Recall IaaS, PaaS, SaaS Taxonomy

On Premises
- Applications
- Data
- Runtime
- Middleware
- Oper. System
- Virtualization
- Servers
- Storage
- Networking

Managed by user
- Managed by vendor

Infrastructure (as a Service)
- Applications
- Data
- Runtime
- Middleware
- Oper. System
- Virtualization
- Servers
- Storage
- Networking

Platform (as a Service)
- Applications
- Data
- Runtime
- Middleware
- Oper. System
- Virtualization
- Servers
- Storage
- Networking

Software (as a Service)
- Applications
- Data
- Runtime
- Middleware
- Oper. System
- Virtualization
- Servers
- Storage
- Networking
Recall IaaS, PaaS, SaaS Taxonomy

• Service, Platform, or Infrastructure as a Service
  • SaaS: service is a complete application (client-server computing)
    • Not usually a programming abstraction
  • PaaS: high-level (language) programming model for cloud computer
    • Turing complete but resource management hidden
  • IaaS: low-level (language) computing model for cloud computer
    • Basic hardware model with all (virtual) resources exposed

• For PaaS and IaaS, cloud programming is needed
  • How is this different from CS101? Scale, fault tolerance, elasticity, ...
  • Example: web servers hosting a popular online store

Killer cloud app: Web servers

• Online retail stores
  • Problem: one client → fraction of one server
    • Computationally demanding: browsing products, forming/rendering web pages, managing client session state
    • Other tasks use specialized services: order taking, billing
  • But: no interaction between clients (unless inventory is low)
  • Solution: parallelism – using more cores to run web server copies

• Downside: need to deal with parallel programming
  • Elasticity requires parallel programs:
    name services, load monitoring, cluster allocation
  • Goal: replace user programming with service configuration
E.g., Obama for America Elastic Load Balancer

What about larger apps?

- Parallel programming is hard – how can cloud frameworks help?

  - A.k.a. “data-parallel”: specify op on element; apply to each in collection
    - Analogy to SIMD operation: single instruction on multiple data
  - Specify an operation on the collection as a whole
    - Union/intersection
    - Permute/sort
    - Filter/select/map
    - Reduce-reorderable:
      - ADD(1,7,2) = (1+7)+2 = (2+1)+7 = 10
    - Reduce-reordered:
      - CONCAT(“the”, “lazy”, “fox”) = “the lazy fox”

(Note the link to MapReduce... it’s no accident)
High Performance Computing Approach

- **HPC:** home for most parallel computing in 90s
- **Killer app:** simulations replacing “wet labs”
  - Examples: weather, cosmology, explosions, etc.
  - Physics same everywhere
  - Define mesh on a set of particles
  - Code simulated physics at one mesh point
  - Iterate and propagate influence
- **Bulk Synchronous Processing (BSP)**
  - Update all mesh points in parallel, use last time point values
  - Form new set of values and repeat
- **Defined “Weak Scaling” for N processors**
  - Strong scaling: same problem finishes N times faster
  - Weak scaling: N times bigger problem finishes at same time
  - Important scaling factor: set problem size to match total available memory

High Performance Computing Frameworks

- Machines cost $O(10^6 - 10^7)$ \(\rightarrow\) emphasis on maximizing utilization
  - Low-level speed and hardware-specific optimizations (esp. network)
  - Preference for expert programmers following established best practices
- **Developed MPI (Message Passing Interface) framework (e.g., MPICH)**
  - Launch N threads with library routines for everything you need:
    - Naming, addressing, membership, messaging, synchronization (barriers), transforms, physics modules, math libraries, etc.
- **Resource allocators and schedulers space-share jobs on physical cluster**
- **Fault tolerance by checkpoint/restart requiring programmer save/restore**
  - Proto-elasticity: kill N-node job and reschedule a past checkpoint on M nodes
- **Very manual, deep learning curve, few commercial runaway successes**
Broadening HPC: Grid Computing

- Grid Computing starts with commodity servers (pre Cloud)
  - 1989 “killer micros” concept to kill off supercomputers
  - Frameworks were less specialized, easier to use (and less efficient!)
    - Beowulf clusters, HTCondor or Sun/Oracle Grid Engine scheduler, Rocks OS distribution
  - For funding reasons grid emphasized geographical sharing
    - Enabling collaboration between multiple institutions
    - So authentication, authorization, single-sign-on, parallel-ftp
    - Heterogeneous workflow (run job A on machine B, job C on machine D)
  - Basic model: jobs selected from batch queue, take over cluster
  - Simplified “pile of work”: when a core becomes free, pick a task from the run queue and run to completion

Cloud Programming: Back to the future

<table>
<thead>
<tr>
<th></th>
<th>HPC/Cluster</th>
<th>Grid</th>
<th>Cloud</th>
</tr>
</thead>
<tbody>
<tr>
<td>Size</td>
<td>Small to medium</td>
<td>Large</td>
<td>Small to large</td>
</tr>
<tr>
<td>Resources</td>
<td>Homogeneous</td>
<td>Heterogeneous</td>
<td>Heterogeneous</td>
</tr>
<tr>
<td>Initial Capital Cost</td>
<td>Very high</td>
<td>High</td>
<td>Very low</td>
</tr>
<tr>
<td>Network type</td>
<td>Private (IB or proprietary)</td>
<td>Private (Ethernet)</td>
<td>Public (Ethernet)</td>
</tr>
<tr>
<td>Hardware type</td>
<td>Top-of-the-line</td>
<td>Commodity or better</td>
<td>Commodity</td>
</tr>
<tr>
<td>Typical ROI</td>
<td>Very high</td>
<td>Medium</td>
<td>High</td>
</tr>
<tr>
<td>Workload diversity</td>
<td>Low</td>
<td>Medium</td>
<td>High</td>
</tr>
</tbody>
</table>

- HPC is **hard mode**: demands too much expertise, details and tuning
- Cloud frameworks all about making parallel programming **easier**
  - Willing to sacrifice efficiency, specialize to application (not machine)
  - Users have a lot of data, need many machines, want no CS training
2005 NIST Arabic-English Competition

- Translate 100 articles
  - 2005: Google wins!
- Qualitatively better 1st entry
- Not most sophisticated approach
  - No one knew Arabic
  - Brute force statistics
- But more data and compute!
  - 200M words from UN translations
  - 1 trillion words of Arabic docs
  - 1000-processor cluster
- Can’t compete without big data

Cloud Programming Frameworks: Case Studies

- MapReduce
  - Two [Sipelstein90] operators (filter/map, reduce) as base of data parallel programming model
- DryadLINQ
  - Compile workflows of different data processing programs into schedulable processes
- Spark
  - Work to keep partial results in memory
  - Declarative programming
- TensorFlow/GraphLab
  - Specialize to iterative machine learning
Motivation

How do you perform *batch processing of large data sets using low cost clusters* with *thousands of commodity machines* which *frequently experience partial failure or slowdowns*?

- Exploit Parallelism
- Exploit Locality
  - Reduce communication
- Tolerate Failure
Batch Processing of Large Datasets on a Cluster

• A framework must offer
  • Parallel programming model
  • Data staging and movement
  • Job orchestration
  • Scheduling
  • Load Balancing
  • Communication
  • Fault Tolerance
  • Performance
  • ...

Motivation for MapReduce

• Simplify the task for humans to write parallel programs
  • Need an abstraction, or programming model, that applies to many computing problems

• Build a framework that automates
  • Parallelization
  • Scaling
  • Data transfer
  • Load balancing
  • Scheduling
  • Mitigating failure or slowdowns
  • ...

Google MapReduce

- Data parallel framework for processing Big Data on large commodity hardware
- Transparently tolerates
  - Data faults
  - Computation faults
- Achieves
  - Scalability and fault tolerance

Commodity Clusters

- MapReduce is designed to efficiently process big data using regular commodity computers

- A theoretical 1000-CPU machine would cost a very large amount of money, far more than 1000 single-CPU or 250 quad-core commodity machines

- **Premise**: MapReduce ties smaller and more reasonably priced machines together into a single cost-effective commodity cluster to solve Big Data problems
Three Strategies

- **Parallelism**
  - Break down jobs into distributed *independent* tasks to exploit parallelism

- **Scheduling**
  - Consider data-locality and variations in overall system workloads for scheduling

- **Fault Tolerance**
  - Transparently tolerate data and task failures

Hadoop MapReduce

- Hadoop is an open source implementation of MapReduce
  - ~2006

- Hadoop presents MapReduce as an analytics engine and under the hood uses a distributed storage layer referred to as Hadoop Distributed File System (*HDFS*)
  - HDFS mimics Google File System (*GFS*)

- Applications in MapReduce are represented as jobs
  - Each job encompasses several map and reduce tasks
  - Map and reduce tasks operate on data *independently* and in *parallel*
MapReduce Overview 1

Think of your solution to processing a large data set as follows:
• Read a large input data set
• Process the input data in chunks independently (e.g., filter)
• Shuffle and Sort intermediate data
• Aggregate or summarize intermediate data independently
• Write the result

MapReduce Overview 2

Think of your solution to processing a large data set as follows:
• Read a large input data set
• Map: Process the data (e.g., filter), produce intermediate data
• Shuffle and Sort intermediate data
• Reduce: Aggregate
• Write the result
MapReduce Overview 3

Think of your solution to processing a large data set as follows:

- **Read a large input data set as** *(key, value) pairs*
- **Map**: Process the data (e.g., filter), *produce (key’, value’)* as intermediate data
- **Shuffle and Sort intermediate data**
- **Reduce**: Aggregate
- **Write the result as** *(key’’, value’’)* pairs

Map Phase + Reduce Phase

- User writes **map()** and **reduce()** functions
- Read a large input data set as (key, value) pairs
- **Map**: Run **map() function in parallel** on all input (key, value) pairs and produce (key’, value’) *partitioned as intermediate data by key’*
- **Shuffle** (hashed intermediate key to a reducer), sort and merge
- **Reduce**: Run **reduce() function** on intermediate data (key’, value’)
- **Write the result as** *(key’’, value’’)* pairs
Map and Reduce Tasks

Individual Map and Reduce tasks in a MapReduce job are idempotent and have no side effects.

- Meaning, given the same input, re-executing a task will produce the same result and will not change the state.

Any task can be restarted without changing the final result.
- The results and the end state of the system are the same, if a task is executed once or many times.

MapReduce Example

- What if we want to count the number of times all house names appeared in the series?

You can have multiple aggregators, each one working on a distinct house.
MapReduce Phases

- MapReduce incorporates two phases
  - Map Phase
  - Reduce phase

Data Distribution

- In a MapReduce cluster, data is distributed to all the nodes of the cluster as it is being loaded

- An underlying distributed file systems (e.g., GFS, HDFS) splits large data files into chunks which are managed by different nodes in the cluster

- Even though the file chunks are distributed across several machines, they form a single namespace
Network Topology In MapReduce

- MapReduce assumes a tree style network topology
- Nodes are spread over different racks in one or more data centers
- The bandwidth between two nodes is dependent on their relative locations in the network topology
- The assumption is that nodes that are on the same rack will have higher bandwidth between them as opposed to nodes that are off-rack

Computing Units: Tasks

- MapReduce divides the workload into multiple independent tasks and automatically schedules them on cluster nodes
- A work performed by each task is done in isolation from other tasks
- The amount of communication which can be performed by tasks is limited mainly for scalability and fault-tolerance reasons
MapReduce Phases

- In MapReduce, splits are processed in isolation by tasks called **Mappers**.
- The output from the mappers is denoted as intermediate output and brought into a second set of tasks called **Reducers**.
- The process of reading intermediate output into a set of Reducers is known as **shuffling**.
- The Reducers produce the final output.
- Overall, MapReduce breaks the data flow into two phases, **map phase** and **reduce phase**.

Keys and Values

- The programmer in MapReduce has to specify two functions, the **map function** and the **reduce function** that implement the Mapper and the Reducer in a MapReduce program.
- In MapReduce data elements are always structured as key-value (i.e., (K, V)) pairs.
- The map and reduce functions receive and **emit** (K, V) pairs.
Input Splits

- A logical representation of data stored in HDFS blocks
- A split can contain reference to one or more HDFS blocks
  - Configurable parameter, default is 1
- Each map task processes one split
  - # of splits dictates the # of map tasks
- By default one split contains reference to one HDFS block
- Map tasks are scheduled in the vicinity of HDFS blocks to reduce network traffic

Partitions

- Map tasks store intermediate output on local disk (not HDFS)
- A subset of intermediate key space is assigned to each Reducer
  - $\text{hash(key)} \mod R$
- These subsets are known as partitions
Hadoop MapReduce: A Closer Look

Input Files

- **Input files** are where the data for a MapReduce task is initially stored.
- The input files typically reside in a distributed file system (e.g., HDFS).
- The format of input files is arbitrary:
  - Line-based log files
  - Binary files
  - Multi-line input records
  - Or something else entirely
InputFormat

- How the input files are split up and read is defined by the \textit{InputFormat}
- InputFormat is a class that does the following:
  - Selects the files that should be used for input
  - Defines the \textit{InputSplits} that break a file
  - Provides a factory for \textit{RecordReader} objects that read the file

\begin{table}
\centering
\begin{tabular}{|c|c|c|c|}
\hline
\textbf{InputFormat} & \textbf{Description} & \textbf{Key} & \textbf{Value} \\
\hline
\texttt{TextInputFormat} & Default format; reads lines of text files & The byte offset of the line & The line contents \\
\hline
\texttt{KeyValueInputFormat} & Parses lines into (K, V) pairs & Everything up to the first tab character & The remainder of the line \\
\hline
\texttt{SequenceFileInputFormat} & A Hadoop-specific high-performance binary format & user-defined & user-defined \\
\hline
\texttt{MyInputFormat} & A user-specified input format & user-defined & user-defined \\
\hline
\end{tabular}
\end{table}

- Several InputFormats are provided with Hadoop:

Files loaded from local HDFS store

InputFormat

File

File
Input Splits

- An *input split* describes a unit of data that a single map task in a MapReduce program will process.

- By dividing the file into splits, we allow several map tasks to operate on a single file in parallel.

- If the file is very large, this can improve performance significantly through parallelism.

- Each map task corresponds to a *single* input split.

RecordReader

- The input split defines a slice of data but does not describe how to access it.

- The *RecordReader* class actually loads data from its source and converts it into (K, V) pairs suitable for reading by Mappers.

- The RecordReader is invoked repeatedly on the input until the entire split is consumed.

- Each invocation of the RecordReader leads to another call of the map function defined by the programmer.
Mapper and Reducer

• The **Mapper** performs the user-defined work of the first phase of the MapReduce program

• A new instance of Mapper is created for each split

• The **Reducer** performs the user-defined work of the second phase of the MapReduce program

• A new instance of Reducer is created for each partition
  • *For each key in the partition assigned to a Reducer, the Reducer is called once*

Partitioner

• Each mapper may emit (K, V) pairs to *any* partition

• Therefore, the map nodes must all agree on where to send different pieces of intermediate data

• The **partitioner** class determines which partition a given (K,V) pair will go to

• The default partitioner computes *a hash value* for a given key and assigns it to a partition based on this result
Sort (merge)

- Each Reducer is responsible for reducing the values associated with (several) intermediate keys.
- The set of intermediate keys on a single node is automatically sorted (merged) by MapReduce before they are presented to the Reducer.

OutputFormat

- The `OutputFormat` class defines the way (K,V) pairs produced by Reducers are written to output files.
- The instances of `OutputFormat` provided by Hadoop write to files on the local disk or in HDFS.
- Several `OutputFormat` are provided by Hadoop:

<table>
<thead>
<tr>
<th><code>OutputFormat</code></th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>TextOutputFormat</td>
<td>Default; writes lines in &quot;key \t value&quot; format</td>
</tr>
<tr>
<td>SequenceFileOutputFormat</td>
<td>Writes binary files suitable for reading into subsequent MapReduce jobs</td>
</tr>
<tr>
<td>NullOutputFormat</td>
<td>Generates no output files</td>
</tr>
</tbody>
</table>
Combiner Functions

- MapReduce applications are limited by the bandwidth available on the cluster.
- It pays to minimize the data shuffled between map and reduce tasks.
- Hadoop allows users to specify a combiner function (just like the reduce function) to be run on a map output only if the reduce function is commutative and associative.

MapReduce In a Nutshell

- MapReduce incorporates two phases, Map and Reduce.
The Shuffle in MapReduce

Job Scheduling in MapReduce

- In MapReduce, an application is represented as a job
- A job encompasses multiple map and reduce tasks
- Job schedulers in MapReduce are pluggable
- Hadoop MapReduce by default **FIFO scheduler** for jobs
  - Schedules jobs in order of submission
  - Starvation with long-running jobs
  - No job preemption
  - No evaluation of job priority or size

Will be presented in detail in the near future!
Task Scheduling in MapReduce

- MapReduce adopts a **master-slave architecture**
- The master node in MapReduce is referred to as **Job Tracker (JT)**
- Each slave node in MapReduce is referred to as **Task Tracker (TT)**
- MapReduce adopts a **pull scheduling** strategy rather than a push one
  - I.e., JT does not push map and reduce tasks to TTs but rather TTs pull them by making requests

Map and Reduce Task Scheduling

- Every TT sends a **heartbeat message** periodically to JT encompassing a request for a map or a reduce task to run

I. **Map Task Scheduling**:

- JT satisfies requests for map tasks via attempting to schedule mappers in the **vicinity** of their input splits (i.e., it considers locality)

II. **Reduce Task Scheduling**:

- However, JT simply assigns the next yet-to-run reduce task to a requesting TT regardless of TT’s network location and its implied effect on the reducer’s shuffle time (i.e., it does not consider locality)
Fault Tolerance in Hadoop

- Data redundancy
  - Achieved at the storage layer through replicas (default is 3)
  - Stored at physically separate machines
  - Can tolerate
    - Corrupted files
    - Faulty nodes
  - HDFS:
    - Computes checksums for all data written to it
    - Verifies when reading
- Task Resiliency (task slowdown or failure)
  - Monitors to detect faulty or slow tasks
  - Replicates tasks
Task Resiliency

- MapReduce can guide jobs toward a successful completion even when jobs are run on a large cluster where probability of failures increases.
- The primary way that MapReduce achieves fault tolerance is through restarting tasks.
- If a TT fails to communicate with JT for a period of time (by default, 1 minute in Hadoop), JT will assume that TT in question has crashed.
  - If the job is still in the map phase, JT asks another TT to re-execute all Mappers that previously ran at the failed TT.
  - If the job is in the reduce phase, JT asks another TT to re-execute allReducers that were in progress on the failed TT.

Speculative Execution

- A MapReduce job is dominated by the slowest task.
- MapReduce attempts to locate slow tasks (stragglers) and run redundant (speculative) tasks that will optimistically commit before the corresponding stragglers.
- This process is known as speculative execution.
- Only one copy of a straggler is allowed to be speculated.
  - Whichever copy of a task commits first, it becomes the definitive copy, and the other copy is killed by JT.
Locating Stragglers

- How does Hadoop locate stragglers?
  - Hadoop monitors each task progress using a progress score between 0 and 1 based on the amount of data processed.
  - If a task’s progress score is less than (average – 0.2), and the task has run for at least 1 minute, it is marked as a straggler.

MapReduce Applications

<table>
<thead>
<tr>
<th>Data Pattern</th>
<th>Shuffle Data/Map Input Ratio</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>Map Input</td>
<td>Shuffle Data</td>
<td></td>
</tr>
<tr>
<td>▶️</td>
<td>◯</td>
<td>0</td>
</tr>
<tr>
<td>▶️</td>
<td>◯</td>
<td>&lt;&lt; 1</td>
</tr>
<tr>
<td>▶️</td>
<td>◯</td>
<td>= 1</td>
</tr>
<tr>
<td>▶️</td>
<td>◯</td>
<td>&gt;&gt; 1</td>
</tr>
</tbody>
</table>
Grep Example

TeraSort Example
DryadLINQ

- **Goal:** Simplify writing data-parallel code
  - Added compiler support for imperative and declarative ops on data
  - Extends the MapReduce model by collectively optimizing workflows
- **Data flows between processes**
  - Graph abstraction: Expressions on data represent workflow between processes
- **Interesting part of the compiler operates on the expressions**
  - Rewrite execution plan to execute faster
  - Inspired by database query optimizations

DryadLINQ

- **Data flowing through a graph abstraction**
  - Vertices are programs (possibly different with each vertex)
  - Edges are data channels (pipe-like)
  - Requires programs to have no side-effects (no changes to shared state)
- **Compiler operates on expressions, rewriting execution sequences**
  - Knows how to partition sets (hash, range and round robin) over nodes
  - Doesn’t always know what processes do, but accepts hints from users
  - Can auto-pipeline, remove redundant partitions, reorder partitions, etc.
Spark: Optimize MR for iterative apps

- MapReduce uses disks for input, tmp, and output
- Want to use memory mostly
- Machine Learning apps iterate over same data to “solve” something
  - Way too much use of disk when the data is not giant
- Spark is MR rewrite: more general (dryad-like graphs of work), more interactive (Scala interpreter), more efficient (in-memory)

Spark Resilient Distributed Datasets (RDDs)

- Spark programs are functional, deterministic → Same input = same result
  - Basis of selective re-execution and automated fault-tolerance
- RDDs: abstraction for parallel, fault-tolerant computation
  - Splits a set into partitions for workers to parallelize operation
  - Fault-tolerance through lineage graphs showing how to recompute data
- Store invocation (code and args) with inputs as a closure
  - Treated as “future” contract: compute now/later at system’s choice (lazy)
  - If code inputs already at node X, “args” is faster to send than results
    - Futures can be used as compression on wire and in replica nodes
Spark Resilient Distributed Datasets (RDDs)

- Many operators are built-ins (well-known properties like Dryad)
  - Spark automates transforms when pipelining multiple built-in operations
- Spark is lazy – only specific operators force computations
  - E.g., materialize in file system (rdd.persist(...);
  - Build programs interactively, computing only when and what user needs
  - Lineage is chain of invocations: future on future ... delayed compute
- Replication/FT: ship and cache RDDs on other nodes
  - Can recompute everything there if needed, but mostly don’t
  - Save space in memory on replicas and network bandwidth
  - Need entire lineage to be replicated in non-overlapping fault domains

Example of Spark “combining functions”

\[
\text{rdd}_x.\text{map(}\text{foo}).\text{map(}\text{bar})
\]

- Function \text{foo()}: take in record \(x\), output record \(y\)
- Function \text{bar()}: take in record \(y\), output record \(z\)
- Spark creates function \text{foo_bar()}: take in record \(x\), output a record \(z\)

Can be optimized
Next up

• Project 2, Part 1: Out today (more from Ankush now!)
• Wednesday: Cloud Storage (Part I)

Readings

Project 2-1/2-2
ETL Processing on AWS using Apache Spark

15-719/18-709
Advanced Cloud Computing

Learning Goals

• Develop a distributed program using Apache Spark that preprocesses a large web crawl dataset.

• Deploy and execute Apache Spark programs on AWS.
**Topic Modeling**

- A machine learning technique for discovery of hidden semantic structures in a text body

---

**Topic Modeling: Process**

1. **Preprocessing**
2. **Model Training**
3. **Training Input**
4. **Topic Model**

---

Kevin Hsieh, Ankush Jain
Text Corpus Preprocessing

• Extract English text from the archived web pages
• Filter out invalid and useless words
• Map words (ASCII strings) to integers for efficient indexing
• Compute the corpus statistics.

Apache Spark

• A fast and general engine for large-scale data processing
• One of the most popular data engines
Spark and RDDs

- RDD = Resilient Distributed Dataset
- A distributed memory abstraction that lets programmers perform in-memory computations on large clusters

Project Goal

- Write a **Apache Spark + Python** program to preprocess a large text corpus. Output the processed corpus and its statistics.
- Your scores for **P2.1** are based on the **correctness** of your program.
- Your scores for **P2.2** are based on the **performance** of your program.
Where to Start

- Go through the Spark primer on TPZ
- Go through the project write-up on TPZ
- Download the starter package
  - Script to download a text corpus
  - Submitter
  - Reference output *(approximate match)*
- Use the student AMI to develop
- Use the `spark-ec2-setup` tool to launch a cluster

Notes on Launching a Spark Cluster

- We use *spot instances* for this project. You need to set a proper bidding price for spot instances
- You need to tag these instances using the cluster setup tool
Submission and Grading – P2.1

• For each test configuration
  1. Launch a Spark cluster
  2. Download the text corpus to HDFS
  3. Run the submitter with proper arguments on the master node

• Test configurations
  • 1 WARC file, 4 slave instances, 30 pt
  • 100 WARC files, 8 slave instances, 50 pt
• Coding quality (manually grading): 20 pt

Submission and Grading – P2.2

• Optimize P2.1 code

• Test configurations
  • 100 WARC files, 8 slave instances, 25 mins, 20 pt
  • 400 WARC files, 8 slave instances, 80 mins, 20 pt
  • 400 WARC files, 16 slave instances, 45 mins, 20 pt
• Coding quality (manually grading): 20 pt
General Tips

- Request spot instance limit increase right away
  - m4.xlarge, us-east-1, type: spot, limit: 20
- Start small, start early
- Don’t use the DataFrame API

Q & A