CSCI E-88 Principles Of Big Data Processing

Harvard University Extension, Fall 2019

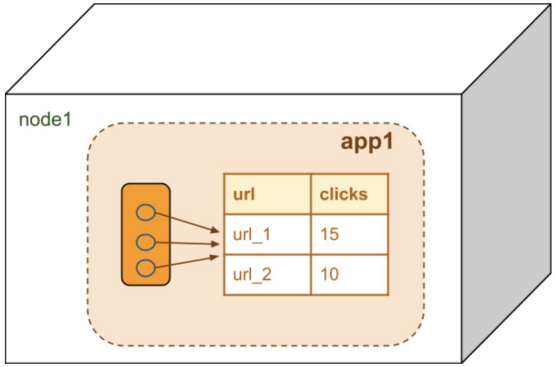
Marina Popova

Lecture 3 - Scaling For Real: MR, HDFS

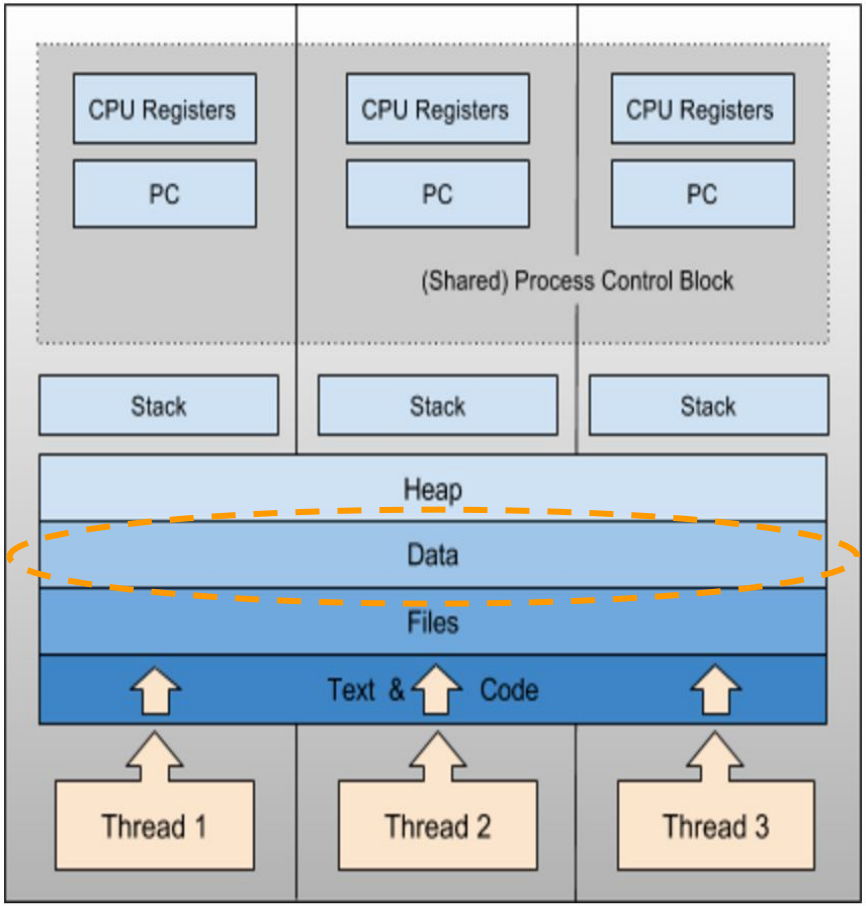
Agenda

- Scaling - continue
- Basics of Parallel Processing
- Introduction to Map-Reduce
- HDFS

Scaling Recap - what we did so far:



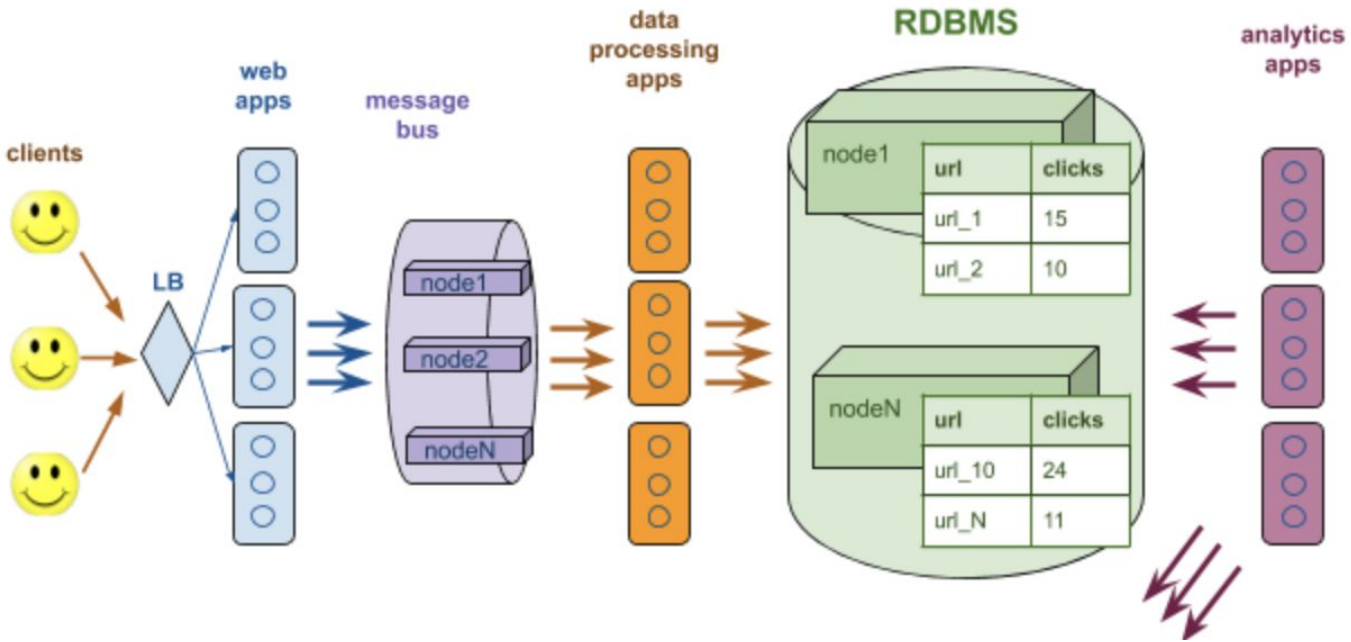
- we scaled our application vertically first - by adding multiple threads
- we used **shared in-memory state** between threads
- we looked at the example data structures to implement this in HW2
- we also saw CPU and IO limitations of a single server in action in HW2



@Marina Popova

Scaling Recap - what we did so far:

- we scaled our app to multiple servers to solve IO and CPU limitations



Scaling Recap:

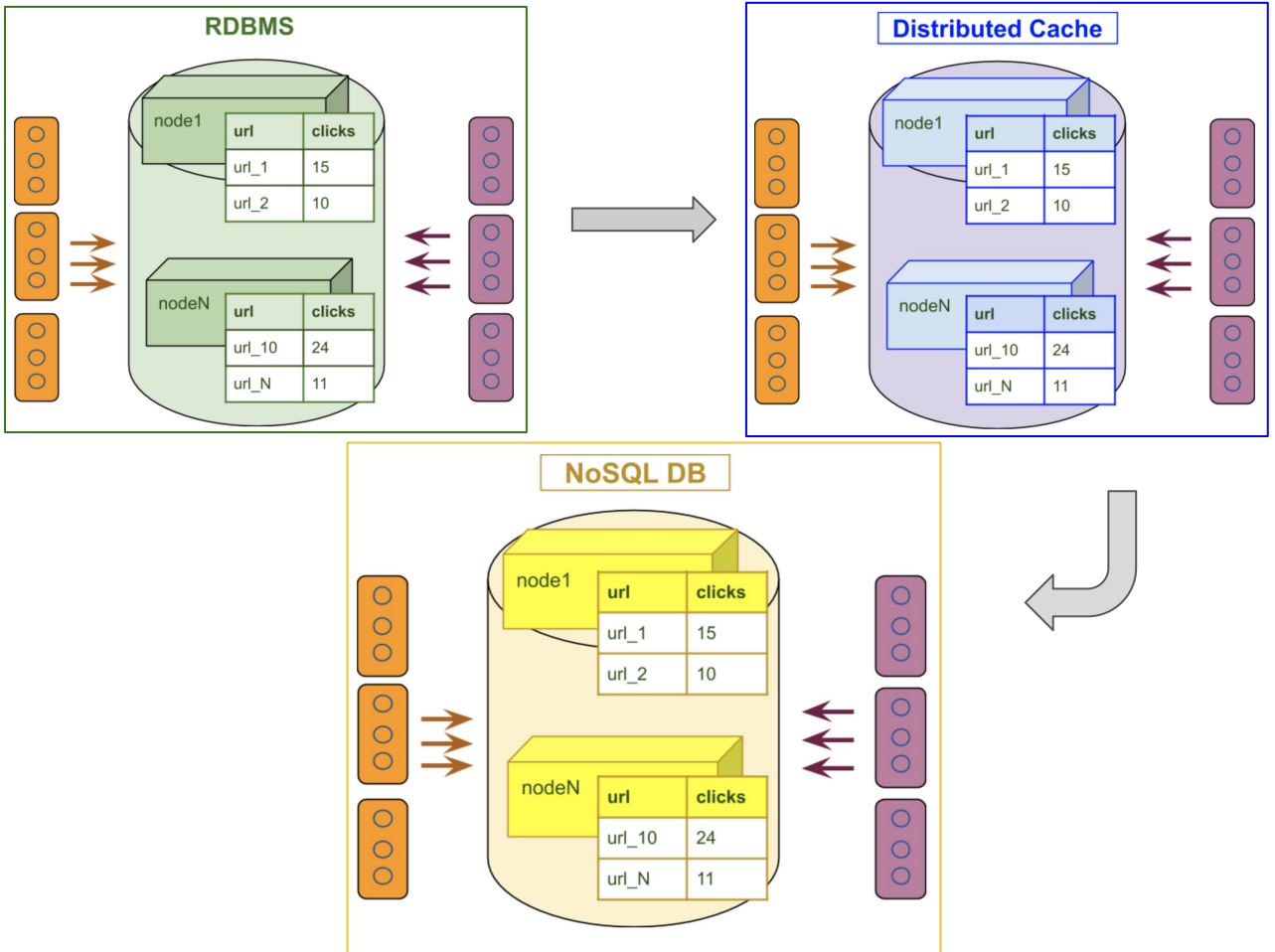
we used **shared mutable state** between all apps:

- using RDBM
- using Distributed cache

we learned about differences between Distributed Cache and RDBMS (Consistent Hashing)

we will learn more details about Redis in the Lab 3

we will implement this approach in the HW3 !



Moving On ….

Eventually, shared mutable state will become a problem:

- too large for distributed cache
- not fast enough
- not reliable enough

Why?

- Single row contention for the same key
- mutable objects are not handled well with data replication (see the Side Notes on DC)
- atomic multi-step operations will become a problem

Side Note: Distributed Cache: mutable vs. immutable objects

Ref: https://stackoverflow.com/questions/33332288/in-process-cache-vs-distributed-cache-on-consistency-with-mutable-immutable-obje

"When you use a distributed cache, each object is replicated among multiple independent machines, multiple *cache nodes*.

If your objects are immutable, replication is not an issue: since the objects never change, any cache instance will deliver exactly the same objects.

As soon as the objects become mutable, **the consistency issues arise**"

Why? --> continue ....

Side Note: Distributed Cache: mutable vs. immutable objects

"when you ask a cache instance for an object, how can you be sure that the object which is delivered to you is up-to-date? What if, while one cache instance was serving you, the object was being modified by another user on another cache instance? In that case, you would not receive the latest version, you would receive a *stale version*.

To deal with this issue, a choice has to be made. One option is to accept some degree of staleness, which allows better performance. Another option is to use some synchronization protocol, so that you never receive stale data: but there obviously is a performance penalty to be paid for this data synchronization between distant cache nodes.

Conversely, imagine that you upload to a cache node some modifications of an object. What if, at the same time, another user uploads some modifications of the same object to another cache node? Should this be allowed, or should it be forbidden by some locking mechanism?

In addition, should object modifications on your cache node become immediately visible to the users of this cache node? Or should they become visible only after they have been replicated to the other nodes?

At the end of the day, mutable objects do make things more complicated when sharing a distributed cache among multiple users. Still, it doesn't mean that these cache should not be used: it just means that it takes more time and more caution to study all available options and choose the appropriate cache for each application."

How to make your processing logic parallelizable?

to truly scale out your processing horizontally, the following core requirements have to be met:

- no shared state
- no shared data
- all operations have to be idempotent:

Idempotent services are services that are guaranteed to produce the same result, no matter how many times they are invoked

Good references:
https://computing.llnl.gov/tutorials/parallel_comp/
http://www.cac.cornell.edu/education/training/StampedeJune2013/ParallelProgramming.pdf
http://selkie.macalester.edu/csinparallel/modules/IntermediateIntroduction/build/latex/IntermediateIntroduction.pdf

Side note: what's the difference between idempotent and atomic operation?
"atomic operation" means that an operation is completed in its entirety, and can be viewed externally only as an indivisible step (regardless of how many steps the actual operation required). It might or might NOT be idempotent!

Basics Of Parallel Processing

Lets look back at the counting Example 1 in Lecture 2 and consider what makes it not parallelizable and how to fix that

1. Shared state **between threads/apps** is used - click_counts
2. The shared state is **mutable**
3. **Incremental algorithms** are used - which modify the counts

Example 2: implement the same click and visitor counting application in a parallelizable manner
Main goals:

- Remove any shared state between parallel threads/processes
- Make thread-scope operations self-contained
- Use data partitioning - each process/thread has separate set of data
- Make all operations idempotent

Basics Of Parallel Processing

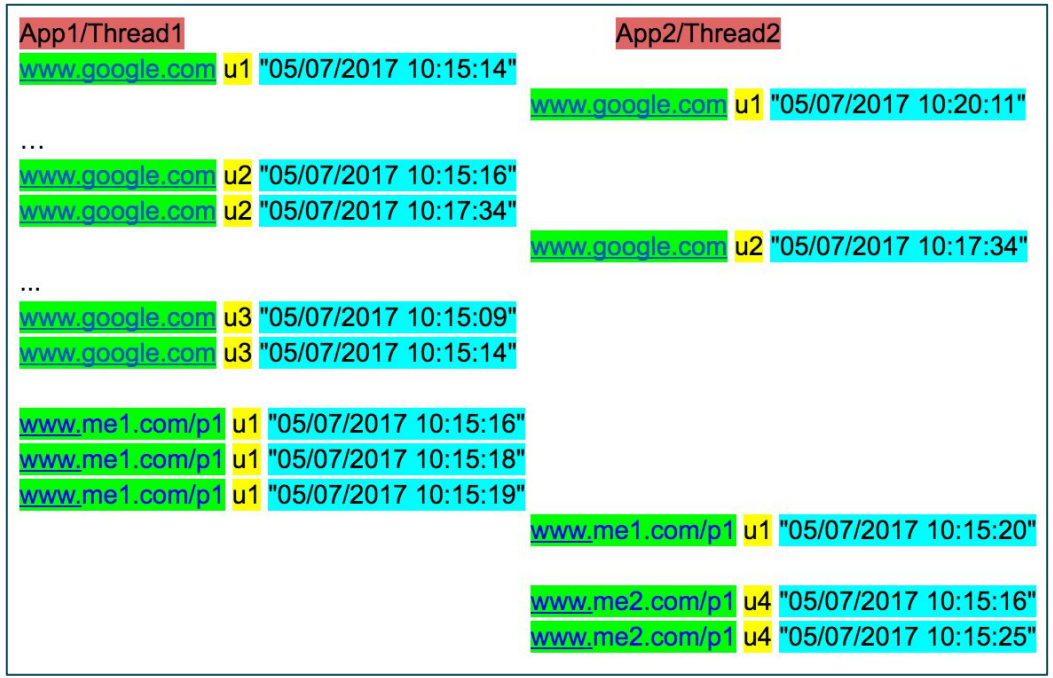
What is the main difference between Example 1 and Example 2 approaches?

Instead of using state that is shared among all threads/processes - like counts - we will use state that is contained within one thread/process only.

This means we will have to combine the results from ALL threads/processes - as a separate distinct phase of the processing !

Example 2: click and visitor counting application - parallelizable

the same input data - split between instances:



results of the queries should be the same:

Query 1: count of unique URLs: **3**

Query 2: count of unique visitors (userIDs) per URL:

- www.google.com : 3 [u1, u2, u3]
- www.me1.com/p1 : 1 [u1]
- www.me2.com/p1 : 1 [u4]

Query 3: count of unique clicks per URL:

- www.google.com :
  - u1 : 2
  - u2 : 3
  - u3 : 2
- www.me1.com/p1 :
  - u1 : 4
- www.me2.com/p1 :
  - u4 : 2

Data structure for in-memory data is the same as before, but it is per-thread/app now:

App1/Thread1

{"www.google.com" --->
{"u1" ---> 1}
{"u2" ---> 2}
{"u3" ---> 2}
"www.me1.com/p1" --->
{"u1" ---> 3}
}

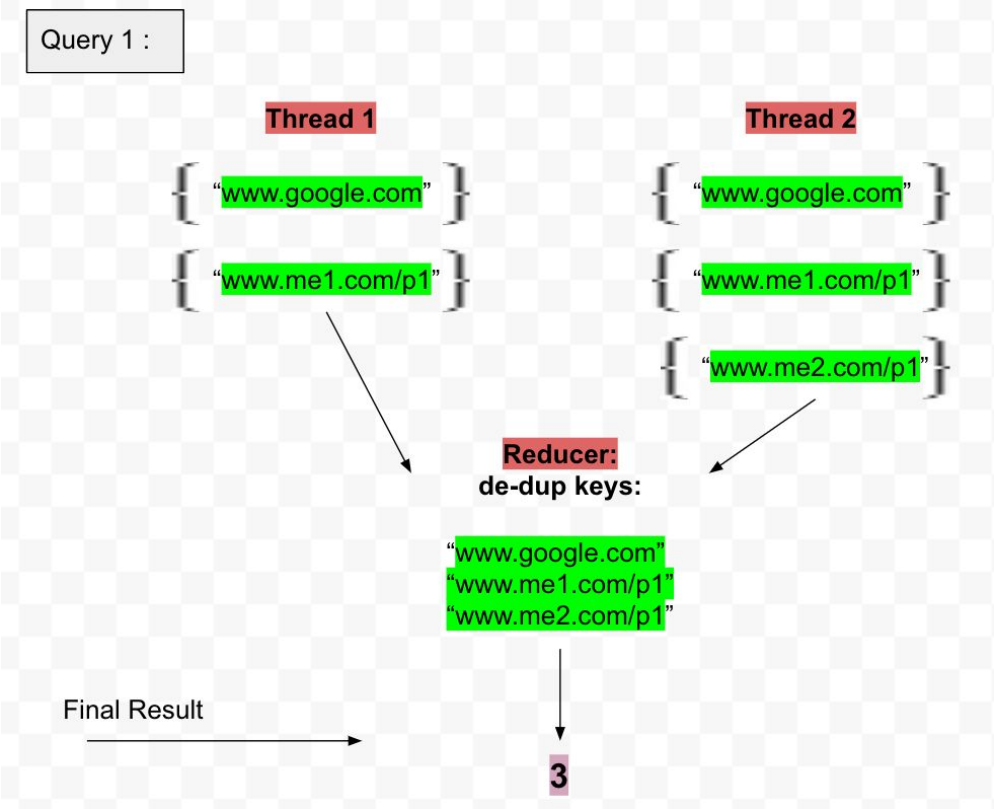
App2/Thread2:

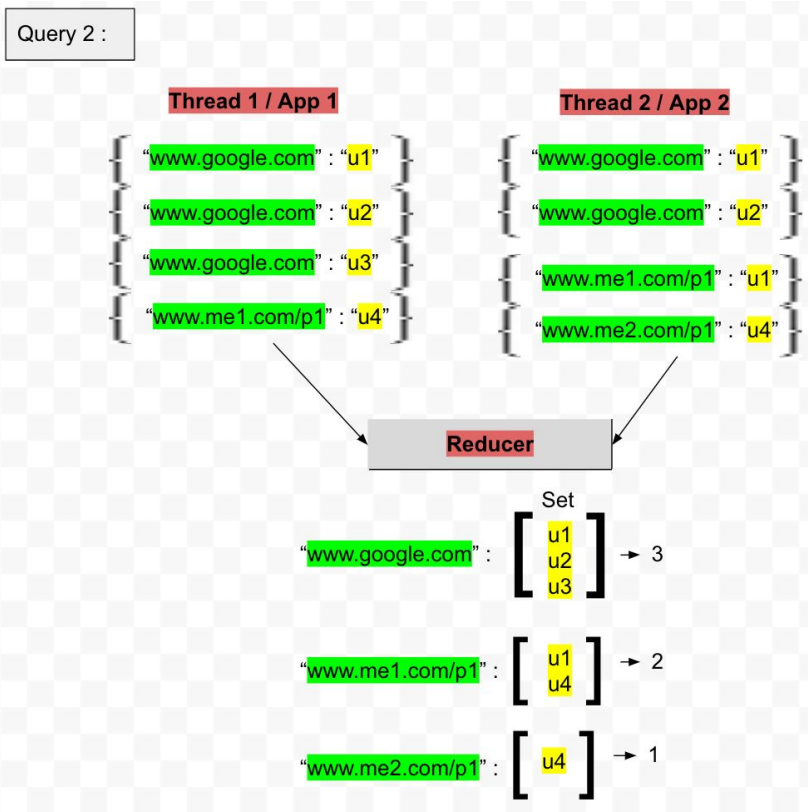
{"www.google.com" --->
{"u1" ---> 1}
{"u2" ---> 1}
"www.me1.com/p1" --->
{"u1" ---> 1}
"www.me2.com/p1" --->
{"u4" ---> 2}
}

The main question is:
How to combine individual results from multiple apps/threads to get the final results?

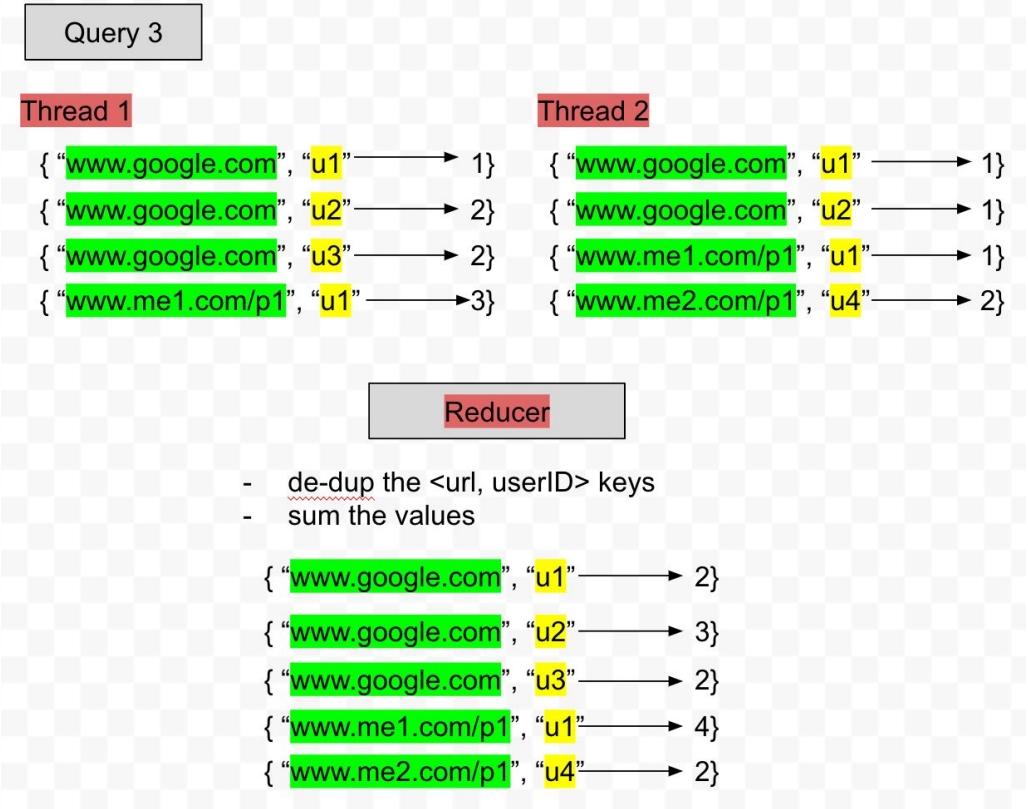
In this implementation approach we are going to use a separate process, called **Reducer**, to collect and compute final results

Lets review what information each App/Tread has to collect and send to the Reducer in order to provide results for the same three Queries


Query 1 :
Thread 1
“www.google.com”
“www.me1.com/p1”
Thread 2
“www.google.com”
“www.me1.com/p1”
“www.me2.com/p1”
Reducer:
de-dup keys:
“www.google.com”
“www.me1.com/p1”
“www.me2.com/p1”
Final Result
3



@Marina Popova


Query 3
Thread 1
{ “www.google.com”, “u1” → 1}
{ “www.google.com”, “u2” → 2}
{ “www.google.com”, “u3” → 2}
{ “www.me1.com/p1”, “u1” → 3}
Thread 2
{ “www.google.com”, “u1” → 1}
{ “www.google.com”, “u2” → 1}
{ “www.me1.com/p1”, “u1” → 1}
{ “www.me2.com/p1”, “u4” → 2}
Reducer
- de-dup the <url, userID> keys
- sum the values
{ “www.google.com”, “u1” → 2}
{ “www.google.com”, “u2” → 3}
{ “www.google.com”, “u3” → 2}
{ “www.me1.com/p1”, “u1” → 4}
{ “www.me2.com/p1”, “u4” → 2}

Basics Of Parallel Processing

Review of properties of this approach:

- Is there any shared state in this implementation?
  - NO
- Is there any shared data?
  - NO
- Is data immutable?
  - YES
- Is processing algorithm incremental?
  - NO - across apps/threads, YES - in the thread scope
- Are all operations idempotent?
  - YES

Basics Of Parallel Processing

Important take-aways:

- We were able to compute uniques only because we had access to the entire data set and could re-compute metrics
- Very important: all counts are calculated over the lifespan of the application - as soon as the application[s] are restarted - the old state is wiped out and a new one is calculated. Options to fix that:
  - use distributed cache with persistence to disk that allows re-loading of the previously stored state
  - persist results periodically into some storage (DB, FS)
  - all approaches are very tricky as there maybe many types of failures (node down, network partitioning), and still many issues to solve, like check-points, recent unsaved data ...

Basics Of Parallel Processing

Important take-aways - state persistence:

- Who is saving the state? Each thread? In which case - how do they know what data to load on startup? What if the number of threads/apps is changed after the restart?
- If it is the one last process (reducer) who is saving the state - which thread gets to load it, as it cannot be double-counted and loaded into multiple threads…. Also, we are getting back to the shared state situation again...
- Many others - just start thinking …. :)
- This might look like storing state is not a good idea - then what do we do?
    - we need to define the right model for the data to be stored - so that your process could access the exact data it needs to answer the required queries. It might be an entire raw set of data, partial calculated metrics or a combination of both - but in any case the data stored has to be immutable, non-sharable and allow for idempotent operations
- We will review approaches to data modeling and storage for raw and pre-calculated data in the next set of lectures

Basics Of Parallel Processing

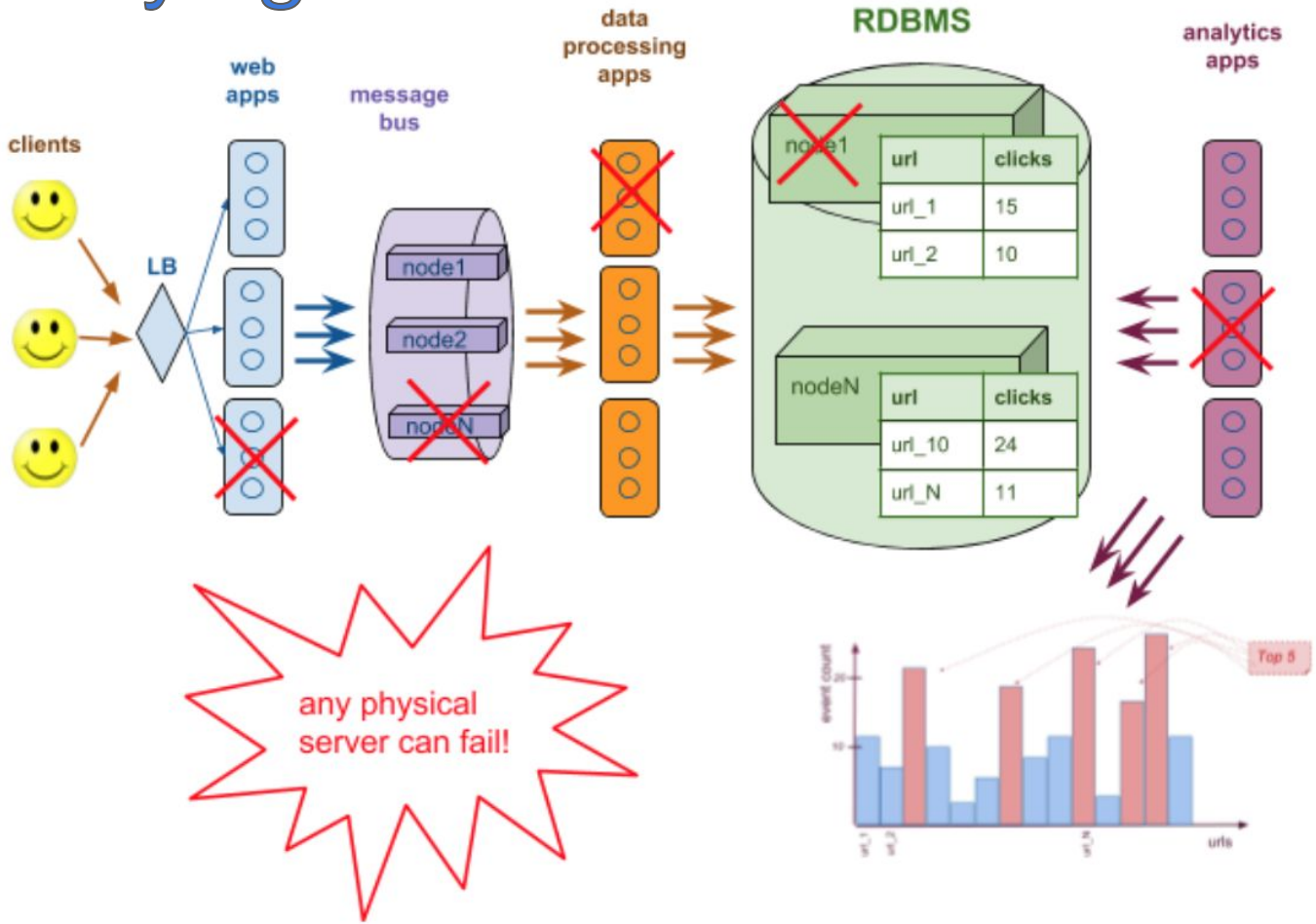
Another important question is: what is the Reducer and where does it leave?

- Obviously, it cannot be part of each thread operation
- Must be a separate process accessible somehow to all other data processing apps/threads
- How would data be passed to it? Many options:
  - Make Reducer expose a REST API and send events to it using that API (slow)
  - Use shared DB (bad)
  - Use distributed FS (doable)
  - Use messaging system like RabbitMQ or Kafka

this is not the only issue to solve! we will review many more soon ...
but first - let's review and generalize the problem area

What are we trying to do ?

we are trying to build a ***reliable and fast distributed system*** *capable of processing Big Data ...*



@Marina Popova

Scaling For Real

Main Challenges and Goals building Distributed systems:

1. **Scalability** - how to distribute processing and data over more and more physical servers?
2. **Fault Tolerance/ Availability** - how to survive failure of individual nodes
3. **Data Consistency** - is my data correct, distributed over multiple nodes? and when some nodes die?
4. **Data Latency** (staleness) - do I collect/store/process incoming data as soon as it is generated?
5. **Query response timings** - how long does it take to query [and visualize] my data?

How do we achieve all this?

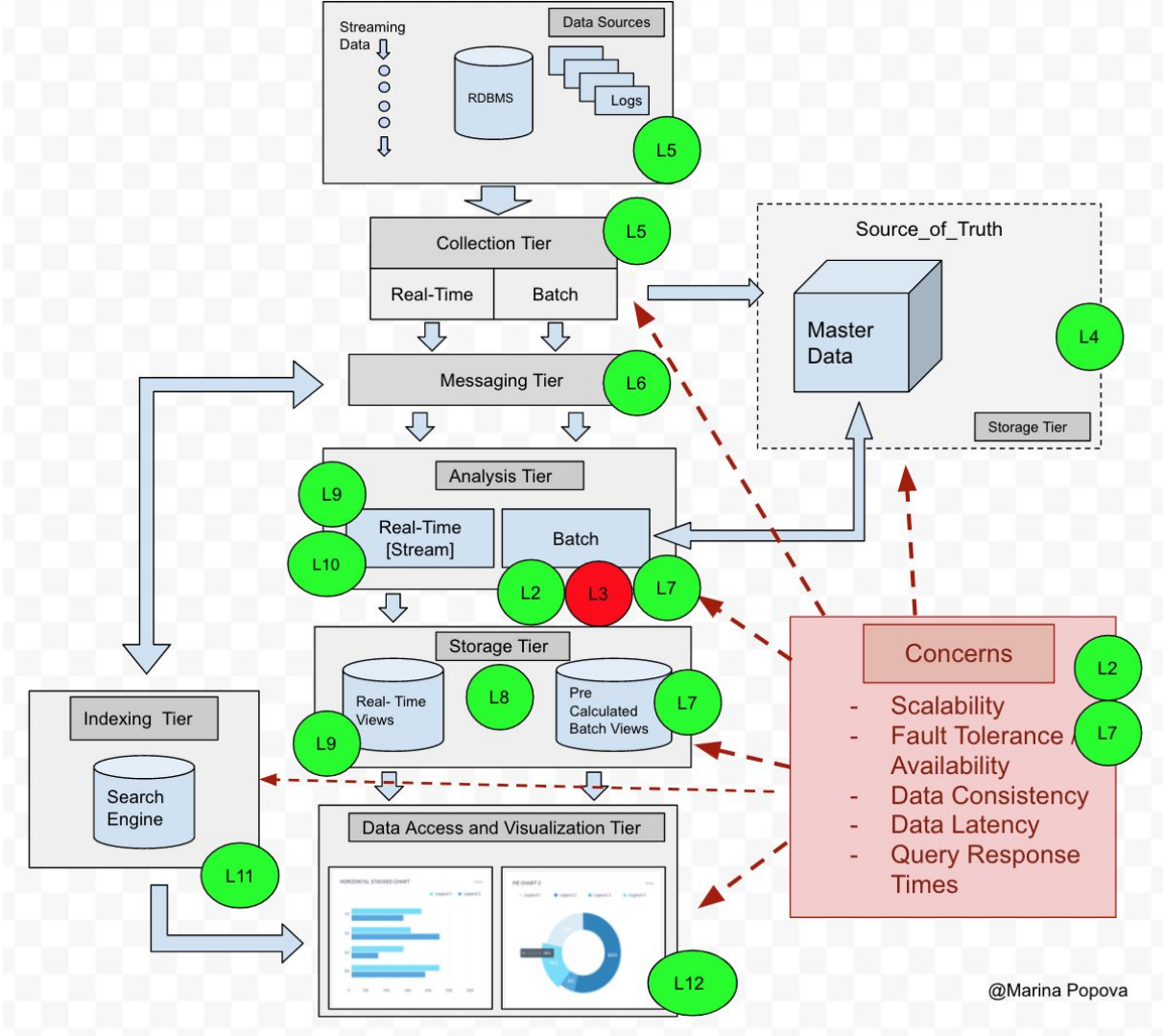
We have to address Scaling of both Data Processing (next) and Data Storage (later)

Scaling of Data Processing

Processing is what old-fashioned ETL was: Extract/Transform/Load

It is applicable to almost amy layer in BDPs

It is crucial for batch processing



Scaling of Data Processing

Main Concerns here:

1. **Scalability**
2. **Fault Tolerance/ Availability**
3. **Data Consistency**
4. **Data latency**

And core principles to achieve these are:

- no shared state
- no shared data
- idempotent operations

How could we implement/build a system like this?
… there are many ways - some are better than others ....

Just for fun:

Lets imagine we had to implement all this ourselves, in our application.

What would we have to do ??

we already saw some parts of such implementation and problems we had to address - let's consider others...

Scaling of Data Processing - Do It Yourself Way

Lets address Failover requirements first:

1. Develop and application that would be aware of all physical servers the instance will run on
2. Use RPC / TCP or some other protocol to actively communicate with apps on other physical servers - to make sure all are up and healthy
3. Decide how to divide work between all instances of the app
4. Decide how to detect when each instance finished its part of the jobs
5. Decide how to collect and combine the results of work of each instance
6. If noticed that one instance is down due to hardware or other failure:
    a. figure out which part of the work was done, and which was not
    b. Re-distribute that part of the work to another healthy app instance
    c. Figure out if the calculated results are still correct so far and if any data was lost due to the crash
    d. If lost - figure out how to get it again from a collection or message tier (if exists), and whether this is possible at all
    e. Find another physical node to run the instance on - bring it into the "ring" of the working nodes

Scaling of Data Processing

Now **lets address Scalability**:

We noticed that are application does not collect/process all incoming data fast enough, maybe we are getting more input data, our app is getting more and more popular …. Data is becoming STALE !
We want to add more instances to process data faster - what will our application has to do?

- App has to provide a way to add , say, 3 more instances for extra data processing
- When added - the app has to decide how to determine what jobs are already running on existing nodes
- Wait till all jobs are done, or interrupt them and figure out how to re-distribute the work among old and newly added instances
- Do it without losing any new incoming data
- Be able to determine if any of the new instances is of a wrong version/configuration - and reject it if so
- ….

Scaling of Data Processing

Now **lets address Data Consistency**:

Lets assume we are storing results of the data processing into an RDBMS os some other data storage
At any point in time - queries to the DB should return correct results

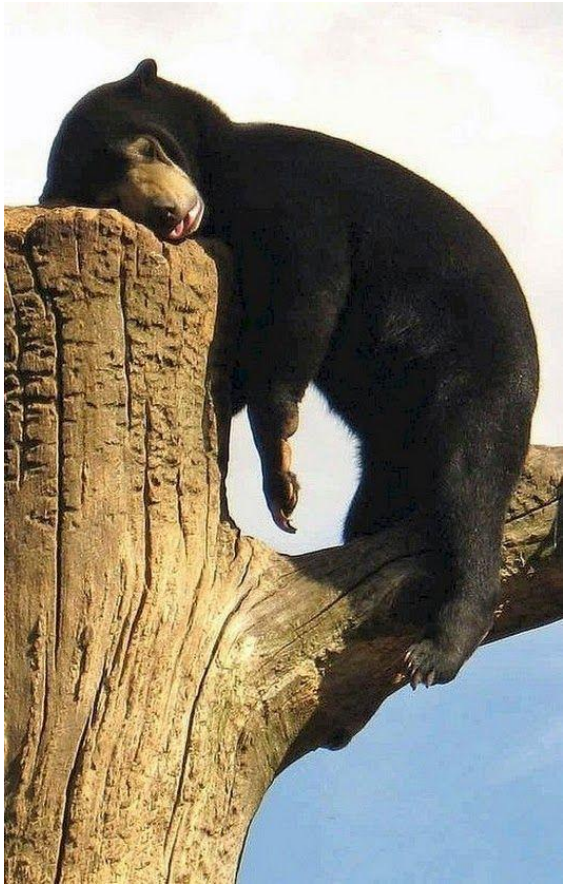
- Our app has to keep track of which results are being written into the DB - from each app instance
- When one instance dies for whatever reason - app has to determine which part of the results it was calculating and which part has already being stored into the DB, if any
- App has to re-distribute the failed work among other nodes
- If any partial results were written into the DB - they have to be cleaned out
- Once the work is re-done by healthy nodes - the results have to be correct - not duplicated or double-counted or under-counted or lost …
- …
-

Scaling of Data Processing

Did we cover it all ?!

NO !

not even scratched the surface ...



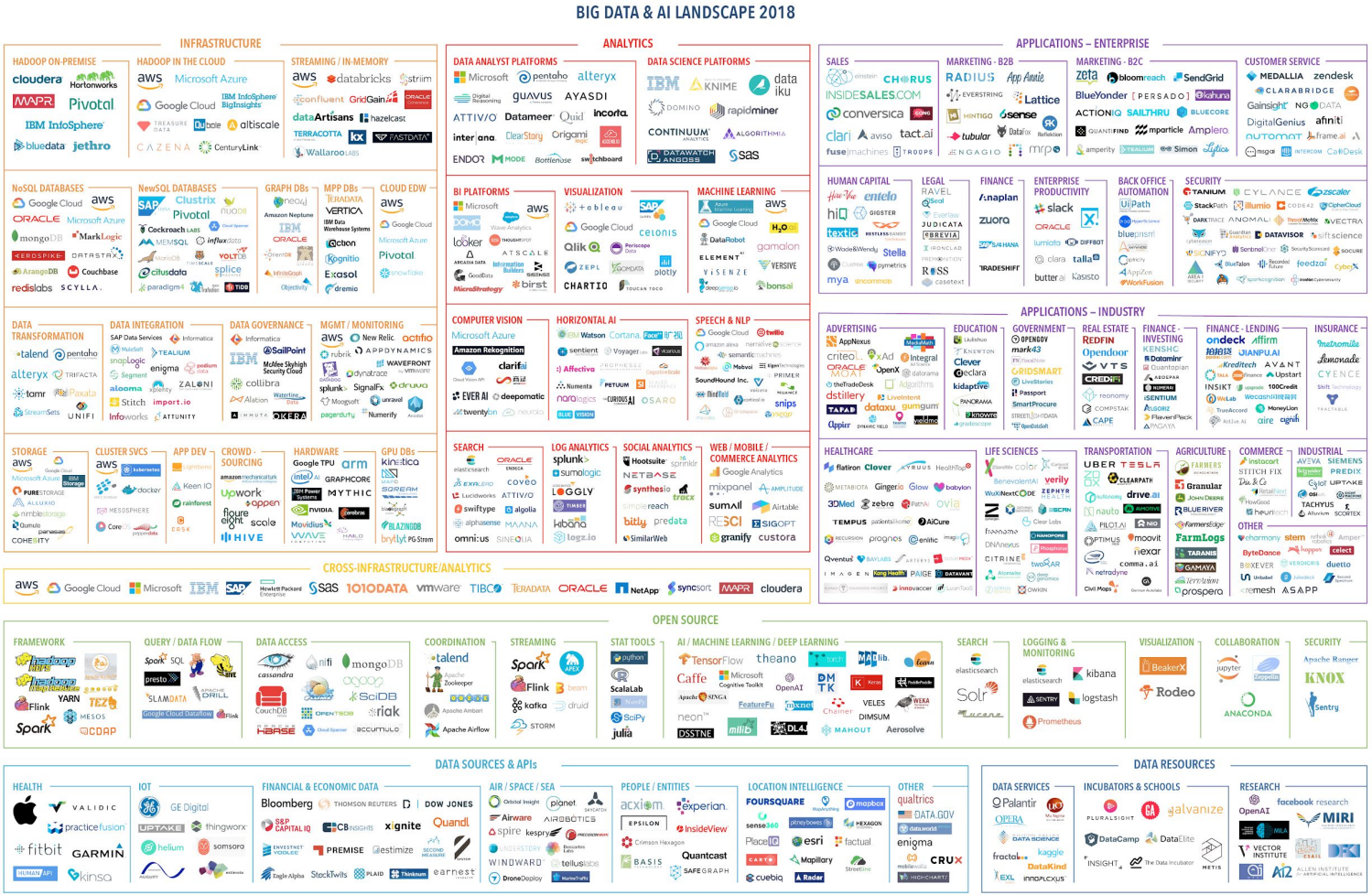
And .. do we really have to do it ourselves ??

@Marina Popova

NO!

There is a lot of help available !

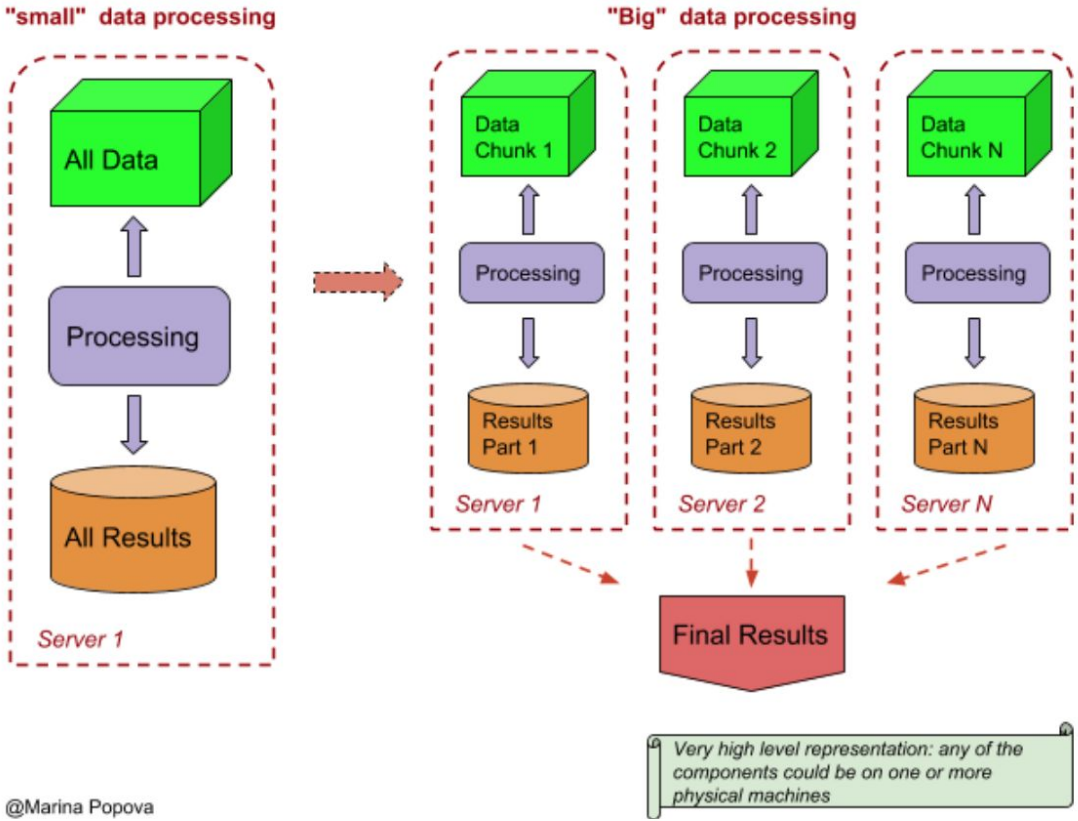
@Marina Popova



Why do we have so much trouble ??

Because we are trying to process BIG data !!

Processing Big vs Small Data



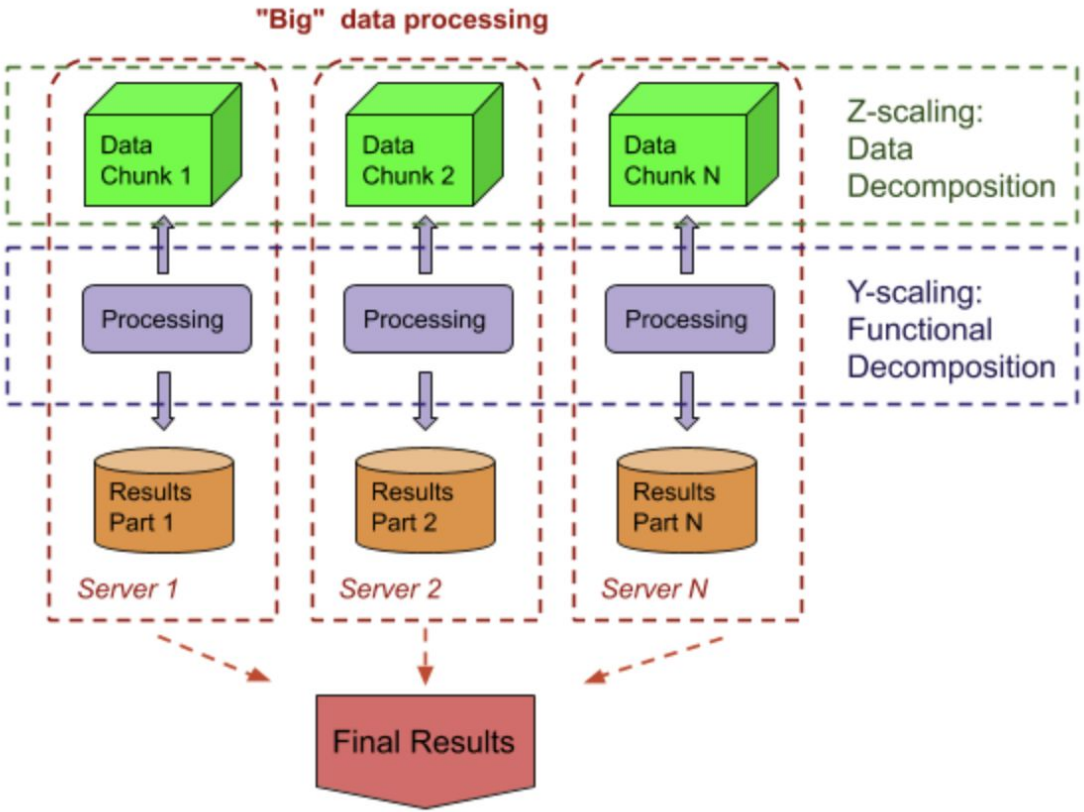
The main difference between processing "Big Data" vs "small data" is that it is no longer feasible to do this using a single physical server's CPU and RAM, for many reasons.

This means, **we have to split the data, computations and results between multiple physical servers** and potentially get the final results by combining the partial results from those servers.

And at the same time satisfy requirements for:

- Scalability
- Fault Tolerance
- Data Consistency
- Data Latency
- Query Performance

Do you recognize XYZ Scaling??



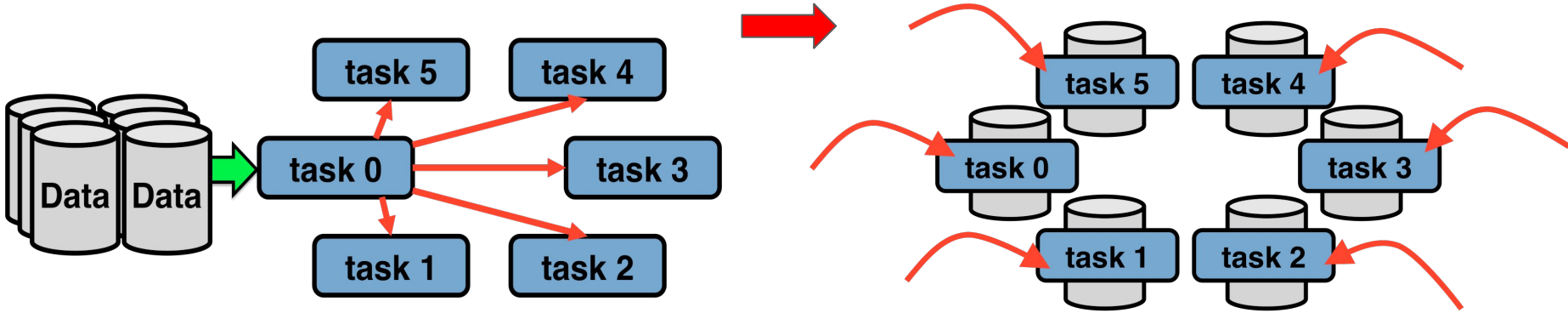
@Marina Popova

Big Data Processing - scaling out

What we observed is a well-known distinction between **Traditional Parallel CPU-intensive** applications and **Data Intensive** applications that use a different way to parallelize computing: Map-Reduce paradigm

CPU-intensive
Data is stored in large storage
Data is brought to Compute

Data-intensive
Data is stored in chunks on compute nodes
Compute is brought to Data



Ref: https://www.glennklockwood.com/data-intensive/hadoop/overview.html

So, how do we do it?
How do we implement this type of scaling?

It's a well known problem with a well known solution...

@Marina Popova

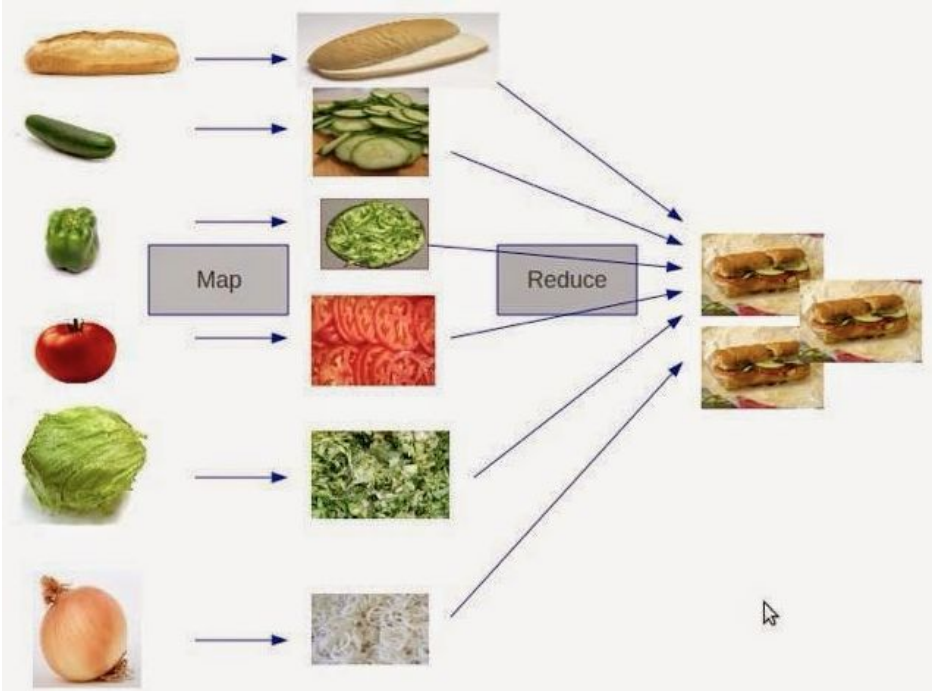
... and the solution is: MR

Map-Reduce paradigm:

A way to process large volumes of data in parallel by dividing the work into a set of independent tasks.

In essence, it is a set of **Map** and **Reduce** tasks that are combined to get final results:

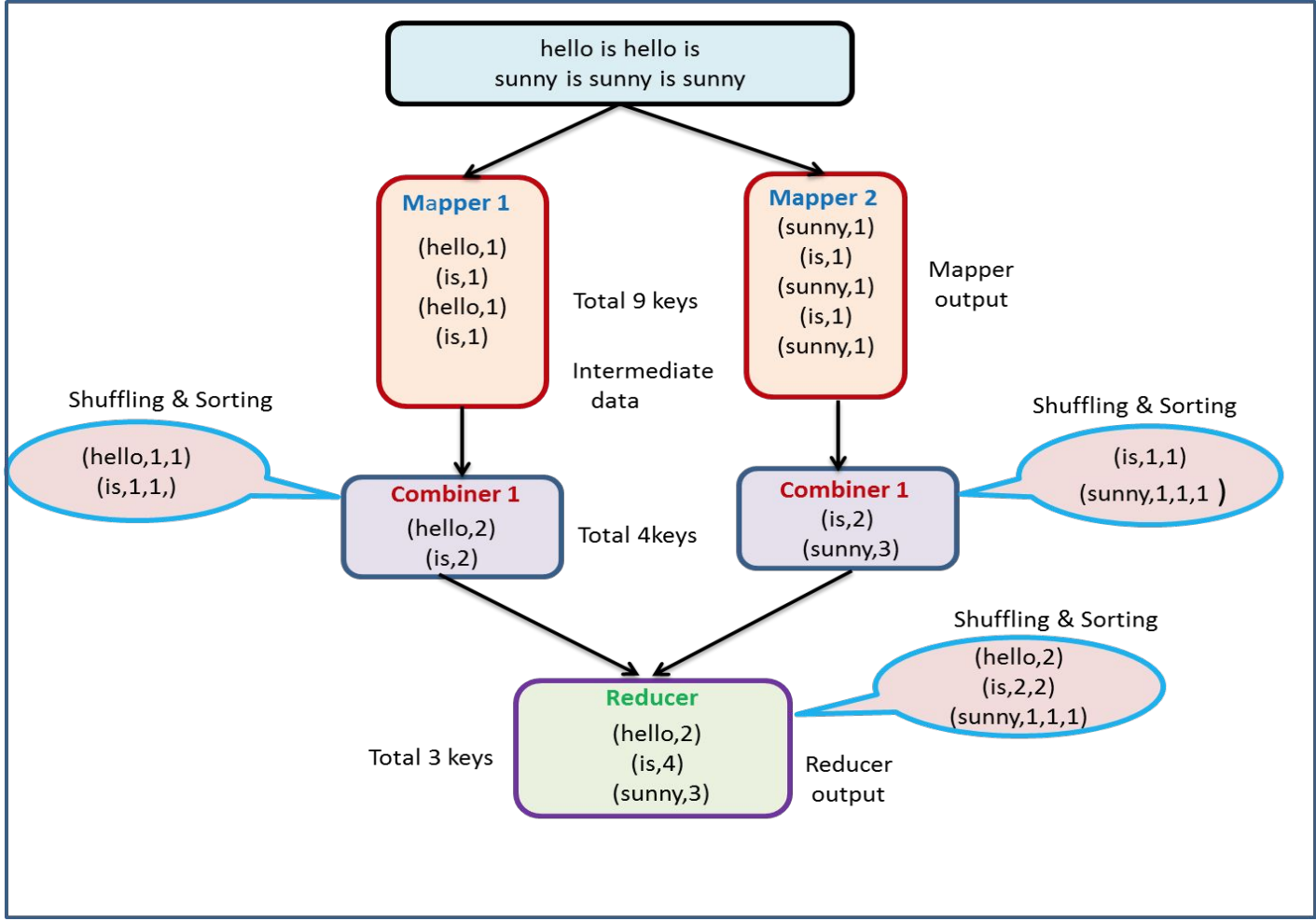
- Map function transforms the piece of data into **key-value pairs** and then the keys are sorted
- Reduce function is applied to **merge the values** based on the key into a single output



Picture source: https://www.datasciencecentral.com/forum/topics/what-is-map-reduce

MR Execution Flow:

1. Input splitting
2. Task allocation
3. Map phase
4. Combiner phase
5. Partition phase
6. Shuffle / Sort phase
7. Reduce phase



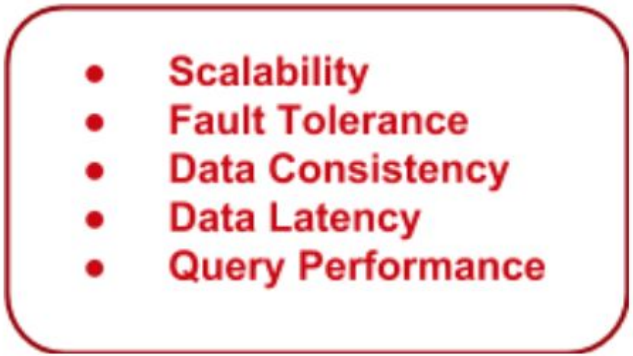
References: http://data-flair.training/blogs/hadoop-mapreduce-introduction-tutorial-comprehensive-guide/

@Marina Popova

Scaling of Data Processing

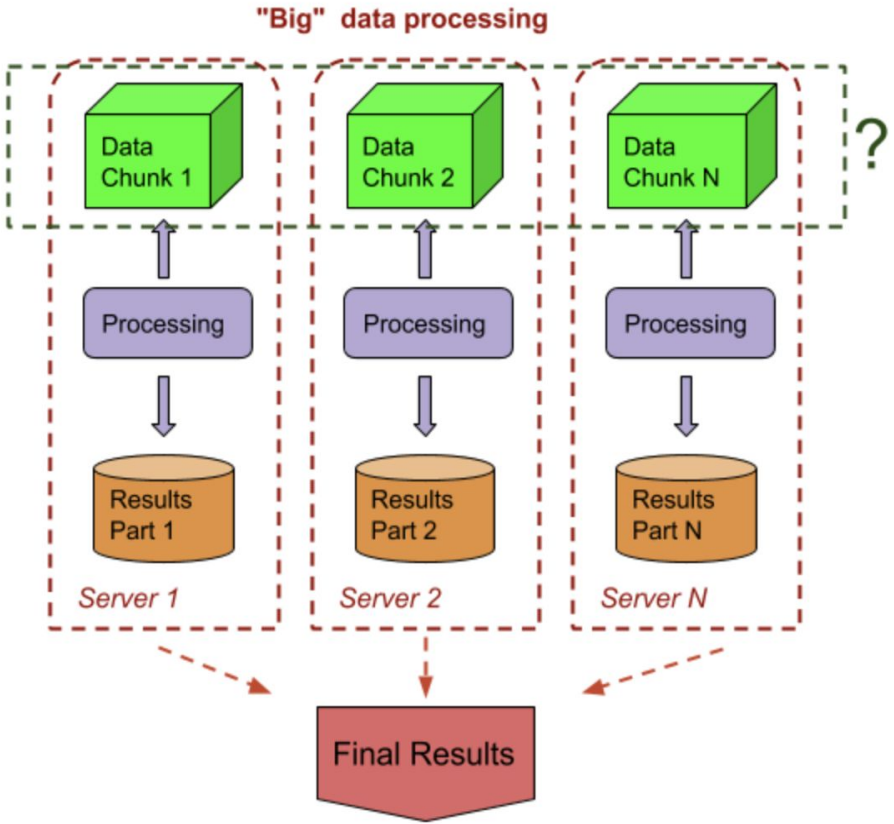
There is more to MR and scaling of Data Processing - how exactly does MR address our key scaling requirements?

- `operation parallelization`
- `idempotent operations`
- `data replication`
  - (and state replication if needed)



We will dive into MR frameworks and implementations in the next Lecture

Scaling of Data Processing - Data Decomposition



Requirements for Data Decomposition are the same:

- Scalability
- Fault Tolerance
- Data Consistency
- Data Latency
- Query Performance

how do we implement this? ...

Scaling of Data Processing - Data Decomposition

You are in luck!
You don't have to implement it yourself!

There are a few options to consider:

- HDFS
- S3
- Google Cloud Storage
- Azure

Distributed File Systems: HDFS

General concepts of Hadoop Architecture

HDFS is part of the Apache Hadoop project.
Apache Hadoop project consists of the following key parts:

1. **Hadoop Common**: The common utilities that support the other Hadoop modules.
2. **Hadoop Distributed File System (HDFS™)**: A distributed file system that provides high-throughput access to application data.
3. **Hadoop YARN**: A framework for job scheduling and cluster resource management.
4. **Hadoop MapReduce**: A **YARN**-based system for parallel processing of large data sets.

Latest Hadoop documentation: http://hadoop.apache.org/docs/current/

Distributed File Systems: HDFS

"The Hadoop Distributed File System (HDFS) is a distributed file system designed to run on commodity hardware. It has many similarities with existing distributed file systems. However, the differences from other distributed file systems are significant. HDFS is highly fault-tolerant and is designed to be deployed on low-cost hardware. HDFS provides high throughput access to application data and is suitable for applications that have large data sets."

Key concepts:

- Deployed across multiple servers, called a **cluster**
- HDFS provides a write-once-read-many access model for files
- Hardware failure is the norm rather than the exception
- Files are spread across multiple nodes, for fault tolerance, HA, and to enable parallel processing
- File blocks are replicated across multiple nodes for fault tolerance and HA

Distributed File Systems: HDFS

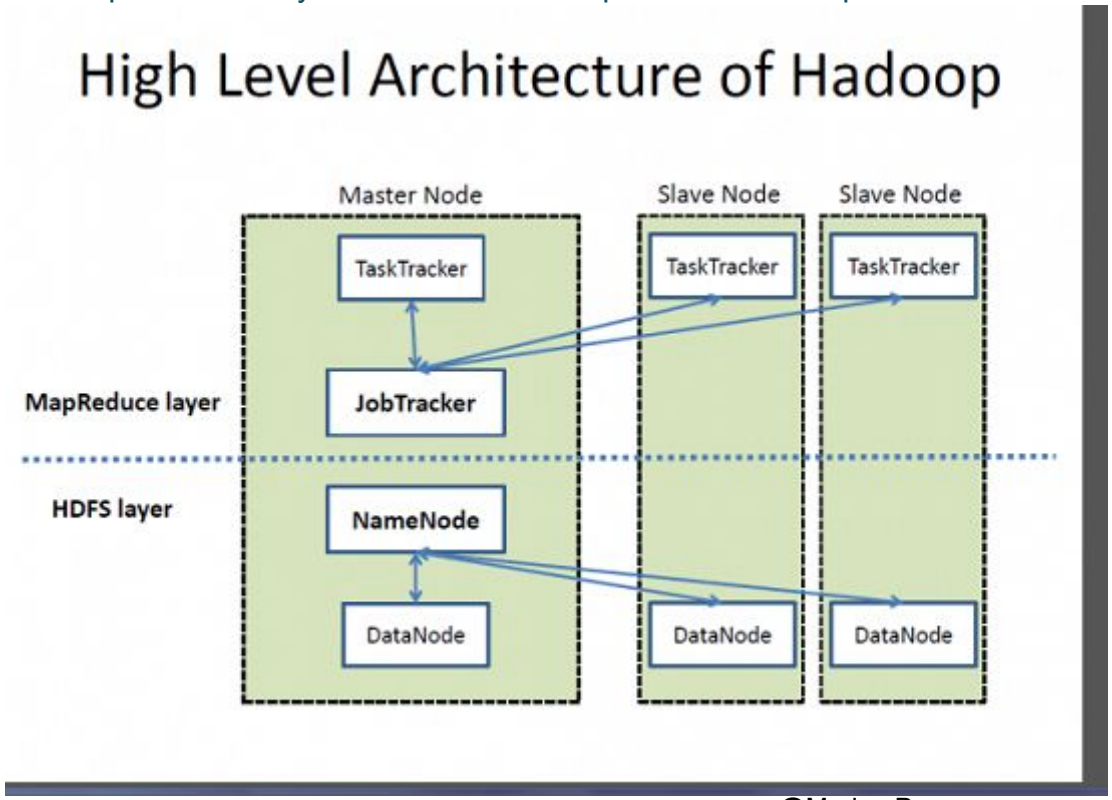
- Hadoop follows a **master slave architecture** design for data storage and distributed data processing using HDFS and MapReduce respectively

- the master node for data storage in hadoop HDFS is the **NameNode**
- the master node for parallel processing of data using Hadoop MapReduce is the **Job Tracker**

- the slave nodes in the hadoop architecture are the other machines in the Hadoop cluster which store data and perform complex computations
- every slave node has a Task Tracker daemon and a DataNode that synchronizes the processes with the Job Tracker and NameNode respectively

Distributed File Systems: HDFS

Ref: https://www.dezyre.com/article/hadoop-architecture-explained-what-it-is-and-why-it-matters/317



Distributed File Systems: HDFS

NameNode and DataNodes

- Critical components of HDFS!
- an HDFS cluster consists of a single NameNode (with one or more slave replicas), a master server that manages the file system namespace and regulates access to files by clients
- In addition, there are a number of DataNodes, usually one per node in the cluster, which manage storage attached to the nodes that they run on.
- Basically, application data and metadata are stored separately in HDFS: Master nodes store metadata, and DataNodes store application data

Distributed File Systems: HDFS

NameNode and DataNodes - continue

- HDFS exposes a file system namespace and allows user data to be stored in files
- The **NameNode** executes file system namespace operations like opening, closing, and renaming files and directories
- The **DataNodes** are responsible for serving read and write requests from the file system’s clients
- The NameNode and DataNode communicate with each other using TCP based protocols.

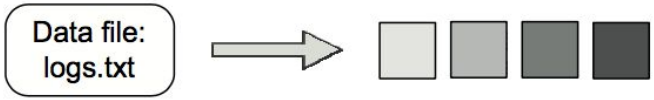
Distributed File Systems: HDFS

More on Data Replication

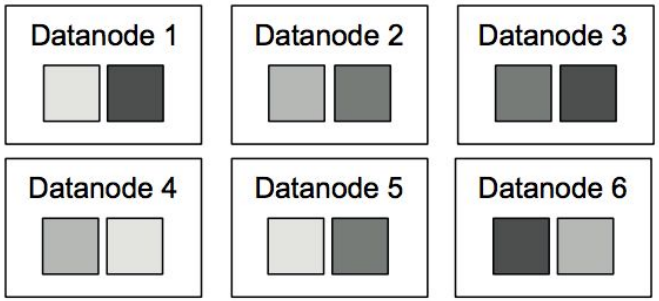
- HDFS is designed to reliably store very large files across machines in a large cluster. It **stores each file as a sequence of blocks**.
- Files/blocks are replicated for fault tolerance. The block size and replication factor are configurable per file.
- All blocks in a file except the last block are usually of the same size
- Files in HDFS are write-once (except for appends and truncates) and have strictly one writer at any time.

- The **NameNode** determines the mapping of blocks to DataNodes.
- The **DataNodes** perform block creation, deletion, and replication upon instruction from the NameNode.

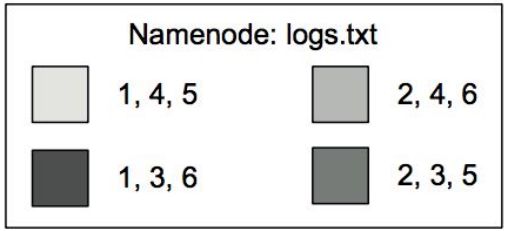
Ref: "Big Data" by Nathan Martz



1. All (typically large) files are broken into blocks, usually 64 to 256 MB.



2. These blocks are replicated (typically with 3 copies) among the HDFS servers (datanodes).



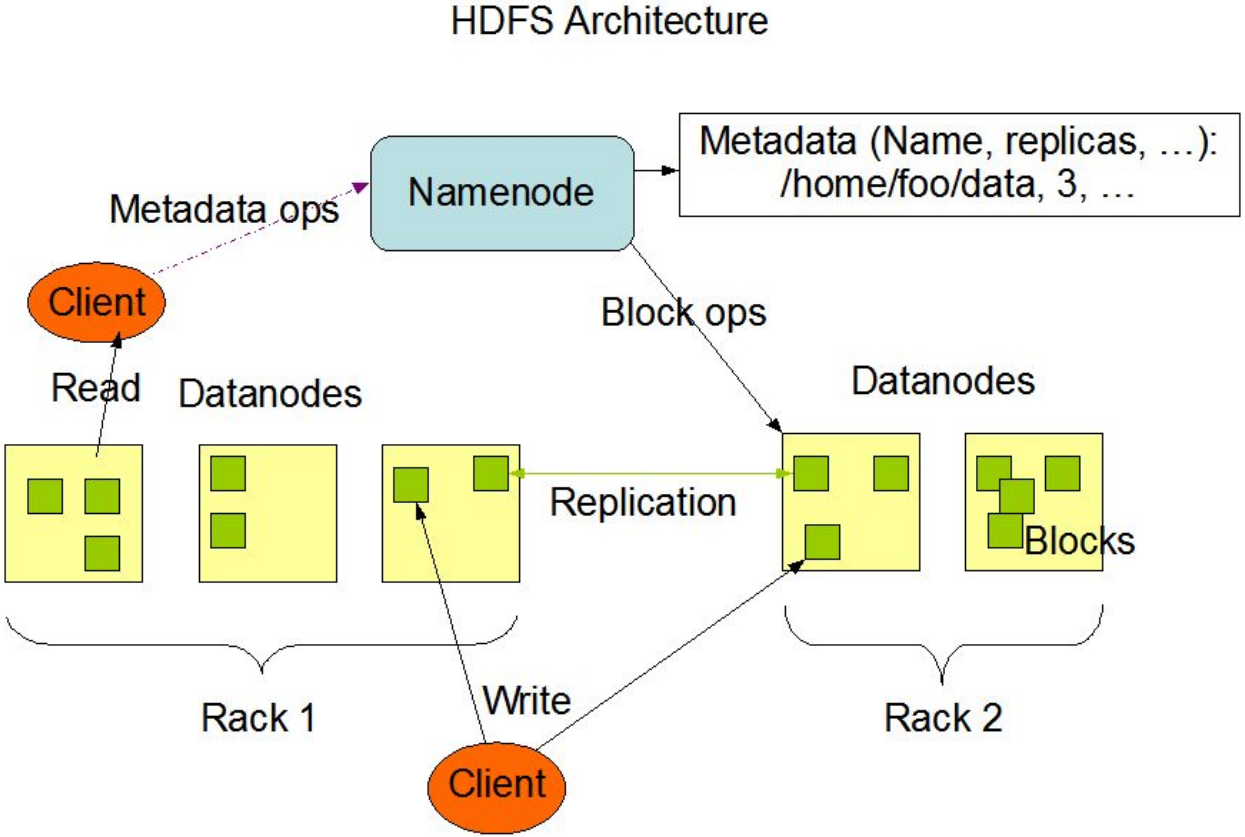
3. The namenode provides a lookup service for clients accessing the data and ensures the blocks are correctly replicated across the cluster.

@Marina Popova

Distributed File Systems: HDFS

Data Replication: NameNode and DataNode team work:

- The NameNode makes all decisions regarding replication of blocks.
- It periodically receives a Heartbeat and a Blockreport from each of the DataNodes in the cluster.
- Receipt of a Heartbeat implies that the DataNode is functioning properly.
- A Blockreport contains a list of all blocks on a DataNode.


HDFS Architecture
Namenode
Metadata (Name, replicas, …):
/home/foo/data, 3, …
Metadata ops
Client
Block ops
Read
Datanodes
Datanodes
Replication
Blocks
Rack 1
Write
Client
Rack 2

Distributed File Systems: HDFS

Replica Placement

- The placement of replicas is critical to HDFS reliability and performance
- Optimizing replica placement distinguishes HDFS from most other distributed file systems
- This is a feature that needs lots of tuning and experience
- The purpose of a **rack-aware replica placement policy** is to improve data reliability, availability, and network bandwidth utilization

Distributed File Systems: HDFS

More on Name Nodes:

- All the files and directories in the HDFS namespace are represented on the **NameNode** by **Inodes** that contain various attributes like permissions, modification timestamp, disk space quota, namespace quota and access times. NameNode maps the entire file system structure into memory.
- Two files, 'fsimage' and 'edits', are used for persistence during restarts
- **'fsimage' file** contains the Inodes and the list of blocks which define the metadata.
- **'edits' file** contains any modifications that have been performed on the content of the fsimage file.
- When the NameNode starts, fsimage file is loaded and then the contents of the edits file are applied to recover the latest state of the file system.
- over the time the edits file grows and consumes all the disk space resulting in slowing down the restart process
- This is when Secondary NameNode comes to the rescue. **Secondary NameNode gets the fsimage and edits log from the primary NameNode at regular intervals** and loads both the fsimage and edit logs file to the main memory by applying each operation from edits log file to fsimage. Secondary NameNode copies the new fsimage file to the primary NameNode and also will update the modified time of the fsimage file to fstime file to track when then fsimage file has been updated

Distributed File Systems: HDFS

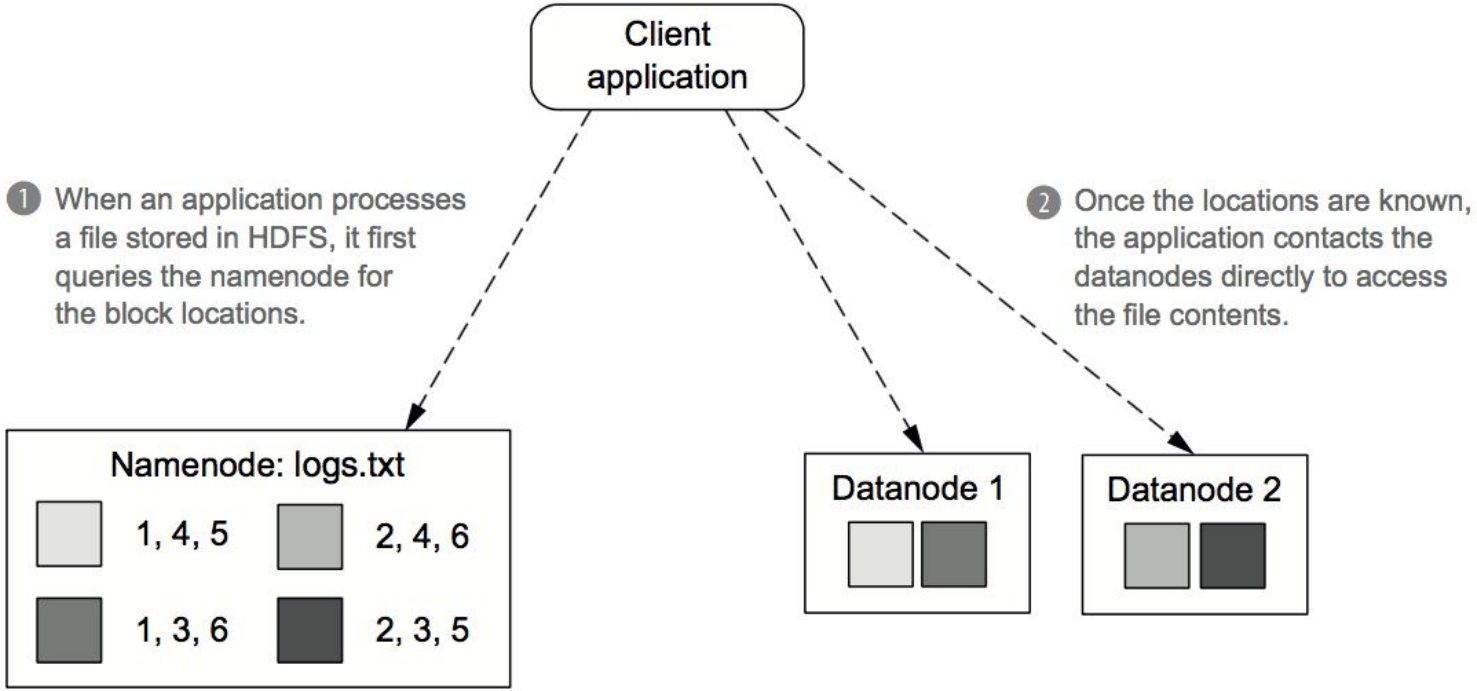
More on Data Nodes:

- DataNode manages the state of an HDFS node and interacts with the blocks
- A DataNode can perform **CPU intensive jobs** like semantic and language analysis, statistics and machine learning tasks, and **I/O intensive jobs** like clustering, data import, data export, search, decompression, and indexing. A DataNode needs lot of I/O for data processing and transfer.
- On startup every DataNode connects to the NameNode and performs a handshake to verify the namespace ID and the software version of the DataNode. If either of them does not match then the DataNode shuts down automatically.

Distributed File Systems: HDFS

What about

Clients?



Distributed File Systems: HDFS

Why such complexity?

- splitting files into blocks enables parallel processing
- horizontal scalability
- HA (High Availability)

Distributed File Systems: HDFS

How do you get HDFS setup?

- Use a VM with pre-installed Hadoop distribution - like Cloudera's QuickStart VM
  - https://www.cloudera.com/downloads/quickstart_vms/5-12.html
  -
- Use AWS EMR :
  - http://docs.aws.amazon.com/emr/latest/ManagementGuide/emr-overview.html
  -
- Install manually on your own Linux server[s]
  - https://www.linuxsecrets.com/en/entry/46performance-tips/2015/04/23/1472-installing-apache-hadoop-2-60-2-70-single-and-multi-node-redhat-centos-scientific-linux-fedora
  -

Distributed File Systems: HDFS

HDFS commands: http://hadoop.apache.org/docs/stable/

Hadoop FS shell

The File System (FS) shell includes various shell-like commands that directly interact with the Hadoop Distributed File System (HDFS) as well as other file systems that Hadoop supports, such as Local FS, HFTP FS, S3 FS, and others. The FS shell is invoked by:

bin/hadoop fs <args>

All FS shell commands take path URIs as arguments. The URI format is scheme://authority/path.

For HDFS the scheme is **hdfs**, and for the Local FS the scheme is **file**. The scheme and authority are optional. If not specified, the default scheme specified in the configuration is used.

If HDFS is being used, hdfs dfs is a synonym.
"hdfs dfs" < -- > "hadoop fs"

Distributed File Systems: HDFS

Example commands:

"hadoop fs -ls /mydir" ==
"hdfs dfs -ls /mydir"

```
[hadoop@ip-172-31-37-32 ~]$ whoami
hadoop
[hadoop@ip-172-31-37-32 ~]$ hdfs dfs -ls /user/hadoop
[hadoop@ip-172-31-37-32 ~]$ hadoop fs -ls /
Found 4 items
drwxr-xr-x   - hdfs hadoop          0 2019-09-17 16:08 /apps
drwxrwxrwt   - hdfs hadoop          0 2019-09-17 16:09 /tmp
drwxr-xr-x   - hdfs hadoop          0 2019-09-17 16:08 /user
drwxr-xr-x   - hdfs hadoop          0 2019-09-17 16:08 /var
[hadoop@ip-172-31-37-32 ~]$ hdfs dfs -ls /
Found 4 items
drwxr-xr-x   - hdfs hadoop          0 2019-09-17 16:08 /apps
drwxrwxrwt   - hdfs hadoop          0 2019-09-17 16:09 /tmp
drwxr-xr-x   - hdfs hadoop          0 2019-09-17 16:08 /user
drwxr-xr-x   - hdfs hadoop          0 2019-09-17 16:08 /var
[hadoop@ip-172-31-37-32 ~]$
```

Distributed File Systems: HDFS

Example commands:

"hdfs dfs -mkdir ~/mydir"

```
[hadoop@ip-172-31-37-32 ~]$ hdfs dfs -mkdir /user/hadoop/marina
[hadoop@ip-172-31-37-32 ~]$ hdfs dfs -ls /user/hadoop
Found 1 items
drwxr-xr-x   - hadoop hadoop          0 2019-09-17 20:03 /user/hadoop/marina
[hadoop@ip-172-31-37-32 ~]$
```

HDFS on EMR: web interfaces

Overview 'ip-172-31-37-32.us-east-2.compute.internal:8020' (active)

Started:	Tue Sep 17 12:08:05 -0400 2019
Version:	2.8.5-amzn-4, r7e6c862e89bc8db32c064454a55af74ddff73bae
Compiled:	Mon Jul 29 19:05:00 -0400 2019 by ec2-user from (HEAD detached at 7e6c862e89)
Cluster ID:	CID-b7c10892-8cc3-4ae3-873e-f9a2633be86f
Block Pool ID:	BP-1160663044-172.31.37.32-1568736482208

Summary

Security is off.

Safemode is off.

1,418 files and directories, 1,379 blocks = 2,797 total filesystem object(s).

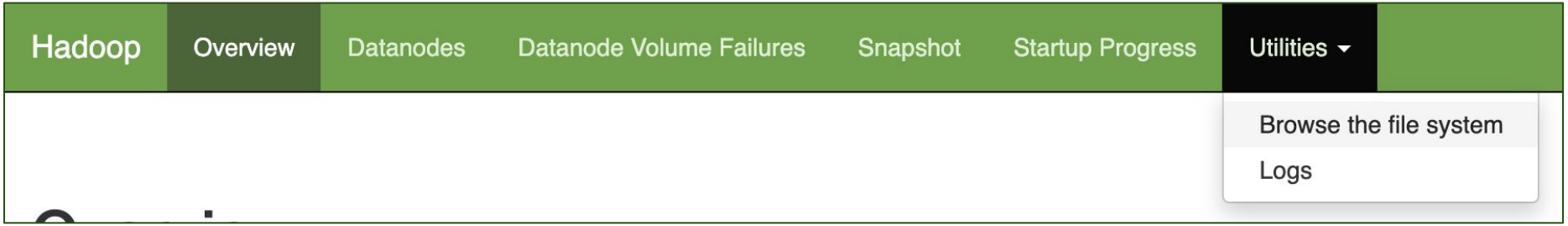
Heap Memory used 289.91 MB of 583.5 MB Heap Memory. Max Heap Memory is 1.6 GB.

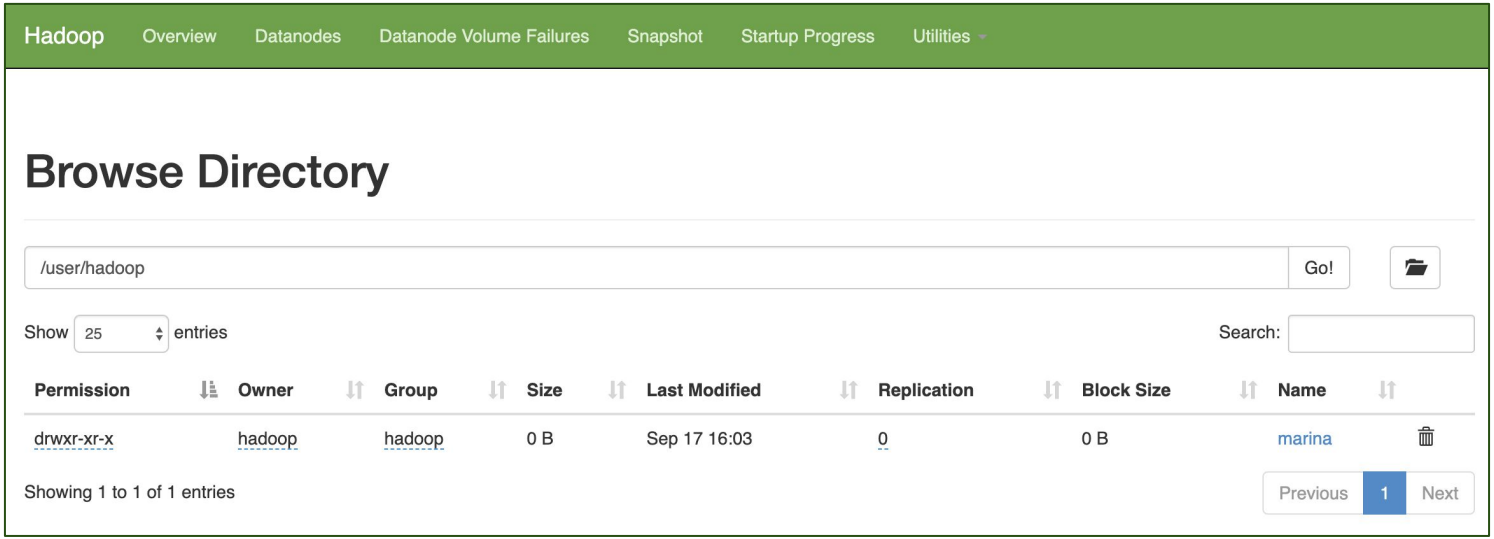
Non Heap Memory used 64.4 MB of 65.75 MB Commited Non Heap Memory. Max Non Heap Memory is <unbounded>.

Configured Capacity:	115.94 GB
DFS Used:	1.95 GB (1.68%)

@Marina Popova

HDFS on EMR: web interfaces





Useful HDFS commands

hdfs dfsadmin

```
hdfs dfsadmin [GENERIC_OPTIONS]
      [-report [-live] [-dead] [-decommissioning]]
      [-safemode enter | leave | get | wait | forceExit]
      [-saveNamespace]
      [-rollEdits]
      [-restoreFailedStorage true |false |check]
      [-refreshNodes]
      [-setQuota <quota> <dirname>...<dirname>]
      [-clrQuota <dirname>...<dirname>]
      [-setSpaceQuota <quota> [-storageType <storagetype>] <dirname>...<dirname>]
      [-clrSpaceQuota [-storageType <storagetype>] <dirname>...<dirname>]
      [-finalizeUpgrade]
      [-rollingUpgrade [<query> |<prepare> |<finalize>]]
      [-metasave filename]
      [-refreshServiceAcl]
      [-refreshUserToGroupsMappings]
      [-refreshSuperUserGroupsConfiguration]
      [-refreshCallQueue]
      [-refresh <host:ipc_port> <key> [arg1..argn]]
      [-reconfig <datanode |...> <host:ipc_port> <start |status>]
      [-printTopology]
      [-refreshNamenodes datanodehost:port]
      [-deleteBlockPool datanode-host:port blockpoolId [force]]
      [-setBalancerBandwidth <bandwidth in bytes per second>]
      [-getBalancerBandwidth <datanode_host:ipc_port>]
      [-allowSnapshot <snapshotDir>]
      [-disallowSnapshot <snapshotDir>]
      [-fetchImage <local directory>]
      [-shutdownDatanode <datanode_host:ipc_port> [upgrade]]
      [-getDatanodeInfo <datanode_host:ipc_port>]
      [-evictWriters <datanode_host:ipc_port>]
      [-triggerBlockReport [-incremental] <datanode_host:ipc_port>]
      [-help [cmd]]
```

Useful HDFS commands

hdfs dfsadmin -report

```
[hadoop@ip-172-31-37-32 ~]$ hdfs dfsadmin -report
Configured Capacity: 124490055680 (115.94 GB)
Present Capacity: 124490055680 (115.94 GB)
DFS Remaining: 122393538560 (113.99 GB)
DFS Used: 2096517120 (1.95 GB)
DFS Used%: 1.68%
Under replicated blocks: 1375
Blocks with corrupt replicas: 0
Missing blocks: 0
Missing blocks (with replication factor 1): 0
Pending deletion blocks: 0

-------------------------------------------------
Live datanodes (2):

Name: 172.31.45.227:50010 (ip-172-31-45-227.us-east-2.compute.internal)
Hostname: ip-172-31-45-227.us-east-2.compute.internal
Decommission Status : Normal
Configured Capacity: 62245027840 (57.97 GB)
DFS Used: 1039806464 (991.64 MB)
Non DFS Used: 0 (0 B)
DFS Remaining: 61205221376 (57.00 GB)
DFS Used%: 1.67%
DFS Remaining%: 98.33%
```

Useful HDFS commands

hdfs dfsadmin -printTopology

```
[hadoop@ip-172-31-37-32 ~]$ hdfs dfsadmin -printTopology
Rack: /default-rack
   172.31.45.227:50010 (ip-172-31-45-227.us-east-2.compute.internal)
   172.31.46.109:50010 (ip-172-31-46-109.us-east-2.compute.internal)

[hadoop@ip-172-31-37-32 ~]$
```

hdfs -version

```
[hadoop@ip-172-31-37-32 ~]$ hdfs version
Hadoop 2.8.5-amzn-4
Subversion git@aws157git.com:/pkg/Aws157BigTop -r 7e6c862e89bc8db32c064454a55af74ddff73bae
Compiled by ec2-user on 2019-07-29T23:05Z
Compiled with protoc 2.5.0
From source with checksum 9a7aa506e8de22caa8349aa16929
This command was run using /usr/lib/hadoop/hadoop-common-2.8.5-amzn-4.jar
[hadoop@ip-172-31-37-32 ~]$
```