MapReduce for Big Data
Distributed Big Data

- Google MapReduce
  - Motivation and History
  - Google File System (GFS)
  - MapReduce:
    - Schema, Example, MapReduce Framework
- Apache Hadoop
  - Hadoop Modules and Related Projects
  - Hadoop Distributed File System (HDFS)
  - Hadoop MapReduce
- Apache Spark
Big Data

- Big Data analytics (or data mining)
  - need to process large data volumes quickly
  - want to use computing cluster instead of a super-computer

- Communication (sending data) between compute nodes is expensive

⇒ model of “move computing to data”
Big Data Processing

Computing cluster architecture:
1000s of computing nodes
10000s Gb of memory
10000s Tb of data storage

- HW failures are rather rule than exception, thus
  1. Files must be stored redundantly
     - over different racks to overcome also rack failures
  2. Computations must be divided into independent tasks
     - that can be restarted in case of a failure
MapReduce: Origins

- In 2003, Google had the following problem:
  1. How to rank tens of billions of webpages by their “importance” (PageRank) in a “reasonable” amount of time?
  2. How to compute these rankings efficiently when the data is scattered across thousands of computers?

- Additional factors:
  1. Individual data files can be enormous (terabyte or more)
  2. The files were rarely updated
     - the computations were read-heavy, but not very write-heavy
     - If writes occurred, they were appended at the end of the file
Google's Solution

- Google found the following solutions:
  - Google File System (GFS)
    - A distributed file system
  - MapReduce
    - A programming model for distributed data processing
Google File System (GFS)

- **Files** are divided into **chunks** (typically 64 MB)
  - The chunks are **replicated** at three different machines
  - The chunk size and replication factor are **tunable**

- **One machine is a master**, the other **chunkservers**
  - The **master** keeps track of all file **metadata**
    - mappings from files to chunks and locations of the chunks
  - To find a file chunk, **client** queries the **master**, and then contacts the relevant **chunkservers**
  - The master’s metadata files are also replicated
MapReduce

- MapReduce is a programming model sitting on the top of a Distributed File System
  - Originally: no data model – data is stored directly in files

- A distributed computational task has three phases:
  1. The map phase: data transformation
  2. The grouping phase
     - done automatically by the MapReduce Framework
  3. The reduce phase: data aggregation

- User must define only map & reduce functions
Map

- **Map function** simplifies the problem in this way:
  - **Input**: a single data item (e.g. line of text) from a data file
  - **Output**: zero or more (key, value) pairs
- **The keys are not typical “keys”:**
  - They do not have to be unique
  - A map task can produce several key-value pairs with the same key (even from a single input)
- **Map phase applies the map function to all items**
Grouping Phase

- **Grouping (Shuffling):** The key-value outputs from the **map** phase are grouped by key
  - Values sharing **the same key** are sent to the same reducer
  - These values are **consolidated** into a single list (**key, list**)
    - This is convenient for the reduce function
  - This phase is **realized by** the MapReduce framework

![Diagram showing grouping phase]

**Intermediate output**
(color indicates key)

**Shuffle (grouping) phase**
Reduce Phase

- **Reduce**: combines values with the same key
  - to achieve the **final result(s)** of the computational task
  - **Input**: (key, value-list)
    - value-list contains all values generated for given key in the Map phase
  - **Output**: (key, value-list)
    - zero or more output records

reduce function

output data
MapReduce, the full picture

- **Input data**
- **Map function**
- **Intermediate output**
  
  *color indicates key*

- **Shuffle (grouping) phase**

- **Input data**
- **Reduce function**

- **Output data**
**Example: Word Count**

Task: Calculate **word frequency** in a set of documents

```python
map(key, value):
    """ key: document name (ignored)
    value: content of document (words) """
    for w in value.split(' '):
        emitIntermediate(w, 1)

reduce(key, values):
    """ key: a word
    values: a list of counts """
    result = 0;
    for v in values:
        result += v
    emit(key, result)
```
Example: Word Count (2)
MapReduce: Combiner

- If the reduce function is commutative & associative
  - The values can be combined in any order and combined per part (grouped)
    - with the same result (e.g. Word Counts)

- ...then it opens space for optimization
  - Apply the same reduce function right after the map phase, before shuffling and redistribution to reducer nodes

- This (optional) step is known as the combiner
  - Note: it’s still necessary to run the reduce phase
**Example: Word Count, Combiner**

Task: Calculate *word frequency* in a set of documents

```python
combine(keyValuePairs):
    """ keyValuePairs: a list counts """
    result = {}
    for k, v in keyValuePairs:
        result[k] = result.get(k, 0) + v
    for k, v in result:
        emit(k, v);
```
Word Count with Combiner

Comp 521 – Files and Databases
MapReduce Framework

- MapReduce **framework** takes care about
  - **Distribution** and parallelizing of the computation
  - **Monitoring** of the whole distributed task
  - The **grouping** phase
    - putting together intermediate results
  - **Recovering** from any failures

- User must define **only** map & reduce **functions**
  - but can define also other additional functions
MapReduce Framework
MapReduce Framework: Details

1. **Input reader** (function)
   - defines how to **read data** from underlying storage

2. **Map** (phase)
   - **master node** prepares $M$ data splits and $M$ idle Map tasks
   - pass individual splits to the Map tasks that run on workers
   - these map tasks are then **running**
   - when a task is **finished**, its intermediate results are stored

3. **Combiner** (function, optional)
   - **combine** local intermediate output from the Map phase
MapReduce Framework: Details

4. Partition (function)
   ○ to **partition** intermediate results for individual **Reducers**

5. Comparator (function)
   ○ sort and **group** the input for each Reducer

6. Reduce (phase)
   ○ **master** node creates **R idle** Reduce tasks on **workers**
   ○ **Partition** function defines a data **batch** for each reducer
   ○ each Reduce task uses **Comparator** to create **key-values pairs**
   ○ function Reduce is **applied** on each key-values pair

7. Output writer (function)
   ○ defines how the **output** key-value pairs are **written out**
Task: Calculate graph of web links
❖ what pages reference (<a href=””>) each page (backlinks)

map(url, html):
""" url: web page URL
html: HTML text of the page ""
for tag, contents in html:
    if tag.type == 'a':
        emitIntermediate(tag.href, url)

reduce(key, values):
""" key: target URLs
    values: a list of source URLs ""
emit(key, values)
Example II: Result

**Input:** (page_URL, HTML_code)

- ("http://cnn.com", "<html>...<a href="http://cnn.com">link</a>...</html>")
- ("http://ihned.cz", "<html>...<a href="http://cnn.com">link</a>...</html>")
- ("http://idnes.cz",
  "<html>... <a href="http://cnn.com">x</a>...
  <a href="http://ihned.cz">y</a>...
  <a href="http://idnes.cz">z</a>... </html>")

**Intermediate output after Map phase:**


**Intermediate result after shuffle phase (the same as output after Reduce phase):**


Comp 521 – Files and Databases Fall 2019
Task: What are the **lengths** of words in the input text

- output = **how many** words are in the text for each length

```python
map(key, text):
    """ key: document name (ignored)
    text: content of document (words) """
    for w in text.split(' '):
        emitIntermediate(length(w), 1)

reduce(key, values):
    """ key: a length
    values: a list of counts """
    result = 0;
    for v in values:
        result += v
    emit(key, result)
```

Same reduce as wordcount
MapReduce: Features

- MapReduce uses a “shared nothing” architecture
  - Nodes operate independently,
    - shares no memory
    - shares no disk
  - Common feature of many NoSQL systems

- Data partitioned and replicated over many nodes
  - Pro: Large number of read/write operations per second
  - Con: Coordination problem – which nodes have my data, and when?
Applicability of MapReduce

- MR is always applicable if the problem is trivially parallelizable

- Two problems:
  1. The programming model is limited (only two phases with a given schema)
  2. There is no data model - it works only on “data chunks”

- Google’s answer to the 2nd problem was BigTable
  - The first column-family system (2005)
  - Subsequent systems: HBase (over Hadoop), Cassandra,...
Apache Hadoop

❖ Open-source MapReduce framework
  ▪ Implemented in Java
  ▪ Named for author's (Doug Cutting) son's yellow toy elephant

❖ Able to run applications on large clusters of commodity hardware
  ▪ Multi-terabyte data-sets
  ▪ Thousands of nodes

❖ A reimplementation and redesign of Google's MapReduce and Google File System

web: http://hadoop.apache.org/
Hadoop: Modules

- **Hadoop Common**
  - Common support functions for other Hadoop modules

- **Hadoop Distributed File System (HDFS)**
  - Distributed file system
  - High-throughput access to application data

- **Hadoop YARN**
  - Job scheduling and cluster resource management

- **Hadoop MapReduce**
  - YARN-based system for parallel data processing

Source: https://goo.gl/NPuuJr
HDFS: Data Characteristics

- Assumes:
  - **Streaming** data access
    - files are read sequentially from the beginning to end
  - **Batch processing** rather than interactive user access

- Very large data sets and files

- **Write-once / read-many**
  - A file once created does not change often
  - This assumption simplifies consistency

- Typical applications for this model:
  - MapReduce, web-crawlers, data warehouses, …
HDFS: Basic Components

- **Master/slave architecture**
- **HDFS exposes a file system namespace**
  - Files are internally **split** into **blocks** and distributed over servers called "DataNodes"
  - Blocks are relatively large (64 MB by default)
- **NameNode** - **master** server
  - Manages the **file system namespace**
    - Opening/closing/renaming files and directories
    - Arbitrates file access
  - Determines **mapping of blocks** to DataNodes
- **DataNode** - **manages file blocks**
  - **Block** read/write/creation/deletion/replication
  - Usually one per physical node
HDFS: Schema
**HDFS: NameNode**

- **NameNode** has a structure called **FsImage**
  - Entire *file system* namespace + mapping of *blocks* to files + file system properties
  - Stored in a file in NameNode’s local file system
  - Designed to be *compact*
    - Loaded in NameNode’s memory (4 GB of RAM is sufficient)

- **NameNode** uses a *transaction log* called **EditLog**
  - to *record* every change to the file system’s meta data
    - E.g., creating a new file, change in replication factor of a file, ..
  - EditLog is stored in the NameNode’s local file system
HDFS: DataNode

- Stores **data blocks as files** on its local file system
  - Each HDFS block is a separate file
  - Has no knowledge about HDFS file system

- When the DataNode **starts up**:
  - It *generates* a list of all HDFS blocks = BlockReport
  - It sends the report to NameNode
HDFS can store very large files across a cluster
▪ Each file is a sequence of blocks
▪ All blocks in the file are of the same size
  • Except the last one
  • Block size is configurable per file (default 128MB)
  • Use of large files promotes high I/O throughput
▪ Blocks are replicated for fault tolerance
  • Number of replicas is configurable per file

NameNode receives HeartBeat and BlockReport from each DataNode
▪ BlockReport: list of all blocks on a DataNode
HDFS: Block Replication

Data is divided into 128 MB per block.

Very large data file

HDFS stores files in a number of blocks.

Each block is replicated to a few separate computers.
Primary objective: to store data reliably in case of:
- NameNode failure
- DataNode failure
- Network partition
  - a subset of DataNodes can lose connectivity with NameNode

In case of absence of a HeartBeat message
- NameNode marks DataNodes without HeartBeat and does not send any I/O requests to them
- The death of a DataNode typically results in re-replication
Hadoop: MapReduce

- Hadoop MapReduce requires:
  - Distributed file system (typically HDFS)
  - Engine that can distribute, coordinate, monitor and gather the results (typically YARN)

- Two main components:
  - **JobTracker** (master) = scheduler
    - tracks the whole MapReduce job
    - communicates with HDFS NameNode to run the task close to the data
  - **TaskTracker** (slave on each node) – is assigned a Map or a Reduce task (or other operations)
    - Each task runs in its own JVM
Hadoop HDFS + MapReduce
Hadoop MapReduce: Schema

The diagram illustrates the MapReduce process within the Hadoop framework. It shows the interaction between the client and the Job Tracker, as well as the flow of data through the Map and Reduce phases.

1. **Input Format** is transformed into **RAM** in the Map phase.
2. **Partition** and **Combine** operations are performed in the Map phase by Task Trackers.
3. **HDFS** stores the input files, and the Map phase partitions these files into **region 1** and **region 2**.
4. In the **Reduce phase**, Task Trackers read the data from HDFS, perform sorting, and then reduce the data. The output is stored back in HDFS.
5. The process is repeated for multiple Map and Reduce stages as indicated by the diagram, showing the iterative nature of the MapReduce algorithm.

The diagram is labeled with Czech terms, which translate to:
- vstupní soubory: input files
- část 1-5: partitions
- region 1-2: regions
- sort: sorting
- reduce(): reduce function
- output format: output format

This visual representation helps in understanding the parallel processing model of MapReduce.
public class Map extends Mapper<LongWritable, Text, Text, IntWritable> {
    private final static IntWritable one = new IntWritable(1);
    private final Text word = new Text();

    @Override
    protected void map(LongWritable key, Text value, Context context) throws ...
    {
        String string = value.toString()
        StringTokenizer tokenizer = new StringTokenizer(string);
        while (tokenizer.hasMoreTokens()) {
            word.set(tokenizer.nextToken());
            context.write(word, one);
        }
    }
}
Hadoop MR: WordCount Example (2)

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

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

- A shallow dive into the Hadoop Eco system
- Primarily, Pig and Hive