Skalering til Big Data

13-04-2018

Christian Thomsen

chr@cs.aau.dk

The course

- 3 days
  - Today: MapReduce & Friends
  - The first Spark if time permits
  - Next time: More Spark
- 9 – 16
  - Breaks as needed
  - Lunch 12:00 – 12:45 (in the NOVI8 building!)
- Two teachers
  - First half: Christian Thomsen – chr@cs.aau.dk
  - Second half: Nguyen Ho – ntth@cs.aau.dk
- Exam
  - 14th + 15th June
  - Based on a mini-project (but we can ask about everything from the course)
  - The mini-project can be made in groups

Literature

- Book:
  - M. Zaharia & B. Chambers. *Spark: The Definitive Guide.*
- Online resources/papers

- Other recommendable books:
  - T. White. *Hadoop: The Definitive Guide*.
  - M. Grover et al. Hadoop Application Architectures
  - V. Mayer-Schönberger & K. Cukier. *Big Data*

Agenda

- MapReduce
- Hadoop
- HDFS
- HBase
- Hive
- Pig
- Spark

Big Data

- More data often beats better algorithms
- We are dealing with larger and larger amounts of data
  - From user-generated content on the web, from scientific observations/simulations, from sensors, from …
  - Terabytes, petabytes
  - Often, we have to process data sets that are too large to handle on a single machine
- Big Data:
  - *‘a broad term for data sets so large or complex that traditional data processing applications are inadequate’* [Wikipedia]
  - Volume
  - Variety
  - Velocity
  - (and many more Vs have been proposed)

Scaling

- If the data size grows, we have to use more hardware to handle it

- We can **scale up** and buy a bigger machine with more and better CPUs, more RAM, larger disks

or

- We can **scale out** and buy (many) more machines
  - We can then use servers with commodity hardware. This is significantly more cost effective than specialized servers

- For large data sets, the only possibility is to scale out

Scaling, cont.

- Just reading a large data set sequentially from a disk takes a lot of time

- It is faster to read different parts of the data set from many disks in parallel

- BUT: We should not make the network a bottleneck when transferring data read at one node to another…

Failures in the cluster

- When we have a cluster with many machines, failures ***will*** happen
- Assume 10,000 servers with a mean-time between failures (MTBF) of 1,000 days ➔ 10 failures each day
- Large-scale computations should not become lost or undeterministic in case of a failure
- If a node crashes, another node should take over its work without user-interaction

Programming

- Programming is difficult…
- Programming of distributed systems is very difficult…
  - race conditions, deadlocks, coordination, etc.
  - the programmer spends a lot of time on the “technical details” instead of the real problem
- The program should be made to scale such that
  - given twice the amount of data, it takes twice as long to process it on the same cluster
  - given a twice as big cluster, it takes the same time to process the double amount of data

MapReduce: Origin and purpose

- Introduced by Google in 2004
- Makes distributed computing on clusters easy
- Highly scalable: Handles TBs of data on clusters of 1000s of machines (scales out)
- The user only has to specify two functions
  - An abstraction: The two functions deal with key/value sets.
- The system can then take care of partitioning, scheduling, failures, etc. (all the tedious work)
  - The user can focus on the important computations
- MapReduce is batch processing system. Brute force!
  - To be used on large amounts of data

The following is partly based on the excellent book by Tom White: “Hadoop: The Definitive Guide”, O’Reilly. You have online access to it from the university library (aub.aau.dk).

MapReduce: Origin and purpose, cont.

- Google published a paper in 2004 about the programming model and implementation
  - Their implementation is not publicly available
  - Patented – but Google said they would not to sue you for using the techniques
- Open source MapReduce frameworks are available
  - The most popular is Apache’s Hadoop MapReduce (written in Java)

Programming model

- Takes a set of key/value pairs as input
- Produces another set of key/value pairs as output
- Keys and values can be primitives or complex types

- The user provides two functions: *map* and *reduce*

- ***map***: `(k1, v1) → list(k2, v2)`
  - Takes an input pair and produces a set of intermediate key/value pairs. MapReduce groups all intermediate pairs with the same key and gives them to *reduce*
- ***reduce:*** `(k2, list(v2)) → list(k3, v3)` **(Hadoop)**
  `list(v2)` **(Google)**
  - Takes an intermediate key and the set of all values for that key. Merges the values to form a smaller set (typically empty or with a single value)

Example: WordCount

- The standard example
- The task is to count occurrences of words in a large collection of documents
- This is easy to do in parallel
    - Several mappers can work on different (parts of) documents in parallel
    - For each word they see, they output an intermediate count of 1 for the word
    - When the mappers are done, several reducers can work in parallel
    - A reducer gets all counts for a certain word and ”reduces” the list of counts (i.e., 1s) to a single value

WordCount, cont.

```
map(String key, String value):
  // key: document name; value: doc. contents
  foreach word in value:
      Emit(word, 1)


reduce(String key, Iterator<int> values):
  // key: a word; values: list of counts
  int result = 0;
  foreach v in values:
      result += v;
  Emit(result);
```

Other examples

- Count of URL access
  - Very similar to the previous example
  - The map function processes log files and outputs (URL, 1) for each access to URL
  - The reduce function adds all intermediate 1’s for each URL

- Reverse web-link graph:
  - The map function is given a document’s URL as key, and the content as value. For each referenced target, it outputs (target, key)
  - The reduce functions just outputs the list of URLs that reference a given target

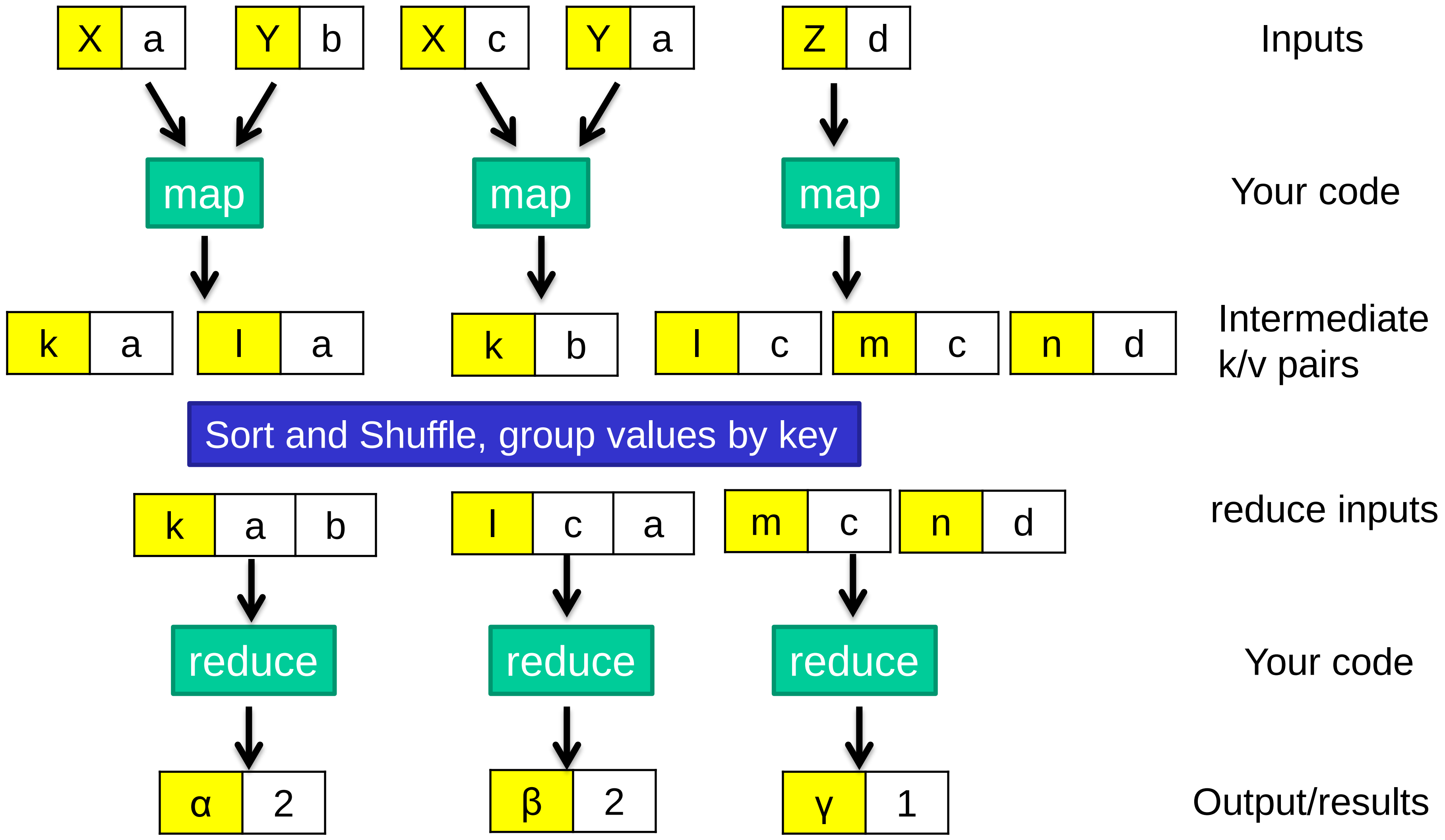
How does it work?

- The map invocations are distributed across many machines such that many map invocations run concurrently
  - Often many thousands of task to assign to hundreds or thousands of nodes
  - The input is automatically partitioned into logical *splits* which can be processed in parallel
  - The input data is stored in a *distributed file system*. The MapReduce runtime systems tries to schedule the map task to the node where its data is located. This is called *data locality*.

- The intermediate key/value pairs are partitioned using a ***deterministic*** partitioning function on the key:
  - By default: hash(key)

- The reduce invocations can then also be distributed across many machines
  - but not until all map tasks have finished

Conceptual view



Refinements

- Partitioning
  - The default partitioning is based on *hash(key)*
  - The user can also make a specialized partitioning function and, e.g., use *hash(ExtractDomain(key))*

- Handling of *stragglers* (tasks that take unusually long time) by starting copies of the slow tasks)
  - This is an optimization – not a reliability means

Refinements - combining

- Combining
  - Consider the WordCount example. A lot of intermediate pairs of the form `(word, 1)` were generated for a given `word` (think about the word ”*the*”, for example)
  - Instead of sending all these over the network, the mappers can combine such pairs (in this case by reusing the reduce code) before the intermediate pairs are copied to the reducer
    How does this affect the reducer?
- Only in certain cases, the reduce function can be used as the combine function. Only when the function is *distributive*:
- $F(k, [v_1, ..., v_n]) =$
  $F(k, [F(k, [v_1, \ldots, v_m]), F(k, [v_{m+1}, \ldots, v_n])]) =$
  $F(k, [F(k, [v_{m+1}, \ldots, v_n]), F(k, [v_1, \ldots, v_m])])$
- In our example, we have r(”the”, [1,1,1,1,1]) = r(”the”, [2, 3])
- Other times, we have to use different functions for combine and reduce

Advantages

- Why is MapReduce better than just writing an efficient distributed program manually?

- It is simple to write a program and run it on hundreds or thousands of machines
  - and the program does not have to be changed if you go from 100 to 1,000 machines
- The user can focus on the real problem

- The framework takes care of the parallelization details
- … and saves the programmer from redoing stuff
  - For example **fault-tolerance** is handled by the framework then used again and again by the user programs
  - This makes the user code simpler, smaller, and cleaner

Performance example

- From Google’s MapReduce paper from 2004
- Performed on a cluster of ~1800 machines (each with 2 Xeon CPUs, 4GB RAM, IDE disks)

- Grep: Scan 10^10 100-byte records (1TB) and look for a (rare) pattern
  - Finishes in ~ 2½ minutes
  - Scans up to 30 GB/sec (see next slide)

- Sort: Sort 10^10 100-byte records
  - The map function extracts a 10-byte sorting key and emits (sortingKey, theText)
  - Finishes in 891 seconds
  - Finishes in 933 seconds if 200 of the workers are killed

Grep performance example

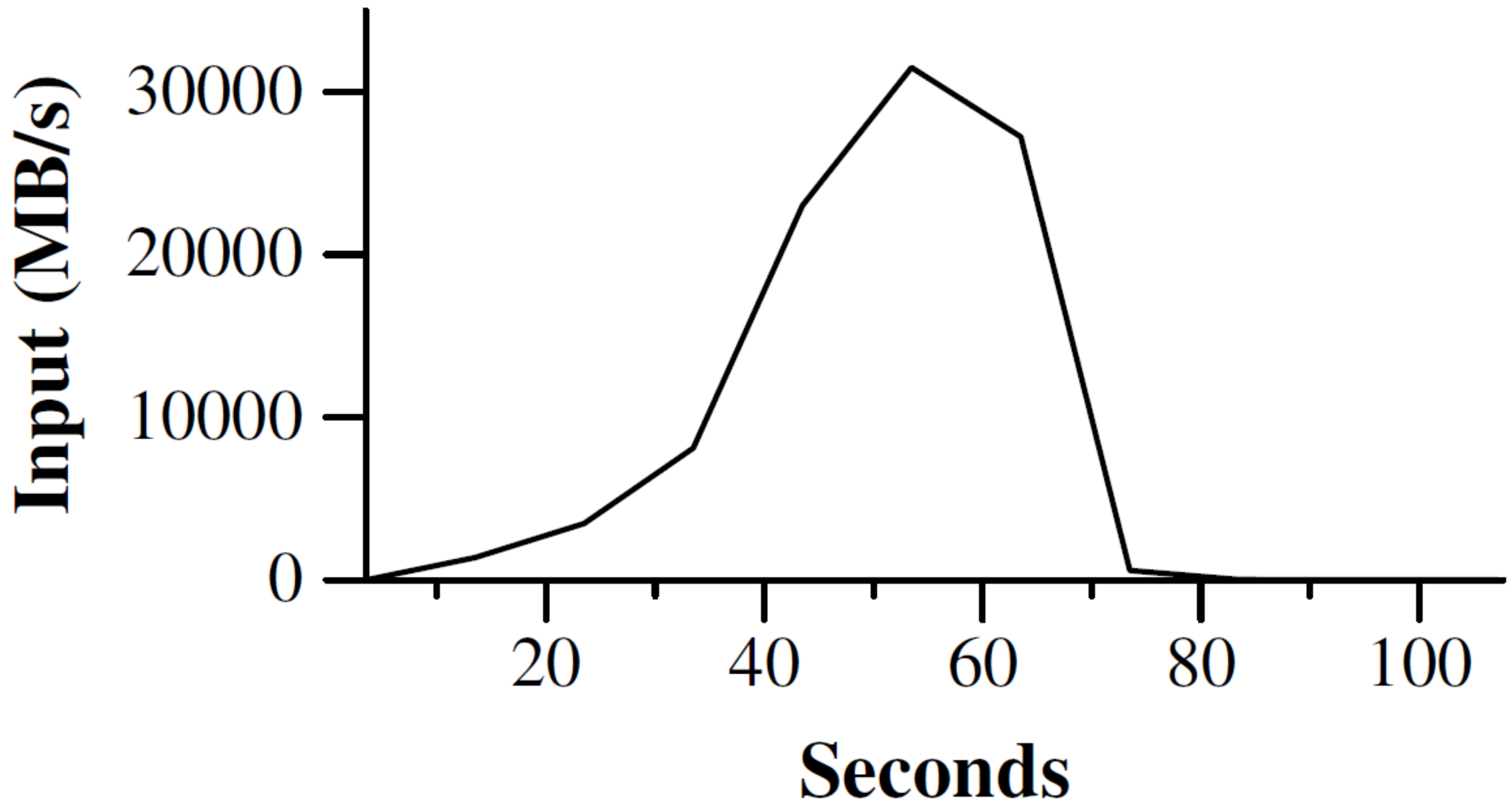


Figure reproduced from J. Dean & S. Ghemawat: ”MapReduce: Simplified Data Processing on Large Clusters”. In OSDI, 2004.

Agenda

- MapReduce
- **Hadoop**
- HDFS
- HBase
- Hive
- Pig
- Spark

YARN = Yet Another Resource Negotiator

- For cluster resource management
- Introduced in Hadoop 2
- MapReduce is just an application; Spark can also run on YARN
- Different kinds of policies supported
- A YARN application has
  - A resource manager (one per cluster)
  - Many node managers (one per node)
  - Many containers to run processes with some amount of resources (memory, CPUs)
- When resources are requested, *locality constraints* can also be given
  - Data local best, rack local 2nd best
  - Instead of meeting a request immediately, YARN can wait a few seconds and see if a better possibility arises

How does it work in Hadoop?

Resource assignment:

- A job is submitted to YARN which creates a container and starts the application master for MapReduce
- The application master requests a container for each map task (i.e., for each input split)
- When 5% of them have finished, the app. master starts to request containers for reduce tasks
- The container runs the task in a dedicated JVM for isolation of failures
- Tasks report their status and progress to the app. master

How does it work in Hadoop?

Inputs and splits:

- **`InputFormat`**s are responsible finding the splits and dividing them into k/v records by means of a **`RecordReader`**
  - A user can implement her own **`InputFormat/RecordReader`**
- A split has a length and a list of locations where it would be good to process the split
- A split is logical and not necessarily related to files
- A split in **`TextInputFormat`** roughly corresponds to an HDFS block, but may include a bit from another HDFS block
  - An HDFS block does not ”respect” line boundaries
  - **`TextInputFormat`** gives offsets as keys and full lines as values
- A minimum size for splits can also be set s.t. a split has data from many HDFS blocks. Typically **not** a good idea. *Why?*

How does it work in Hadoop?

Mapper:

- Each mapper processes one input split
- The map function is invoked for each k/v pair in the logical split
  - Data may have to be fetched from another node
- The map output is first written to a memory buffer (def.100MB)
  - When it is partly (def. 80%) full, partitioning, in-memory sorting, and combining are done and the data is spilled to disk in the background
  - When there is no more input in the split, the spill files are merged into a sorted, partitioned file (if more than 3 spills, combining is done again)
- The output is written to local disk and may be compressed
- Reducers can now fetch their partitions by means of HTTP
  - The data should not be deleted before all reducers are done. *Why?*

How does it work in Hadoop?

Reducer:

- The app. master tells reducers where to fetch (map output) data from (many nodes)
- A reducer starts copying as soon as the data is ready
- It has a configurable no. of threads (def. 5) for copying
- Copies data to an in-mem. buffer and merges & spills to disk
  - The combiner runs again when merging
- All the map outputs are merged while maintaining the sort order
- The reduce function is invoked for each key
- Each reducer produces one output file
- Output written to the final destination, typically HDFS
  - If HDFS, the first replica is written to a node-local disk

Commit

- An **`OutputCommitter`** ensures that jobs and tasks succeed/fail in a clean way

- If tasks produce files, they should write to their *working directory* from which output files will be promoted by **`FileOutputCommitter`** to the final output directory on success

Fault-tolerance in Hadoop

- A task can fail with a runtime exception
  - E.g., **`NullPointerException`**, **`IndexOutOfBoundsException`**
- The task is marked as failed and its container released
- Hanging tasks can also be marked as failed
  - By default if it does not send progress reports within 10 minutes
- A failed task will be retried on another node manager
  - A node manager with many failures will be *blacklisted*
- After a number of failures (def. 4), the job fails
- It is also possible to configure Hadoop to allow a max. percentage of failed tasks

- The application master will also be retried in case of failure

Amount of mappers

- In Hadoop, the number of mappers is decided by the framework based on:
  - A hint from the programmer
  - Number of input files and number of blocks in the distributed file system
    - Do not have too many small files

- "*The right level of parallelism for maps seems to be around 10-100 maps per-node, although it has been set up to 300 maps for very cpu-light map tasks. Task setup takes a while, so it is best if the maps take at least a minute to execute*" [Apache Hadoop Tutorial]

Amount of reducers

- The number of reducers is decided by the programmer
  - The default is 1 – typically not a good choice!
- From Apache Hadoop Tutorial:
- The right number of reduces seems to be 0.95 or 1.75 multiplied by (#nodes * max. #containers per node).
- With 0.95 all of the reduces can launch immediately
- With 1.75 the faster nodes will finish their first round of reduces and launch a second wave
  - Increasing the number increases overhead,
    but increases load balancing and lowers the cost of failures.
- The scaling factors are slightly less than whole numbers to reserve a few reduce slots for speculative tasks and failed tasks.

Best practices for MapReduce

- Add more jobs rather than complexity to jobs
  - The code will be more maintainable and composable
- Mappers and reducers should use as little memory as possible
  - To leave memory for shuffling
  - Do not build huge lists, sets, … of all input data to map or reduce
- Go for few large and splittable input files instead of many small files
  - 1GB can, e.g., be stored as 1 file with 8 blocks of 128MB or 10,000 small files, i.e., 10,000 blocks of 0.1MB

MapReduce criticism

- Even simple jobs have high startup costs due to the MapReduce framework
- MapReduce is general, but low-level
  - For example, joins are cumbersome to implement
- MapReduce programs are often not ”elegant”
- Long development cycle:
  - Write code, compile, pack, submit jobs, fetch results
- Data is read repeatedly – no sharing between different jobs
- To connect two MR jobs, the output of the 1st job is written to disk and then read by the 2nd job
  - (and then you have to take care of deleting the intermediate data)
- We will see alternative solutions later…

Hardware

- MapReduce/Hadoop made for commodity hardware
  - Standard hardware available from competing vendors
  - ”Commodity” != ”low-end” (many failures lead to high costs)
- Typical HW could be similar to this (your mileage may vary)
  - Servers which fit into rack units
  - Two CPUs; 8 – 16 cores; medium speed – the fastest too expensive
  - 64 – 256 GB ECC RAM (ECC = Error Correcting Code)
    - Or even more: The OS can use the memory to cache disk data
  - 12 – 24 SATA disks (2-4TB);
    - The power consumption is proportional to the number of disks; not their capacity
    - RAID ***not*** recommended (except for the HDFS namenode);
      HDFS already gives data redundancy and handles disk failures nicely

- See, e.g., ”Hadoop: The Definitive Guide” or https://docs.hortonworks.com/HDPDocuments/HDP2/HDP-2.6.2/bk_cluster-planning/content/ch_hardware-recommendations_chapter.html

Cloud computing

- Some companies have their own clusters
  - This requires big investments in hardware and maintenance staff
  - Is the cluster used often enough to justify the investment?
- Storage and computations can be moved to the cloud
- Datacenters provide resources “as needed”
  - Access to a variable number of ***virtual*** machines
  - Easy to scale up and down as needed
  - Pay as you go
  - No or very little start-up cost for the user
  - Very powerful instances for a short time may cost nearly the same as less powerful instances for a longer time
- The service provider takes care of maintenance tasks
- For example Amazon EC2 and Microsoft Azure (also preconfigured instances)
- Databricks also provides a cloud solution for Spark

Object storage

- When the virtual machines are given back, their disks are also given back…
- The service providers also provide long-term object storage where you pay for the amount of used storage
- The data locality is gone

- For example AWS S3 and Microsoft Azure Blob Service
  - They can be presented to Hadoop as a file system …
  - … but with some caveats, e.g., eventual consistency, non-atomic directory renames

Agenda

- MapReduce
- Hadoop
- **HDFS**
- HBase
- Hive
- Pig
- Spark

HDFS

- HDFS = Hadoop Distributed File System
- A distributed file system is needed to scale out
  - Computations can be pushed to the data

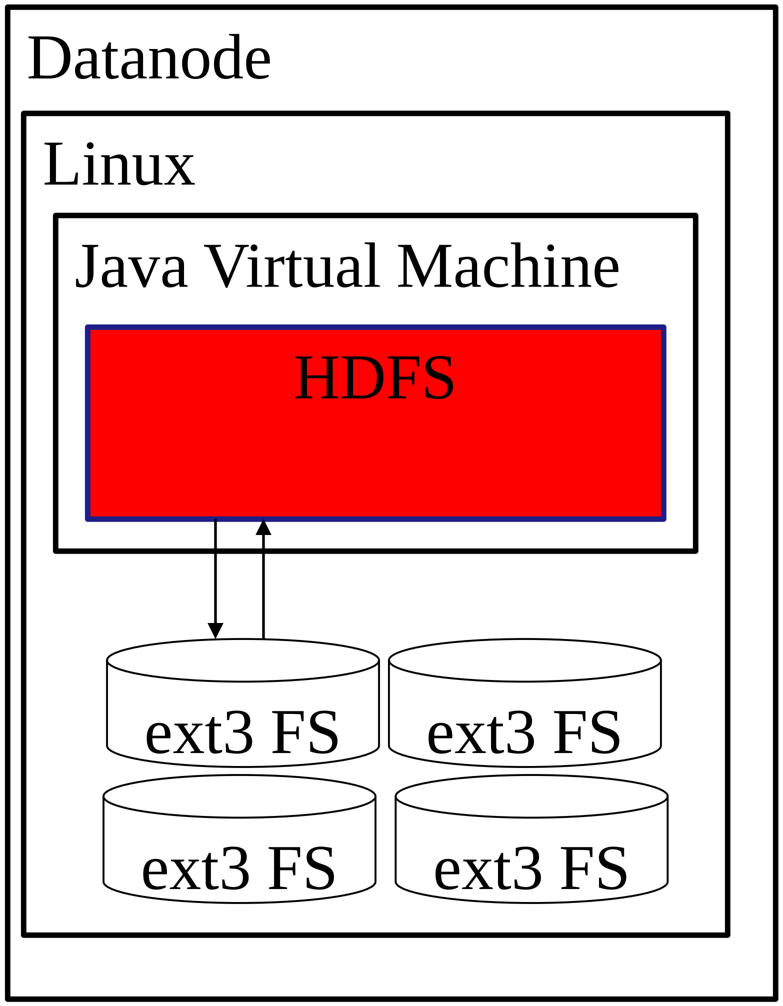
- The distributed file system should
  - Tolerate failures without dataloss (i.e., be made for commodity HW)
  - Support very large data amounts and very large files
  - Support efficient reading of the large files
    - The throughput is much more important than the latency

- HDFS designed for this
- HDFS divides files into blocks that are replicated
  - Default block size = 128 MB; default replication factor = 3

HDFS

- HDFS has two types of nodes:
  - A namenode manages the file system tree and metadata
    - What directories exist, who owns the file, where the blocks are placed
  - A datanode holds blocks that can be read by clients
- HDFS is somewhat inspired by UNIX file systems:
  - A file has an owner, a group, and a mode
  - Commands: ls, mv, cat, …
- But does not support random updates: A file can have a *single* writer which appends at *the end* of the file
- Reading everything or much of it is much more efficient than random reads

HDFS



The HDFS blocks are stored as normal files in the OS-managed file-system

HDFS checksums

- HDFS checksums its data
  - A checksum is added when writing data and checked when reading data
  - Default: 4 bytes checksum for every 512 bytes data
- Each datanode keeps a log of checksum verifications, i.e., when clients have read the data successfully
- A background scanner also does verifications
- When an error is detected, a new replica can be made from one of the good ones

HDFS reading

- The client call **`open`** on **`FileSystem`**
- A wrapped instances of **`DFSInputStream`** is returned
- It contacts the namenode and gets locations for the first blocks
- It then connects to the closest datanode holding the 1st block. After reading the 1st block, it transparently connects to the closest datanode holding the 2nd block etc.
- If there is an exception, it connects to another datanode holding the block.
  - Checksums also used to verify that the data was read correctly
- This scales well
  - Clients read data directly from datanodes
  - The namenode is only involved in telling where blocks are present (and it keeps this information in memory)

HDFS writing

- The namenode is contacted to create the file
- A wrapped instance of **`DFSOutputStream`** is returned
- The client writes data to this object which asks the namenode to create new blocks as needed
- The first datanode forwards data to the next, etc.
- Acknowledgements going back the other way
- The required minimum number of successful block writes can be configured
  - Default: 1 (in which case asynchronous replication will ”do the rest”)
- Default replication strategy:
  - 1st replica on the same node as the client;
  - 2nd replica on a node in another rack;
  - 3rd replica on another node in the same rack as the 2nd

Coherency

- Data in the latest block can be lost in case of failures
- Applications can call methods to achieve guarantees

- When **`hflush`** returns succesfully all written data has reached all relevant datanodes and can be seen by readers

- **hsync** guarantees the same and in addition that the datanodes have written the data to disk

The CAP Theorem

It is impossible for a distributed computer system to simultaneously provide all three of the following guarantees

- Consistency
  - Every read receives the most recent write or an error
- Availability
  - Every request receives a (non-error) response – without guarantee that it contains the most recent write
- Partition tolerance
  - The system continues to operate despite an arbitrary number of messages being dropped (or delayed) by the network between nodes

[Wikipedia]

- Sometimes described as
  “Consistency, Availability, or Partitioning? Pick two!”

HDFS and the CAP Theorem

- But we cannot really choose that partitioning won’t happen…
- So we have to choose between consistency or availability

- HDFS offers consistency
- If three nodes are down (or in other partitions), a block may be unavailable
- But often, (some of) the three datanodes with the block will be accessible such that availability is provided

- Recommended reading: Eric Brewer: *”CAP twelve years later: How the ”rules” have changed”*. Computer 45(2):23—29. Available from http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=6133253

Agenda

- MapReduce
- Hadoop
- HDFS
- **HBase**
- Hive
- Pig
- Spark

HBase motivation

- There is often a need to work with very large data volumes
- HDFS is for bulk processing and does not support efficient read/write random access
- What if we only need to find or update info about a certain user/the contents of a specific web page/…?
- An RDBMS is good at this – but scalability is a problem for very large data sets.
- ”Hadoop: The Definitive Guide” tells *how the typical RDBMS scaling story runs*:
  - Get an RDBMS instance with a nice schema
  - add memcached
  - buy a very powerful server with fast disks
  - denormalize data to avoid joins
  - stop server-side computations
  - prematerialize most complex queries
  - Drop indexes and triggers

Bigtable and HBase

- Google published a paper about their **Bigtable** in 2006
    - F. Chang et al. ”Bigtable: A Distributed Storage System for Structured Data”. In OSDI, 2006
- ”a distributed storage system … designed to scale to … petabytes of data across thousands of commodity servers”
- ”A Bigtable is sparse, distributed, persistent multi-dimensional sorted map
- The map is indexed by a row key, column key, and a timestamp”
- Built to scale linearly
- Not an RDBMS (no SQL, no FKs, no joins, …)
- Apache HBase is an open-source implementation in Java

The following is partly based on Tom White: “Hadoop: The Definitive Guide”, O’Reilly and M. Grover et al: “Hadoop Application Architectures”, O’Reilly

HBase data model

- Data is organized into named **tables** (but they are not like relational tables!)
- Tables have **rows** and **columns** and where they intersect, there are *versioned* **cells**
  - By default, the insertion time is used as version number, but it can also be specified explicitly
  - The number of cell versions to keep can be configured
- A row has a *single and unique* **row key**. Rows are physically stored in order based on this key
- Both row keys and cells are (uninterpreted) arrays of bytes

HBase column families

- Columns belong to **column families**
- Naming of a column: **`columnfamily:qualifier`**
- Column families are defined up-front
  - One or more column families per table
- All rows have the same column families
- Rows can have different columns which are specified as needed
- All columns of a column family are physically stored together
  - Put columns with similar access patterns and sizes in the same family

HBase table example

	Column family podcast	Column family info			
<row key>	**podcast:mp3**	**info:host**	**info:title**	**info:length**	**info:comm ents**
0000a	<bytes>	A And	Rap rap	1:43	
0000b	<bytes>	Pluto	Vuf vuf	0:10	OK
0000c	<bytes>; <bytes>	Pinoccio	Noget	0:10; 0:12	
…	…	…	…	…	…

Each cell version has a timestamp (not shown here)

HBase operations

- The supported operations are similar to the operations, which can be performed on a hash table. You can thus:
- *Put* a row into a table
- *Get* a row based on the key value
- *Scan* a from some key value to another (or scan all rows)
  - There are both forward and backward scans
- *Increment* the value for a given key value and column
- *Delete* rows

- Operations on a single row are atomic

- You ”talk” to HBase via its APIs or its own shell

How HBase stores data

- Tables are partitioned into **regions**
- First there is only one region, but when a region gets bigger than a threshold, it is split into two
  - (You can also specify an amount of regions at creation time)
- A region is denoted by its table and its key range
- Regions are distributed to **regionservers**
- A special catalog table keeps track of the regions
- It is then possible to lookup where the region for key *k* is
  - (and clients cache this information)
- HBase writes its data out to files – typically in HDFS
  - HBase aims for locality when assigning regions to regionservers

Adding data

- When data is inserted into a region, the regionserver writes to a log and stores the new data in a cache in memory
- When the cache reaches a threshold size, it is flushed to a file
- Only the present data – not ”missing” data ➔ good for sparse data
- When reading data, the cache is consulted first and then flush files from newest to oldest
- When data is deleted, a marker in inserted in the flush

- Compactions merge files
- When a major compaction is done, deleted and expired data is not written to the new file

Agenda

- MapReduce
- Hadoop
- HDFS
- HBase
- **Hive**
- Pig
- Spark

Hive – data warehousing on MapReduce

- Hive is a data warehousing solution for MapReduce created by Facebook
  - The goal was to bring the well-known concepts (such as tables) to Hadoop and enable use of an SQL-like query language
- Before Hive, Facebook used a commercial RDBMS
  - But the data amounts grew from 15TB to 700TB in three years
  - Data processing took too long time
  - MapReduce was a compelling proposition for Facebook
- MapReduce requires the user to write map and reduce functions
  - You need to take care of some low-level details in Java if you use Hadoop – this can be challenging for many users

WordCount – the actual code for Hadoop

```
package org.myorg;

import java.io.IOException;
import java.util.*;

import org.apache.hadoop.fs.Path;
import org.apache.hadoop.conf.*;
import org.apache.hadoop.io.*;
import org.apache.hadoop.mapreduce.*;
import org.apache.hadoop.mapreduce.lib.input.FileInputFormat;
import org.apache.hadoop.mapreduce.lib.input.TextInputFormat;
import org.apache.hadoop.mapreduce.lib.output.FileOutputFormat;
import org.apache.hadoop.mapreduce.lib.output.TextOutputFormat;

public class WordCount {

 public static class Map extends Mapper<LongWritable, Text, Text, IntWritable> {
    private final static IntWritable one = new IntWritable(1);
    private Text word = new Text();

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

 public static class Reduce extends Reducer<Text, IntWritable, Text, IntWritable> {

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

 public static void main(String[] args) throws Exception {
    Configuration conf = new Configuration();

        Job job = new Job(conf, "wordcount");

    job.setOutputKeyClass(Text.class);
    job.setOutputValueClass(IntWritable.class);

    job.setMapperClass(Map.class);
    job.setReducerClass(Reduce.class);

    job.setInputFormatClass(TextInputFormat.class);
    job.setOutputFormatClass(TextOutputFormat.class);

    FileInputFormat.addInputPath(job, new Path(args[0]));
    FileOutputFormat.setOutputPath(job, new Path(args[1]));

    job.waitForCompletion(true);
 }

}
```

Code from apache.org

Hive – data warehousing on MapReduce

- With Hive, the user writes SQL-like queries and Hive translates them into MapReduce* jobs
  - *) MapReduce is deprecated, but still supported in Hive; Hive is moving to Tez and Spark

- Simple example: Equi-join
  - Mappers can read from the relevant tables and do filtering
  - Reducers can combine the selected data from the different tables
- A query will often lead to several MapReduce# jobs
  - #) But Tez and Spark can avoid this
- The MapReduce jobs are not generated on-demand, but are standard building-blocks which get configured and ordered by XML files created by Hive

Data model

- Hive structures data into tables with rows consisting of a specified number of columns associated with types

- Hive supports primitive types (ints, floats, strings) and complex types (maps, lists, structs)

- It is possible to add user-defined types and (de-)serializers
  - Data stored in a legacy format can then be read by Hive without an initial import/transform step

Data storage

- A table is stored in a directory in the HDFS
  - **`LOAD DATA`** simply moves a file into the directory (w/o parsing); during query processing all files for a tables are read
- **Partitions** are stored as subdirectories
  - The values of the partitioning key(s) are stored in the directory names
- **`/user/hive/warehouse/MyTable/Year=2012`**
- Data can also be **bucketed** (with or without partitioning)
  - Similar to hash partitions in RDBMSs
  - Gives files of managable sizes
  - When two data sets are bucketed on the same key and have *b* and *nb* buckets ($n, b \in \mathbb{Z}^+$), respectively, joins can be done efficiently

Data storage

- It is also possible to use **external** tables, i.e., data that is ”owned” by another application
  - **`CREATE EXTERNAL TABLE exttbl(a int, b int) LOCATION ‘/some/path’`**
- It is also possible to explicitly define delimiters and use other (de-)serializers
  - Existing log files can then appear as logical tables in Hive
- Column-oriented storage is also supported

Metastore

- The metastore is like a system catalog and stores metadata
- There should be low latency
- → the metastore is NOT stored in HDFS
- Instead an RDBMS is used
  - The default is the Java-based DerbyDB, but any JDBC-compliant RDBMS can be used

Querying

- Hive supports SQL-like queries written in HiveQL
  - including subqueries, joins, group bys, and aggregations
- The SELECT and FROM clause may be swapped in HiveQL
  - Useful for “multi table inserts” where data is written to many tables, but where the input data is only scanned once:
  - **FROM VeryLargeDataSet**

```
 INSERT OVERWRITE FirstTarget
 SELECT att1, MAX(att2)
 WHERE rating > 100
 GROUP BY att1

 INSERT OVERWRITE SecondTarget
 SELECT att1, MIN(att2)
 WHERE RATING < 10
 GROUP BY att1;
```

Querying

- The user can also use MapReduce programs in queries

- WordCount in Hive:

```
FROM (
    MAP doctext USING ‘mymapper’ AS (word, count)
    FROM docs
    CLUSTER BY word
) a
REDUCE word, cnt USING ‘myreducer’;
```

Word Count again

- But it can be simpler and more SQL-like as in
- `CREATE TABLE docs(line STRING);`
- `LOAD DATA INPATH ‘docs’ OVERWRITE INTO TABLE docs`

-
  ```
  CREATE TABLE word_counts AS
  SELECT word, count(1) AS count FROM
  (SELECT  explode(split(line, ‘\s’)) AS word FROM docs)w
  GROUP BY word
  ORDER BY word;
  ```

- Example taken from Capriolo, Wampler, and Rutherglen: ”Programming Hive”, O’Reilly

Recent Hive development

- Record-level INSERTs, UPDATEs, or DELETEs also supported now
- Transactional support, but all operations are auto-committed
- Indexing
- Tez and Spark
- Live Long and Process (LLAP) functionality
  - Caching, pre-fetching, some query processing done in long-living daemons
  - ”Small/short queries are largely processed by this daemon directly, while any heavy lifting will be performed in standard YARN containers.”
  - https://cwiki.apache.org/confluence/display/Hive/LLAP

Agenda

- MapReduce
- Hadoop
- HDFS
- HBase
- Hive
- **Pig**
- Spark

Pig

- Yahoo! proposed *Pig* where the user specifies sequences of transformations in the language *Pig Latin*
- Pig raises the abstraction compared to MapReduce and provides a high-level procedural language (”*the sweet spot between MapReduce and SQL*”)
- Example:
  - **`A = LOAD ‘users’ AS (name:chararray, age:int);`**
  - **`B = LOAD ‘scores’ AS (username:chararry, pts:int);`**
  - **`C = JOIN A BY name, B BY username;`**
  - **`D = FILTER C BY age > 18;`**
  - **`E = FOREACH D GENERATE name, age, pts;`**
  - **`STORE E INTO ‘results’`**;

The following is also partly based on Tom White: “Hadoop: The Definitive Guide”, O’Reilly.

The Pig philosophy

- **Pigs Eat Anything**
  - Pig can operate on data whether it has metadata or not. It can operate on data that is relational, nested, or unstructured. And it can easily be extended to operate on data beyond files, including key/value stores, databases, etc.
- **Pigs Live Anywhere**
  - Pig is intended to be a language for parallel data processing. It is not tied to one particular parallel framework. It has been implemented first on Hadoop, but we do not intend that to be only on Hadoop.
- **Pigs Are Domestic Animals**
  - Pig is designed to be easily controlled and modified by its users. […]
- **Pigs Fly**
  - Pig processes data quickly. We want to consistently improve performance, and not implement features in ways that weigh pig down so it can't fly.

[http://pig.apache.org/philosophy.html]

Pig features

- Designed for ad-hoc data analysis of very large data sets
  - Batch-processing/scans, not random lookups → no indexes
  - Read-only → no transactional concerns
- Ships with powerful functionality: join, group, order, filter, …
- Extensible and customizable via UDFs
  - UDFs also tend to be more reusable than MR programs
- Has interactive shell (”Grunt”) for running Pig commands

Pig features cont.

- Flexible, nested data model
- Can operate over plain input files without a schema
  - Pig does not take over control of the data – co-exisistence
- Only parallelizable operations
- Pig Latin programs are translated into Hadoop MapReduce jobs
  - (or Tez or Spark jobs)

Pig data model

- Atoms
  - int, long, float, double
  - chararray
  - bytearray
- Tuples
  - Sequence of fields: (20, ‘test’, (‘hello’, 1))
- Bags
  - Unordered collection of tuples: {(1,2), (3), (3), (‘alfa’, ‘beta’, 9)}
- Maps
  - Set of key/value pairs: [‘alfa’#1, ‘beta’#2, ’gamma’#{(‘hi’), (3)}]

Schemas

- A relation is what you get from LOAD or the other relational operators (FILTER, FOREACH .. GENERATE, JOIN, …)
- A relation *may* have a schema with names and types, but is not required to (unlike SQL databases)
  - **`f1 = LOAD ‘somefile’ AS (name:chararray, age:int);`**
  - **`f2 = LOAD ‘anotherfile’;`**
  - If no name is given, fields are referenced by $0, $1, …
  - If no type is given, bytearray is assumed
  - **`DESCRIBE f1;`** shows the schema for f1
- If a value cannot be cast to the expected type, null is used
  - No exception thrown – large datasets are likely to have bad data.
- Often Pig can guess the schema of a new relation

Pig Latin

- A Pig Latin program consists of a sequence of steps, each carrying out a single transformation
  - The operations may be executed in a different order
- ILLUSTRATE can help you to understand your program
  - **`ILLUSTRATE x;`**
  - Can generate example data to show interesting cases

- As Pig sees statements, a *logical* plan is built
  - But data is not loaded, transformed, … immediately
- The execution starts when data must be output:
  - **`DUMP`** and (sometimes) **`STORE`**
- A *physical* plan is then prepared and executed
  - **`EXPLAIN`** shows plans for relations

Multiquery execution

- STORE can sometimes also wait:
  - `A = LOAD ‘inputfile’;`
  - `B = FILTER A BY $1 == ‘test’;`
  - `C = FILTER A BY $1 != ‘test’;`
  - `STORE B INTO ‘testdata’;`
  - `STORE C INTO ‘nottestdata’;`
- One scan of A is enough to write both B and C to files

Pig Latin cont.

- Pig Latin has many built-in functions as well

- Aggregation/SQL-like: COUNT, COUNT_STAR, AVG, MAX, MIN, SUM
- Strings: LOWER, UPPER, TRIM, ENDSWITH, …
- Math: COS, SIN, LOG, RANDOM, …
- DateTime: GetYear, ToUnixTime, HoursBetween, …

- And it is easy to add your own UDF

- For a complete list of built-in functions, see http://pig.apache.org/docs/r0.17.0/func.html

WordCount in Pig

```
A = LOAD ‘file‘ AS (line:chararray);
B = FOREACH A GENERATE
                  flatten(TOKENIZE((line)) AS word;
C = GROUP B BY word;
D = FOREACH C GENERATE group, COUNT(B);
```

- From http://pig.apache.org/docs/r0.17.0/basic.html:
  “*The result of a GROUP operation is a relation that includes one tuple per group. This tuple contains two fields:*
  - *The first field is named "group" [...] is the same type as the group key.*
  - *The second field takes the name of the original relation and is type bag.*”
  - We can then,e.g., do COUNT(X), MAX(X.$0), SUM(X.y),… on the 2nd field X

Other operators in Pig

- `X = ORDER Y BY $0 DESC, $2;  --ASC is def.`
- `D = UNION A, B, C;  --schemas can vary!`
- `A = LIMIT B 40;`

-
  ```
  SPLIT A INTO B IF $0<100, C IF $0==100,
      D OTHERWISE;
  ```

Parallelism in Pig

- One mapper per HDFS block
- By default, one reducer per GB input data up to a limit (which defaults to 999, but can be overridden)
- PARALLEL can also be used to explicitly define the number of reducers:
  - `j = JOIN A BY $0, B BY $3 PARALLEL 100;`

Agenda

- MapReduce
- Hadoop
- HDFS
- HBase
- Hive
- Pig
- **Spark**

Spark

- Many other cluster computing frameworks for batch processing have been proposed
- A very good proposal is ***Spark***
  - Originally from UC Berkeley
  - Now the most active Apache project

- Spark can keep datasets in memory between jobs
  - Avoids the expensive I/O that MapReduce has to do
  - In particular good for iterative algorithms and interactive analysis
- Spark **does not use** MapReduce, but is closely integrated with Hadoop
  - Can run on YARN (can also run in other modes, e.g., stand-alone)
  - Can work with HDFS; does not have its own DFS

Spark

- Easy interactive use with REPL (read, eval, print, loop)
- Spark offers a rich set of APIs (Scala, Java, and Python)
- Considered easy to use

- WordCount in Spark (using the Python API):

```
lines = spark.textFile("hdfs://...")

res = lines.flatMap(lambda line: line.split())
    .map(lambda word: (word, 1))
    .reduceByKey(lambda a, b: a+b)
```

[Example from spark.apache.org]

Resilient Distributed Datasets

- Spark uses *Resilient Distributed Datasets (RDDs)* which are **immutable** partitioned record collections
  - **Warning:** RDDs are very low-level; Spark now also has ”Structured APIs” as we will see next time

- An RDD can be created from
  - a collection of objects (the collection is *”parallelized”*)
  - data in stable storage (HDFS files etc.)
  - coarse-grained transformations on existing RDD(s) (e.g., map, filter, and join)

- RDDs represented by objects with methods
- RDDs can be recomputed as needed or kept in memory
  - Can be spilled to disk if there is not enough RAM

RDD partitions

- An RDD has *partitions* – atomic pieces of the dataset
- The partitions can be stored on different nodes

- An RDD for an HDFS file has one partition per HDFS block and *prefers* to have this partition on the same node as the HDFS block

RDDs in Spark

- RDDs are by default not replicated across nodes
  - The user can, however, choose to do it
- It is enough to write RDD data to the local memory
  - Much faster than writing to the network or the disks
- An RDD’s lineage is logged
  - It is then easy to recreate (partitions of) RDDs
    - Holds both for RDDs not stored and RDDs lost due to node failure
    - Partitions can also be recreated in parallel
- RDDs do not need to be materialized at all times
- No replication of RDDs ➔ space for more RDDs in the memory

RDDs

- Two categories of operations on RDDs:
  - **Transformations** generate new RDDs from existing (e.g., map, filter)
  - **Actions** trigger a computation on RDDs and do something with the result (e.g., count, save)

- Transformations are lazy
  - RDDs are not computed until an action is used (i.e., a result is requested or data is exported to storage)
  - This allows Spark to pipeline transformations

Using Spark

- Spark can read from any Hadoop MapReduce input source
- The user writes a *driver* program that launch *workers*.
  - It is also possible to use Spark interactively (here with Scala)
- **`lines = spark.textFile(”hdfs://…”)`**
- **`errors = lines.filter(_.startsWith(”Error”))`**
- **`errors.persist() // errors is now in mem.`**
- **`errors.count()`**
- **`errors.filter(_.contains(”MySQL”))`**
  `.map(_.split(‘\t’)(3)).collect()`

(example borrowed from [Zaharia et al. 2012])

- Like MapReduce, Spark will schedule tasks on the nodes with the relevant data. Memory prioritized higher than disk.

Persistence / caching

- An RDD may be (re-)computed for each action on it…
- RDDs can be persisted
- Call persist() to indicate that an RDD should be persisted
- By default, Spark keeps persisted RDDs in memory (and spills to disk if out of RAM)
- The RDD will then be much faster to use next time
  - Very important for iterative algorithms and interactive use

- An optional argument to persist can set the storage level
- General hint: Use memory!
  - Only spill to disk if recomputations are expensive

Persistence – storage levels

- **MEMORY_ONLY**:
  - Java objects in the JVM. If out of mem., some partitions will not be cached and will be recomputed as needed
- **MEMORY_AND_DISK**:
  - Java objects in the JVM. If out of mem, store some partitions on disk and read them from there when needed
- **MEMORY_ONLY_SER**:
  - Serialized Java objects in byte arrrays. If out of mem., some partitions will not be cached and will be recomputed as needed
- **MEMORY_AND_DISK_SER**:
  - As above – but uses disk instead of recomputing partitions
- **DISK_ONLY**
- ***_2**:
  - Replicates each partition on two cluster nodes → fast fault recov.

Partitioning

- Typically 2-4 partitions for each CPU
- Spark tries to set the number automatically
  - The user can also set it – e.g., **`parallize(data, 10)`**, **`repartition(20)`**

- The user can also define the partitioner function
- If two RDDs are partitioned by the same partitioner, joins are efficient

Wide & narrow dependencies

- For some operations, each parent partition is used only by one child partition
  - For example **map** and **union**
  - If two RDDs are partioned in the same way, **join** is another example
  - We talk about *narrow* dependencies

- For other operations, each parent partition is used by many child partitions
  - For example **groupByKey** (values for a given key may be in any partition) or **repartition**
  - We talk about *wide* dependencies

Shuffling

- For narrow dependencies, pipelined execution on one node can compute all needed parent partitions
  - No network traffic
- For wide dependencies, data needs to be *shuffled* between nodes
  - All values for a given key should go to the same node
- For shuffling, Spark uses *map tasks* to organize data and *reduce tasks* to aggregate it
  - Names from MapReduce – not related to Spark’s map and reduce
- Spark writes to a memory buffer, sorts on target partition, and then writes out to a file
  - These files are kept as long as the resulting RDDs exist

Transformations and actions

- Spark offers many transformations, e.g.:
  - filter(p)
  - map(f) (1:1 – not like map in MapReduce)
  - flatMap(f) (0 or more outputs for one input value – like map in MR)
  - union(rdd) and intersection(rdd)
  - join(rdd) joining RDD<K,V> and RDD<K, W>
  - reduceByKey(f) on an RDD<K,V> returns RDD<K, Iterable<V>>,
    f must be associative and is used sim. to a ”combiner” on the map side
- … and actions, e.g.:
  - reduce(f) for associative and commutative f: $A \times B \rightarrow C$
  - count()
  - collect()
  - take(n)

- See http://spark.apache.org/docs/latest/rdd-programming-guide.html#transformations and http://spark.apache.org/docs/latest/rdd-programming-guide.html#actions

Using Spark

- Many of the other cluster programming models can be efficiently implemented with Spark!
  - Incl. MapReduce, HaLoop, DryadLINQ, Pregel
  - Build a graph with MapReduce, then proces it with Pregel in the same program

- Spark offers good performance – up to 20X-40X faster than Hadoop MapReduce (see the paper for details)
  - Best performance gains when the data fits in the cluster’s memory
  - Degrades gracefully when there is not enough memory and partitions become stored on disk. Similar to other systems then.

Performance example

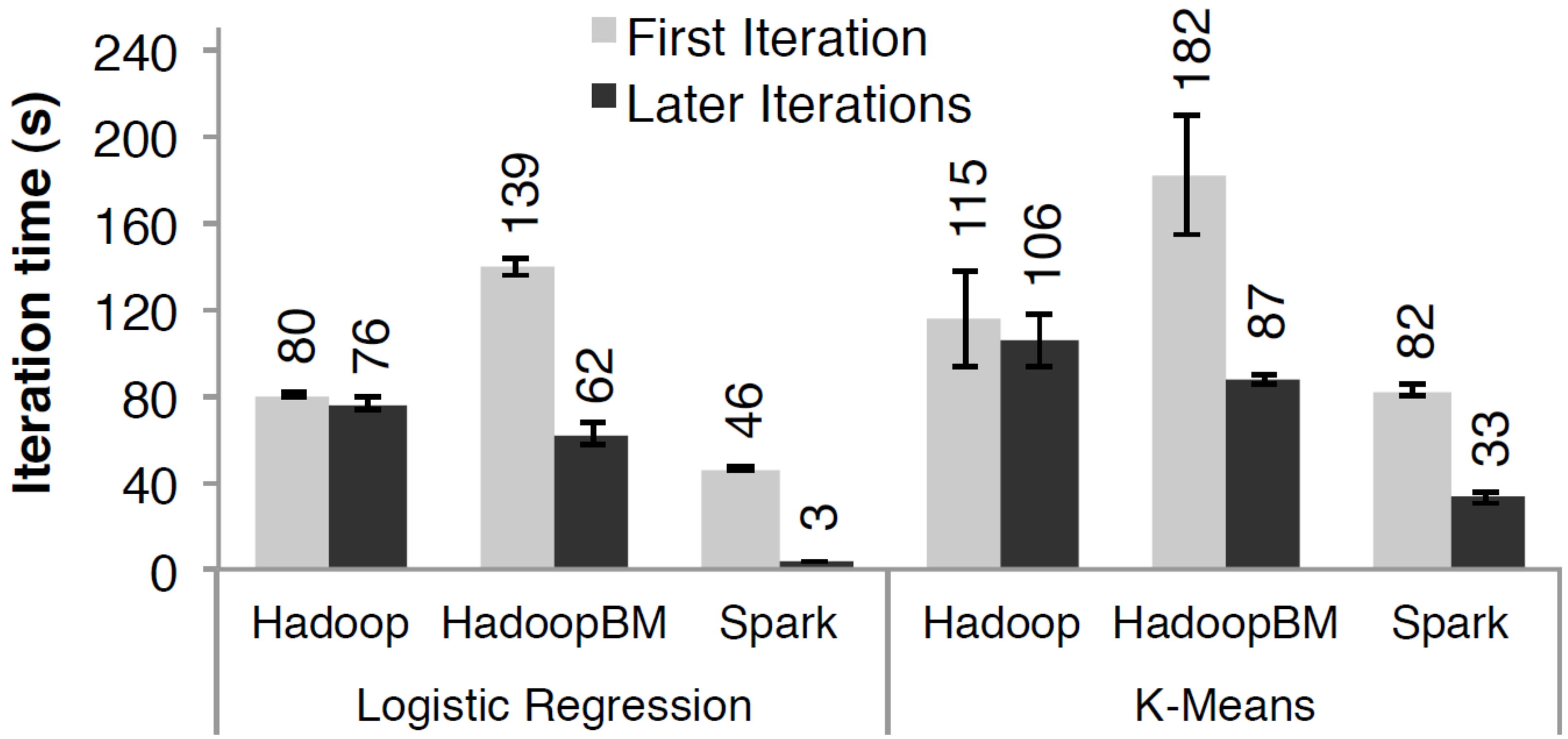


Figure from Zaharia et al., 2012

Shark

- *Shark* was Hive ported to Spark
  - Shark was compatible with Hive
  - Fault-tolerant SQL support on thousands of nodes
- Shark was up to 100X faster than Hive
  (when the data fits in the cluster’s memory)
  - Ran on Spark using RDDs
  - Exploited in-memory columnar storage and compression
  - Supported ”map pruning” based on gathered statistics s.t. irrelevant data can be disregarded
- Shark is discontinued and was followed by Hive on Spark and Spark SQL

- More about Spark SQL next time