Programming in the Cloud

Gabriel Scalosub

Based on various papers/resources (see list at the end)

Outline

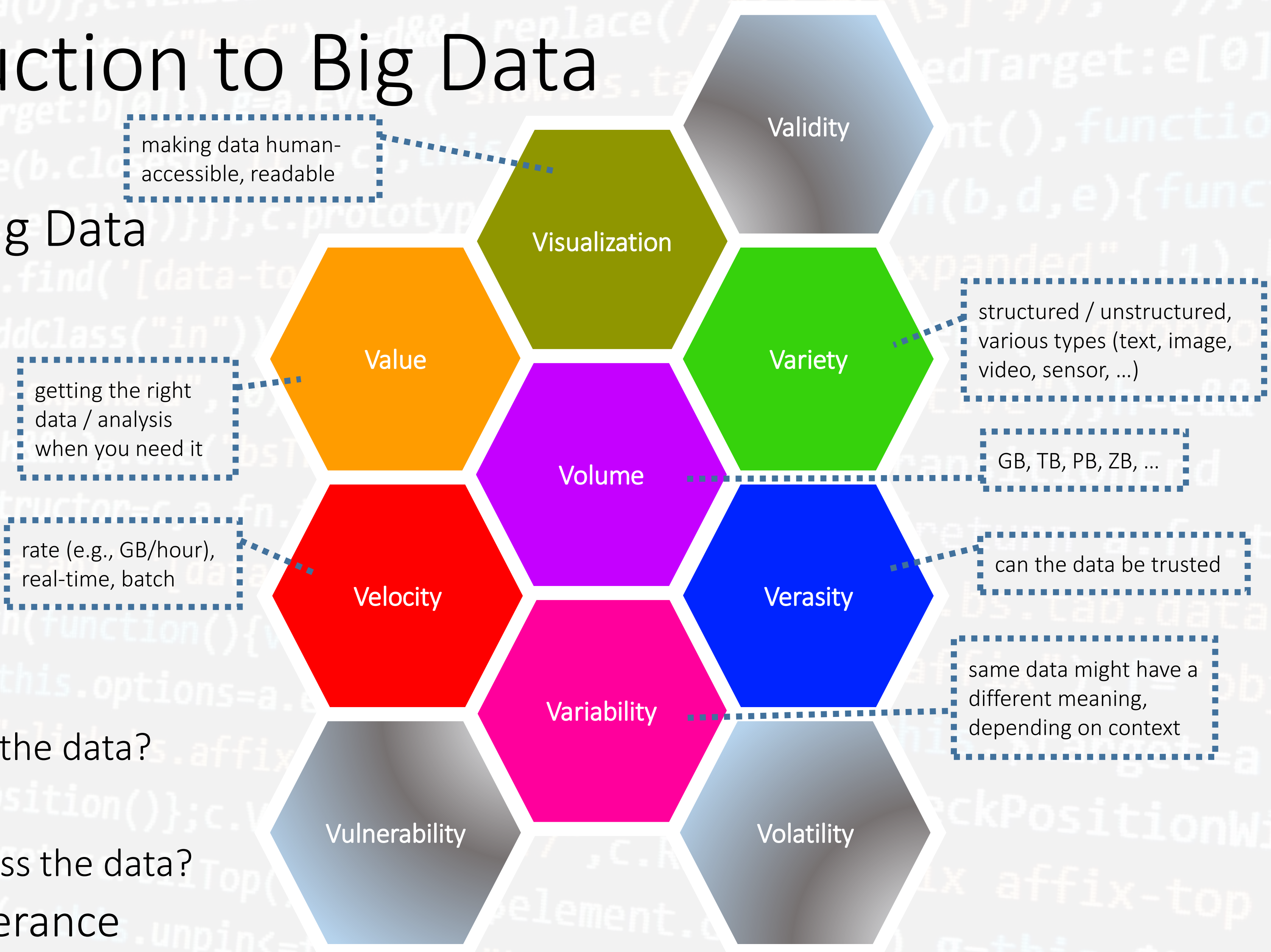
- Cloud Programming Paradigms
  - (Ultra) Quick Introduction to Big Data
  - Hadoop and MapReduce

Outline

- Cloud Programming Paradigms
  - (Ultra) Quick Introduction to Big Data
  - Hadoop and MapReduce

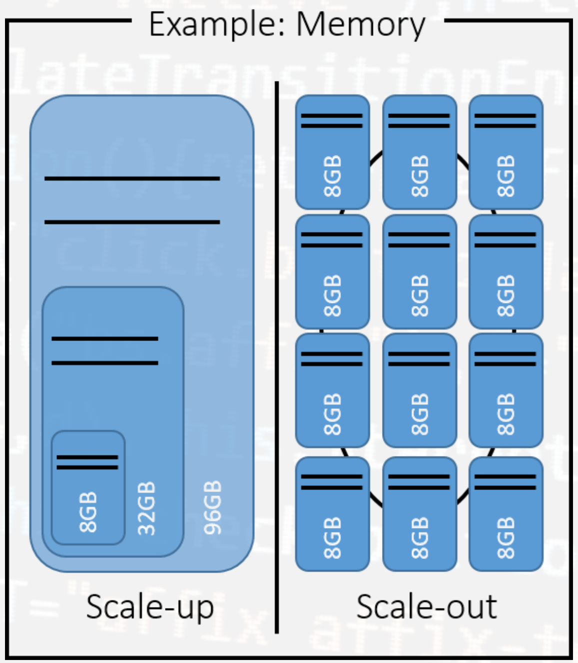
(Short) Introduction to Big Data

- Characterization of Big Data
  - V…
  - V^3
  - V^5
  - V^7
  - V^{10}, …
- Main challenges:
  - Storage
    - How/where to store the data?
  - Processing:
    - How/where to process the data?
  - Scalability & fault tolerance



(Ultra) Quick Introduction to Big Data

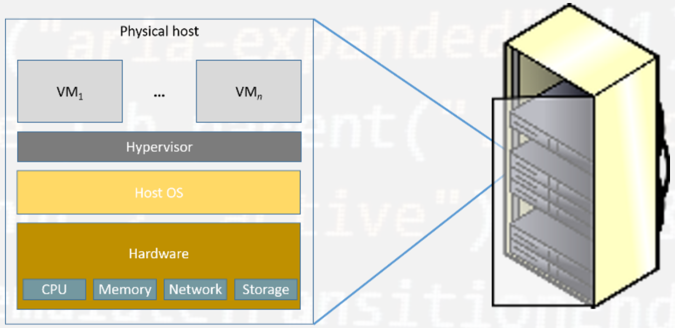
- (Classical) vertical scaling vs. horizontal scaling*
  - Vertical: stronger HW, faster CPUs, larger memory, larger BW, larger storage
    - Stops being scalable beyond some point
  - Horizontal: parallelizing code, use more COTS HW
    - If done correctly, can be scaled unlimitedly
    - No problem handling/analyzing many TBs of data
- Distributed file-system
  - Store data "everywhere" / "all-over"
    - Easy/fast (parallel) access
    - Fault-tolerance

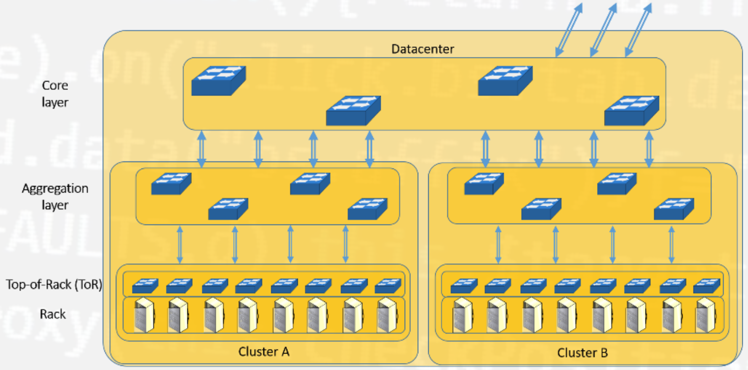


* Previous lecture…

(Ultra) Quick Introduction to Big Data

- Processing locality
  - Goal: avoid costly data transfers
    - time, congestion, $$$...
  - Perform data processing close to where data is stored
    - Same host
    - Same rack
    - Same cluster
  - Big Data: stored "everywhere" / "all-over"
  - Make processing tasks "local"
    - Process only small chunks of data available "close by"
    - Minimize data transfers
  - We've seen similar things before:
    - Traffic/data aware placement...



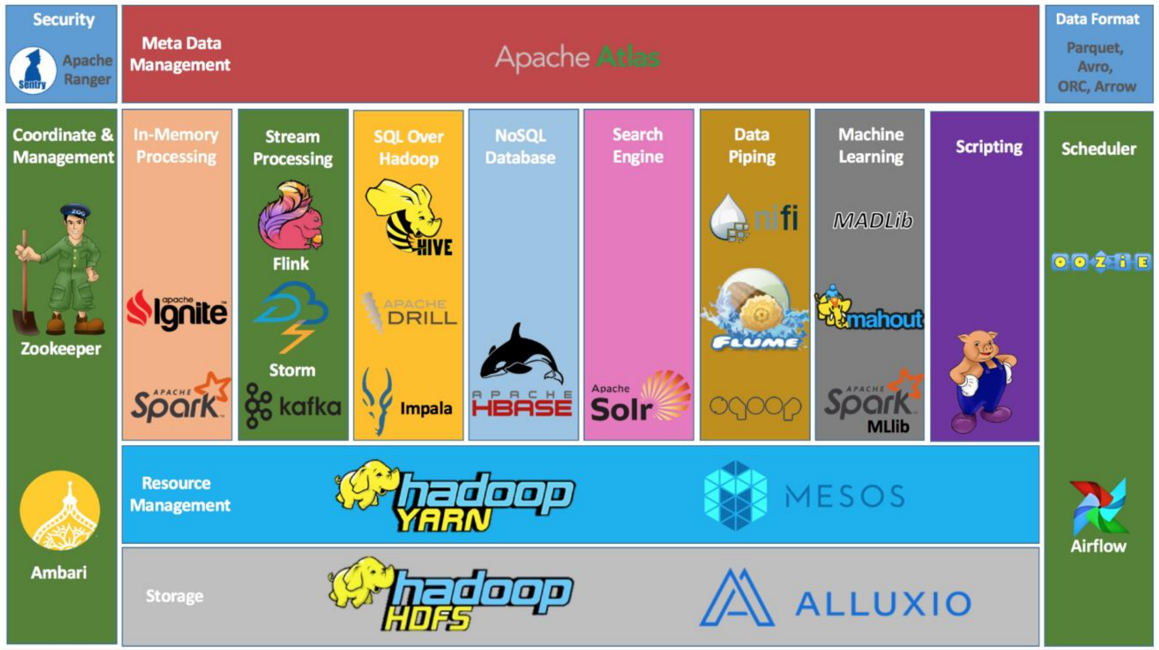


Outline

- Cloud Programming Paradigms
  - (Ultra) Quick Introduction to Big Data
  - Hadoop and MapReduce

Hadoop and MapReduce

- Hadoop: Big Data Framework
  - Hadoop distributed file system (HDFS)
    - Developed and open-sourced by Yahoo (2010)
      - Following initial design of the GFS published by Google (2003)
  - MapReduce programming paradigm
    - Published by Google (2004)
- Extended by YARN (Hadoop 2-)
  - Yet Another Resource Negotiator
    - Manage computing resources
    - Processing tasks scheduling
- Many extensions of the ecosystem
  - Hive, Spark, Kafka, …



Du, “Apache Hive Essentials”, 2018

Hadoop Distributed File System (HDFS)

- Hadoop cluster
  - Racks of storage (disks), connected to the network
  - Master-slave architecture
    - Master: namenode
    - Slave(s): datanode(s)
- namenode
  - Maintains entire file namespace structure in memory
    - Great for not-too-many very large files, ill-suited for great-many (billions) small files
- datanode
  - Stores actual data, in blocks
- Might look familiar if you know P2P…

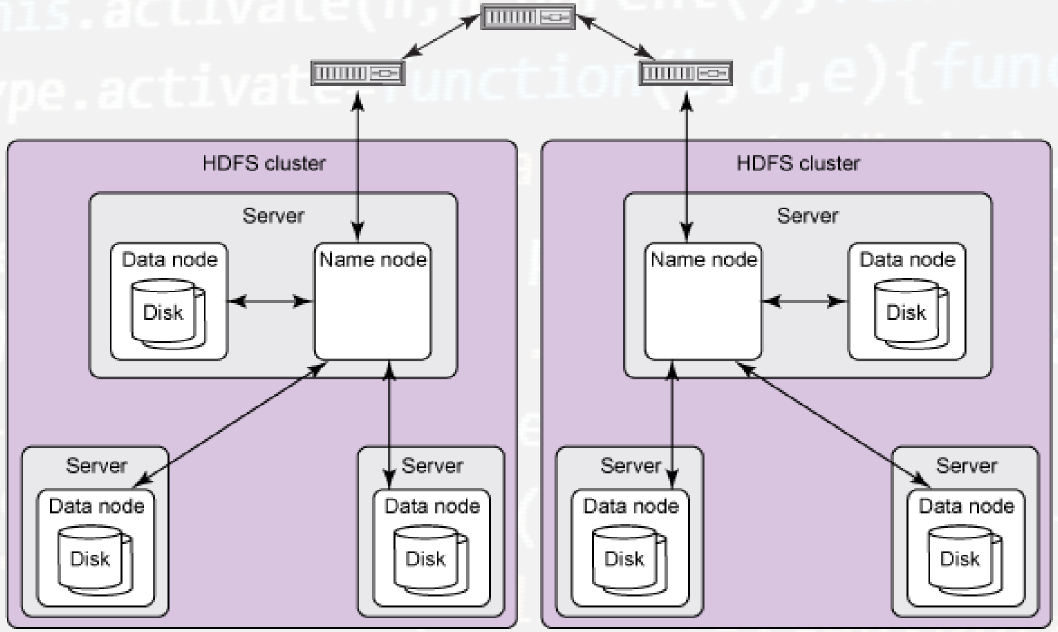


Image: Hanson (2011)

Hadoop Distributed File System (HDFS)

- HDFS data blocks
  - Data stream broken into blocks
  - Block size: 128MB (default in YARN)
    - Much larger than disk block size (~KB)
    - Makes disk seek-time negligible compared to disk transfer rate
    - Example:
      - Disk with 10ms seek-time, 100MB/s transfer rate
      - Goal: Find BlockSize (MB) s.t. seek-time is 1% of transfer time
      - 0.01 * (BlockSize/100) = 10ms ⇒ BlockSize ~ 100MB

true ballpark figures

Targeting:
- Streaming file access
- Not random-access

- Each block stored in multiple disks
  - Redundancy for parallel read & fault tolerance
  - Default #copies: 3
    - May be increased for higher parallelizability

Hadoop Distributed File System (HDFS)

- HDFS namenode
  - Manages FS namespace, and all FS tree, files, directories, and metadata
  - FS structure Information stored persistently on local disk
    - Image file, edit-log file (occasionally merged into image file)
  - Files blocks locations ephemeral (i.e., in memory only)
    - Reconstructed from datanodes, sending periodical reports to namenode
  - Single point of failure…
    - Addressed by HDFS high-availability (HA): standby namenode
- HDFS datanode
  - Stores file blocks, managed-by / reports-to namenode
- HDFS client
  - Issues requests (read/write) to namenode (CLI / RESTful API)

Hadoop Distributed File System (HDFS)

- Client performing Read
  - NN gives list of DNs with file first blocks
    - Sorted by proximity to client
    - Topology aware!
  - Client constantly reads from FSDIS
    - Oblivious to file/DN locations
  - FSDIS opens/closes DNs connections
    - Requests DNs of subsequent blocks directly from NN
    - Verifies checksums for blocks received from DNs
    - No unnecessary open connections
      - sequential

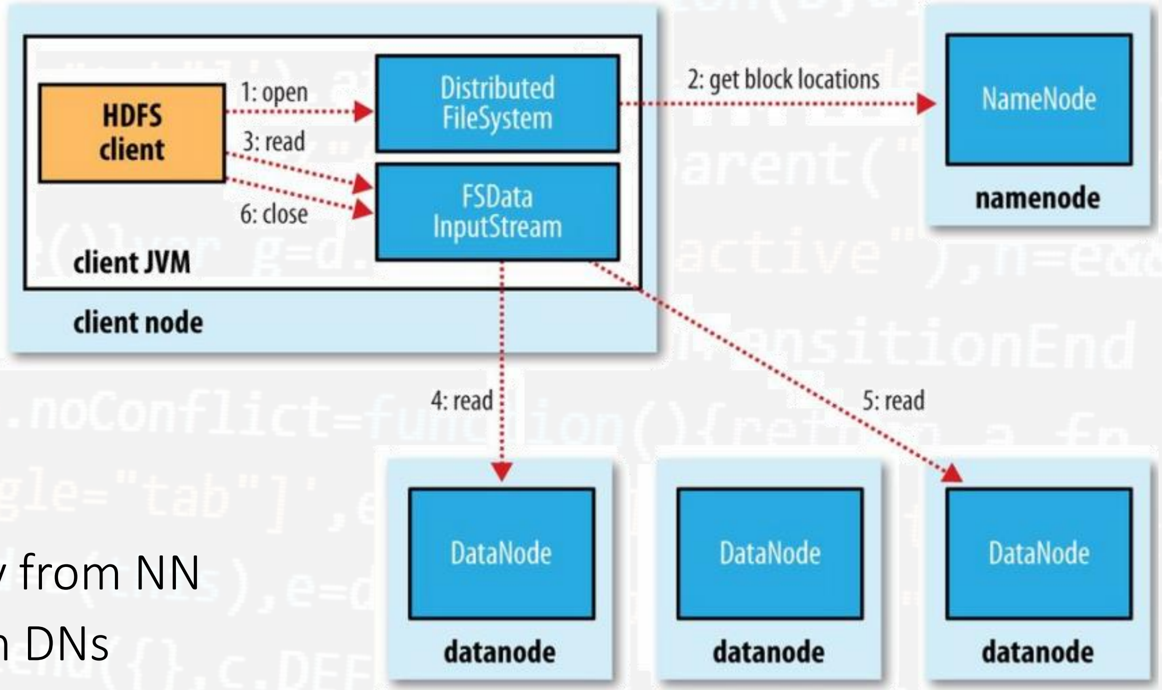


Image: White (2015)

Hadoop Distributed File System (HDFS)

- Client performing Write
  - Client request causes NN to create file
    - NN verifies permissions, existence, etc.
  - NN provides pipeline of nodes
    - Will store copies of the block
    - New pipeline whenever new block required
  - Client writes data packets to local buffer
    - Consumed by FSDOS
    - FSDOS awaits ack from all datanodes for clearing packets
      - Strict consistency
  - Failures handled by NN
    - As long as block is available on some DN
    - Asynchronous replication / fault recovery

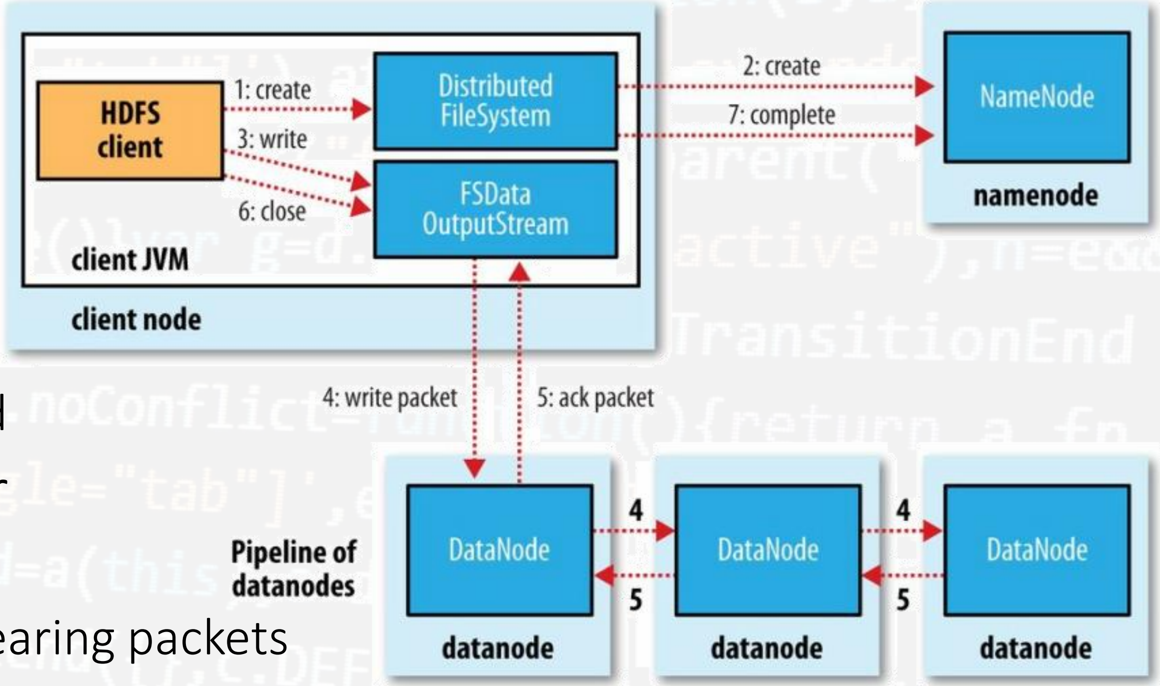


Image: White (2015)

Hadoop Distributed File System (HDFS)

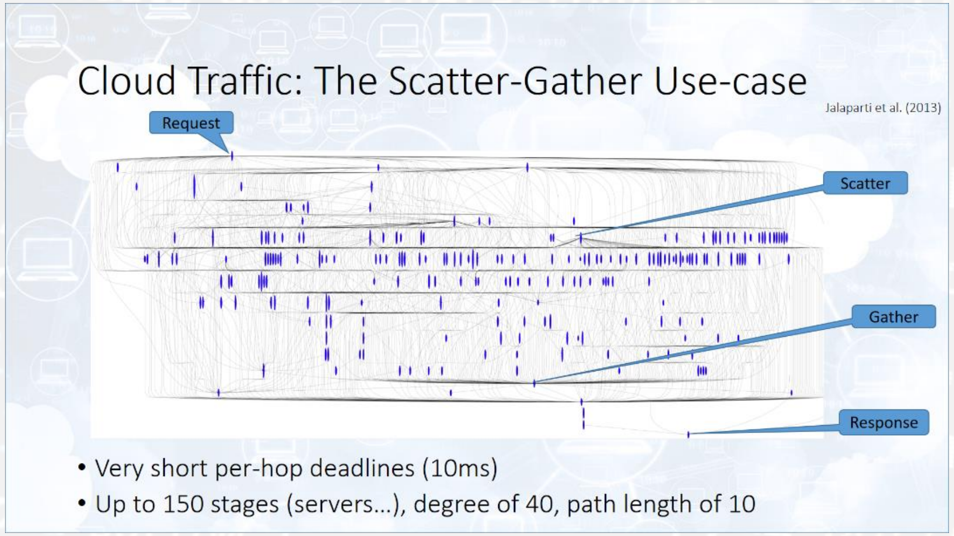
- Features
  - Abstracts away the physical underlying storage
  - Meant to handle very large files (GB, TB)
    - Not ideal for small files
  - Distributed: network-based file system
  - Streaming data access
    - Write-once (and append), Read-many-times
  - Scalable & cost-efficient: horizontal scaling
    - Just add more machines…
  - Fault-tolerance: replication & recovery
  - High-throughput: supports high rates
    - Latency is a secondary objective

Programming over Hadoop

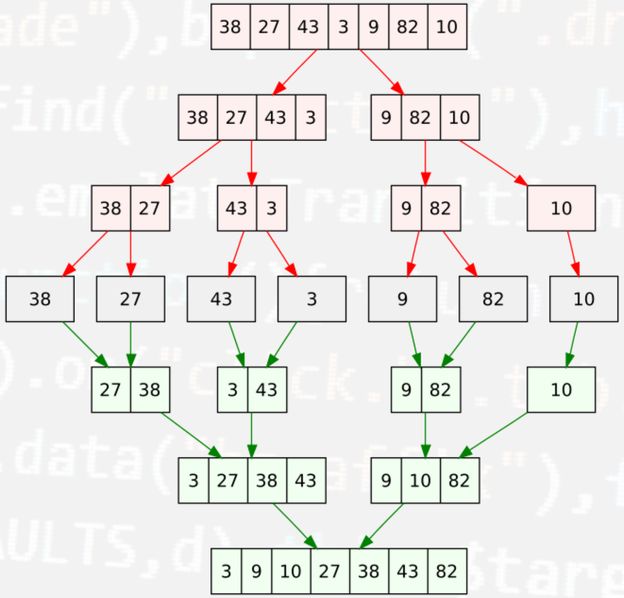
- And now, how to go beyond just read-write

MapReduce: Introduction

- Divide and Conquer
  - Split main big task into smaller subtasks
  - Solve smaller subtasks recursively
    - If small enough, solve directly
  - Combine solutions of subtasks into solution of main task
- Scatter-gather



- Example: merge-sort



MapReduce Example: Word Count

- The Word Count problem:
  - Input: (very big) file
  - Output: number of times each word appears in file
- Case 1: File size > Memory, $\{(\text{word,count})\}_{\text{word}}$ < Memory
  - Maintain hash-table with words as keys, update counter as you scan file
- Case 2: File size > Memory, $\{(\text{word,count})\}_{\text{word}}$ > Memory
  - Unix pipelining:

    extract_words(file) | sort | uniq –c

    - “**extract_words(file)**” emits words in file, one by one (separate lines)
    - Unix “**sort**” actually performs a variant of merge-sort under-the-hood, with temp files
    - Unix “**uniq –c**” outputs the count for each unique entry

MapReduce Example: Word Count

- From Unix pipelining to MapReduce

extract_words(file) | sort | uniq –c

- Map()
  - Scan input file, record-by-record
    - Record=line, in our case
  - Extract keys from each record
    - Words, in our case
    - (Also “count” values, trivially 1, in our case)

- Reduce()
  - Aggregate / summarize / filter / transform
    - Count unique words, in our case
  - Write result to file

- Group by key
  - Sort&Shuffle
    - Well, just sort, in our case

MapReduce Paradigm

- Map – Group (Sort & Shuffle) – Reduce
  - Input: set of (k, v) key-value pairs

provided by the MR framework

provided by the programmer

- $\text{Map}(k, v) \rightarrow \text{list}((k', v'))$
  - Single Map() call for each key-value pair in input
  - Map extracts list of intermediate key-value pairs
  - Potentially unrelated to key-value pairs in input
- $\text{Reduce}\big(k', \text{list}(v')\big) \rightarrow \text{list}((k', v''))$
  - Single Reduce() call for each unique key k'
  - All intermediate key-value pairs with same key are reduced together(!)

- Group (Sort & Shuffle)
  - Gets all(!) Map() lists corresponding to some set of keys k'
    - Possibly from different Map() calls
  - Sorts all key-value pairs in lists by key
  - Shuffles key-value-lists pairs to reducers
    - Usually with some hash-function

      $$h(k') \bmod R$$
    - Load balancing
    - Each Reduce() gets all lists with same key
    - Sometimes referred as “partitioning”

MapReduce Paradigm

- Map – Group (Sort & Shuffle) – Reduce
  - Input: set of (k, v) key-value pairs
  - $\mathrm{Map}(k, v) \rightarrow \mathrm{list}((k', v'))$
    - Single Map() call for each key-value pair in input
    - Map extracts list of intermediate key-value pairs
    - Potentially unrelated to key-value pairs in input
  - $\mathrm{Reduce}\big(k', \mathrm{list}(v')\big) \rightarrow \mathrm{list}((k', v''))$
    - Single Reduce() call for each unique key k'
    - All intermediate key-value pairs with same key are reduced together(!)

provided by the programmer

Word Count

```
map(key, value):
  // key: document name, value: document contents
  for each word w in value:
    EmitIntermediate(w, "1");
```

```
reduce(key, values):
  // key: a word, values: a list of counts
  result = 0;
  for each v in values:
    result += ParseInt(v);
  Emit(AsString(result));
```

MapReduce Paradigm

- Map – Group (Sort & Shuffle) – Reduce
  - Input: set of (k, v) key-value pairs

provided by the programmer

  - $\mathrm{Map}(k, v) \rightarrow \mathrm{list}((k', v'))$
    - Single Map() call for each key-value pair in input
    - Map extracts list of intermediate key-value pairs
    - Potentially unrelated to key-value pairs in input
  - $\mathrm{Reduce}\big(k', \mathrm{list}(v')\big) \rightarrow \mathrm{list}((k', v''))$
    - Single Reduce() call for each unique key k'
    - All intermediate key-value pairs with same key are reduced together(!)

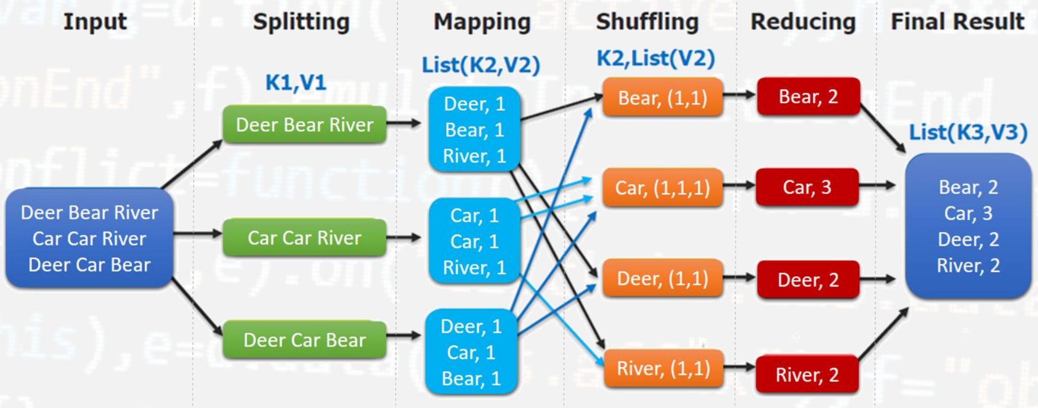


Image: Pattamsetti (2017)

MapReduce: Some Finer Details

- Combiners
  - Deals with mappers producing many key-value pairs with same key
    - E.g., very common words (think “The”, “a”, etc.)
  - Mapper does pre-aggregation before grouping
    - Similar to Reduce, but partial view
    - Possible for associative operations
      - E.g., sum, count, max, etc.
    - Minimizes mapper-reducer traffic

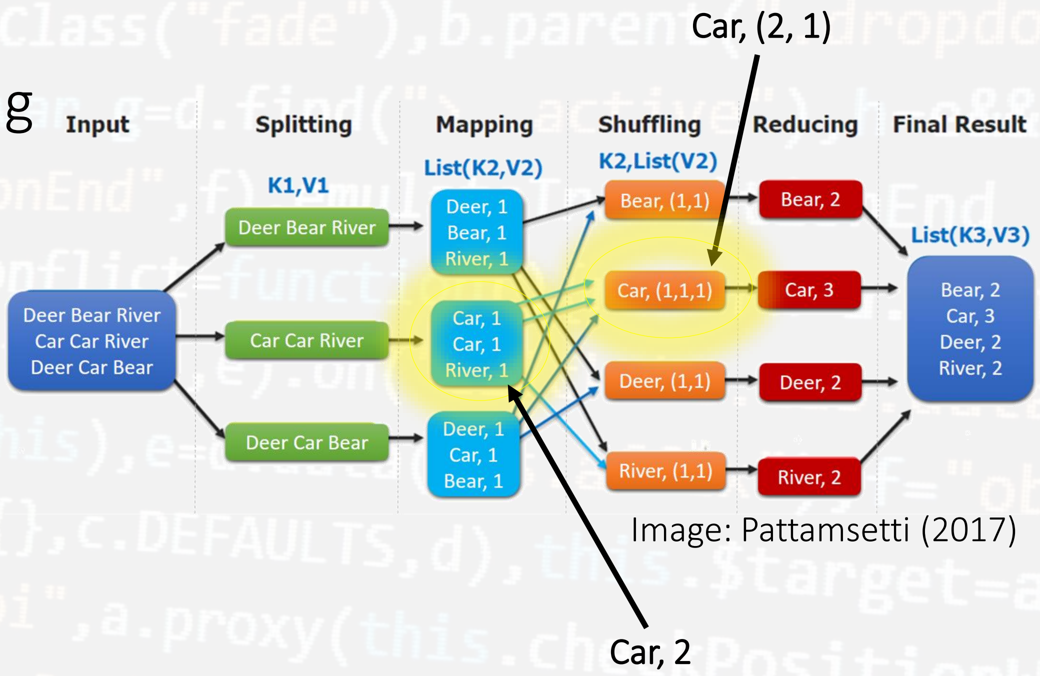


Image: Pattamsetti (2017)

MapReduce: Some Finer Details

- Combiners
  - Deals with mappers producing many key-value pairs with same key
    - E.g., very common words (think “The”, “a”, etc.)
  - Mapper does pre-aggregation before grouping
    - Similar to Reduce, but partial view
    - Possible for associative operations
      - E.g., sum, count, max, etc.
    - Minimizes mapper-reducer traffic
- Dealing with stragglers
  - MR is “bottlenecked” by slowest mapper/reducer
  - Usually solved by spawning additional mappers/reducers
    - And taking the result from the first that finishes
- Failures
  - Reset tasks (Map/Reduce)

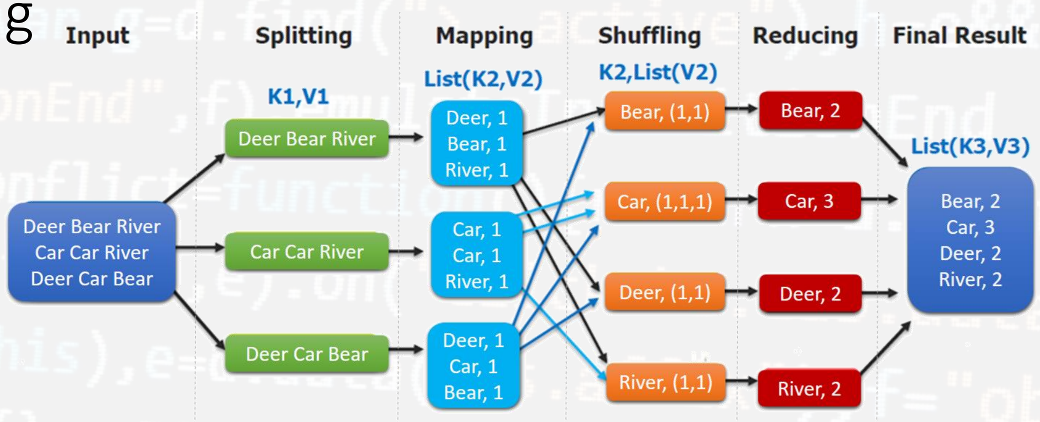


Image: Pattamsetti (2017)

MapReduce: Runtime & Workflow

- Master node coordinating MR
  - Client issues job
  - MR handles spawning mapper / reducer tasks across machines
    - Mappers scheduled close to stored data
  - Mappers intermediate results kept in local FS
    - Master is notified, directs reducers to intermediate results location
  - Synchronous / map-reduce barrier
    - Mappers should finish before reducers kick-in
  - Managing failures and inter-process communications

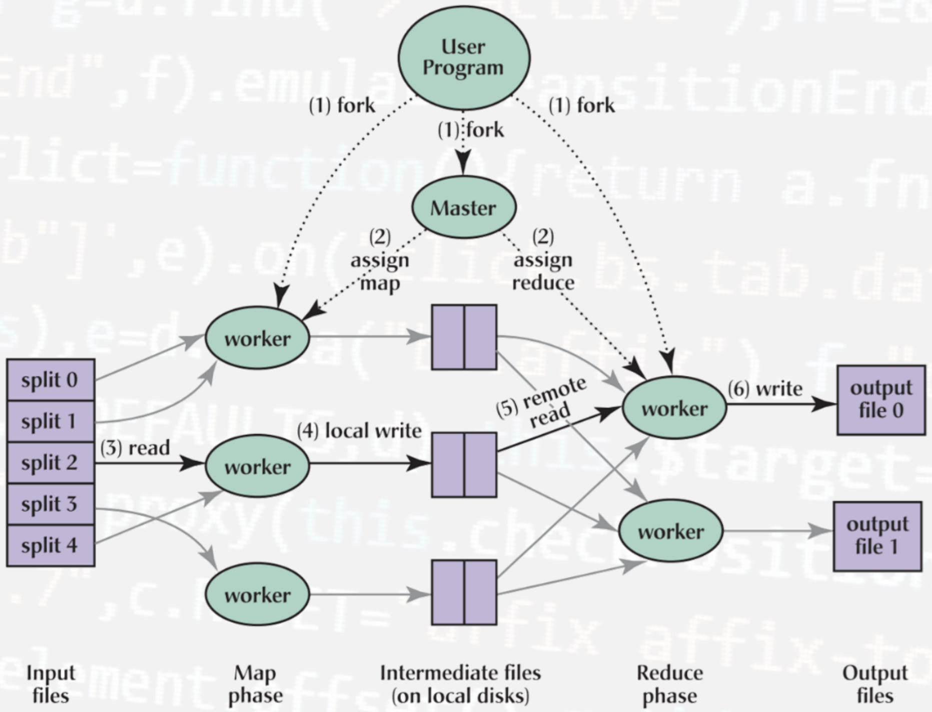


Image: Dean & Ghemawat (2004)

MapReduce Applications

- Distributed grep
  - Mapper: grep lines matching pattern
  - Reducer: Identity
- Inverted index
  - Mapper: extracts <word,doc_id> pairs by going over words appearing in doc_id
  - Reducer: concatenate all doc_id for any given word, output <word,list(doc_id)>
  - Application: reverse web-link graph, <target,list(source)>
- Distributed sorting
  - Mapper: local sorting of list
  - Reducer: merge sorting of sorted lists
- Many many others…

- HW:
  - Complex business intelligence
  - Prediction mechanism
  - Stay tuned…

“MapReduce-inspired” Problems & Frameworks

- Network aware MapReduce
  - Beyond mere locality
  - Take available network resources into account
    - Mappers scheduling, shuffling/partitioning to reducers
- Machine learning
  - Distributed training of Neural Networks
  - Highly non-trivial

Word Count: Full Unix Pipeline (Simple) Example

```
cat input_file | tr -s ' ' | tr ' ' '\n' | sort | uniq -c | sort -k 1nr,1 | less
```

Word Count: Full Unix Pipeline (Simple) Example

```
cat input_file | tr -s ' ' | tr ' ' '\n' | sort | uniq -c | sort -k 1nr,1 | less
```

Extract lines

Shrink multiple spaces to single space

Break every word onto single line

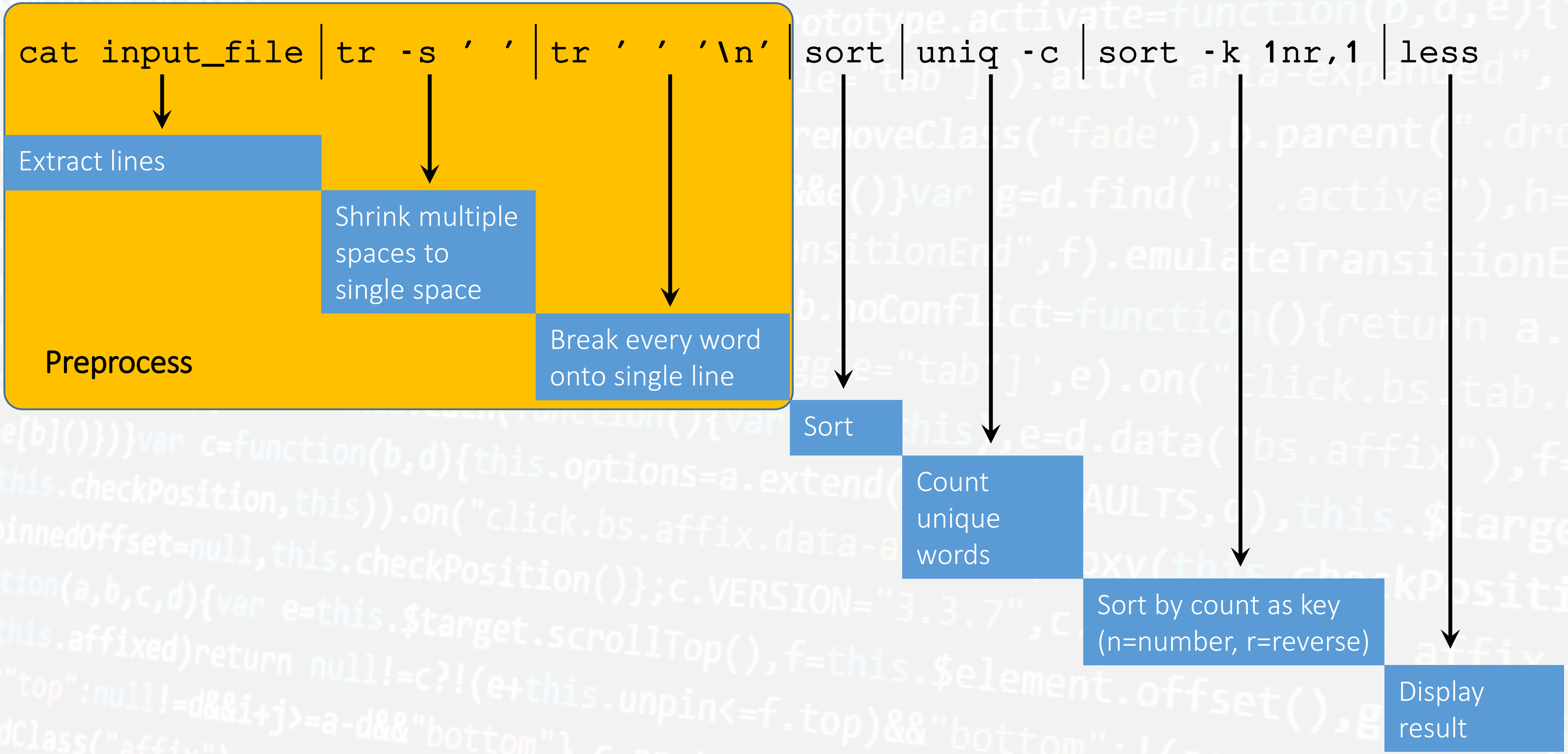
Sort

Count unique words

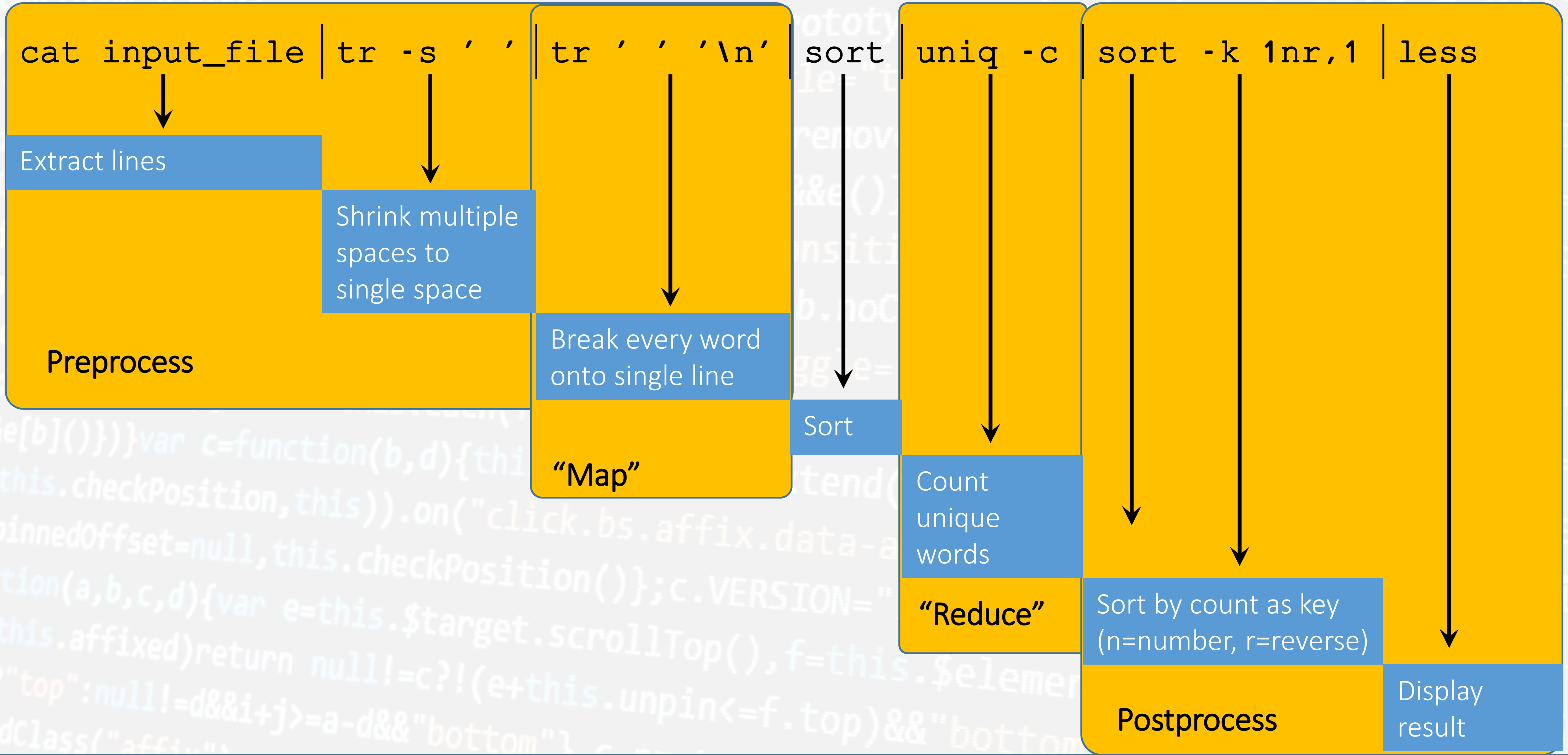
Sort by count as key (n=number, r=reverse)

Display result

Word Count: Full Unix Pipeline (Simple) Example



Word Count: Full Unix Pipeline (Simple) Example



Word Count \w MR: A Pythonish Implementation

- Demo
  - Python
  - Using Linux-sort

www.michael-noll.com

(Partial) Bibliography

- Laney, “3D Data Management: Controlling Data Volume, Velocity and Variety”, Gartner (2001)
- Zikopoulus et al., “Understanding Big Data”, McGraw-Hill (2012)
- Du, “Apache Hive Essentials”, Packt (2015)
- Ghemawat et al., “The Google File System”, SOSP (2003)
- Shvachko et al., “The Hadoop Distributed File System”, MSST (2010)
- Hanson, “An introduction to the Hadoop Distributed File System”, https://www.ibm.com/developerworks/library/wa-introhdfs/ (2011)
- White, “Hadoop – The Definitive Guide”, O'Reilly (2015)
- Dean and Ghemawat, “MapReduce: Simplified Data Processing on Large Clusters”, OSDI (2004)
- Pattamsetti, “Distributed Computing in Java 9”, Packt (2017)
- Noll, “Writing An Hadoop MapReduce Program In Python”, https://www.michael-noll.com/