dm22s1

Topic 03 : Data Handling

Part 02 : Big-Data-Frameworks

Dr Bernard Butler
Department of Computing and Mathematics, WIT.
(bernard.butler@setu.ie)

Autumn Semester, 2022

Outline

- Batch processing with Hadoop
- Stream processing with Spark

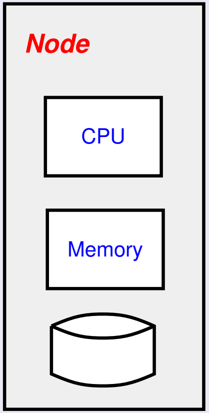
Overview of Large Scale Infrastructures

- Data infrastructures need to be *distributed* to scale to very large data sets
- This means that computation happens on processing *nodes* that are connected by a network.
- The challenge is to manage the resources (within and between nodes) effectively so that computation can proceed concurrently on the data, while minimising bottlenecks when data and results need to be shared across the network.

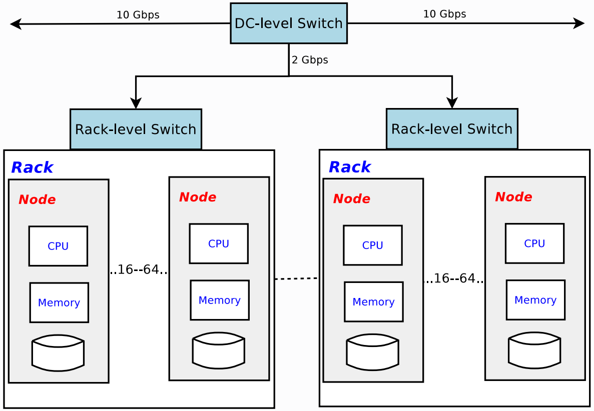
Small versus large scale

Modern data infrastructures scale *out* rather than *up*

Small Scale



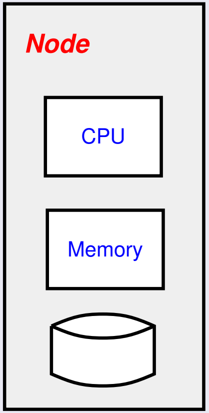
Large (cloud) Scale



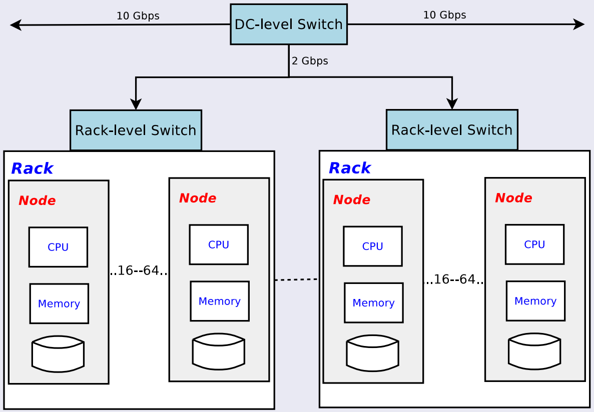
Small versus large scale

Modern data infrastructures scale *out* rather than *up*

Small Scale



Large (cloud) Scale



Architectural considerations

- change locus of control
  - Traditional (e.g., Data Warehousing): move the data to the (database) server for computation
  - "Big Data": move the computation to the data (programs are smaller and more transportable than massive data, but need to be designed as such)
- minimise communication across network links
  - exploit data locality (where known)
  - rearrange computation steps
- distribute the computation evenly (avoid "hot spots")

Note that some of these aims could be in conflict, so performance management is not trivial!

Static versus Streaming data and Computations

Big Data Phase 1: 2007-2012

- Data flows into the *data lake* and is stored there;
- Tasks are distributed to work on data subsets within the lake;
- Tasks start together, on receipt of a control signal;
- The overall *batch* job ends when the last task ends;
- More data arrives and the cycle repeats.

Big Data Phase 2: 2013-date

- Workflows (control and data flows) are defined;
- Data streams in from many sources, creating *back pressure*;
- Data is in motion, so storing it has secondary importance;
- Data streams are processed in place, preferably in-memory;
- Processing is continuous, there are no (arbitrary) start and end points.

Batch processing applies to chunks of bounded data. It sacrifices latency for consistent snapshots.
Streaming treats data as unbounded, coming from continuous processes.

Static versus Streaming data and Computations

Big Data Phase 1: 2007-2012

- Data flows into the *data lake* and is stored there;
- Tasks are distributed to work on data subsets within the lake;
- Tasks start together, on receipt of a control signal;
- The overall *batch* job ends when the last task ends;
- More data arrives and the cycle repeats.

Big Data Phase 2: 2013-date

- Workflows (control and data flows) are defined;
- Data streams in from many sources, creating *back pressure*;
- Data is in motion, so storing it has secondary importance;
- Data streams are processed in place, preferably in-memory;
- Processing is continuous, there are no (arbitrary) start and end points.

Batch processing applies to chunks of bounded data. It sacrifices latency for consistent snapshots. Streaming treats data as unbounded, coming from continuous processes.

Introducing Hadoop

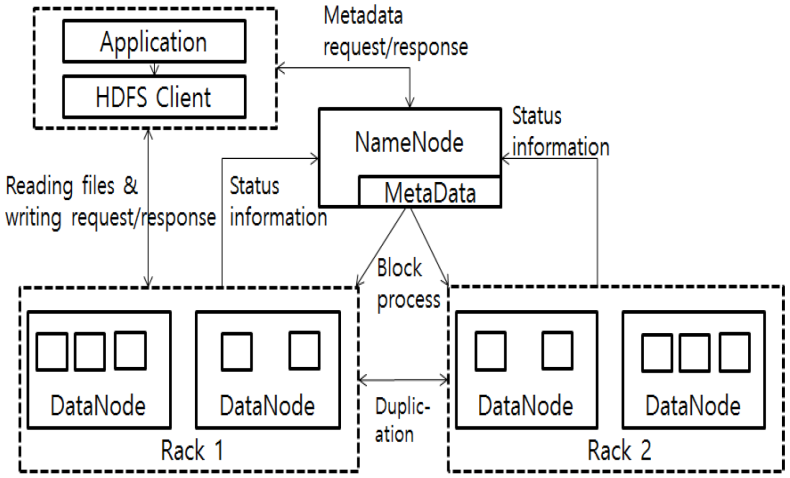
- Google had published papers on the (distributed) Google Filesystem (GFS) and how it supported massively-distributed, concurrent computations structured as *map-reduce* batch jobs.
- Yahoo engineers (Doug Cutting and others) created an equivalent and open-sourced it as Apache Hadoop:
  1. Hadoop Distributed File System (HDFS) for distributed, replicated, reliable storage;
  2. Map-Reduce programming framework for distributing tasks to data and collecting the results (legacy);
  3. YARN resource management on hadoop clusters (Hadoop 2.0 and later).
- Hadoop was a sensation and spawned many related Apache (and other) projects Apache HBase, Apache Hive and Apache Mahout, often sponsored by large web companies.
- It (and particularly HDFS) continues to be the foundation for many Big Data initiatives.

YARN decouples HDFS and map-reduce, to support other programming frameworks on the same Hadoop cluster.

HDFS overview

- Cluster of commodity (not specialised!) Linux-based compute nodes *with attached storage*, contrast with databases instances and SAN
- Offers *global namespace*: file looks like a single entity—storage details are hidden to users
- Optimised for Read and Append operations
- Files are split into chunks (partitions) of standard size (16-64MB is common); partitions are replicated with rack-awareness to minimise correlated failures
- Master (`namenode`) stores file -> storage and other metadata: master-slave with other nodes
- Secondary namenodes can exist, but manage snapshots of the metadata
- The vast majority of nodes are `datanode`s which store the actual data
- HDFS monitors the "health" of the cluster and moves partition replicates as needed; re-partitioning is expensive
- Client applications that add or remove files issue a system call that is routed transparently to the namenode and HDFS uses the metadata to complete their request.

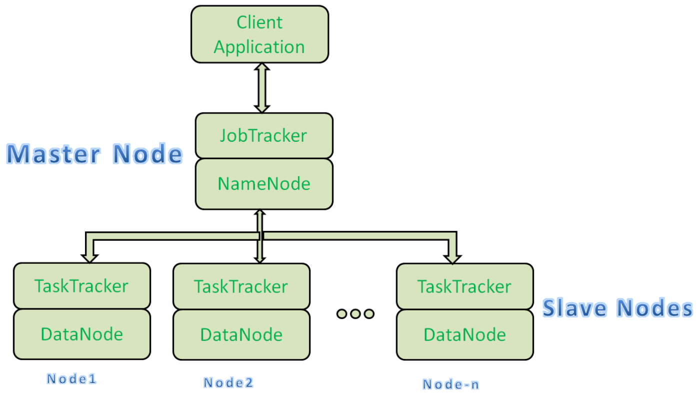
HDFS architecture and protocol



For a “friendly” introduction to the HDFS protocol, see this cartoon by Maneesh Varshney*.

*https://wiki.scc.kit.edu/gridkaschool/upload/1/18/Hdfs-cartoon.pdf

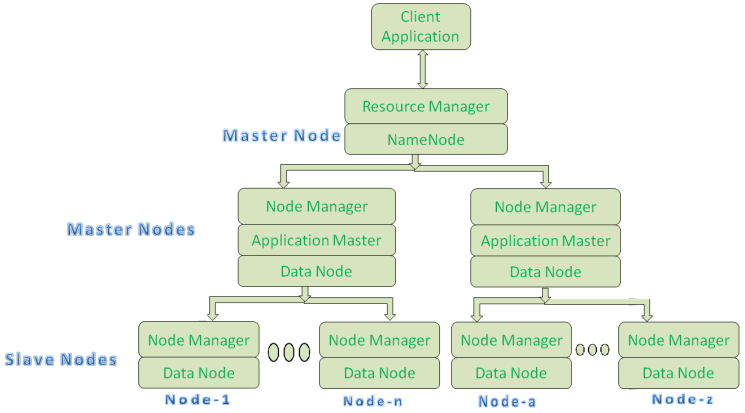
Hadoop 1.x components



Hadoop 1.x Components Architecture

Note the close coupling between the map-reduce `JobTracker` and `TaskTracker` and the underlying HDFS `NameNode` and `DataNode` components.

Hadoop 2.x components



Hadoop 2.x High-Level Architecture

Note that the map-reduce components have been replaced with more general `ResourceManager` and `NodeManager` components.

Introduction to map-reduce

Input: set of key-value pairs `(k,v)`

Map

- `Map(k,v)` $\rightarrow$ `<k`$^{(1)}$`,v`$^{(1)}$`>*`
- In words, the function takes a *single* key-value and outputs a set of (derived) key-value pairs
- The Map call depends on the task, but there is always one per key-value pair

Reduce

- `Reduce(k`$^{(1)}$`,<v`$^{(1)}$`>*)` $\rightarrow$ `<k`$^{(1)}$`, v`$^{(2)}$`>*`
- In words, the function takes *all* values `v`$^{(1)}$ having the same key `k`$^{(1)}$ are reduced together and processed in `v`$^{(1)}$ order
- The Reduce call also depends on the task, but there is always one per unique key `k`$^{(1)}$.

The easiest way to understand this algorithm is to look at an example. The hadoop documentation describes how map-reduce can be applied to word counting.

Introduction to map-reduce

Input: set of key-value pairs `(k,v)`

Map

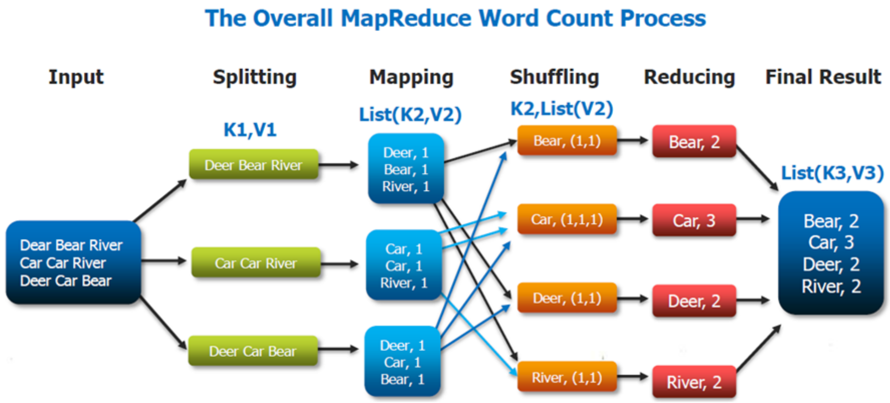
- `Map(k,v)` $\rightarrow$ `<k`$^{(1)}$`,v`$^{(1)}$`>*`
- In words, the function takes a *single* key-value and outputs a set of (derived) key-value pairs
- The Map call depends on the task, but there is always one per key-value pair

Reduce

- `Reduce(k`$^{(1)}$`,<v`$^{(1)}$`>*)` $\rightarrow$ `<k`$^{(1)}$`, v`$^{(2)}$`>*`
- In words, the function takes *all* values `v`$^{(1)}$ having the same key `k`$^{(1)}$ are reduced together and processed in `v`$^{(1)}$ order
- The Reduce call also depends on the task, but there is always one per unique key `k`$^{(1)}$.

The easiest way to understand this algorithm is to look at an example. The hadoop documentation describes how map-reduce can be applied to word counting.

Map-Reduce for Word-Counting



This diagram (credit: NYU) shows the control and data flows. *Splitting* is explicit here. *Mapping* emits one word per line and can include repeated lines. *Shuffling* (in a *Combiner*) is optional, added to reduce the load on the *Reduce* phase if the function is commutative and associative. Often $\#\{\text{Mappers}\} \gg \#\{\text{Reducers}\}$ so matching the load (e.g., by word popularity) can be tricky. Split, Map, Shuffle and Reduce functions can run in parallel on their own data, but delays may occur at each step: "slowest ship in the fleet" problem.

Analysis of Map-Reduce

- Programmability: by abstracting common functions, programmers can focus on the details of their tasks and leave the rest to the framework;
- Recovery: If a TaskNode fails but the data is still available
    - the YARN scheduler can redistribute its tasks to other TaskNodes of the same kind (e.g., Mappers)
    - the stand-in TaskNode can start when ready, but overall execution time will increase
- Performance: High performance is not guaranteed
    - Load balancing in such a rigid allocation model (e.g., how many Mappers?) can be difficult
    - Need to get the right balance between computation and (within-cluster) communication effort, e.g., collocate Reducers with Mappers?
    - Often bottlenecks arise in network links between the cluster and external systems
- Limitations: Rigidly prescribed task decomposition is not always a good fit
    - When processing streaming (unbounded) data
    - For iterative applications over the same data (common in Machine Learning)

What now for Hadoop?

Hadoop 3 vs Hadoop 2

Hadoop 3 did not introduce architecture changes but made 2 big efficiency improvements:

- instead of 3x replication (200% space overhead), it now uses *erasure coding* (50% space overhead), and
- the YARN *timeline* feature (which schedules when resources are allocated to tasks) was made more efficient and scalable.

Another Apache project that has outgrown its Hadoop roots is Zookeeper, for distributed coordination.

HDFS, YARN and Zookeeper are the Hadoop features that are most used by other Big Data projects.

Newer processing frameworks offer performance and other advantages, but still rely on those 3 features.

Big Data Frameworks for Streaming Data

Apache Spark: "Analytics 3G"

- Spark started as an in-memory replacement for map-reduce, but can persist to HDFS
- Up to 100 times faster than hadoop because of in-memory processing, etc.
- Spark Streaming generalised this to support micro-batches (pseudo-streaming)
- Spark was earlier and has built a larger community
- Direct support for Machine Learning (Spark ML)

Apache Flink: "Analytics 4G"

- Designed as an in-memory stream-processing engine
- Faster than Spark, without performance dips when micro-batch hands over to next
- Batch processing is a special case where the data is finite
- Potential of real time processing, very efficient, mostly in niche applications for now
- Scheduler is very powerful and can *automatically* optimize job performance

Honourable mentions: Apache Storm, Apache Samza, Apache Zookeeper, Apache Kafka

Big Data Frameworks for Streaming Data

Apache Spark: "Analytics 3G"

- Spark started as an in-memory replacement for map-reduce, but can persist to HDFS
- Up to 100 times faster than hadoop because of in-memory processing, etc.
- Spark Streaming generalised this to support micro-batches (pseudo-streaming)
- Spark was earlier and has built a larger community
- Direct support for Machine Learning (Spark ML)

Apache Flink: "Analytics 4G"

- Designed as an in-memory stream-processing engine
- Faster than Spark, without performance dips when micro-batch hands over to next
- Batch processing is a special case where the data is finite
- Potential of real time processing, very efficient, mostly in niche applications for now
- Scheduler is very powerful and can *automatically* optimize job performance

Honourable mentions: Apache Storm, Apache Samza, Apache Zookeeper, Apache Kafka

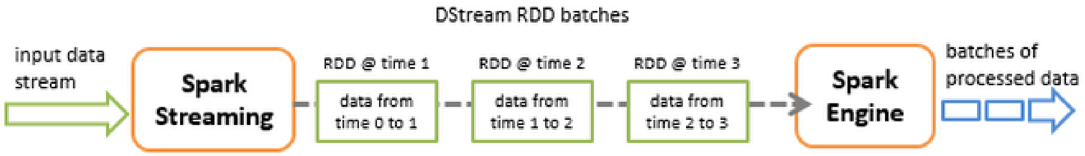
Message Queues for Streaming Data

- Message Queues act as sources for Stream Processors (c.f., files and Batch Processors)
- Producers write (enqueue) messages that Consumers read (dequeue) *asynchronously*
- Apache Kafka (from LinkedIn and Confluent) integrates well with Spark, Flink, etc.
- Producers append messages to topic queues that are *committed* and distributed to different servers as (replicated) *partitions*
- Each message is assigned an *offset* in its topic queue; Consumers uses this to restart/replay etc. Message ordering is important!
- Messages are *retained* on disk (for a set period) for resilience, but most processing is real-time
- Kafka guarantees message order recovery, committed messages not being lost (assuming at least one replicate exists!), consumers read only committed messages
- Kafka is a very common provider of data and recent versions have added the ability to perform some operations on the data.
- Kafka scales so well that it offers a general purpose message delivery scheme for microservices etc.

Resilient Distributed Data sets: Introduction

- Immutable (Read-Only after it has been created) Collection of *objects* directly usable by a program
- Can be created in two ways:
  - Parallelize an existing `Collection` object
  - Read and convert data from a file, hadoop input object, etc.
- partitioned into chunks; each partition is distributed to a server
- stored in memory (use disk only as a last resort!)
- partitions are recomputed on failure
- software is applied to RDD and is executed concurrently - this is transparent to the programmer

Other data structures include DataSets (improved performance) and Dstreams (discretised-time RDDs), see below:



RDD Operations

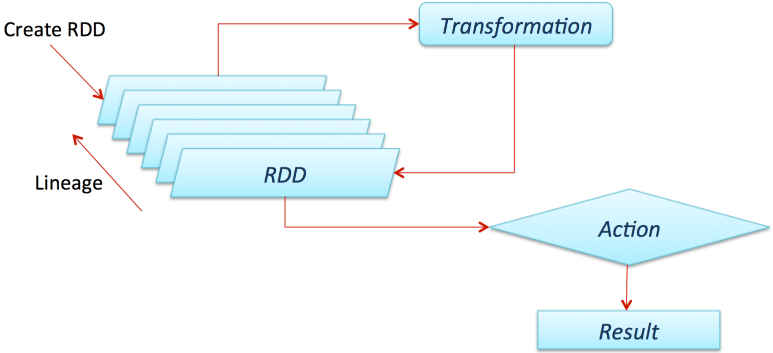
Spark supports Transformations and Actions on RDDs

A Transformation

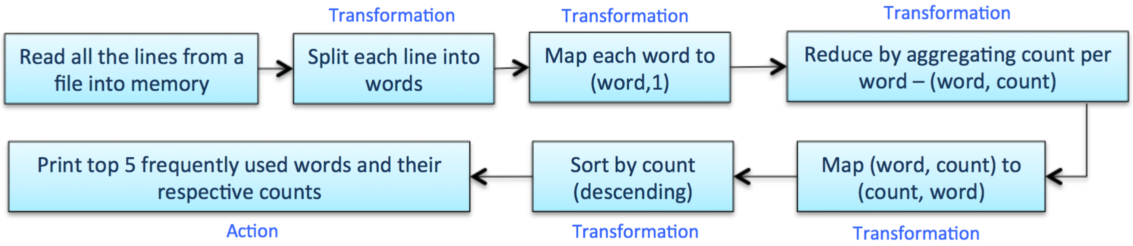
Maps RDD_1 to RDD_2

An Action

Derives a (non-distributed) *value* x from RDD_1



Spark: word-count example


Read all the lines from a file into memory
Transformation
Split each line into words
Transformation
Map each word to (word,1)
Transformation
Reduce by aggregating count per word – (word, count)
Transformation
Map (word, count) to (count, word)
Transformation
Sort by count (descending)
Action
Print top 5 frequently used words and their respective counts

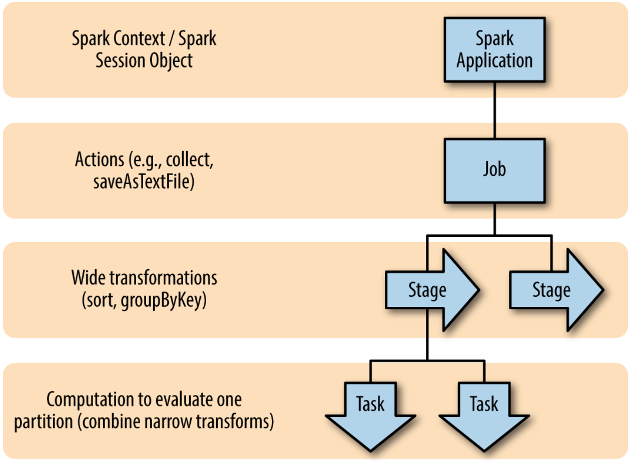
Note the similarities with the hadoop example on slide 12. Also note that the Spark programming model (in terms of the RDD abstraction) is richer and the distinction between Transformation and Action is emphasised.

Example Transformations and Actions

Transformation	Action
map(func)	reduce(func)
filter(func)	collect()
sample(. . .)	count()
union(other)	take(n)
distinct(. . .)	takeSample(. . .)
groupByKey()	saveAsTexFile(path)
join(other)	saveAsSequenceFile(path)
cartesian(other)	saveAsObjectFile(path)
pipe(cmd)	countByKey()
coalesce(numPartitions)	foreach(func)

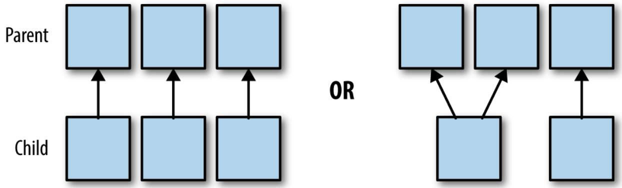
Since Spark uses lazy evaluation, Transformations are not evaluated until required by a downstream Action. This helps performance and fault tolerance.

Spark Application Tree


Spark Context / Spark Session Object
Spark Application
Actions (e.g., collect, saveAsTextFile)
Job
Wide transformations (sort, groupByKey)
Stage
Stage
Computation to evaluate one partition (combine narrow transforms)
Task
Task

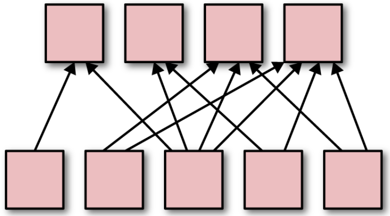
A Spark application is centred on Actions, which pull RDDs that are subject to Transformations. The more complex Transformations need further refinement as shown.

Narrow vs Wide Transformation Dependencies



Dependencies are narrow only if

- they can be determined at design time, and if
- each parent has at most one child partition.



Wide Dependencies

Example narrow Transformations are `map` and `coalesce` and wide Transformations are `sort`, `groupByKey` and `join`.

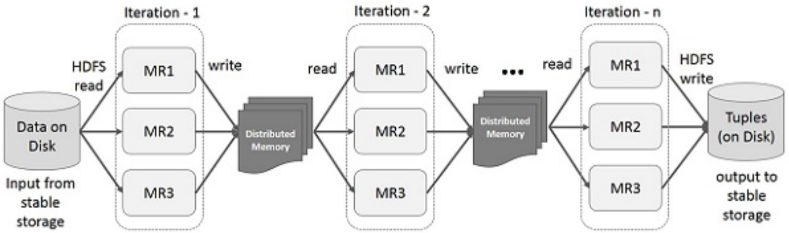
Challenge: Operating on unbounded data from a stream

Many traditional algorithms assume the data is finite

But some operations need to be adapted to work with unbounded data:

- Queries over a *sliding window*
- Filtering data streams with fixed criteria
- Counting distinct elements
- Computing *moments* (e.g., running averages)
- Counting itemsets (hot subsets)

Practical algorithms have been devised for each of these. However iterative algorithms are easier to arrange than with map-reduce on bounded data.



Streaming Challenge 1: Fixed Size Sample from Unbounded Data

How do we generate a fixed-size sample representing the data seen so far?

Reservoir sampling

Let the sample be S with required size s.

- `init`: Store all the first s elements of the stream to S
- `update`: Suppose we have seen $n - 1$ elements, and now the n^{th} element arrives ($n > s$).
  - With probability $P = \frac{s}{n}$, keep the new n^{th} element, otherwise discard it.
  - If we picked the n^{th} element, then it replaces one of the s elements in the sample S, picked uniformly at random.
- `query`: Return the current sample S.

This algorithm maintains a sample S so that, after n elements, it contains each element seen so far with probability $\frac{s}{n}$.

Imagine filling the reservoir first, then randomly adding more water while allowing existing water to escape, keeping the reservoir level constant.

Streaming Challenge 2: Average (arithmetic mean) from Unbounded Data

Assuming a stationary process, how do we compute its mean value, using the data seen so far?

Numerically stable mean

Let the stream of values be $x_1, x_2, \ldots$ and let $\bar{x}(k)$ be the mean of the k values seen so far.

- `init`: Set $\bar{x}(1) = x_1$.
- `update`: Suppose we have seen $n-1$ elements, and now the n^{th} element arrives ($n > 1$).
    - Use the following *stable* calculation to update the mean: $\bar{x}(n) = \bar{x}(n-1) + \frac{x_n - \bar{x}(n-1)}{n}$.
    - The additive update step will eventually underflow when n gets too large, but use of the naive updating formula would have encountered difficulties for much smaller n.
- `query`: Return the current mean $\bar{x}(n)$.

The (naive) unstable formula is $\bar{x}(n) = \frac{(n-1)\bar{x}(n-1) + x_n}{n}$. Mathematically, each of the two formulas can be derived from the other.

This works for a stationary process because the data generating process does not change with time, so the running mean converges to the true mean over time.

Review of Big Data Infrastructures

- Big Data requires new infrastructures for storage and operations
- Storage and computing paradigms need to work well together
- Initially the focus was on bounded data and batch processing (hadoop)
- This was facilitated by the falling cost of storage and computation and the growth of commodity/utility computing (the cloud)
- More recently, the need to reduce *latency* (delay between data arrival and it being processed) has changed priorities
- Data is now seen as a stream (“the Twitter fire-hose”). Spark and Flink take account of this.
- New abstractions, data structures and algorithms are being developed
- Data Mining can benefit from these new infrastructures; work is underway to implement machine learning on such infrastructures
- Techniques such as *deep learning* become feasible only because there is enough data, and they can build upon the new infrastructures
- ***Next week***: *we aim to get a preliminary understanding of data in this “data deluge”*