Introduction to Hadoop

Igor Yakushin
ivy2@uchicago.edu

September 28, 2019

Login and slides

- Use ssh client to get to Hadoop cluster

  ```
  ssh -Y YourUserName@hadoop.rcc.uchicago.edu
  ```

- Use your normal university username, password, 2fa
- To get slides and labs, point the browser to:

  https://github.com/igory1999/Graham_Introduction_to_Hadoop

- After logging into Hadoop cluster, get code and slides:

  ```
  git clone https://github.com/igory1999/Graham_Introduction_to_Hadoop.git
  ```

What is Hadoop?

- A framework to process huge amount of data
- Implemented in Java
- Runs on a cluster of commodity computers, from a few to a few thousand nodes
- Consists of two major components:
  - HDFS - distributed file system; everything you put there is automatically divided into blocks that are replicated and spread across the cluster;
  - MapReduce - an approach to perform calculations on such data in parallel.

Hadoop zoo

- There is a zoo of other Hadoop related tools
  - Hive - SQL interface to Hadoop built on top of HDFS and MapReduce
  - Impala - another SQL interface to Hadoop, allegedly much faster
  - HBase - noSQL fast distributed database on top of HDFS
  - Pig - data processing languague built on top of HDFS and MapReduce
  - Spark - for performance reasons no longer relies on MapReduce, can be used without Hadoop; its native language is Scala but it can also be used from Java, Python, R
  - Sqoop - copy data between relational database and Hadoop
  - Flume - copy data into Hadoop, typically used to store logs from a cluster for subsequent analysis
  - YARN - resource allocation manager, used to submit jobs
  - Zookeeper - centralized service for maintaining configuration information, naming, providing distributed synchronization
  - Hue - web GUI to many of the above tools
- In Big Data Platform class you'll mostly use pySpark via JupyterHub interface, HDFS, Hive, Pig, Hue.

Hadoop vs traditional RDBMS

RDBMS	**Hadoop**
do not scale well beyond a few terabytes	easily scales to petabytes by adding more computers to the cluster
for large datasets runs on very expensive enterprise server	data is scattered on a cluster of cheap commodity computers
works on structured data; one needs to predefine the data schema	can accept any unstructured data and worry about interpreting and reinterpreting it later
data needs to be normalized to avoid duplication and enforce constraints	works best on a single denormalized table

Hadoop vs traditional RDBMS

RDBMS	**Hadoop**
can be faster for queries on small subset of data	might introduce too much overhead for such queries
ACID - compliant	in general - not ACID-compliant
natively speaks SQL	natively implements MapReduce approach in Java on top of which some subset of SQL might be supported
is good for banks to keep track of transactions	is good for data scientists to try various ideas on a huge sets of data

ACID compliance

‘

- **A**tomicity - The database transaction must completely succeed or completely fail
- **C**onsistency - During the database transaction, RDBMS progresses from one valid state to another. The state is never invalid
- **I**solation - The client's database transaction must occur in isolation from other clients attempting to transact
- **D**urability - Once transaction is committed, it will remain so, even in the event of power loss, crashes, or errors. The data operation that was part of the transaction must be reflected in nonvolatile storage.

Hadoop has no concept of transaction so is not ACID compiliant.

Distributions

There are many Hadoop distributions that bundle different set of tools and add their own:

- Apache - free, open source;
- Cloudera - we are using Cloudera 6.3;
- Hortonworks - Hortonworks and Cloudera recently merged but they still maintain two separate distributions and working on the joint one
- HDInsight - Microsoft Azure's Cloud based Hadoop Distribution
- MapR
- ...

Many distributions, for example Cloudera, provide virtual machines to play with

HDFS

- Any file that is put into HDFS is automatically split into blocks (by default 128M)
- Each block is replicated (by default 3 times)
- The blocks are spread across the cluster so that
  - calculations can be done in parallel on different blocks to increase performance
  - the file system is fault-taulerant with respect to disk/node/rack failure
  - if a node goes down, HDFS takes care of creating more replicas automatically
  - when a node comes back, extra replicas are destroyed
  - if a new node is added, data spreads on it automatically
  - there is a NameNode that keeps track of where the blocks are
- The calculations are done on DataNodes
- Hadoop tries to do computations where the data is to minimize communication between nodes which is much slower

HDFS: user interface

```
$ hdfs dfs -ls /user/$USER
drwx------   - ivy2 ivy2          0 2016-08-12 20:11 /user/ivy2/.Trash

$ hdfs dfs -mkdir /user/$USER/test2

$ hdfs dfs -put /usr/share/dict/linux.words /user/$USER/test2/

$ hdfs dfs -ls -h  /user/$USER/test2/
-rw-r--r--   3 ivy2 ivy2      4.7 M 2016-08-12 20:12 /user/ivy2/test2/linux.words

$ hdfs dfs -setrep 4  /user/$USER/test2/linux.words
Replication 4 set: /user/ivy2/test2/linux.words

$ hdfs dfs -ls -h  /user/$USER/test2/
-rw-r--r--   4 ivy2 ivy2      4.7 M 2016-08-12 20:12 /user/ivy2/test2/linux.words

$ hdfs dfs -mv /user/$USER/test2/linux.words /user/$USER/test2/words.txt

$ hdfs dfs -get /user/$USER/test2/words.txt

$ hdfs dfs -rm /user/$USER/test2/words.txt
```

- Lab 1

MapReduce

- **map** - convert each record into (*key*, *value*) pairs
- **reduce** - apply some reduce operation to the resulting set of (*key*, *value*); for example, sum values for each key, or find max, or min, etc.
- Obviously, maps can be done in parallel, independently from each other, on different nodes, for each record
- Between local map and global reduce, perhaps, apply reduce locally on each node before doing it globally and sort the results
- The native way to write your own MapReduce is to extend the corresponding Java classes from MapReduce API
- These days people rarely have to implement their own MapReduce but rather use high level tools like Pig, Hive, Spark, HBase, etc. unless you are a programmer implementing such a high level tool

Examples of applying MapReduce approach

- Count how many times each word occurs in a text
  - map - for each word w_i in a record (for example, a line in a text file) output $(w_i, 1)$
  - reduce - for each w_i sum the values
- Count the average number of social contacts a person has grouped by age
  - map - for each person p_i of an age a_i in a record, output $(a_i, (1, contacts_i))$
  - reduce - for each age a_i, sum separately the first and second component of the value to obtain number of people of a given age and total number of contacts, divide the latter by the former

Examples of applying MapReduce approach

- Finding common friends, like in Facebook. For each person a list of friends is given:

```
A -> B C D
B -> A C D E
C -> A B D E
D -> A B C E
E -> B C D
```

  - map - key is a person and a friend, sorted; value is the list of friends. For example, for C the output from map is:

```
(A C) -> A B D E
(B C) -> A B D E
(C D) -> A B D E
(C E) -> A B D E
```

  - reduce - group the results by key, for example:

```
(A B) -> (A C D E) (B C D)
```

    and find the intersection of value lists:

```
(A B) -> (C D)
```

MapReduce: using Java API

- Import various Hadoop related modules:

```
import java.io.IOException;
import java.util.StringTokenizer;

import org.apache.hadoop.conf.Configuration;
import org.apache.hadoop.fs.Path;
import org.apache.hadoop.io.IntWritable;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.mapreduce.Job;
import org.apache.hadoop.mapreduce.Mapper;
import org.apache.hadoop.mapreduce.Reducer;
import org.apache.hadoop.mapreduce.lib.input.FileInputFormat;
import org.apache.hadoop.mapreduce.lib.output.FileOutputFormat;
```

MapReduce: using Java API

- Inside your own WordCount class, create a class that extends Mapper:

```
public class WordCount {

  public static class TokenizerMapper
       extends Mapper<Object, Text, Text, IntWritable>{

       private final static IntWritable one = new IntWritable(1);
       private Text word = new Text();

       public void map(Object key, Text value, Context context
                       ) throws IOException, InterruptedException {
           StringTokenizer itr = new StringTokenizer(value.toString());
           while (itr.hasMoreTokens()) {
               word.set(itr.nextToken());
               context.write(word, one);
           }
       }
  }
}
```

- TokenizerMapper overwrites map method to split each line into words and return $(word, 1)$ for each *word*.

MapReduce: using Java API

- IntSumReducer extends Reducer class and overwrites its reduce method to count number of words:

```
public static class IntSumReducer
    extends Reducer<Text,IntWritable,Text,IntWritable> {
    private IntWritable result = new IntWritable();

    public void reduce(Text key, Iterable<IntWritable> values,
                       Context context
                       ) throws IOException, InterruptedException {
        int sum = 0;
        for (IntWritable val : values) {
            sum += val.get();
        }
        result.set(sum);
        context.write(key, result);
    }
}
```

MapReduce: using Java API

- main function sets up configuration, launches MapReduce job and writes the results to a file:

```
    public static void main(String[] args) throws Exception {
        Configuration conf = new Configuration();
        Job job = Job.getInstance(conf, "word count");
        job.setJarByClass(WordCount.class);
        job.setMapperClass(TokenizerMapper.class);
        job.setCombinerClass(IntSumReducer.class);
        job.setReducerClass(IntSumReducer.class);
        job.setOutputKeyClass(Text.class);
        job.setOutputValueClass(IntWritable.class);
        FileInputFormat.addInputPath(job, new Path(args[0]));
        FileOutputFormat.setOutputPath(job, new Path(args[1]));
        System.exit(job.waitForCompletion(true) ? 0 : 1);
    }
}
```

MapReduce: using Java API

```
source env.sh
mkdir wordcount_classes
javac -cp /opt/cloudera/parcels/CDH/lib/hadoop/*:\
          /opt/cloudera/parcels/CDH/lib/hadoop/client-0.20/*\
          -d wordcount_classes WordCount.java
jar -cvf wordcount.jar -C wordcount_classes/ .

hdfs dfs -mkdir /user/$USER/wordcount
hdfs dfs -rm -r -f /user/$USER/wordcount/output
hdfs dfs -put  /software/matlab-2014b-x86_64/\
               toolbox/distcomp/examples/integration/old/pbs/README
               /user/$USER/wordcount/

hadoop jar wordcount.jar WordCount
           /user/$USER/wordcount/README /user/$USER/wordcount/output

hdfs dfs -ls /user/ivy2/wordcount/output
hdfs dfs -cat wordcount/output/part-r-00000
hdfs dfs -cat wordcount/output/part-r-000* | sort > out.txt
hdfs dfs -getmerge wordcount/output merged.txt
cat merged.txt | grep sort
```

- Lab 2

MapReduce: using streaming

- Streaming interface allows one to write MapReduce in any language although the functionality is limited and the performance is worse than what one can get by using native Java API.
- Mapper takes input from stdin, breaks it into records - lines in a text file - and is expected to print to stdout pairs of key and value separated by Tab; everything before the first Tab is considered a key, the rest of the line is considered a value
- Similarly reducer is expected to receive such pairs in stdin and prints its results to stdout
- When launching a job, one needs to use Hadoop's streaming jar and specify mapper, reducer, input and output files as options

MapReduce: using streaming

- In this example we implement WordCount.java functionality in python
- mapper.py:

```
import sys

for line in sys.stdin:
    line = line.strip()
    words = line.split()
    for word in words:
        print '%s\t%s' % (word, 1)
```

MapReduce: using streaming

- reducer.py:

```
from operator import itemgetter
import sys

current_word = None
current_count = 0
word = None

for line in sys.stdin:
    line = line.strip()
    word, count = line.split('\t', 1)
    try:
        count = int(count)
    except ValueError:
        continue

    if current_word == word:
        current_count += count
    else:
        if current_word:
            print '%s\t%s' % (current_word, current_count)
        current_count = count
        current_word = word

if current_word == word:
    print '%s\t%s' % (current_word, current_count)
```

MapReduce: using streaming

- To run a job:

```
hadoop jar /opt/cloudera/parcels/CDH/lib/hadoop-mapreduce/hadoop-streaming.jar \
       -input /user/$USER/wordcount/README \
       -output /user/$USER/wordcount/streaming-out-py \
       -file mapper.py -mapper mapper.py -file reducer.py -reducer reducer.py
```

- Lab 3

HBase

- **H**adoop data**Base**
- Consists of tables
- Each table is sparse, distributed, persistent, multidimensional sorted map, indexed by rowkey, column family, column, timestamp
- Can store structured, semistructured, unstructured data
- Does not care about types
- Not a relational database, does not speak SQL natively, does not enforce relationship in data
- Designed to run on a cluster of computers, scale horizontally as you add more machines to the cluster
- The main operations are: create (table), put (value into cell), get (value from cell), scan (values from cells)
- Various auxiliary operations: alter, list, describe, ...

HBase

```
Row Key     Column Family: {Column Qualifier:Version:Value}
-------------------------------------------------------------------
00001       CustomerName:  {'FN': 1383859182496:'John',
                            'LN': 1383859182858:'Smith',
                            'MN': 1383859183001:'Timothy',
                            'MN': 1383859182915:'T'}
            ContactInfo:   {'EA': 1383859183030:'John.Smith@xyz.com',
                           'SA': 1383859183073:'1 Hadoop Lane, NY 11111'}
00002       CustomerName:  {'FN': 1383859183103:'Jane',
                            'LN': 1383859183163:'Doe',
            ContactInfo:   {'SA': 1383859185577:'7 HBase Ave, CA 22222'}
```

- Internally HBase table is stored in HFiles - different set of files for different column families
- HFiles for the same column family are periodically merged together or split and distributed among the nodes to maintain high performance and fault-taulerance
- On top of HDFS
- One can specify how many latest versions of data to keep in HBase table or to query versions in a particular date-time range

HBase: shell, create, put, list, describe

```
$ hbase shell

> create 'CustomerContactInfo', 'CustomerName', 'ContactInfo'
> put 'CustomerContactInfo', '00001', 'CustomerName:FN', 'John'
> put 'CustomerContactInfo', '00001', 'CustomerName:LN', 'Smith'
> put 'CustomerContactInfo', '00001', 'CustomerName:MN', 'T'
> put 'CustomerContactInfo', '00001', 'ContactInfo:EA', 'John.Smith@xyz.com'
> put 'CustomerContactInfo', '00001', 'ContactInfo:SA', '1 Hadoop Lane, NY 11111'
> put 'CustomerContactInfo', '00002', 'CustomerName:FN', 'Jane'
> put 'CustomerContactInfo', '00002', 'CustomerName:LN', 'Doe'
> put 'CustomerContactInfo', '00002', 'ContactInfo:SA', '7 HBase Ave, CA 22222'
>list
=> ["CustomerContactInfo"]
>describe 'CustomerContactInfo'
Table CustomerContactInfo is ENABLED
CustomerContactInfo
COLUMN FAMILIES DESCRIPTION
{NAME => 'ContactInfo', DATA_BLOCK_ENCODING => 'NONE', BLOOMFILTER => 'ROW',
REPLICATION_SCOPE => '0', VERSIONS => '1', COMPRESSION => 'NONE',
MIN_VERSIONS => '0', TTL => 'FOREVER', KEEP_DELETED_CELLS => 'FALSE',
BLOCKSIZE => '65536', IN_MEMORY => 'false', BLOCKCACHE => 'true'}
{NAME => 'CustomerName', DATA_BLOCK_ENCODING => 'NONE', BLOOMFILTER => 'ROW',
REPLICATION_SCOPE => '0', VERSIONS => '1', COMPRESSION => 'NONE', MIN_VERSIONS => '0',
TTL => 'FOREVER', KEEP_DELETED_CELLS => 'FALSE', BLOCKSIZE => '65536',
IN_MEMORY => 'false', BLOCKCACHE => 'true'}
```

HBase: alter, scan, VERSIONS

```
> alter 'CustomerContactInfo', NAME => 'CustomerName', VERSIONS => 5
> describe 'CustomerContactInfo'
Table CustomerContactInfo is ENABLED
CustomerContactInfo
COLUMN FAMILIES DESCRIPTION
{NAME => 'ContactInfo', DATA_BLOCK_ENCODING => 'NONE', BLOOMFILTER => 'ROW',
REPLICATION_SCOPE => '0', VERSIONS => '1', COMPRESSION => 'NONE',
MIN_VERSIONS => '0', TTL => 'FOREVER', KEEP_DELETED_CELLS => 'FALSE',
BLOCKSIZE => '65536', IN_MEMORY => 'false', BLOCKCACHE => 'true'}
{NAME => 'CustomerName', DATA_BLOCK_ENCODING => 'NONE', BLOOMFILTER => 'ROW',
REPLICATION_SCOPE => '0', VERSIONS => '5', COMPRESSION => 'NONE',
MIN_VERSIONS => '0', TTL => 'FOREVER', KEEP_DELETED_CELLS => 'FALSE',
BLOCKSIZE => '65536', IN_MEMORY => 'false', BLOCKCACHE => 'true'}
> put 'CustomerContactInfo', '00001', 'CustomerName:MN', 'Timothy'
> scan 'CustomerContactInfo', {VERSIONS => 2}
ROW        COLUMN+CELL
00001      column=ContactInfo:EA, timestamp=1471196578957, value=John.Smith@xyz.com
00001      column=ContactInfo:SA, timestamp=1471196578988, value=1 Hadoop Lane, NY 11111
00001      column=CustomerName:FN, timestamp=1471196578805, value=John
00001      column=CustomerName:LN, timestamp=1471196578859, value=Smith
00001      column=CustomerName:MN, timestamp=1471197270641, value=Timothy
00001      column=CustomerName:MN, timestamp=1471196578901, value=T
00002      column=ContactInfo:SA, timestamp=1471196579070, value=7 HBase Ave, CA 22222
00002      column=CustomerName:FN, timestamp=1471196579016, value=Jane
00002      column=CustomerName:LN, timestamp=1471196579042, value=Doe
```

HBase: get, disable, drop, quit

```
> get 'CustomerContactInfo', '00001'
COLUMN                  CELL
 ContactInfo:EA         timestamp=1471196578957, value=John.Smith@xyz.com
 ContactInfo:SA         timestamp=1471196578988, value=1 Hadoop Lane, NY 11111
 CustomerName:FN        timestamp=1471196578805, value=John
 CustomerName:LN        timestamp=1471196578859, value=Smith
 CustomerName:MN        timestamp=1471197270641, value=Timothy
> get 'CustomerContactInfo', '00001', {COLUMN => 'CustomerName:MN'}
COLUMN                  CELL
 CustomerName:MN        timestamp=1471197270641, value=Timothy
> disable 'CustomerContactInfo'
> drop 'CustomerContactInfo'
> quit
```

- Lab 4

HBase: clients

Besides hbase shell, one can use HBase with

- MapReduce
- Hive
- Pig
- Spark
- Impala

Pig: introduction

- High level language - **Pig Latin**
- Compiler translates Pig Latin into MapReduce jobs
- It is a **dataflow language** where you define a data stream and a set of transformations applied to it.
- Operations: load, store, dump, filter, foreach, group, join, order by, distinct, limit, sample, etc.
- You can specify data types to help Pig to optimize a program or you can let it figure it out

Pig: how to run

Pig programs can run in three ways:

- as a script
  - on a local computer

```
pig -x local milesPerCarrier.pig
```

  - on a cluster

```
pig -x mapreduce milesPerCarrier.pig
```

- interactively using Grunt interpreter

```
pig -x local
```

- Embedded in other languages such as Java, Python, JavaScript

Pig: examples

```
records = LOAD 'pig/words.csv' USING PigStorage(',') AS (W, N:int);
mrecs = GROUP records ALL;
tot = FOREACH mrecs GENERATE SUM(records.N);
DUMP tot;
```

```
in = load 'pig/mary.txt' as (line);
-- TOKENIZE splits the line into a field for each word.
-- flatten will take the collection of records returned by
-- TOKENIZE and produce a separate record for each one, calling the single
-- field in the record word.
words = foreach in generate flatten(TOKENIZE(line)) as word;
grpd = group words by word;
cntd = foreach grpd generate group, COUNT(words);
store cntd into 'cntd.out';
```

```
records = LOAD 'words.csv' USING PigStorage(',') AS (W, N:int);
r1 = filter records by N < 10;
dump r1;
r2 = foreach r1 generate N*N as N2, W;
describe r2;
r3 = join r1 by W, r2 by W;
```

- Lab 5

Hive: introduction

- Hive provides Hadoop with SQL access to data
- Hive server, sitting on top of HDFS and MapReduce, accepts connections from various Hive clients via, for example, ODBC, JDBC drivers and converts SQL into MapReduce jobs
- Hive supports a large subset of SQL including joins
- One can create indexes to improve performance
- When creating a table, one specifies where the data is stored: textfile, HBase table or any other of numerous data formats supported by Hadoop and residing on HDFS

Hive: example

```
$ hive
hive> set hive.cli.print.current.db=true;
hive> CREATE DATABASE my1;
hive> USE my1;
hive (my1)> CREATE TABLE t1(W STRING, N INT) ROW FORMAT DELIMITED FIELDS TERMINATED BY ',';
hive (my1)> LOAD DATA INPATH 'words.csv' INTO TABLE t1;
hive (my1)> select count(*) from t1;
hive (my1)> select max(N) from t1;
```

- To avoid collision, everybody should use a separate database named by username
- Interactive usage is only good for experimentation, don't use it for homework. Instead, prepare a sql script, for example, `my.sql`, and execute it as follows:

  `hive -f my.sql > out.txt 2> err.txt`

  and submit three files as homework: `my.sql`, `out.txt` and `err.txt`.
- You can also run sql on hive database as follows:

  `hive --database my1 -e 'select count(*) from t1;'`
- Lab 6

Sqoop: introduction

- Sqoop is used to import/export data from/to relational database
- In the process of importing one can execute SQL to downselect or denormalize data
- Can import into HBase, Hive or HDFS
- Can export from HDFS only
- Uses JDBC driver to connect to databases

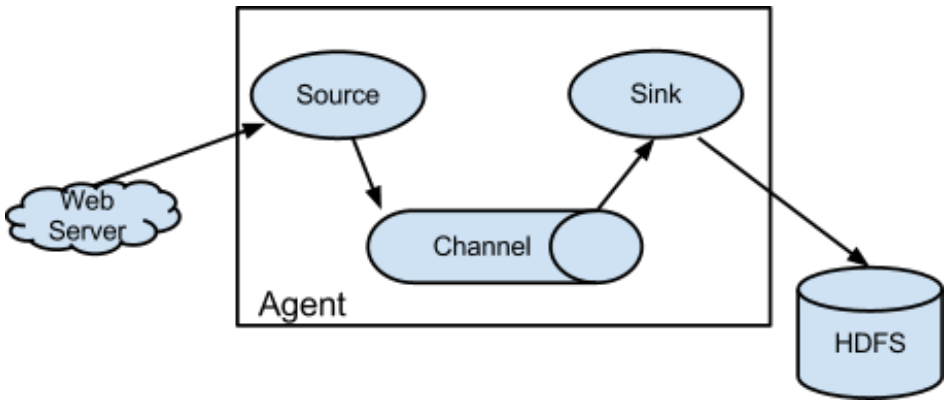
Sqoop: examples

```
$ sqoop import \
--connect jdbc:mysql://localhost/serviceorderdb \
--username root -P \
--table serviceorders -m 1 \
--class-name serviceorders \
--target-dir /usr/biadmin/serviceorders-import \
--bindir .
Enter password:
```

```
$ sqoop import --connect jdbc:mysql://localhost/serviceorderdb \
--username root -P -m 2 \
--query 'SELECT customercontactinfo.customername, customercontactinfo.
contactinfo FROM customercontactinfo JOIN
serviceorders ON customercontactinfo.customernum = serviceorders.customernum
WHERE $CONDITIONS' \
--split-by serviceorders.serviceordernum \
--boundary-query "SELECT min(serviceorders.serviceordernum),
max(serviceorders.serviceordernum) FROM serviceorders" \
--target-dir /usr/biadmin/customers \
--verbose
```

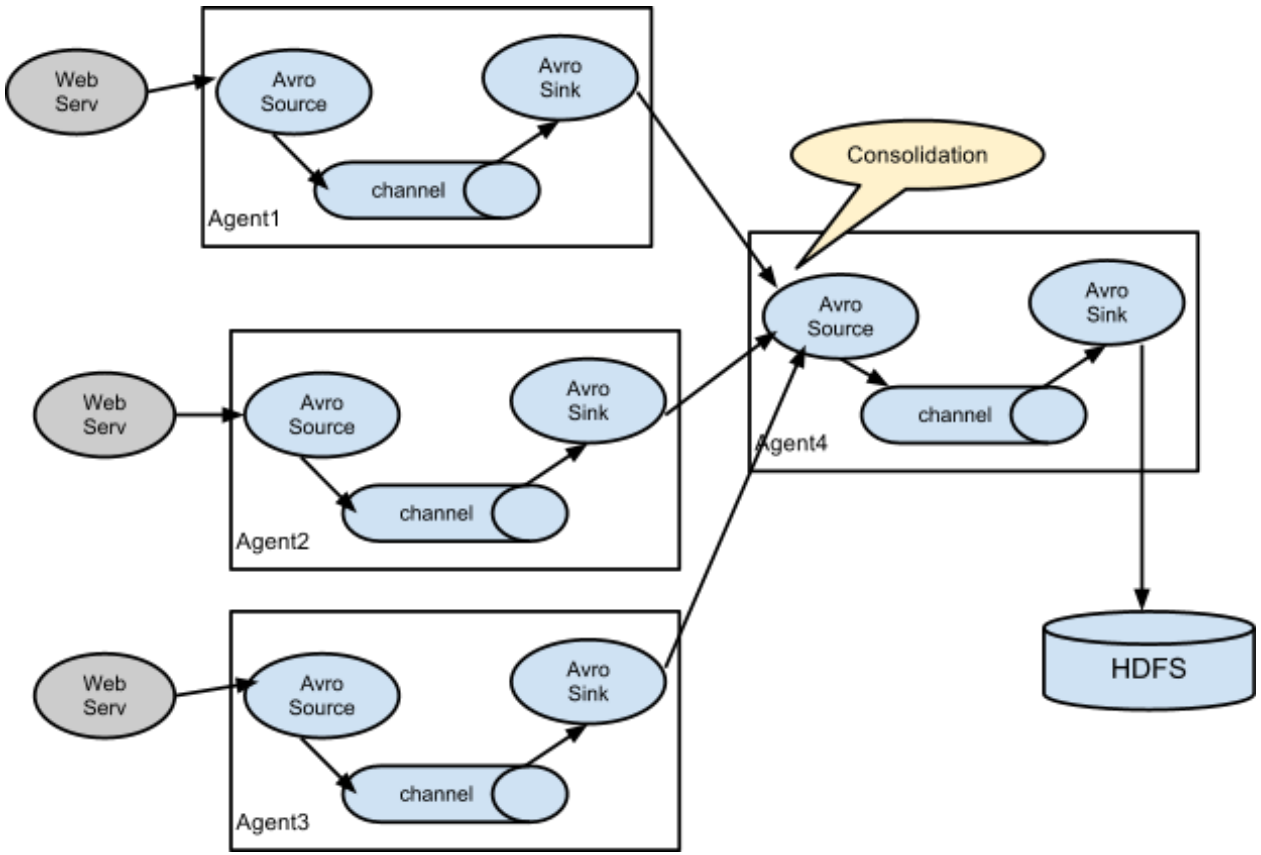
Flume

- Apache Flume is used to propagate data from various sources into Hadoop.
- A typical example is to use Flume to collect logs from a cluster
- Flume agent
  - runs on each node
  - can accept multiple input channels, transform data and send it to multiple output sinks, local or remote;



Flume

- Flume agents can be connected in a tree: data is collected from all the nodes in a cluster and inserted into Hadoop or Solr or file.



Spark

- Apache Spark - fast general-purpose cluster computing system
- API in Scala, Java, Python, R
- Rich set of higher-level tools:
  - Spark SQL
  - MLib - library for machine learning
  - GraphX - library for graph processing
  - Spark Streaming - scalable, high-throughput stream processing of live data streams.
- It can be used interactively or in batch
- It works in Hadoop cluster, in general purpose cluster like midway, on your laptop

Spark: RDD

- Spark's primary abstraction - **Resilient Distributed Dataset (RDD)**
- RDD - collection of records partitioned across the nodes and can be operated on in parallel
- RDD can be created from HDFS files in various supported formats or by transforming other RDDs
- One can ask Spark to cache RDD in memory or disk for fast reuse
- RDDs automatically recover from node failure
- RDD supports two types of operations:
  - **Transformations** - create a new dataset from an existing one. Lazy evaluation - evaluated only when required by action.
  - **Actions** - return a value to the driver program after running a computation on the dataset.

Spark: RDD: transformations

Examples of RDD transformations:

- map(func) - transform each element by a applying a function
- filter(func) - select records satisfying boolean function
- sample(withReplacement, fraction, seed) - Sample a fraction fraction of the data, with or without replacement, using a given random number generator seed.
- union(otherDataset), intersection(otherDataset)
- distinct([numTasks]))
- groupByKey([numTasks])
- sortByKey([ascending], [numTasks])
- pipe(command, [envVars])

Spark: RDD: actions

- 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.
- count()
- take(n) - Return first n elements of the results
- 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. This is usually done for side effects such as updating an accumulator variable (see below) or interacting with external storage systems.

Spark: Shared variables

- Spark's second abstraction - **shared variables**
- By default variables are not shared between tasks
- Two kinds of shared variables supported:
  - **Broadcast variables** - can be used to cache a value in memory on all nodes
  - **Accumulators** - such as counters, sums; one can only "increment" those variables; can be used to store intermediate results of reduce operation

Spark: line counting example

- The examples are given in python
- Usual Python libraries, like numpy, can be used

```
#!/usr/bin/python
from pyspark import SparkContext
user=os.getenv("USER")
filename = "hdfs:///user/%s/test1/input/linux.words"%(user)
sc = SparkContext("yarn-client","SparkExample")

filedata = sc.textFile(filename).cache()

numAs = filedata.filter(lambda s: 'a' in s).count()
numBs = filedata.filter(lambda s: 'b' in s).count()

print "Lines with a: %s, lines with b: %s" % (numAs, numBs)
```

- The above example runs on a text file in HDFS but can as well be connected to files in many other supported formats.
- The code can be submitted to Hadoop with spark-submit command

Spark: using pyspark interpreter

- When using pyspark interpreter, you do not have to import spark modules, this is already done for you
- Otherwise, you are just inside a normal python interpreter

```
$ pyspark
>>> lines = sc.textFile(``wordcount/README'')
>>> lines.map(lambda line: len(line.split())).reduce(lambda a,b: a if (a>b) else b)
>>> pairs = lines.map(lambda s: (s,1))
>>> counts = pairs.reduceByKey(lambda a, b: a+b)

# put an array on the cluster
>>> data = [1, 2, 3, 4, 5]
>>> distData = sc.parallelize(data)

# broadcast variables
>>> broadcastVar = sc.broadcast([1, 2, 3])
>>> broadcastVar.value
    [1, 2, 3]

# accumulators
>>> accum = sc.accumulator(0)
>>> sc.parallelize([1, 2, 3, 4]).foreach(lambda x: accum.add(x))
>>> accum.value
    10
```

- Lab 7

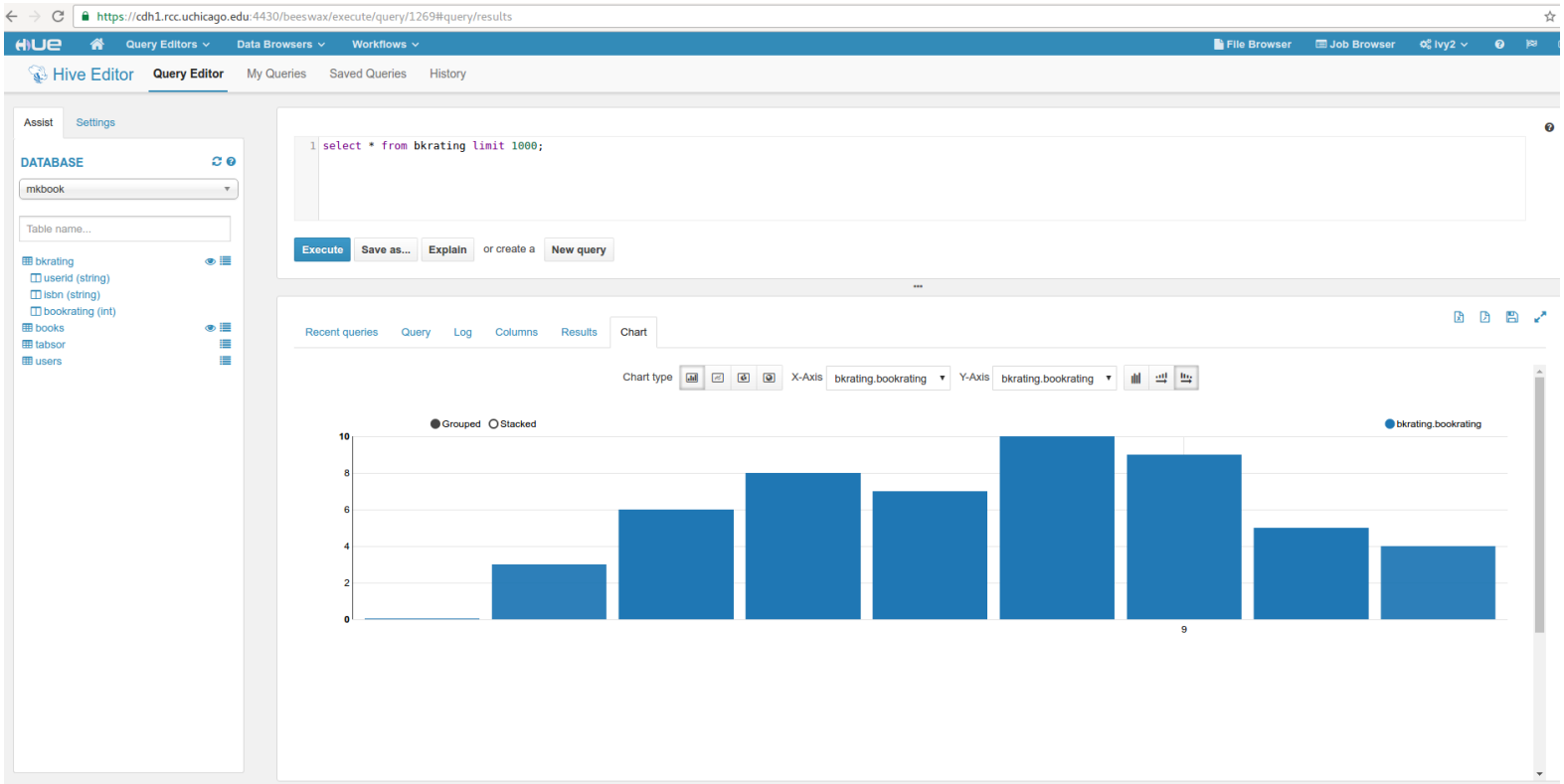
Spark: Running on general purpose cluster

- Unless you rely on HDFS, HBase or some other Hadoop-specific componenet, you can easily use Spark without Hadoop on a general purpose cluster.
- It is installed on midway
- Lab 8

GUI

There are various GUI interfaces to Hadoop, for example:

- JupyterHub - `https://hadoop.rcc.uchicago.edu`
- Hue - `https://hadoop.rcc.uchicago.edu:8888`



References

- Apache Hadoop documentation http://hadoop.apache.org
- Cloudera Hadoop documentation https://www.cloudera.com/documentation.html
- “Hadoop for dummies” by D. deRoos, P. C. Zikopoulos, R. B. Melnyk, B. Brown, R. Coss
- Presentation by Bryon Gill at XSEDE Workshop in Big Data: https://www.psc.edu/hpc-workshop-series/big-data-february-2018

- ”Learning Spark” by H. Karau, A. Konwinski, P. Wendell, M. Zaharia
- ”HBase in action” by N. Dimiduk, A. Khurana
- ”Programming Pig” by A. Gates
- “Hadoop. The Definitive Guide” by Tom White