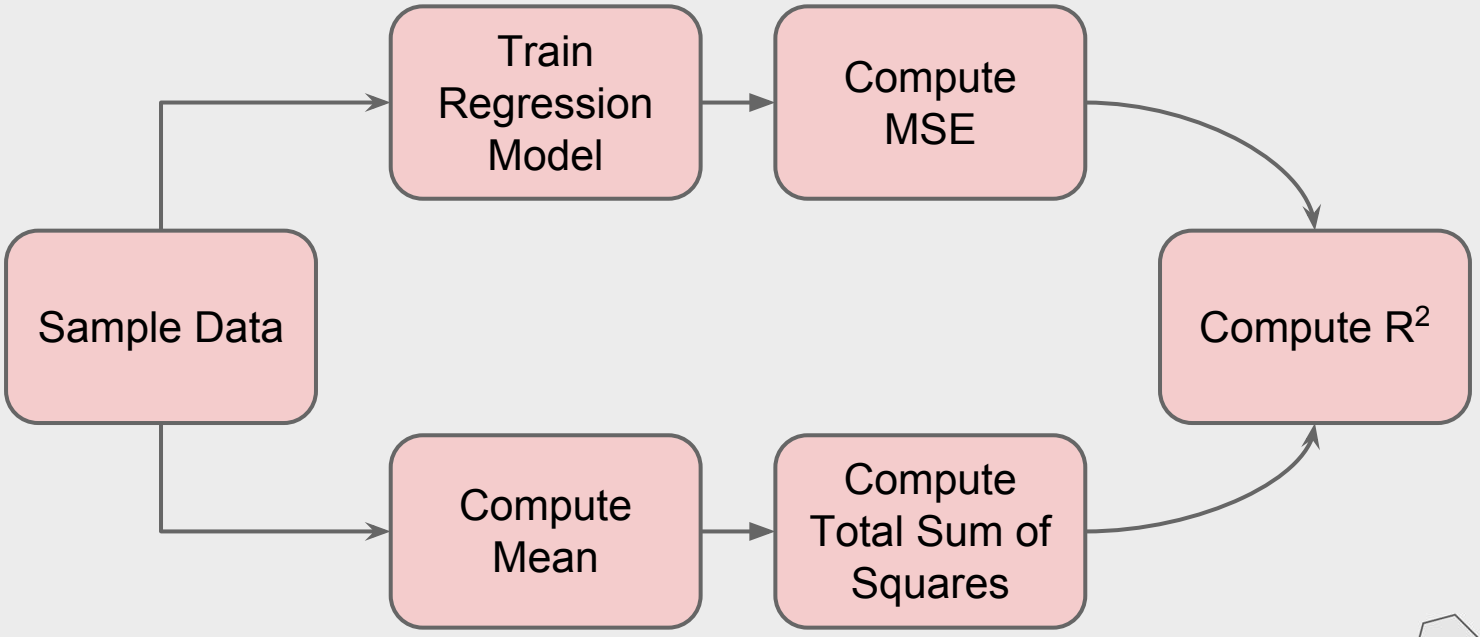


```
fields = ("fixed_acidity", "volatile_acidity","citric_acid", … )

def parse(row):
    return dict(zip(fields, [float(item) for item in row]))

class ThetaMapper(Mapper):

    def __init__(self, *args, **kwargs):
        super(ThetaMapper, self).__init__(*args, **kwargs)

        with open('theta.json', 'r') as thetas:
            self.thetas = json.load(thetas)

    def compute_cost(self, values):
        yhat = 0
        for key, val in self.thetas.items():
            yhat += values.get(key, 1) * val

        return (values['quality'] - yhat) ** 2

    def map(self):
        for wine in self:
            self.emit(1, self.compute_cost(wine))

    def __iter__(self):
        reader = csv.reader(self.read(), delimiter=";")
        for wine in map(parse, reader):
            yield wine
```



```
def cost(row, clf=None):
    """Computes the square error given the row."""
    return (row[0] - clf.predict(row[1:])) ** 2


def main(sc):
    # Load the model from the pickle file
    with open('clf.pickle', 'rb') as f:
        clf = sc.broadcast(model.load(f))

    # Create an accumulator to sum the squared error
    sum_square_error = sc.accumulator(0)

    # Load and parse the blog data
    blogs = sc.textFile("blogData").map(float)

    # Map the cost function and accumulate the sum.
    error = blogs.map(partial(cost, clf=clf))
    error.foreach(lambda cost: sum_square_error.add(cost))

    # Print and compute the mean.
    print sum_square_error.value / error.count()
```




Train Regression Model
Compute MSE
Sample Data
Compute R^2
Compute Mean
Compute Total Sum of Squares

Two Approaches to Big Data ML

Decompose to Memory

Use decomposition methods to reduce input domain into something that can fit into memory (filtering, sampling, summarization, indexing).

Compute model in memory on a single machine (128 GB).

Evaluate model on cluster.

Boost Weaker Models

Use distributed implementations for models that are in Mahout or Spark to perform computation.

Boost or bag the models together to produce a stronger model.

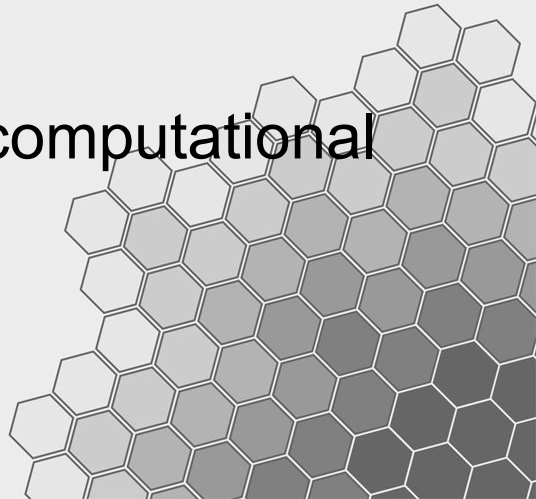
Performs better with some models, e.g. Random Forest.

Parallel Canopy Clustering

Canopy clustering is an unsupervised *pre-clustering* algorithm often used as a preprocessing step for K-Means clustering or Hierarchical clustering.

This algorithm is intended to speed up other clustering algorithms like K-Means or Hierarchical clustering with very large data sets that make these clustering algorithms impractical.

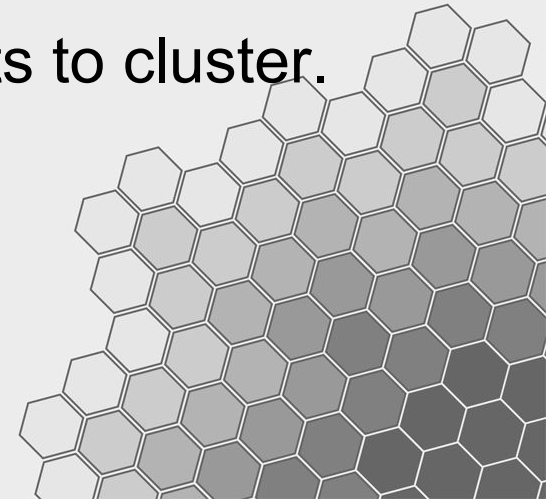
Basically a form of “blocking” - reducing our computational space.

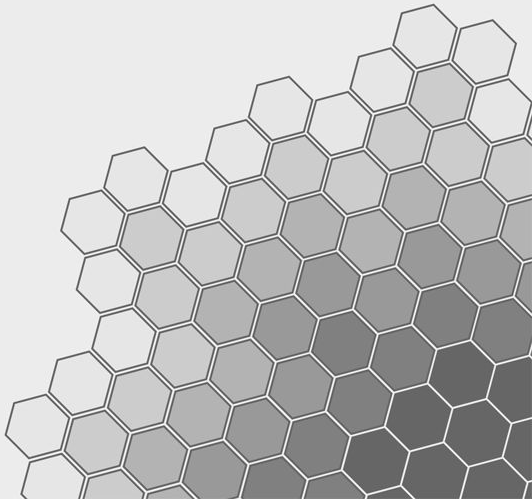


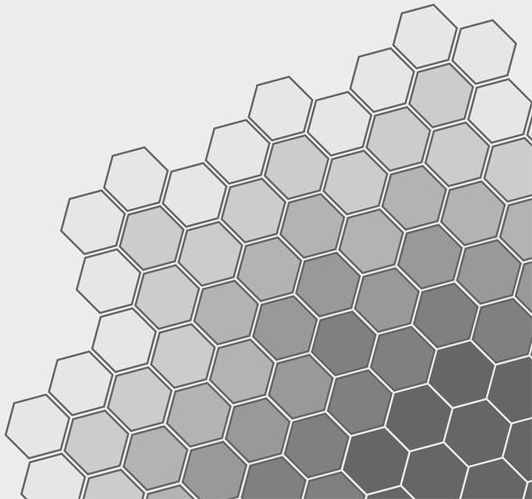
Canopy Clustering

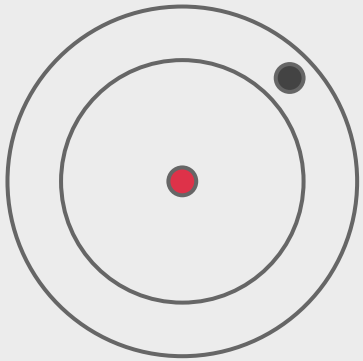
The algorithm begins with two thresholds T_1 and T_2 the loose and tight distances respectively, where $T_1 > T_2$.

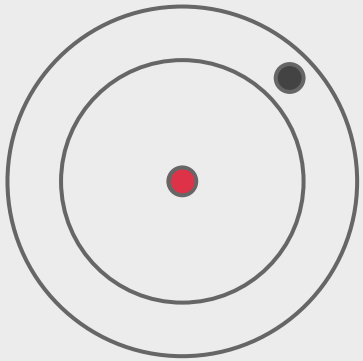
1. Remove a point from the set and start a new “canopy”
2. For each point in the set, assign it to the new canopy if the distance is less than the loose distance T_1.
3. If the distance is less than T_2 remove it from the original set completely.
4. Repeat until there are no more data points to cluster.

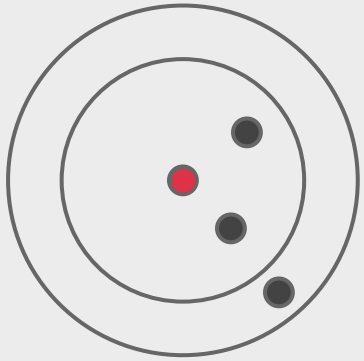


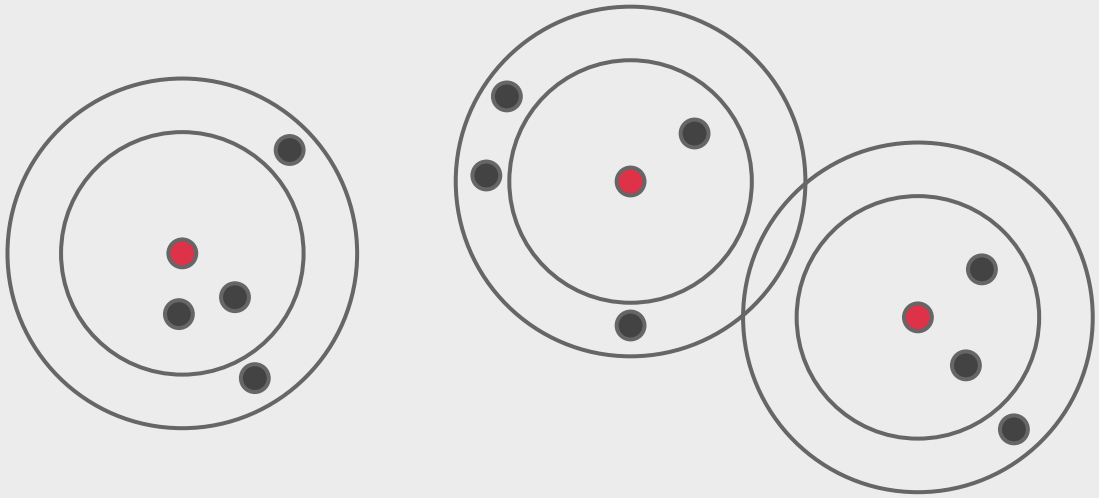


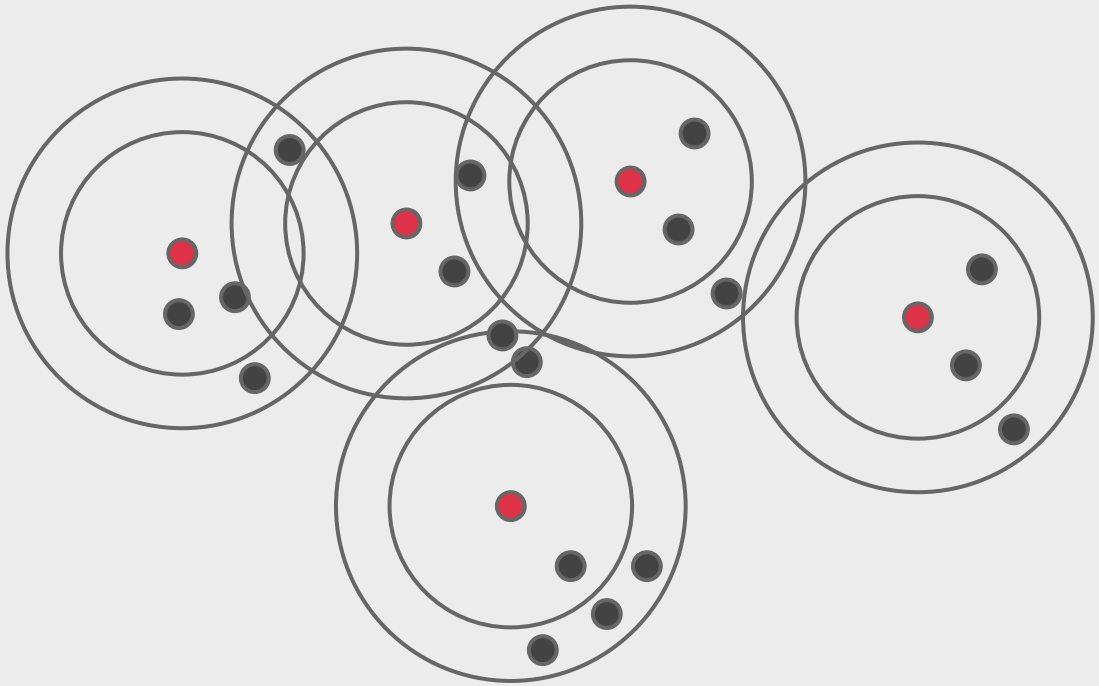




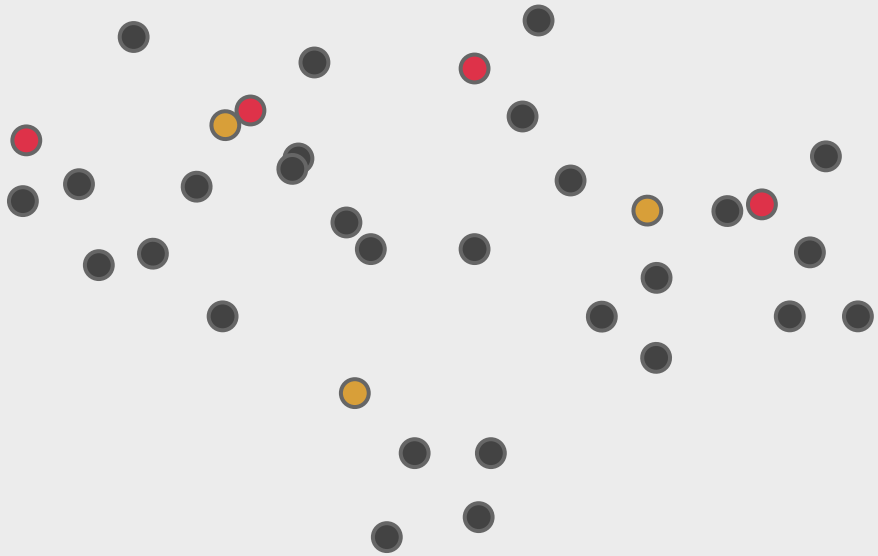


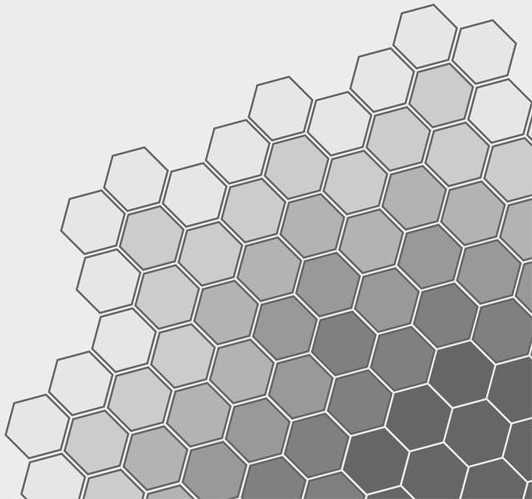




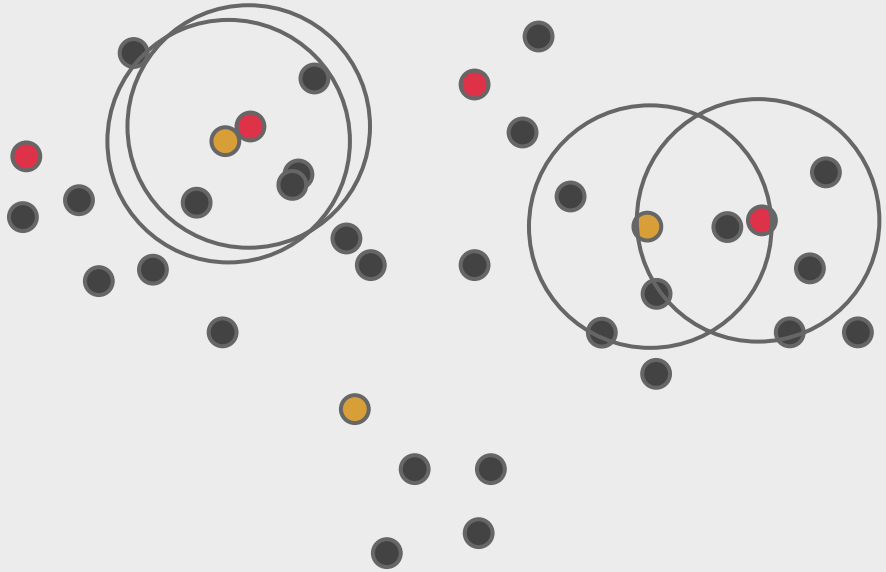


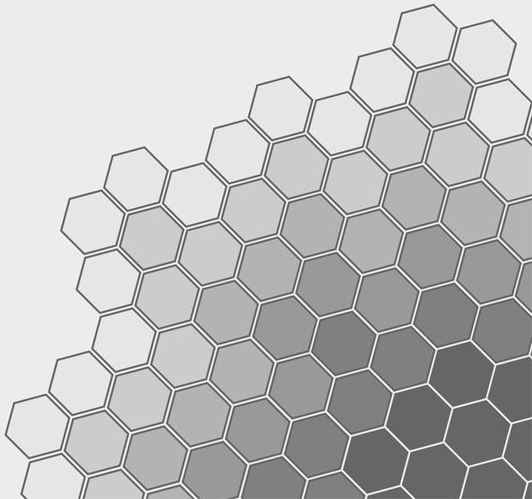
Mapper A - Red Center
Mapper B - Gold Center





Reducer computes redundant centers that are within the threshold of each other.



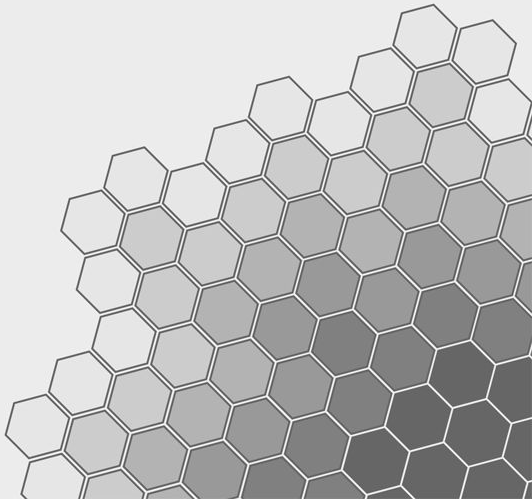


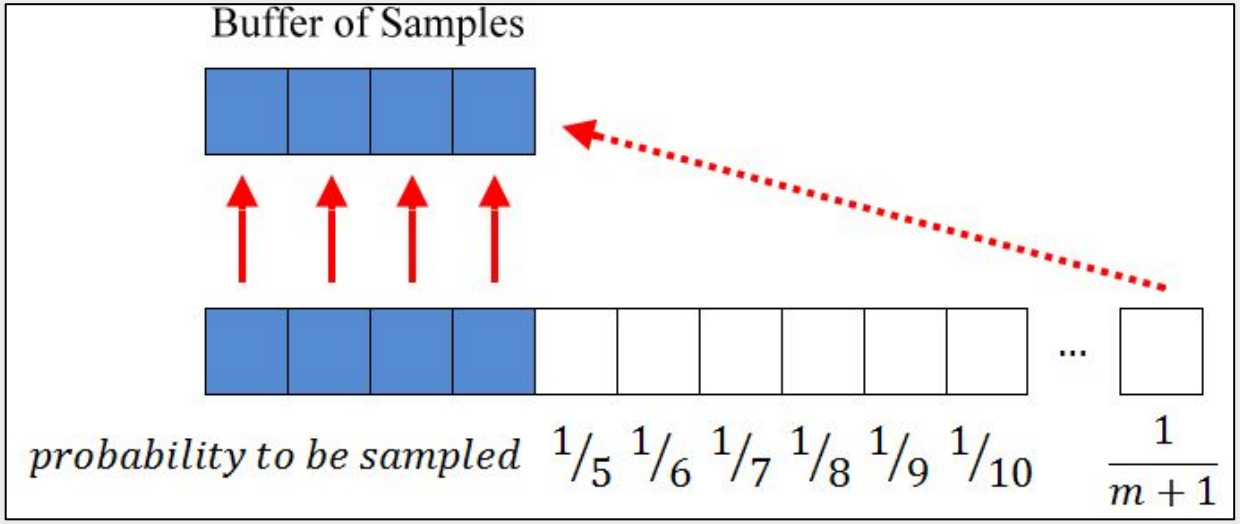
Canopy Clustering: MR Jobs

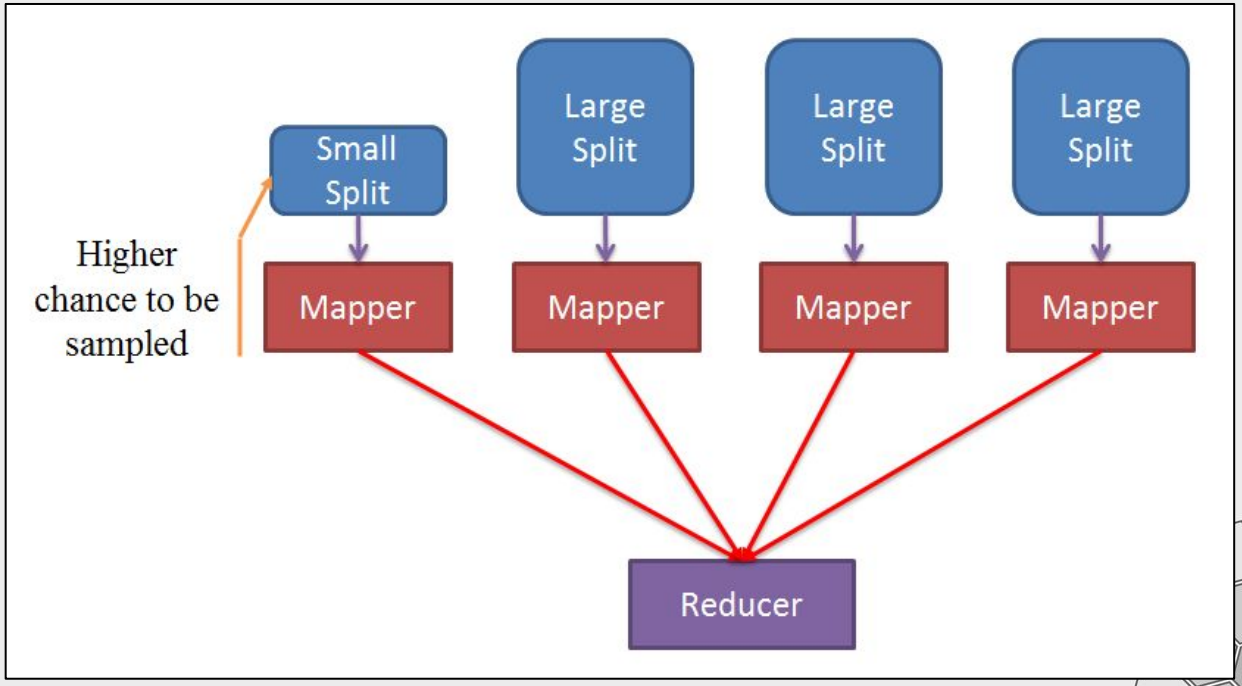
1. Iterative MapReduce job to compute canopies
2. CanopyMapper goes through points to see if the point fits within the threshold of a previously discovered canopy, otherwise emits a new canopy.
3. CanopyReducer merges canopy centers from mappers.
4. Final MapReduce job emits the canopy that each point belongs to.
5. Use a custom partitioner to ensure that each reducer sees canopies that are close to each other.
6. Perform K-Means, etc on canopies emitted by reducer.

Sample % of the Dataset

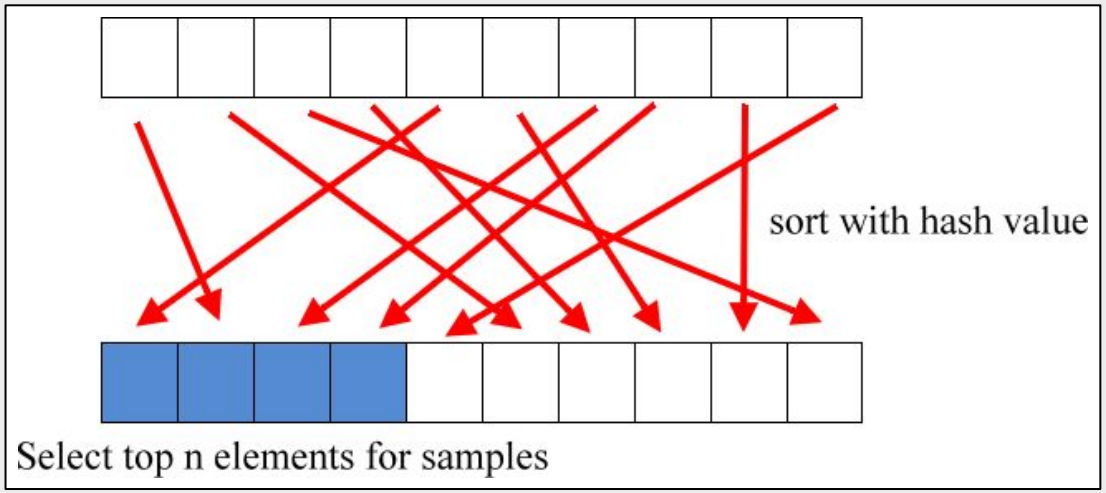
```
import random

class PercentSampleMapper(Mapper):

    def __init__(self, *args, **kwargs):
        self.percentage = kwargs.pop("percentage")
        super(PercentSampleMapper, self).__init__(*args, **kwargs)

    def map(self):
        for _, val in self:
            if random.random() < self.percentage:
                self.emit(None, val)

if __name__ == '__main__':
    mapper = PercentSampleMapper(sys.stdin, percentage=0.20)
    mapper.map()
```

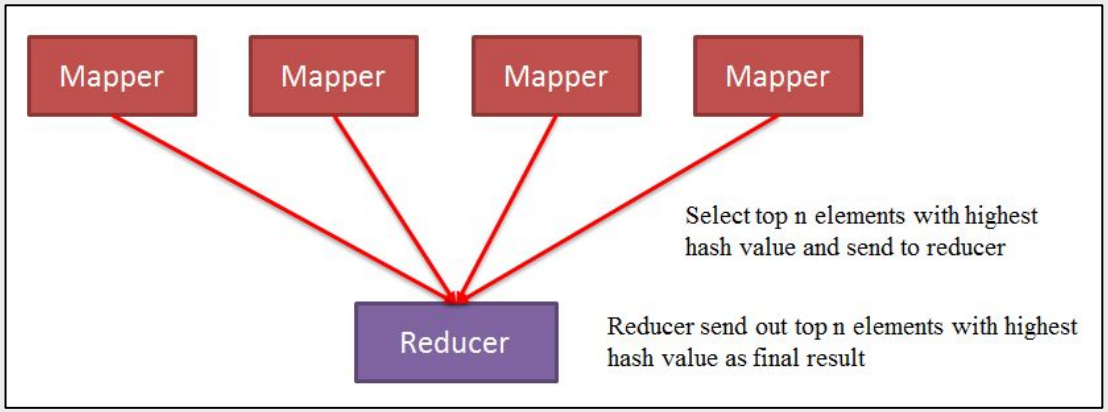






Reservoir Sampling Problem

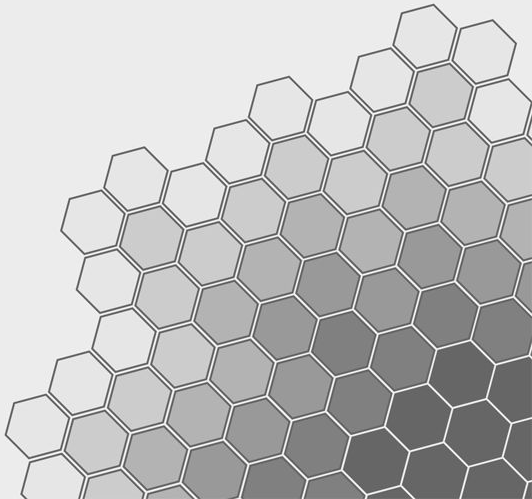




Solution: Top N of Top N Random Numbers

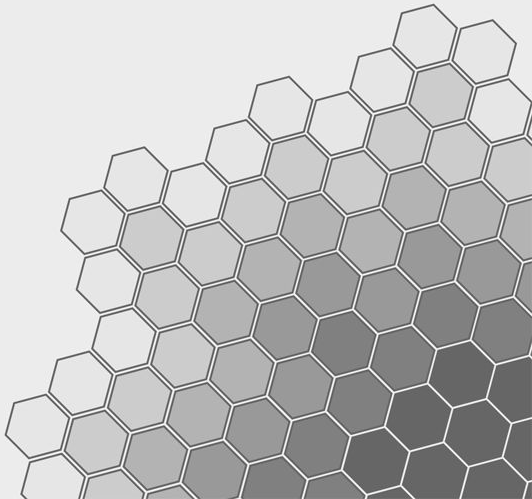
Distributed Reservoir

```
import random, heapq

class SampleMapper(Mapper):

    def __init__(self, n, *args, **kwargs):
        self.n = n
        super(SampleMapper, self).__init__(*args, **kwargs)

    def map(self):
        # initialize our heap as a list with n zeros
        heap = [0 for x in xrange(self.n)]

        for value in self:
            # maintain a heap of only n largest values
            heapq.heappushpop(heap, (random.random(), value))

        for item in heap:
            # emit the sampled values
            self.emit(None, item)
```

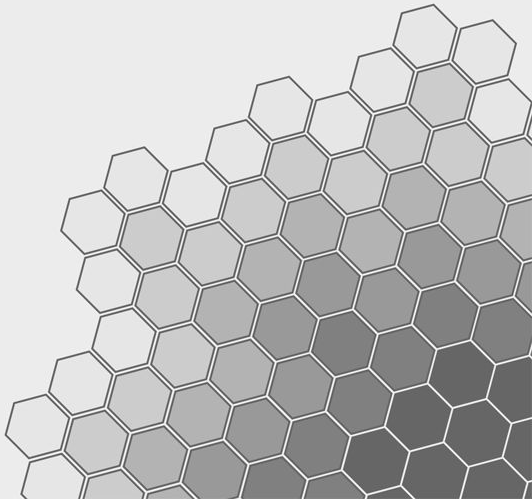


Distributed Reservoir

```
class SampleReducer(Mapper):

    def __init__(self, n, *args, **kwargs):
        self.n = n
        super(SampleReducer, self).__init__(*args, **kwargs)

    def reduce(self):
        # initialize our heap as a list with n zeros
        heap = [0 for x in xrange(self.n)]

        for _, values in self:
            for value in values:
                heapq.heappushpop(heap, make_tuple(value))

        for item in heap:
            # emit the sampled values
            self.emit(None, item[1])
```

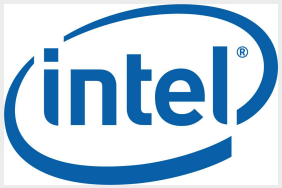


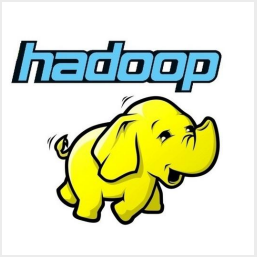
The Hadoop Ecosystem



Hadoop Distributions

Hadoop has developed a Linux-like cloning, with a variety of distributions from trusted vendors; enabling businesses with professional support for the open source product.





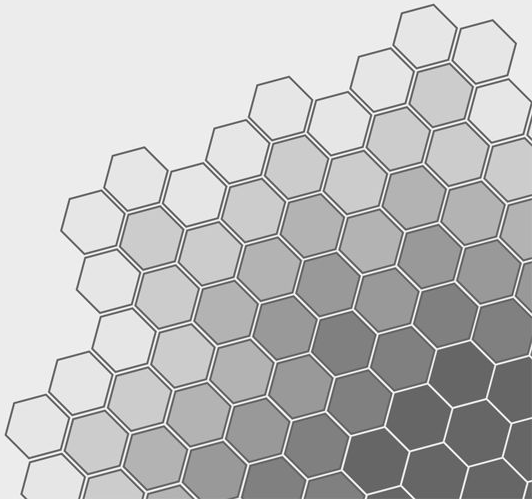
Hadoop Ecosystem

Sqoop

Designed to transfer data between Hadoop and Relational Databases.

Uses JDBC connectors to connect to relational databases which means most databases are supported.

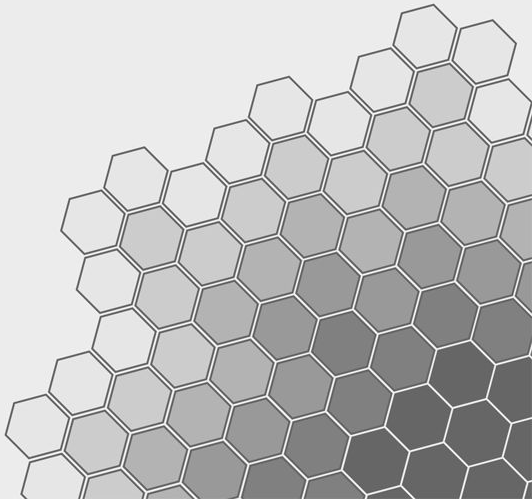
Sqoop can import and export data to HDFS, Hive, or even HBase.



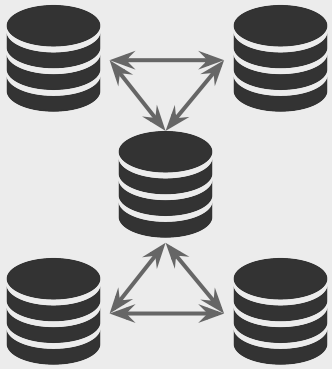
NoSQL Databases

HDFS allows for easy *horizontal scaling* of data - by the addition of more machines for storage. However, traditional relational databases usually scale via vertical scaling: the addition of more powerful machines.

The rise of Big Data has led to the advent of *NoSQL* databases - non-tabular data stores whose data formats allow for easy horizontal scaling and geographic distribution.



NoSQL Databases fall into four general categories:



Key-Value Store
A hash table stores a unique key and a pointer to a value somewhere in the data set. Simple, reliable and the basis of many NoSQL systems.

Riak, SimpleDB

Columnar Stores
Designed particularly for distributed databases with large data sets, keys point to multiple columns that are arranged together by column family.

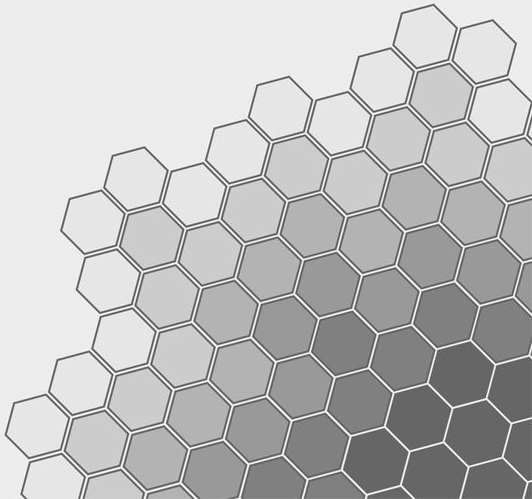
Cassandra, HBase

Document Databases
Storage of semi-structured documents, particularly JSON. Keys and values are stored together as documents and can be nested.

MongoDB, CouchDB

Graph Databases
Data is stored in nodes and the relationships between data as edges, allowing for flexible graph queries and traversals.

Neo4j, Titan

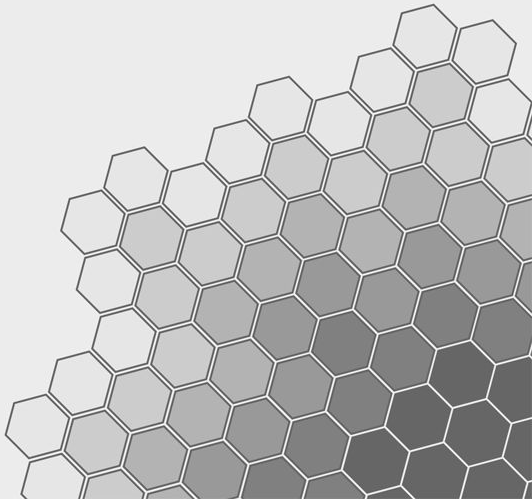


HBase

The Hadoop database - allows random, realtime read/write access to big data via columnar storage. HBase implements a distributed NoSQL database based off of Google's Bigtable.

HBase is highly compressed and has fast access via an in-memory key store.

HBase is accessed via MapReduce, through a REST API and a Java programming interface.

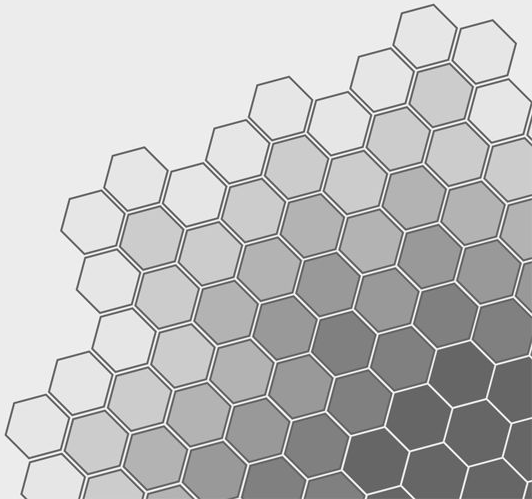


Cassandra

Linearly scalable columnar data store that is also based off of Google BigTable; and one of the premier columnar NoSQL databases.

Not part of Hadoop, nor does Cassandra require HDFS or any part of the Hadoop cluster to function.

However, it is commonly included on Hadoop clusters, and Cassandra includes a Hadoop InputFormat and OutPut so that data in Cassandra can be analyzed with MapReduce jobs or have outputs stored in the columnar storage.



Accumulo

Yet a third NoSQL database - like HBase also built on top of Hadoop, also a columnar key-value store. But, unlike HBase; Accumulo was built the the NSA.

Accumulo has instant cell-level read/write access as well as high consistency for the data store. It also has cell-level security.

Each cell is assigned per-user permissions which allows massive databases to scale but also include security and privacy restrictions at scale.