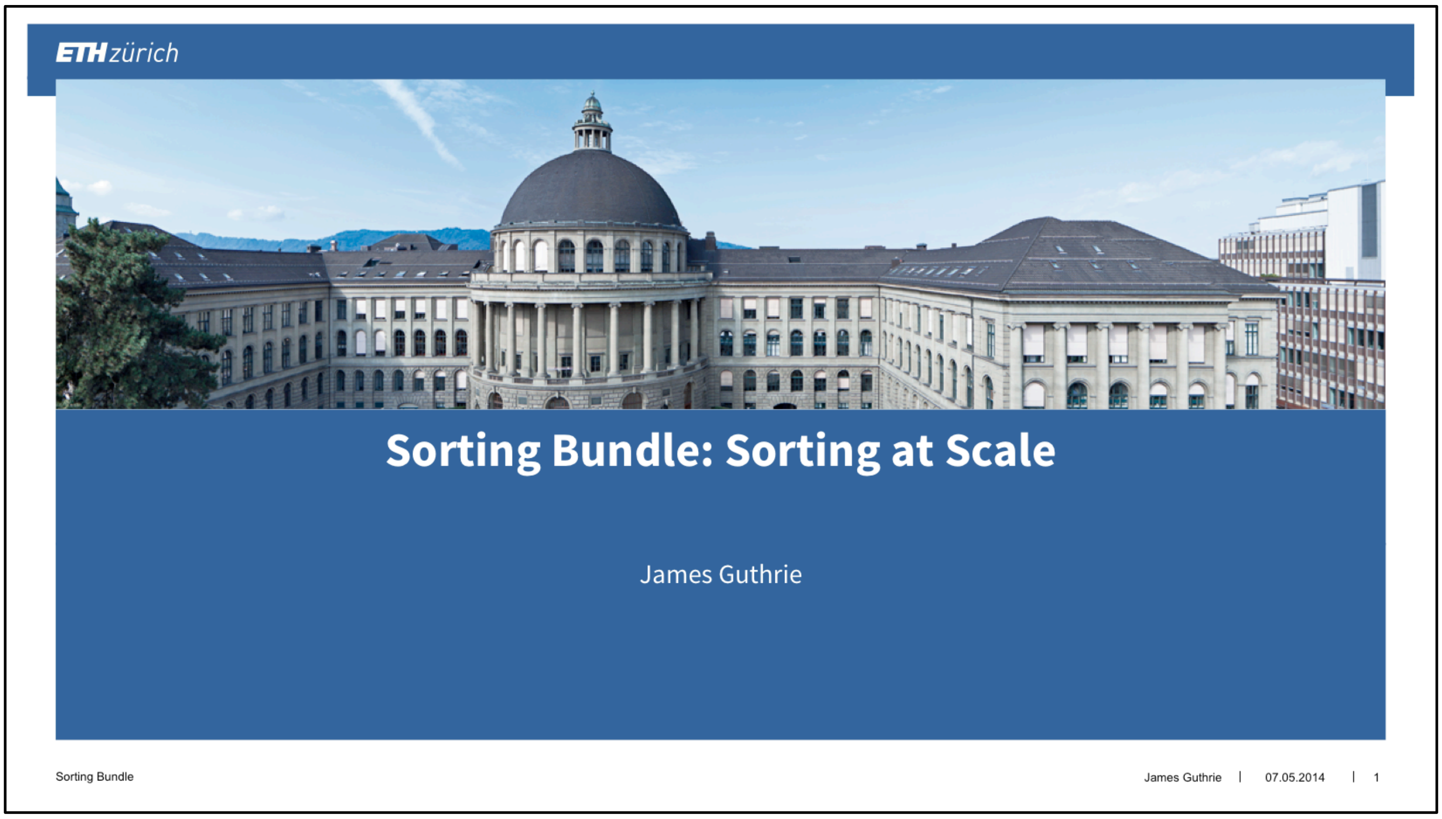
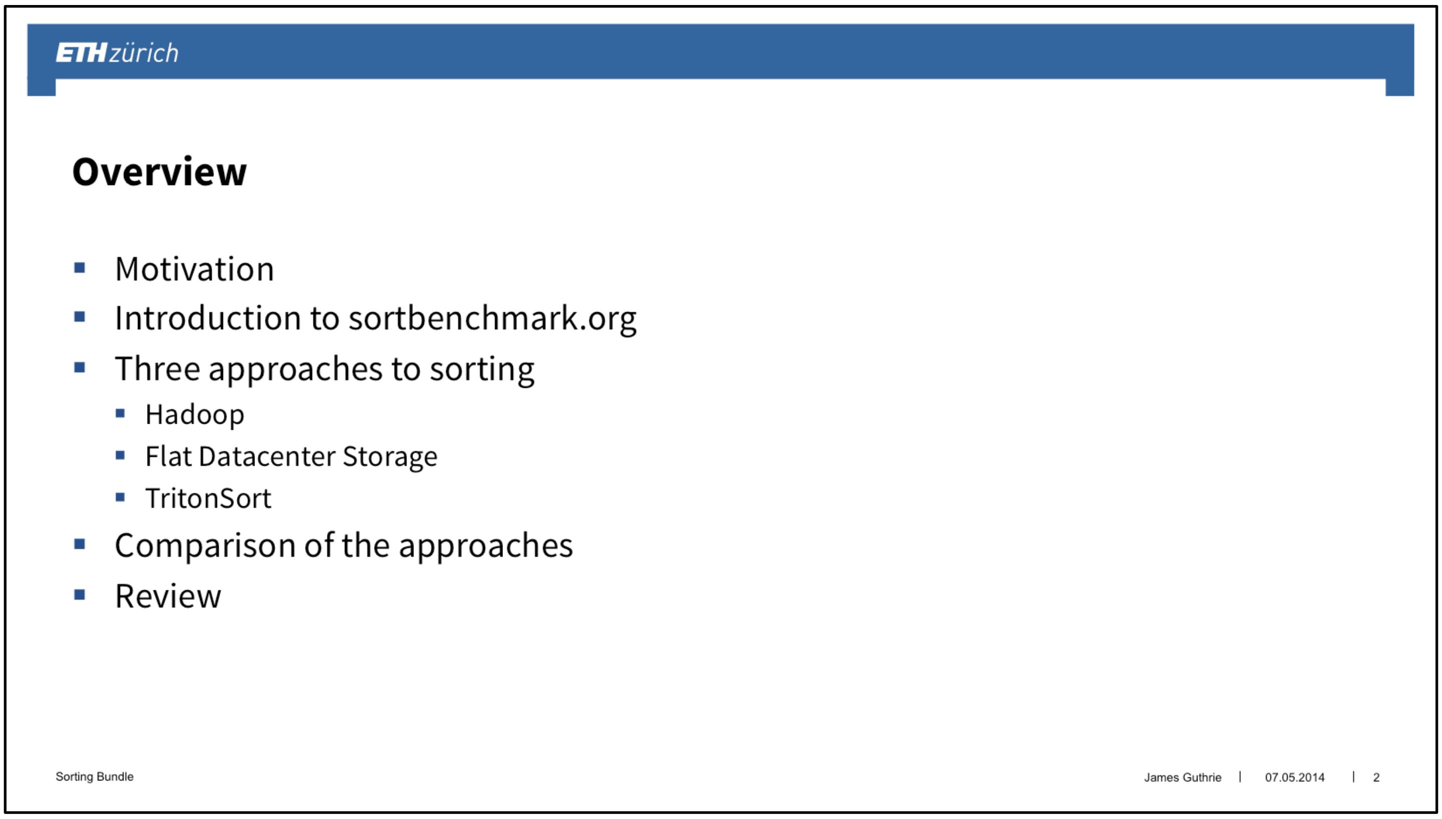


ETH zürich

Sorting Bundle: Sorting at Scale

James Guthrie

The task in the sorting bundle was to select a couple of papers from the sortbenchmark.org website, find something interesting to present and hopefully show you something new and interesting today.


ETH zürich

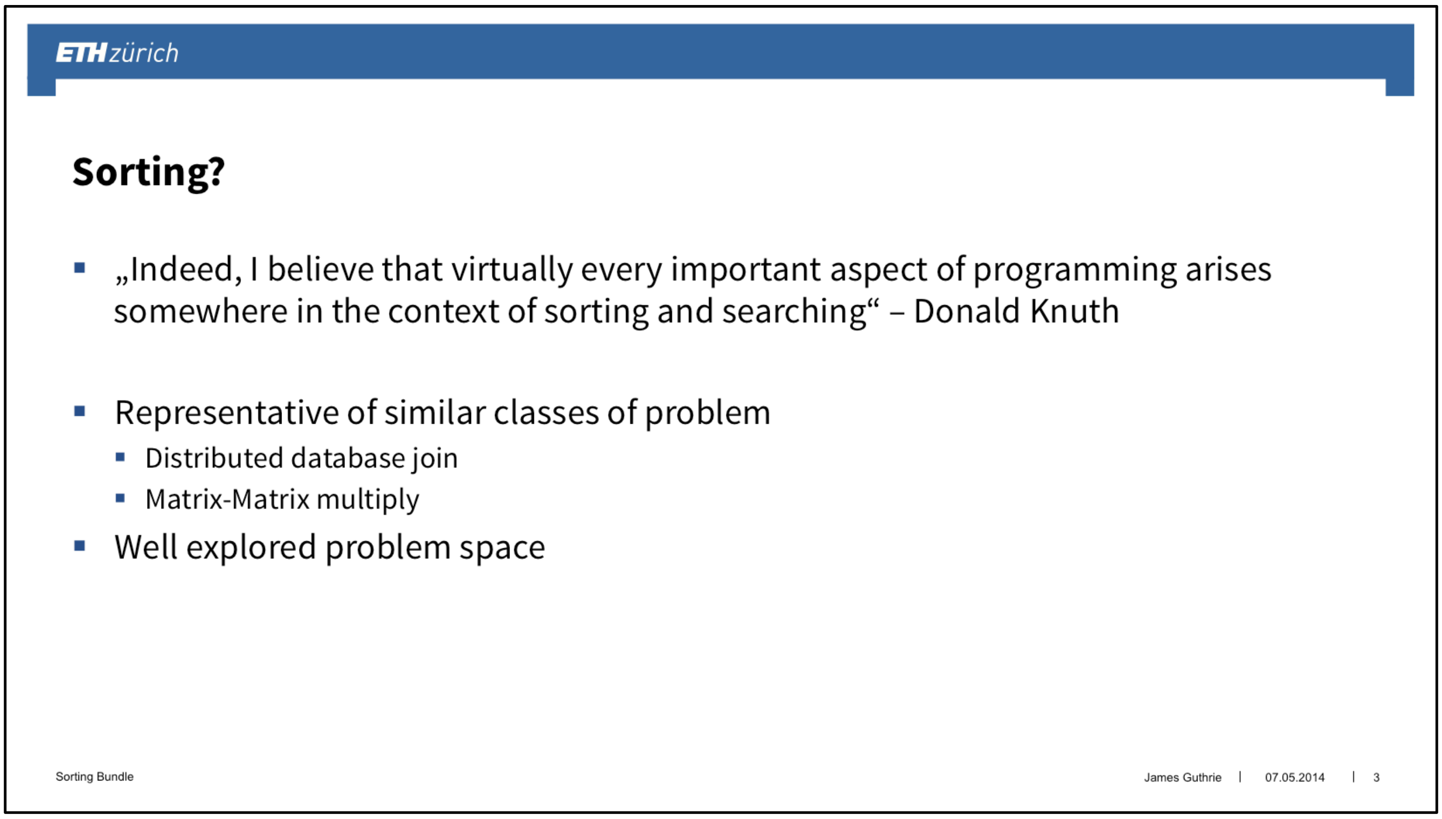
Overview

- Motivation
- Introduction to sortbenchmark.org
- Three approaches to sorting
  - Hadoop
  - Flat Datacenter Storage
  - TritonSort
- Comparison of the approaches
- Review

Sorting Bundle

James Guthrie | 07.05.2014 | 2

Today we're going to start with a brief motivation behind sorting, then I'll introduce you to sortbenchmark.org, after which we will take a look at three different approaches to sorting. Finally we will compare the three approaches and do a small review of what was presented today.



ETH zürich

Sorting?

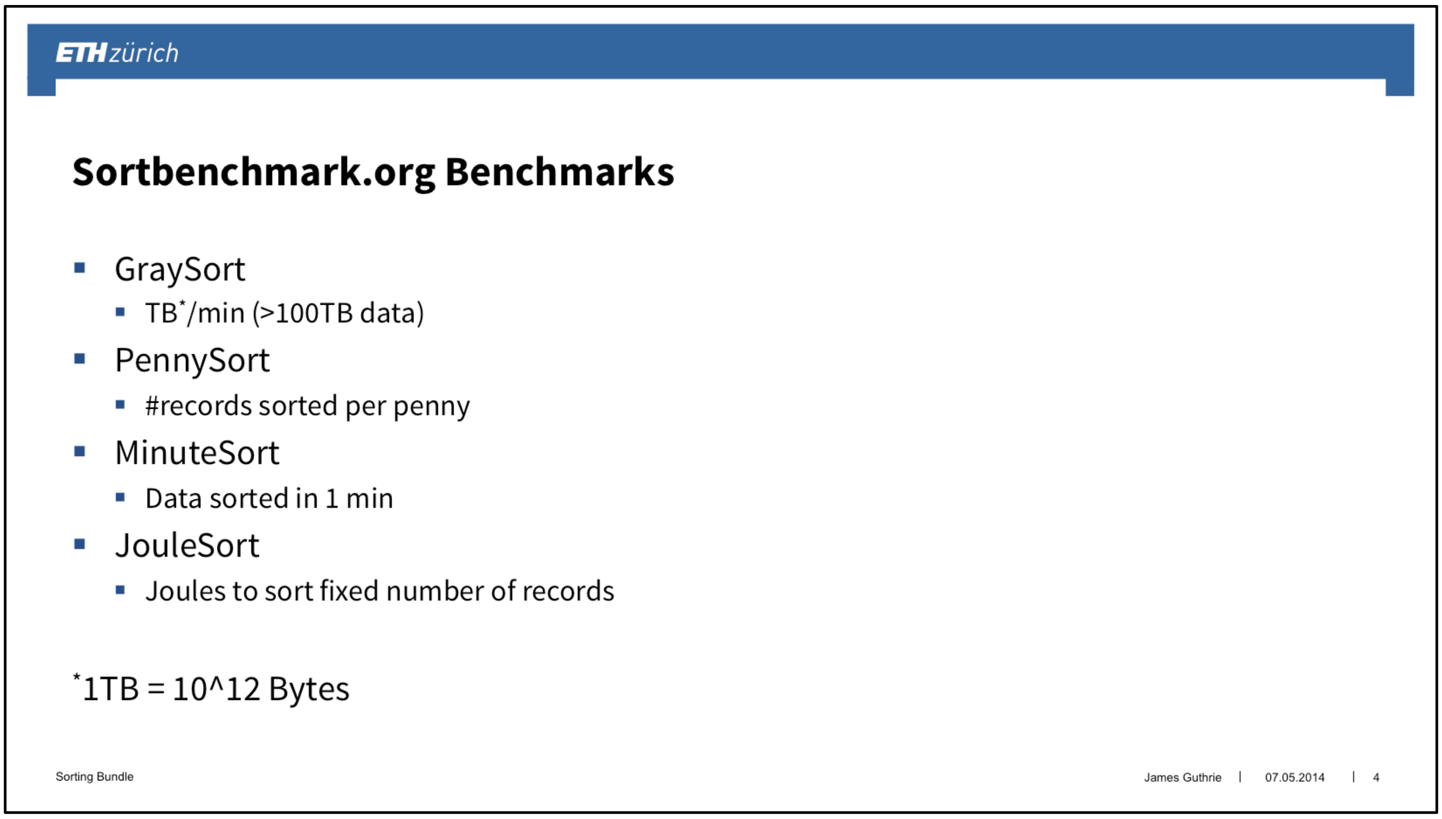
- „Indeed, I believe that virtually every important aspect of programming arises somewhere in the context of sorting and searching“ – Donald Knuth

- Representative of similar classes of problem
  - Distributed database join
  - Matrix-Matrix multiply
- Well explored problem space

Sorting Bundle James Guthrie | 07.05.2014 | 3

(***) Knuth is convinced that sorting plays a role in almost every aspect of programming.

(***) But sorting isn’t just important because Knuth says so, it also represents similar classes of problem (in their I/O complexity) and is a well-explored problem space.


ETH zürich

Sortbenchmark.org Benchmarks

- GraySort
  - TB*/min (>100TB data)
- PennySort
  - #records sorted per penny
- MinuteSort
  - Data sorted in 1 min
- JouleSort
  - Joules to sort fixed number of records

*1TB = 10^12 Bytes

Sorting Bundle James Guthrie | 07.05.2014 | 4

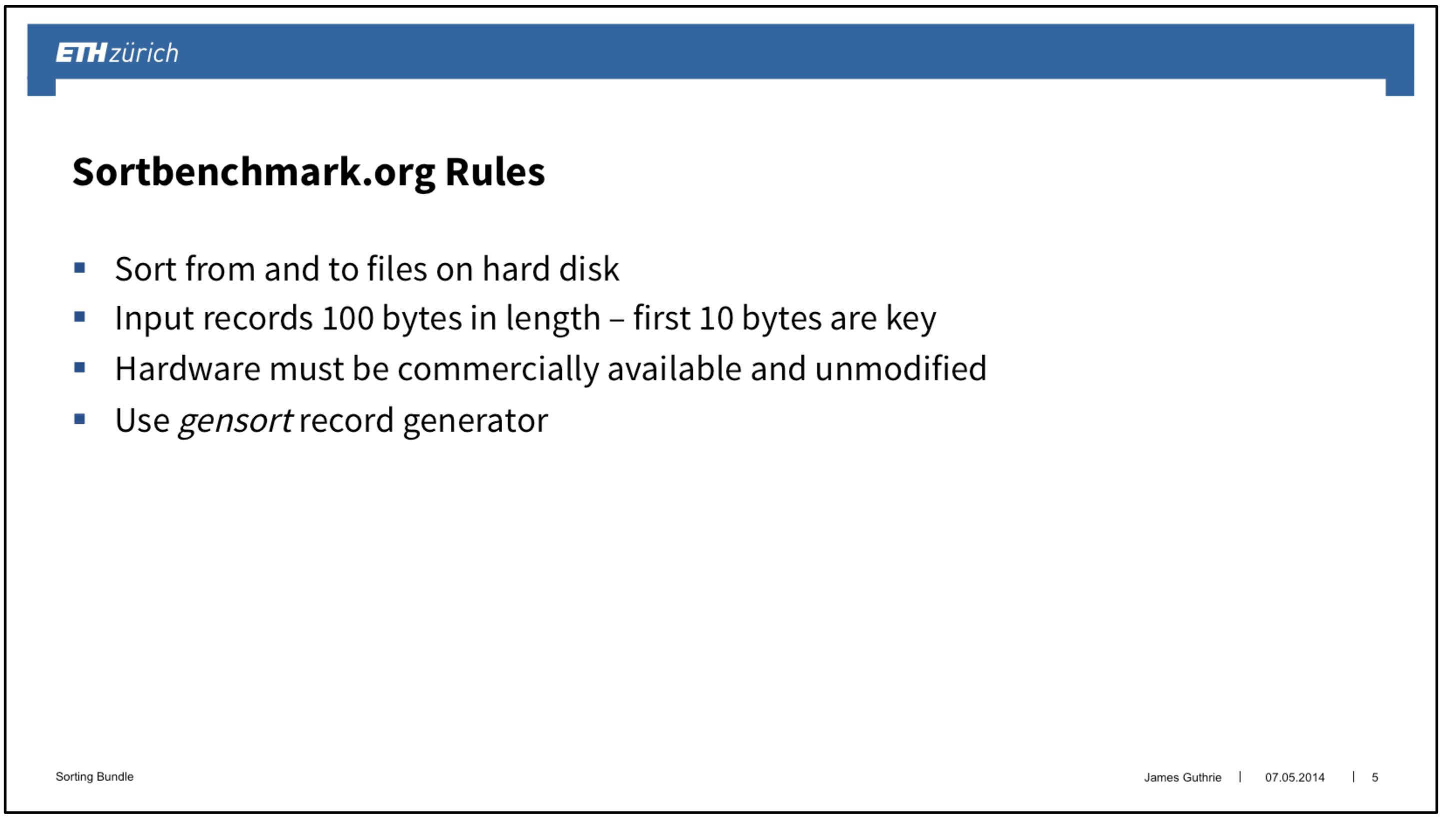
Sortbenchmark.org is a website which lists rankings of the best sorting systems, divided among a few different types of benchmark.

(***) The GraySort, named after the founder of the sorting benchmark Jim Gray, is a measure of brute-force sorting power.
(***) The metric is the sort rate in Terabytes/minute on a large dataset. The size of the dataset increases as technology progresses and currently sits at minimum of 100TB. Note that a Terabyte is denoted here as 10^12 bytes, not 2^40 bytes (TiB).

(***) The PennySort is a measure of the price/performance ratio of computing hardware, or how affordable computation is.
(***) The metric is #records sorted per penny of compute time. It assumes a lifetime of three years, through which the component cost of the hardware is divided to come up with the number of seconds of compute time one gets for a penny.

(***) The MinuteSort is a measure of how much sorting can be done in a minute.
(***) The metric is (surprisingly) the amount of data sorted in a minute.

(***) The JouleSort is a measure of how energy-efficient the sort is.
(***) The metric is the number of joules required to sort a given number of records (also sometimes the #records per joule). This is broken down into four different categories, from 10GB to 100TB of records (10^8, 10^9, 10^10 and 10^12 records).


ETH zürich

Sortbenchmark.org Rules

- Sort from and to files on hard disk
- Input records 100 bytes in length – first 10 bytes are key
- Hardware must be commercially available and unmodified
- Use *gensort* record generator

Sorting Bundle James Guthrie | 07.05.2014 | 5

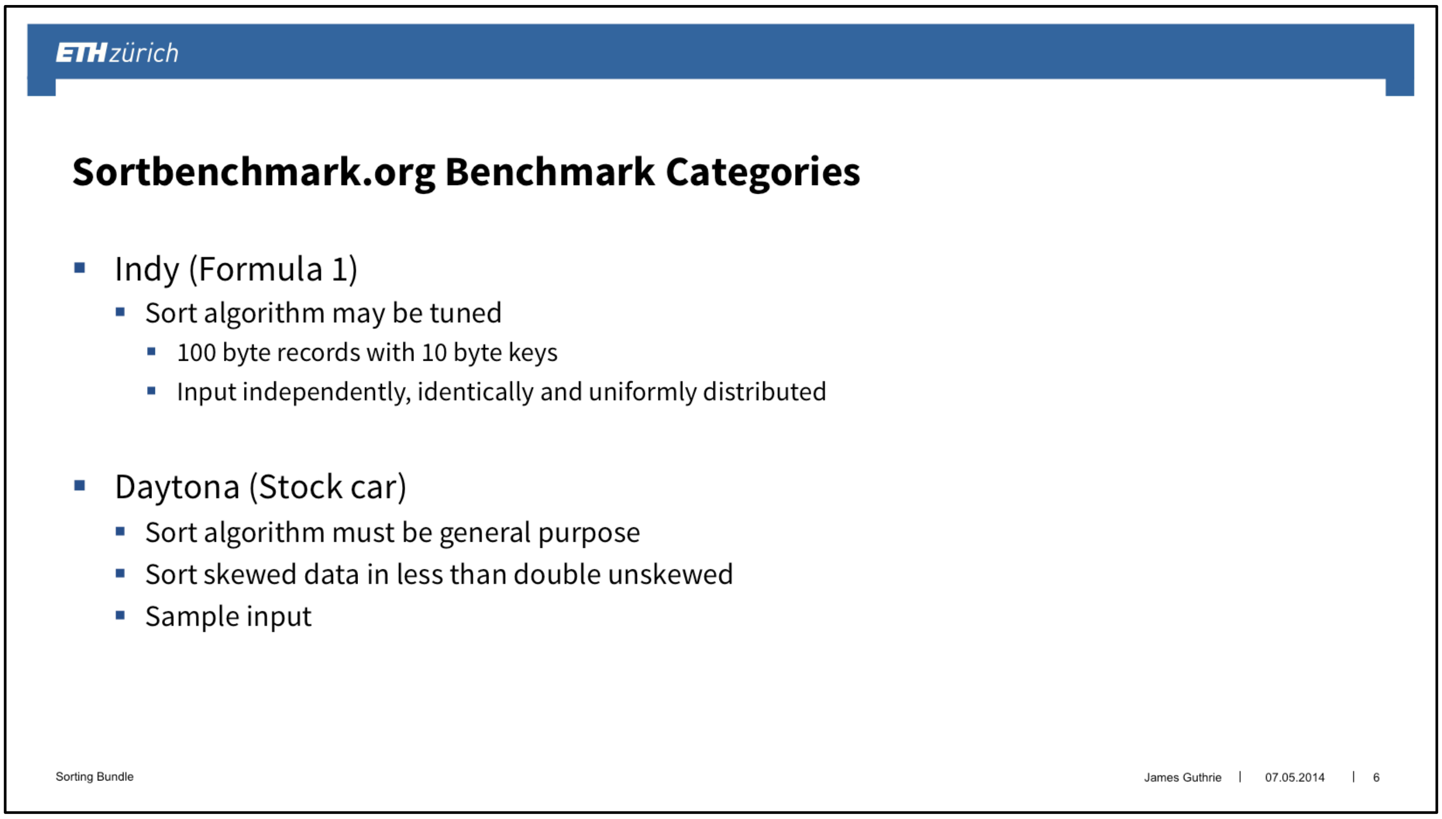
There are a few rules that must be adhered to for the results to be accepted into the sort benchmark.

The input data must reside on the hard disks before the sort begins and the sorted output data must reside on the hard disks when the sort ends.

The input records to be sorted are 100 bytes long. The first 10 bytes of each record are the key by which the record must be sorted.

The hardware used must be commercially-available and unmodified (no overclocking or tuning)

The sort application should use the gensort generator provided by sortbenchmark.org



ETH zürich

Sortbenchmark.org Benchmark Categories

- Indy (Formula 1)
  - Sort algorithm may be tuned
    - 100 byte records with 10 byte keys
    - Input independently, identically and uniformly distributed
- Daytona (Stock car)
  - Sort algorithm must be general purpose
  - Sort skewed data in less than double unskewed
  - Sample input

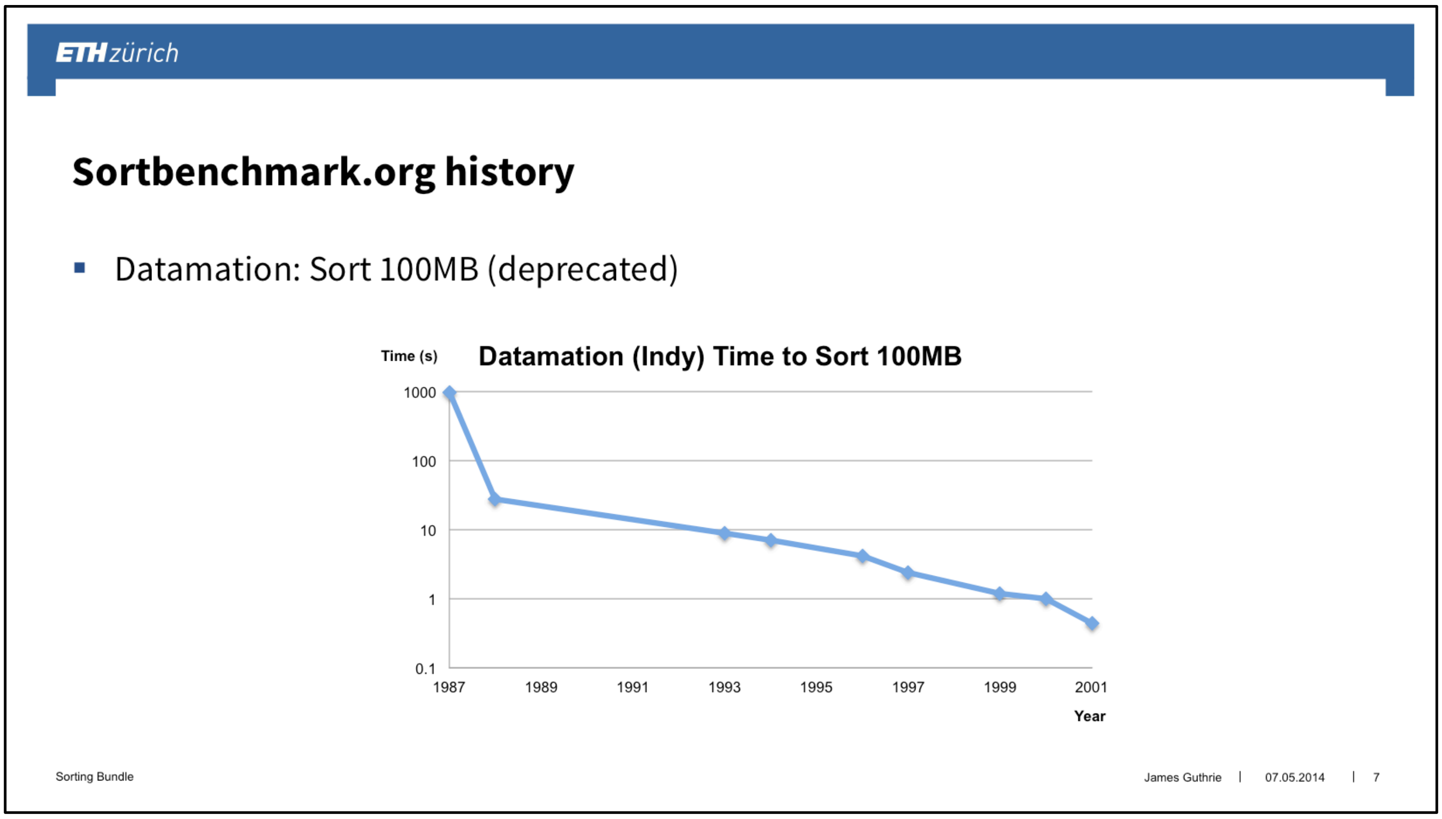
Sorting Bundle James Guthrie | 07.05.2014 | 6

Additionally there are two categories of benchmark.

In the first category, called “Indy” (or Formula 1 if you are unfamiliar with “Indy”) the sort algorithm may be tuned to take advantage of the properties of the input data. Specifically that the records to be sorted are 100 bytes long and that the key to sort them by is the first 10 bytes. It may also be assumed that the input data is independently, identically and uniformly distributed.

In the second category (called “Daytona” or “Stock car”) the sort algorithm must be general purpose i.e. able to handle records and keys of arbitrary length. Additionally, the sort algorithm may not make any assumptions about the distribution of the data.

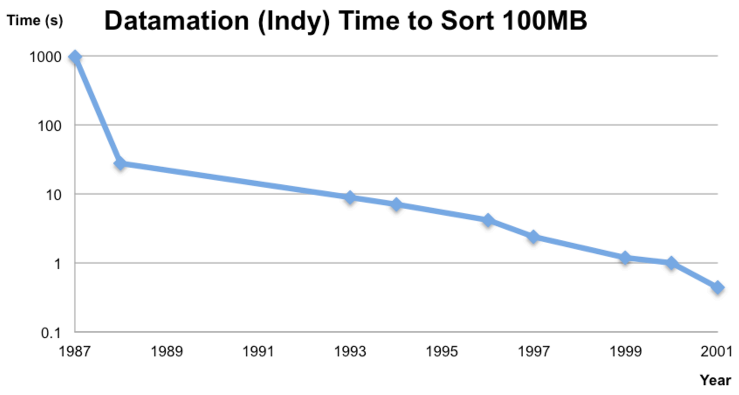
The Daytona approaches generally seem to first sample the input data to determine the distribution and then run the Indy algorithm with the calculated distributions.



ETH zürich

Sortbenchmark.org history

- Datamation: Sort 100MB (deprecated)



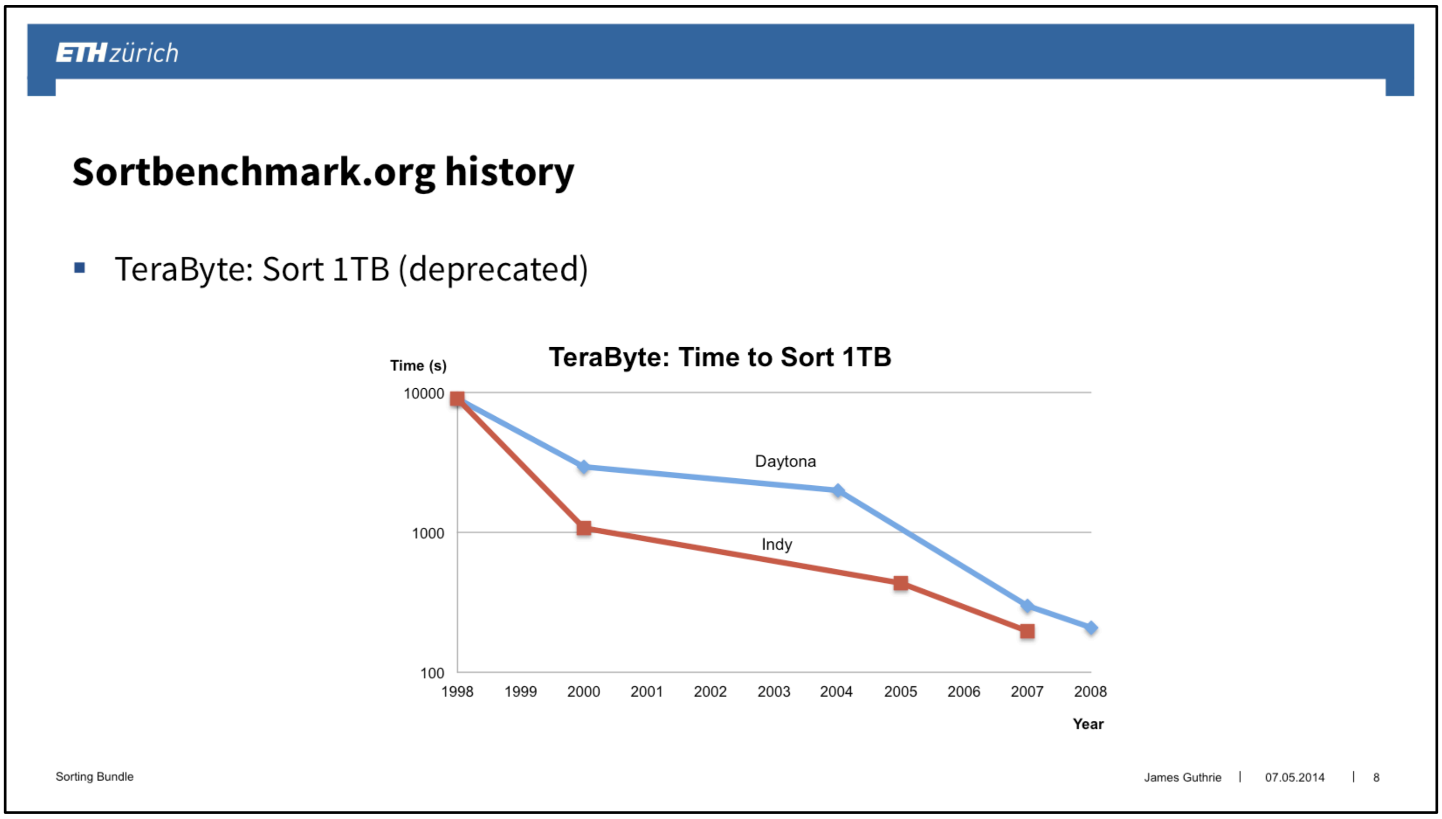
Sorting Bundle James Guthrie | 07.05.2014 | 7

Now that we have a feeling for what the sort benchmark is about, let's see what has happened historically.

The origins of the sort benchmark are the "Datamation Sort", which was the original sort benchmark. The metric is time to sort 100MB (in seconds). At this point in time the Daytona category didn't exist yet, so these results are Indy only.

Note that the y axis on this graph is logarithmic! What we see is that the time taken to sort 100MB went from 980s in 1987 to 0.44s in 2001, when the Datamation Sort was retired.

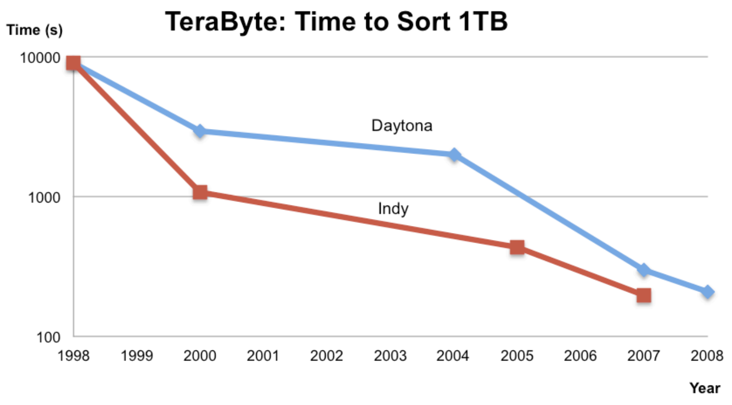
The record-holder in the sort benchmarks has constantly changed from year to year.



ETH zürich

Sortbenchmark.org history

- TeraByte: Sort 1TB (deprecated)



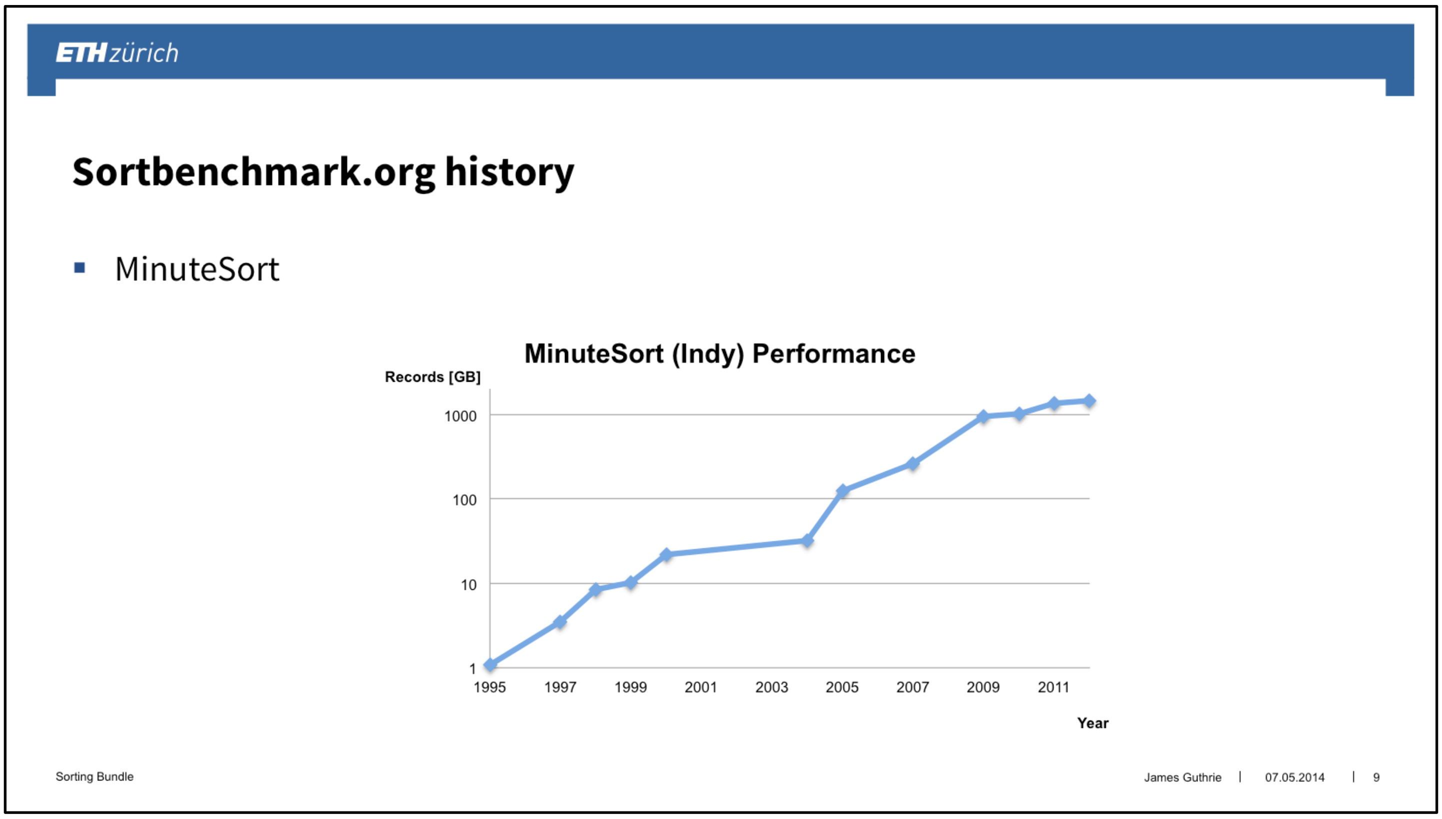
Sorting Bundle

James Guthrie | 07.05.2014 | 8

Next up in the historical timeline is the TeraByte Sort, which I assume came to life when they realised that the DataMation Sort was coming to the end of its useful life. Here the goalpost was shifted to a Terabyte of data to be sorted.

We see that the time taken to sort 1TB of data went from 9060s in 1998 to 196s (indy) and 208s (Daytona) in 2008, the last year that the benchmark was held.

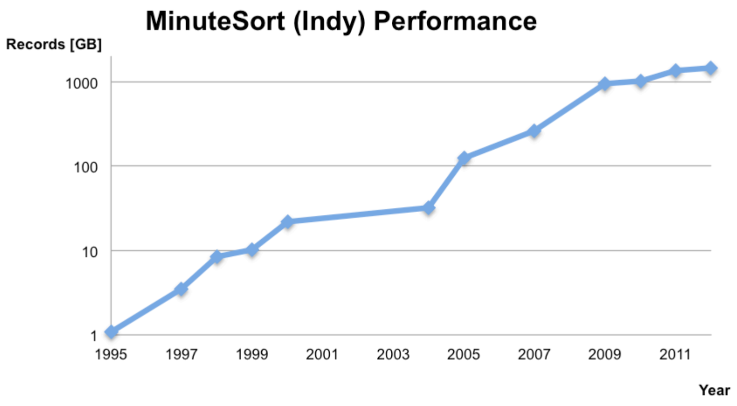
These sorts were both similar to the modern GraySort, with the important distinction that in the GraySort the metric is TB/min and not time taken to sort an amount of data. This allows the volume of data to be sorted to increase with advances in technology.



ETH zürich

Sortbenchmark.org history

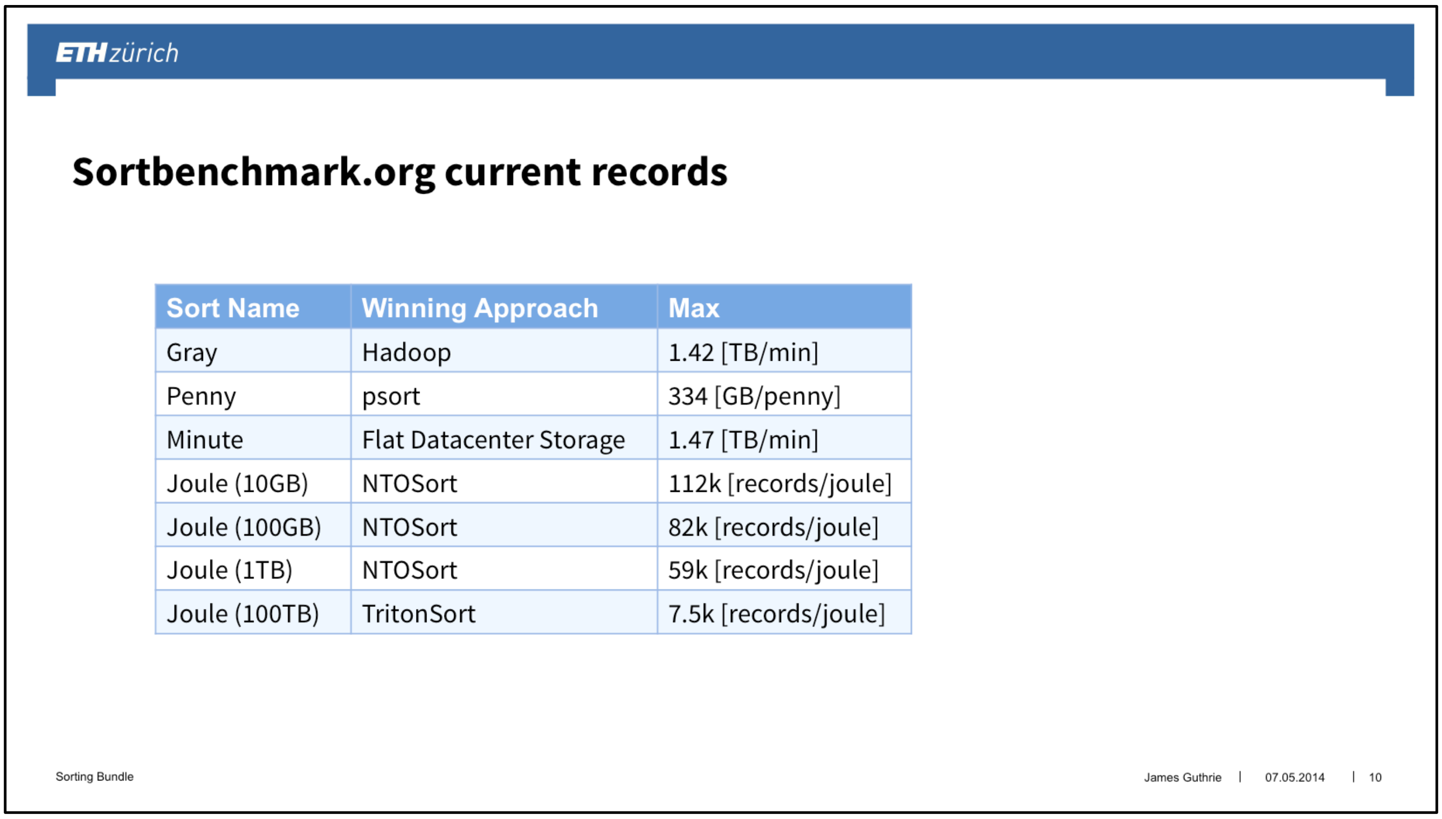
- MinuteSort



Sorting Bundle

James Guthrie | 07.05.2014 | 9

Finally we see the MinuteSort performance, which from 1995 to 2011 has gone from sorting 1.08GB to 1.47TB in a minute.


ETH zürich

Sortbenchmark.org current records

Sort Name	Winning Approach	Max
Gray	Hadoop	1.42 [TB/min]
Penny	psort	334 [GB/penny]
Minute	Flat Datacenter Storage	1.47 [TB/min]
Joule (10GB)	NTOSort	112k [records/joule]
Joule (100GB)	NTOSort	82k [records/joule]
Joule (1TB)	NTOSort	59k [records/joule]
Joule (100TB)	TritonSort	7.5k [records/joule]

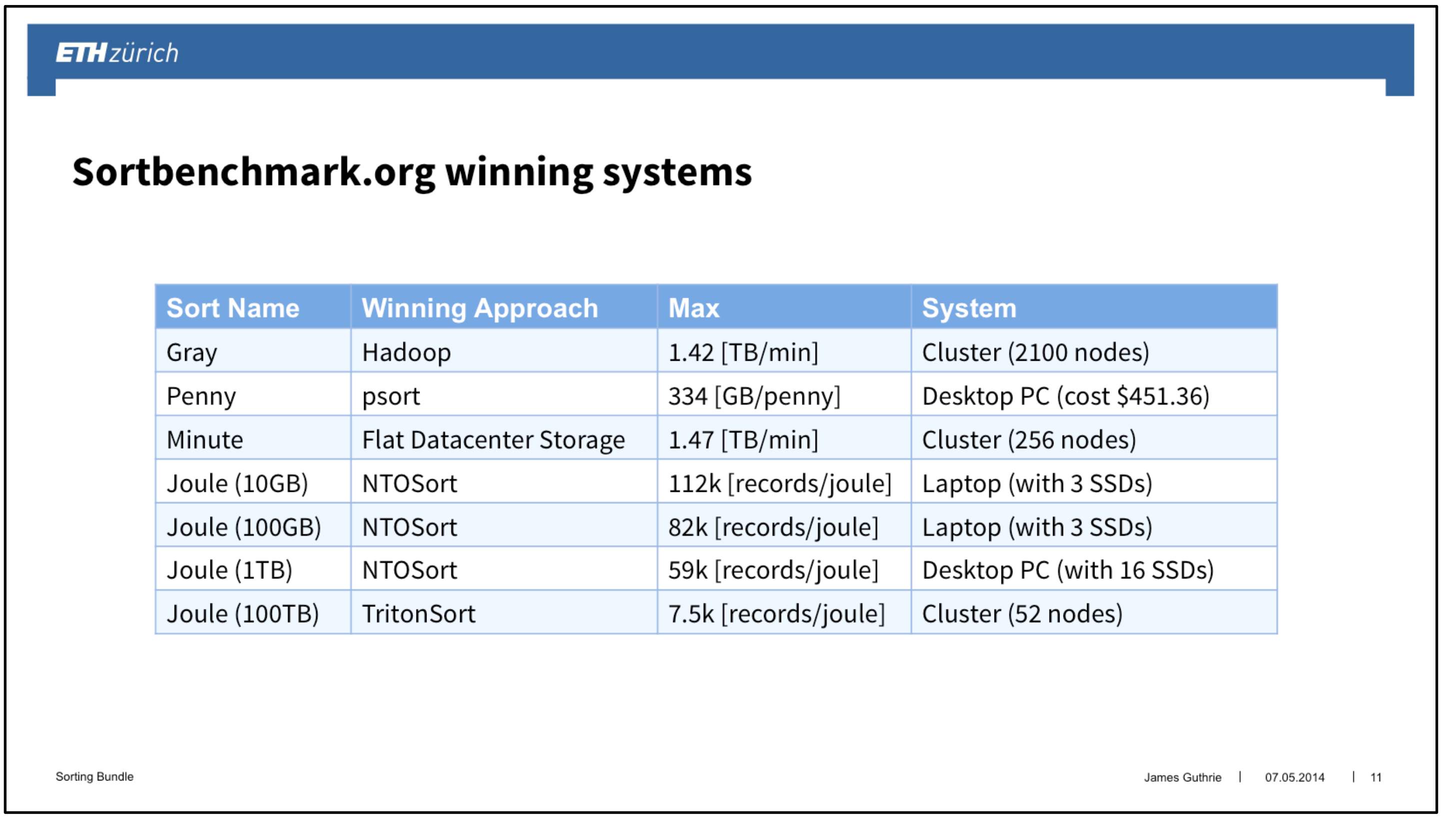
Sorting Bundle   James Guthrie | 07.05.2014 | 10

So what do the current world records look like?

(***) Note that the records shown are all for the Indy category.

Okay, so there are a lot of numbers here - I think the most interesting thing to look at is the progression in the JouleSort: the number of records per joule steadily decreases (with increasing input size) and then takes a huge jump between 1TB and 100TB.

On the next slide we'll see what may have been the reason for this.



ETH zürich

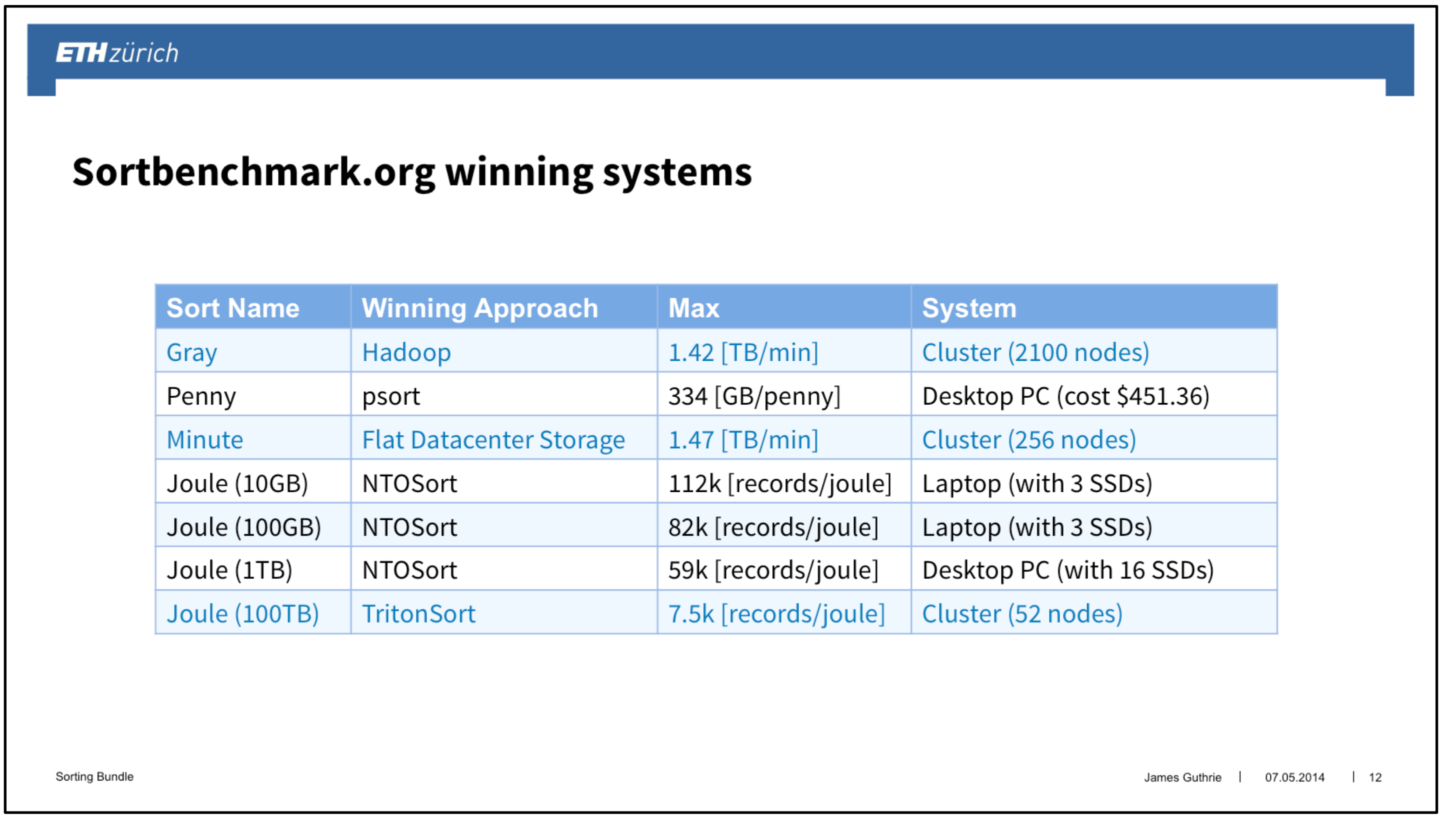
Sortbenchmark.org winning systems

Sort Name	Winning Approach	Max	System
Gray	Hadoop	1.42 [TB/min]	Cluster (2100 nodes)
Penny	psort	334 [GB/penny]	Desktop PC (cost $451.36)
Minute	Flat Datacenter Storage	1.47 [TB/min]	Cluster (256 nodes)
Joule (10GB)	NTOSort	112k [records/joule]	Laptop (with 3 SSDs)
Joule (100GB)	NTOSort	82k [records/joule]	Laptop (with 3 SSDs)
Joule (1TB)	NTOSort	59k [records/joule]	Desktop PC (with 16 SSDs)
Joule (100TB)	TritonSort	7.5k [records/joule]	Cluster (52 nodes)

Sorting Bundle   James Guthrie | 07.05.2014 | 11

Here we see what kind of system won the respective benchmark, which gives us an idea of why the JouleSort with 100TB of data performed much worse than the JouleSort with 1TB of data: the former was done on a cluster of 52 nodes whereas the latter on a single desktop PC. What I also find interesting is that the winning entries for JouleSort under 1TB are laptop computers which are designed to be as energy efficient as possible.

The winning system in the PennySort is whatever they could cobble together for the smallest amount of money possible ($450 for the whole computer).


ETH zürich

Sortbenchmark.org winning systems

Sort Name	Winning Approach	Max	System
Gray	Hadoop	1.42 [TB/min]	Cluster (2100 nodes)
Penny	psort	334 [GB/penny]	Desktop PC (cost $451.36)
Minute	Flat Datacenter Storage	1.47 [TB/min]	Cluster (256 nodes)
Joule (10GB)	NTOSort	112k [records/joule]	Laptop (with 3 SSDs)
Joule (100GB)	NTOSort	82k [records/joule]	Laptop (with 3 SSDs)
Joule (1TB)	NTOSort	59k [records/joule]	Desktop PC (with 16 SSDs)
Joule (100TB)	TritonSort	7.5k [records/joule]	Cluster (52 nodes)

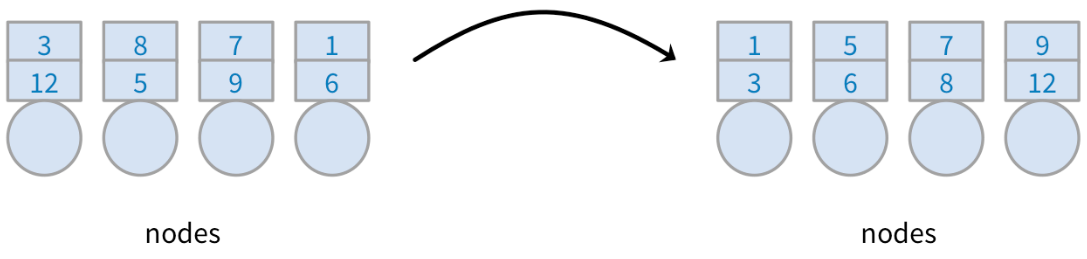
Sorting Bundle James Guthrie | 07.05.2014 | 12

Seeing as we are in a seminar about distributed computing, possibly the most interesting for us are the three clusters which win in the GraySort, MinuteSort and JouleSort (100TB) categories.

I thought it would be interesting to present the various approaches taken by the different winners and try to understand what it was about the different approaches that made them better suited to win in the category that they did.

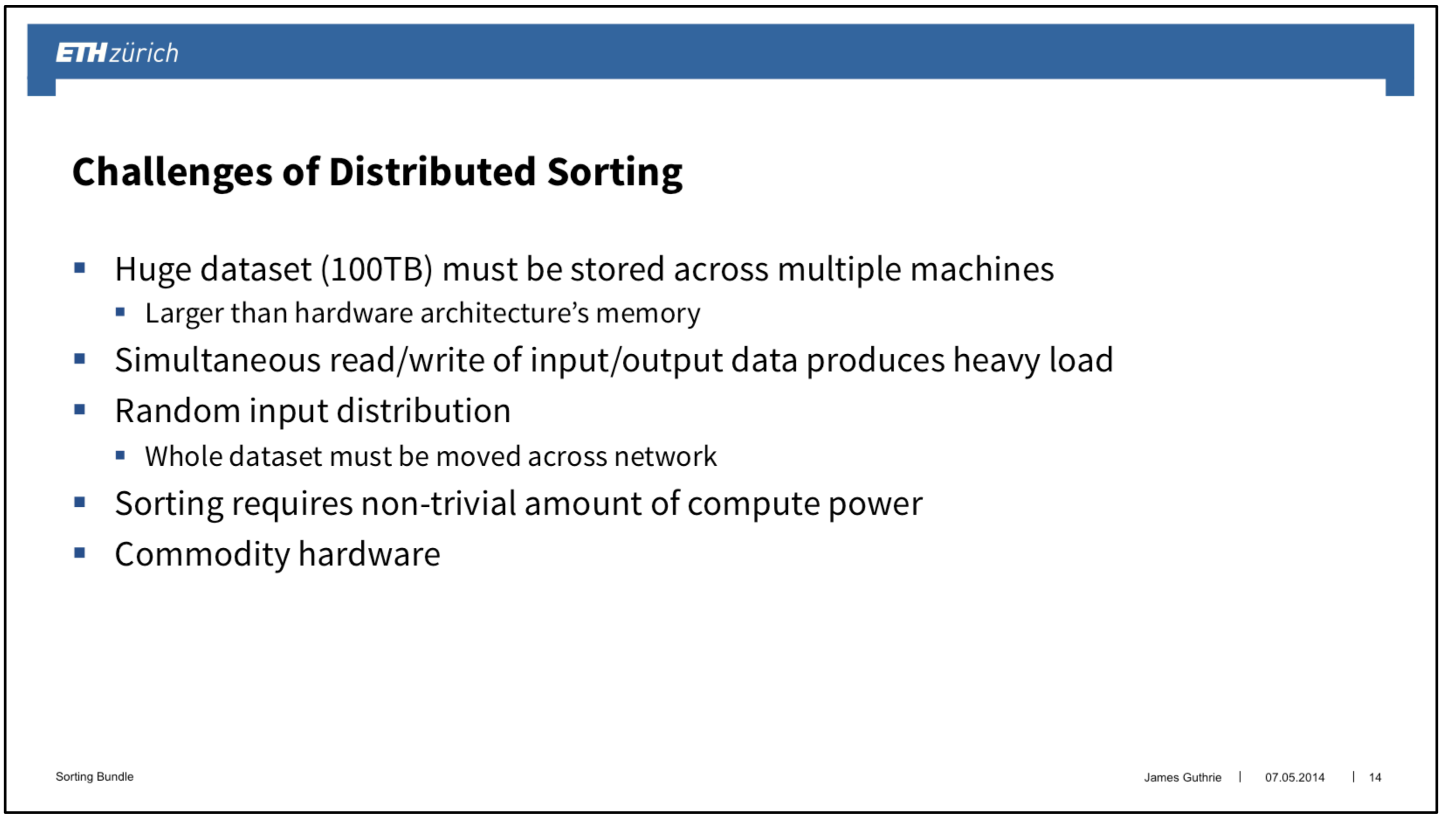
ETH zürich

Setting The Scene For Distributed Approaches



So we've already seen the rules established for the sort benchmark, but how should they be applied in a distributed approach?

Firstly, the unsorted input data may be split into individual files.
(***) Here we have a simplified version which shows four nodes, each with an input file with two records.
(***) What we want to see is a transformation which results in the sorted data being split among individual files
A concatenation of the sorted output files must result in the sorted version of the input data

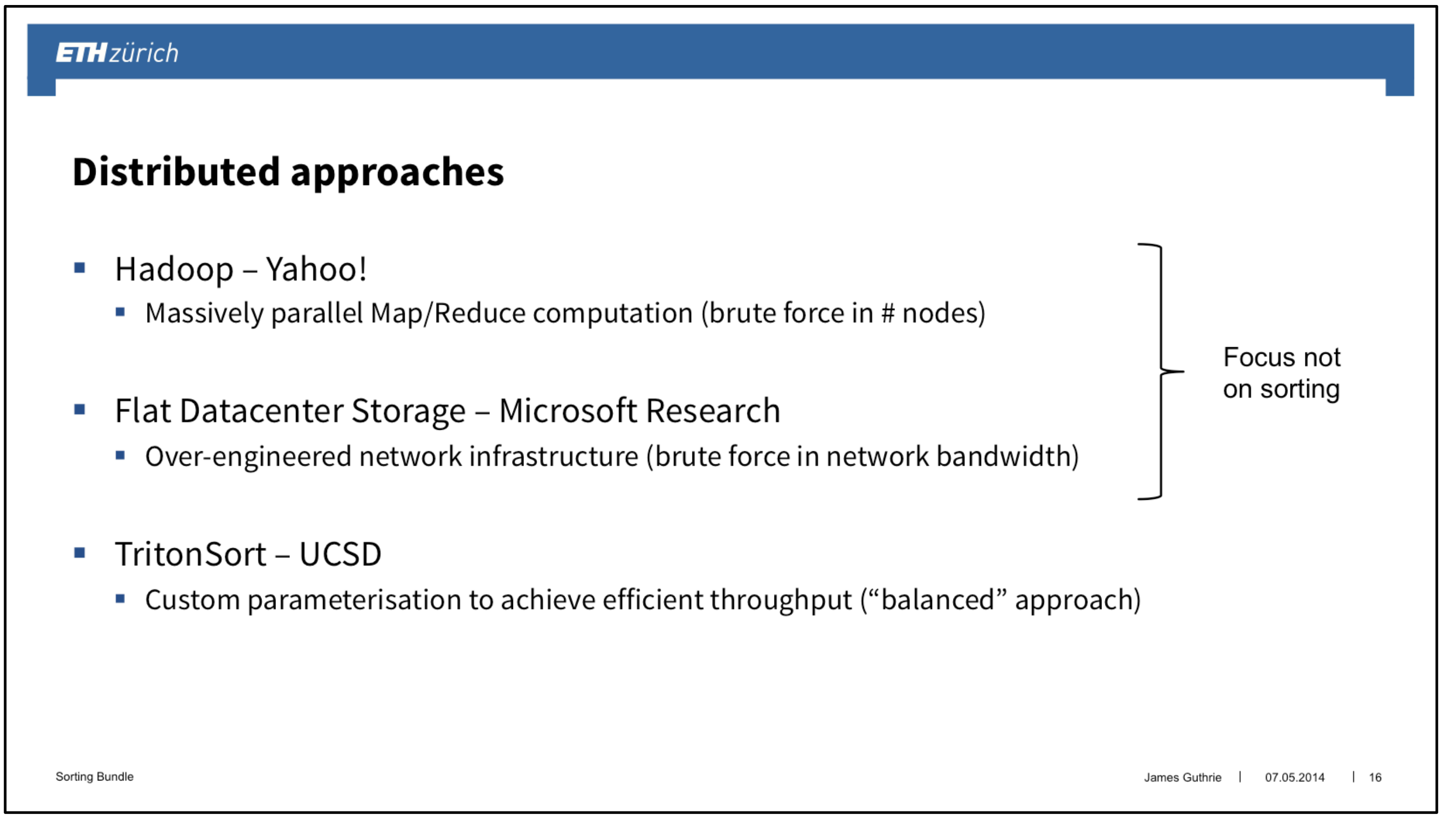

ETH zürich

Challenges of Distributed Sorting

- Huge dataset (100TB) must be stored across multiple machines
  - Larger than hardware architecture's memory
- Simultaneous read/write of input/output data produces heavy load
- Random input distribution
  - Whole dataset must be moved across network
- Sorting requires non-trivial amount of compute power
- Commodity hardware

Sorting Bundle James Guthrie | 07.05.2014 | 14

There are also a number of challenges that the benchmarks pose which make a distributed approach both necessary and non-trivial.



ETH zürich

Distributed approaches

- Hadoop – Yahoo!
  - Massively parallel Map/Reduce computation (brute force in # nodes)
- Flat Datacenter Storage – Microsoft Research
  - Over-engineered network infrastructure (brute force in network bandwidth)
- TritonSort – UCSD
  - Custom parameterisation to achieve efficient throughput ("balanced" approach)

$\left.\begin{array}{l} \\ \\ \end{array}\right\}$ Focus not on sorting

Sorting Bundle

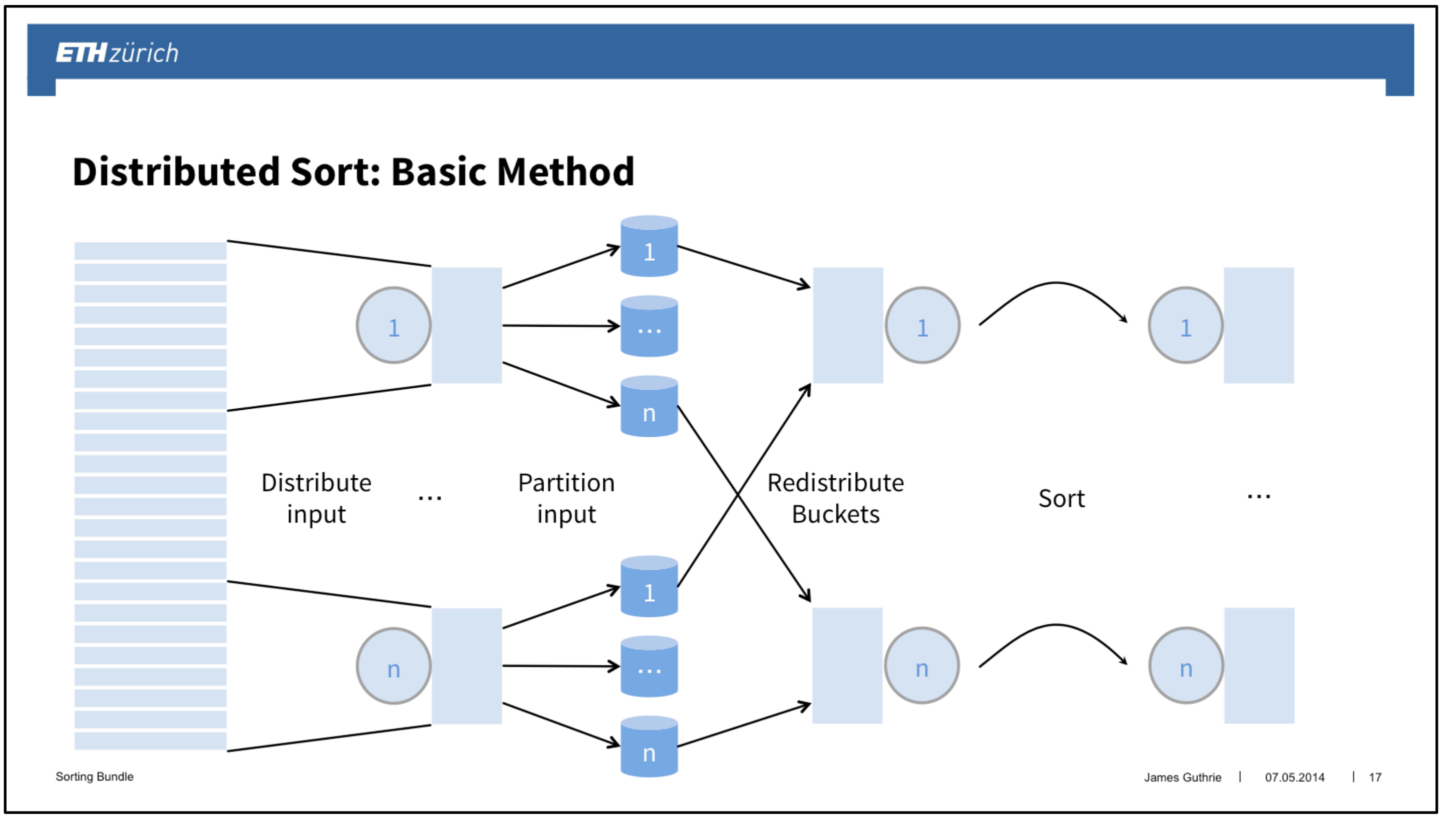
James Guthrie | 07.05.2014 | 16

Let's take a slightly closer look at the three distributed approaches that we're going to look at today.

(***) Hadoop's approach is a massively parallel Map/Reduce computation, where the computational power comes from the sheer number of nodes.

(***) Flat Datacenter Storage's approach is to over-engineer the network infrastructure – sort of a brute force in network bandwidth.

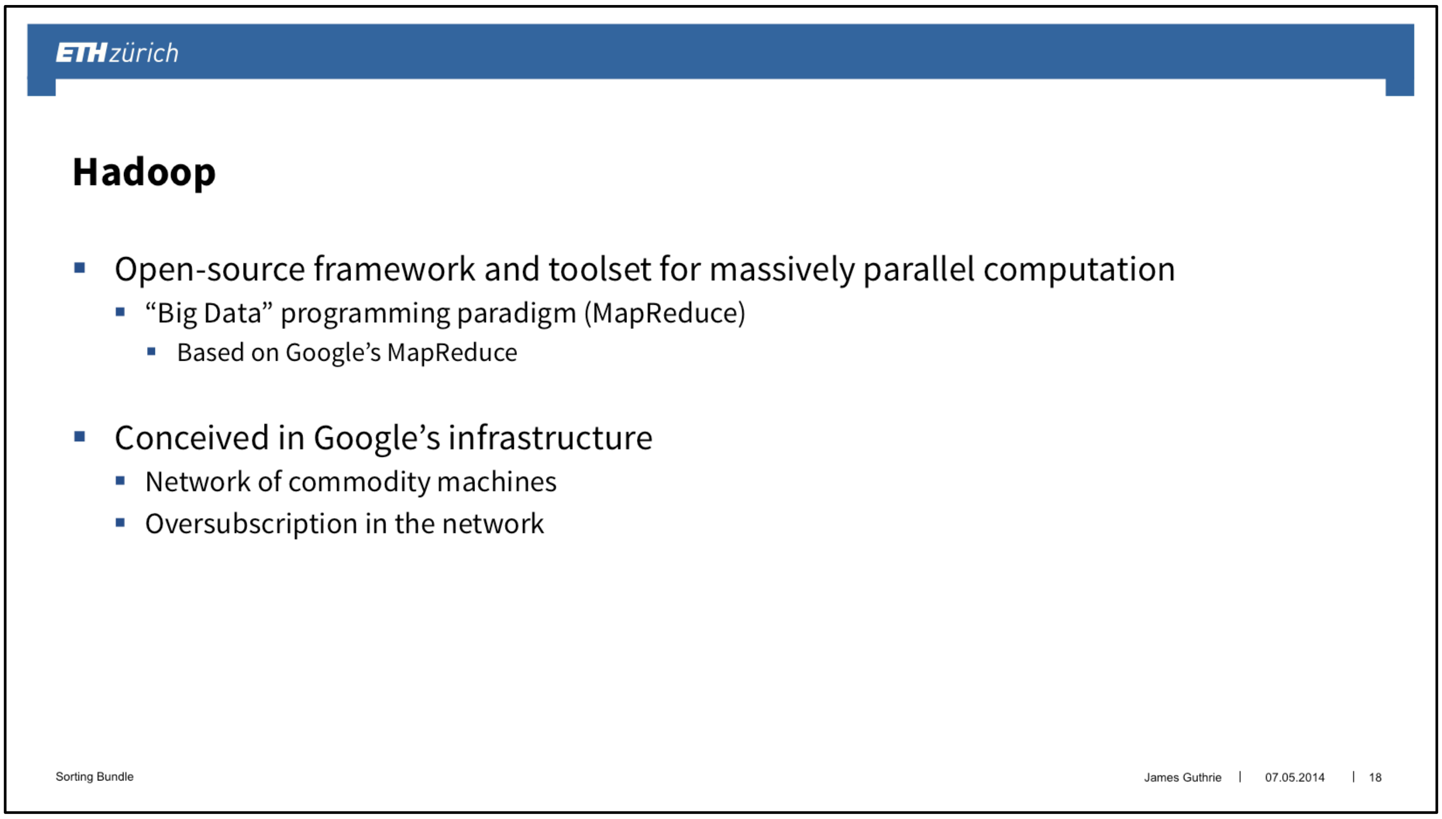
(***) TritonSort's approach is to parameterise the individual components in such a way as to achieve incredibly efficient processing.



As all of the distributed approaches use the same basic method to sort the data, I will present it here before we look at how the individual approaches perform the sort.

So we start with the input dataset, a large number of records, whose keys are evenly distributed across the keyspace.

(***) Let's assume that we have n nodes.
(***) We divide the input space evenly such that each node has an equal number of input records.
What we want to do with the input records is move them to the node where they will be when all the data is sorted. We're not really sorting yet, just distributing the input data.
We can do this because each node knows what the key distribution is, so they can (individually) deterministically allocate each input record to a destination node. Node 1 will have the first 1/nth of the keyspace assigned to it, node x the x 1/nth of the data.
(***) So now each node partitions its keyspace into n buckets, each containingan approximately equal number of records (because the input space is evenly distributed).
When the input space has been distributed into buckets, the buckets are redistributed. Each node receives n-1 buckets, one from every other node.
(***) Node 1 receives all buckets labeled "1" from all other nodes
(***) and node n receives all buckets labeled "n" from all other nodes
(***) Now each node merely has to sort all of the data in the buckets it received and then the story is over. The complete dataset is sorted from node 1 to n.


ETH zürich

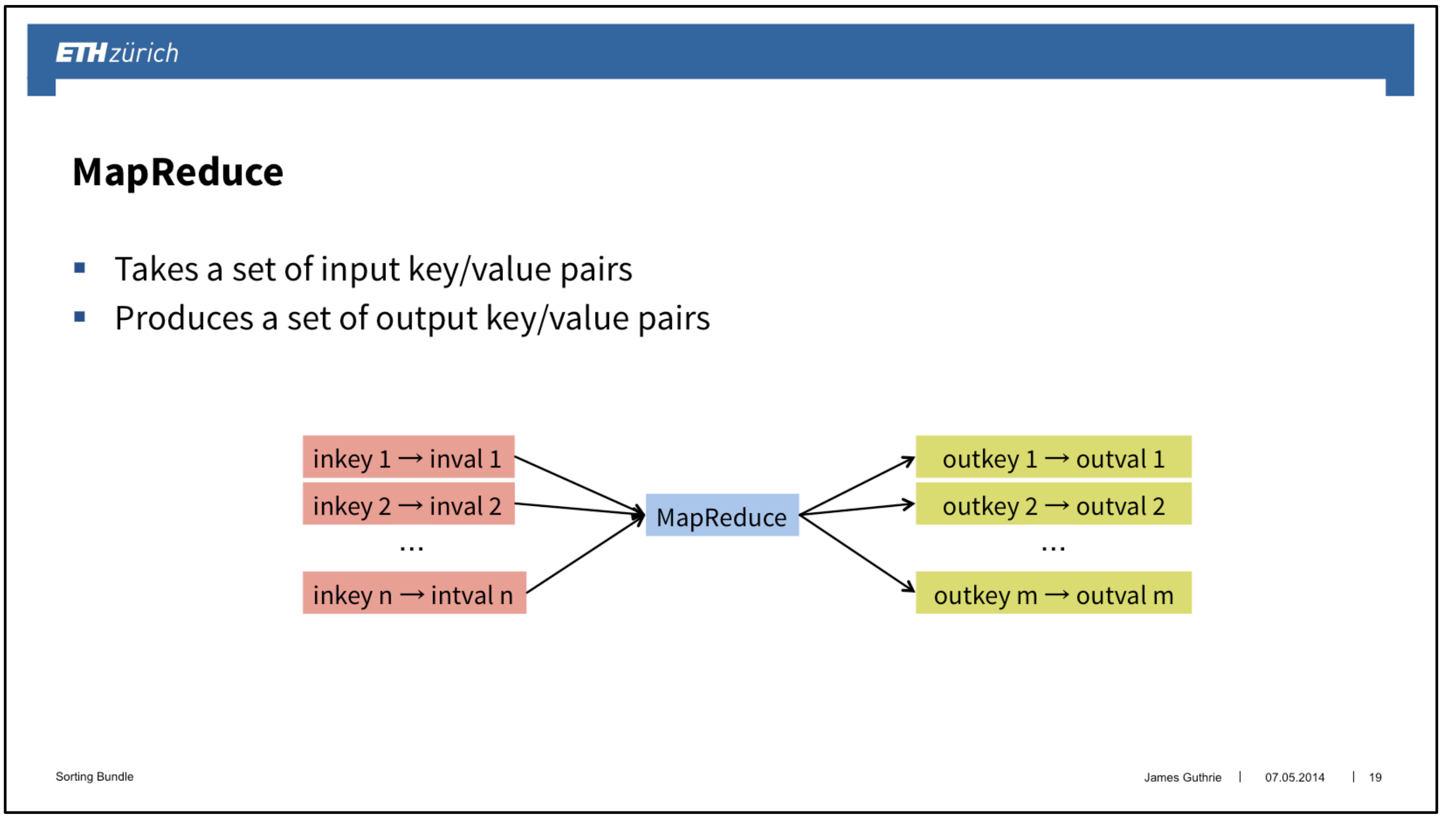
Hadoop

- Open-source framework and toolset for massively parallel computation
  - "Big Data" programming paradigm (MapReduce)
    - Based on Google's MapReduce
- Conceived in Google's infrastructure
  - Network of commodity machines
  - Oversubscription in the network

Sorting Bundle James Guthrie | 07.05.2014 | 18

Hadoop is an open-source framework based on the MapReduce programming paradigm, conceived at Google and most importantly in Google's infrastructure.

The constraint in Google's infrastructure is that it's a network of commodity machines – most importantly that there is oversubscription in the network. This means that as one goes up the hierarchy of machines, there is not enough available bandwidth for all leaf nodes to communicate over the root node.


ETH zürich

MapReduce

- Takes a set of input key/value pairs
- Produces a set of output key/value pairs

$$\begin{tikzcd}
\text{inkey 1} \to \text{inval 1} \arrow[dr] & & \text{outkey 1} \to \text{outval 1} \\
\text{inkey 2} \to \text{inval 2} \arrow[r] & \text{MapReduce} \arrow[ur] \arrow[r] \arrow[dr] & \text{outkey 2} \to \text{outval 2} \\
\dots & & \dots \\
\text{inkey n} \to \text{intval n} \arrow[uur] & & \text{outkey m} \to \text{outval m}
\end{tikzcd}$$

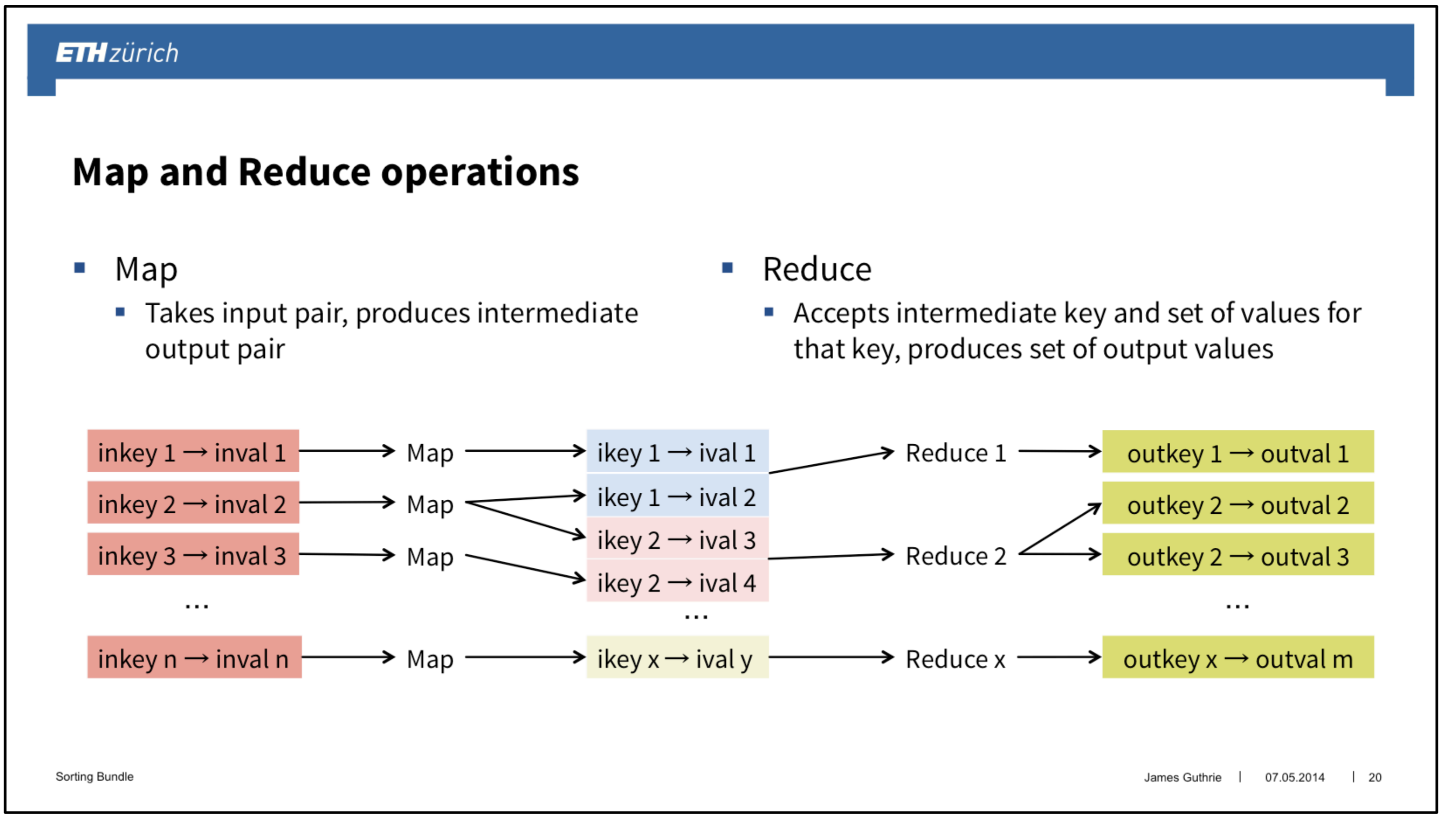
Sorting Bundle

James Guthrie | 07.05.2014 | 19

Some of you may have heard of the MapReduce programming paradigm.

MapReduce is essentially a function that
(***) takes a set of input key/value pairs
(***) and produces a set of output key/value pairs

That’s the 30 000 foot view of MapReduce.



ETH zürich

Map and Reduce operations

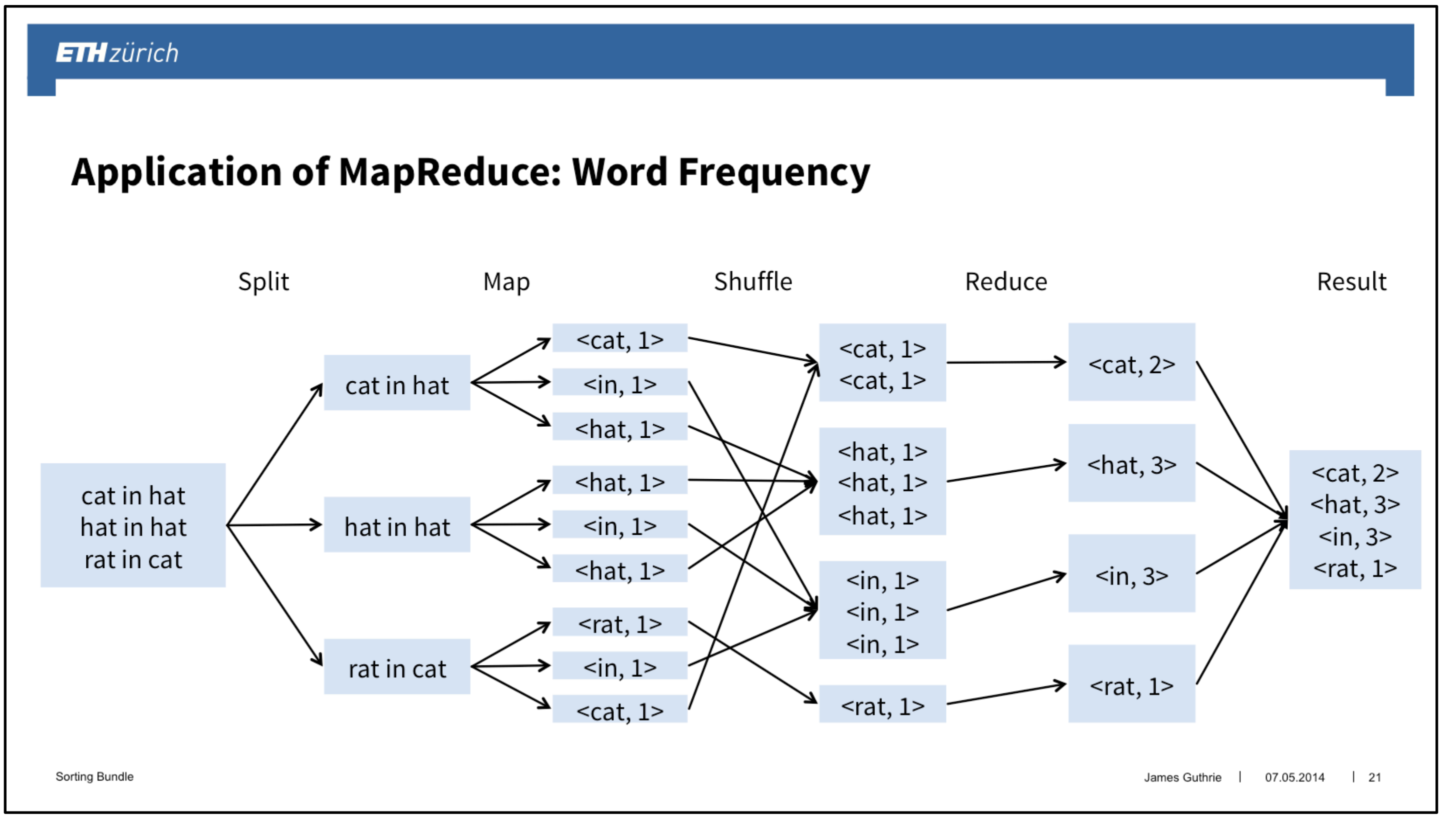
- Map
  - Takes input pair, produces intermediate output pair
- Reduce
  - Accepts intermediate key and set of values for that key, produces set of output values

$$\begin{tikzcd}
\text{inkey 1} \rightarrow \text{inval 1} \arrow[r] & \text{Map} \arrow[r] & \text{ikey 1} \rightarrow \text{ival 1} \arrow[r] & \text{Reduce 1} \arrow[r] & \text{outkey 1} \rightarrow \text{outval 1} \\
\text{inkey 2} \rightarrow \text{inval 2} \arrow[r] & \text{Map} \arrow[r] \arrow[dr] & \text{ikey 1} \rightarrow \text{ival 2} & & \text{outkey 2} \rightarrow \text{outval 2} \\
\text{inkey 3} \rightarrow \text{inval 3} \arrow[r] & \text{Map} \arrow[dr] & \text{ikey 2} \rightarrow \text{ival 3} \arrow[r] & \text{Reduce 2} \arrow[r] \arrow[ur] & \text{outkey 2} \rightarrow \text{outval 3} \\
 & & \text{ikey 2} \rightarrow \text{ival 4} & & \\
\cdots & & \cdots & & \cdots \\
\text{inkey n} \rightarrow \text{inval n} \arrow[r] & \text{Map} \arrow[r] & \text{ikey x} \rightarrow \text{ival y} \arrow[r] & \text{Reduce x} \arrow[r] & \text{outkey x} \rightarrow \text{outval m}
\end{tikzcd}$$

So let’s take a closer look at MapReduce.

MapReduce is made up of two functions:

1. a “Map”, which takes an input key/value pair and produces an intermediate key/value pair.
2. a “Reduce” function which takes all values belonging to an intermediate key/value pair and outputs one or more values.


ETH zürich

Application of MapReduce: Word Frequency

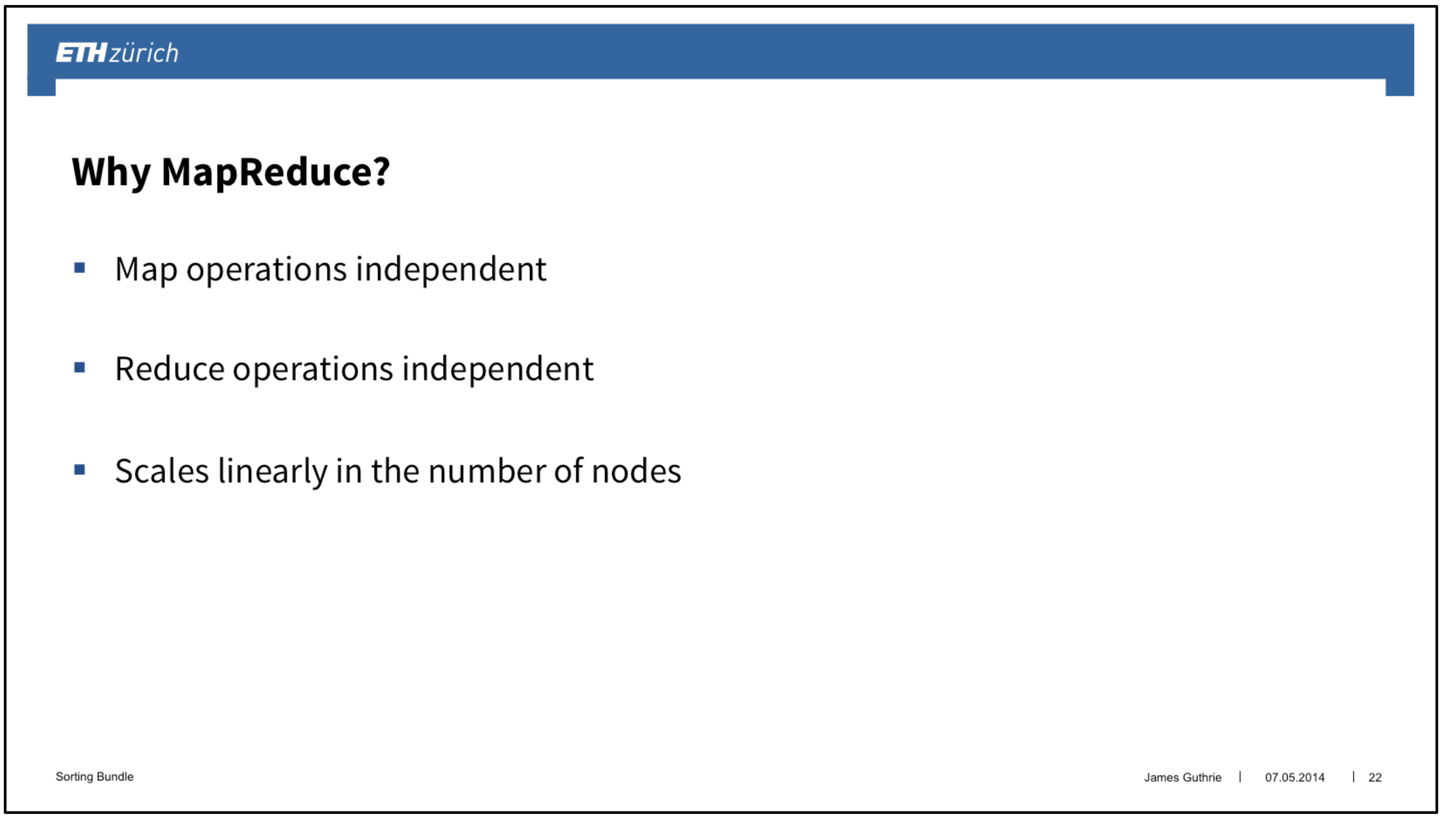
Sorting Bundle

James Guthrie | 07.05.2014 | 21

So how do we actually apply MapReduce to a problem?

Let’s take a look at the problem of counting the frequency of words in a file. We start with the sample file on the left with three lines of three words each.

(***) In the split phase we split the input data into three pieces (three chosen arbitrarily in this case)
(***) The map function in this case outputs each word it saw in the input as a key with the value ‘1’ (essentially “I saw one <x>”)
(***) This for all maps. Note that also when “hat” appears twice that the output is not <hat, 2> (which we could also do, but for simplicity’s sake we will ignore)
(***) Next comes the shuffle phase in which all values belonging to the same key are collected together
(***) The Reduce stage takes all the values from the shuffle and in this case sums up the number of times a word was seen.
(***) The output of the reduce stage forms the output of the entire computation


ETH zürich

Why MapReduce?

- Map operations independent
- Reduce operations independent
- Scales linearly in the number of nodes

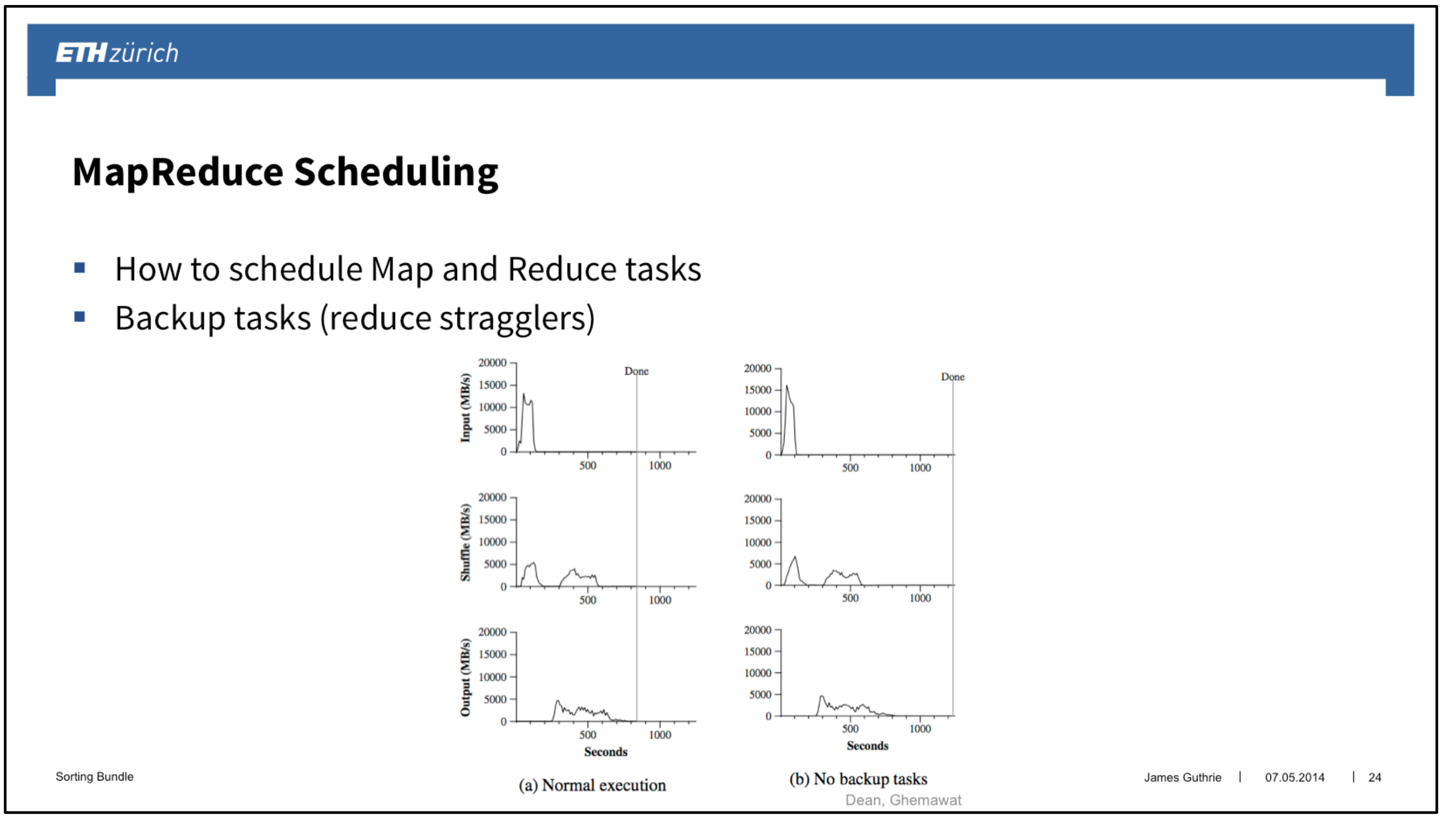
Sorting Bundle James Guthrie | 07.05.2014 | 22

So what is so great about MapReduce?

(***) Well the first thing to note is that Map is very parallel. Each map task uses a disjoint part of the dataset, making them independently and simultaneously schedulable.

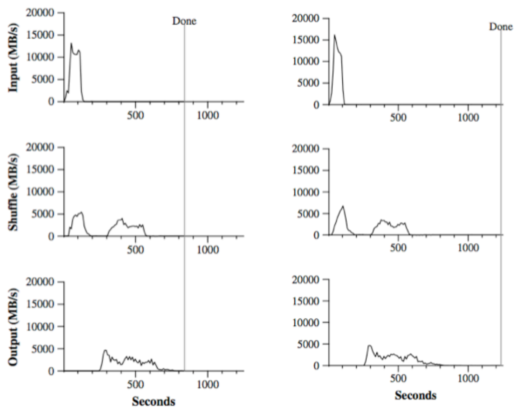
(***) In the same way, the Reduce tasks can also be run in parallel.

(***) The great thing about these “independent” operations is that MapReduce is linearly scalable: if the Map is compute-bound, throw more computers at it. And this has been the approach that “big data shops” have taken. According to some counts, Yahoo! has over 40000 machines in clusters.


ETH zürich

MapReduce Scheduling

- How to schedule Map and Reduce tasks
- Backup tasks (reduce stragglers)



(a) Normal execution (b) No backup tasks

Dean, Ghemawat

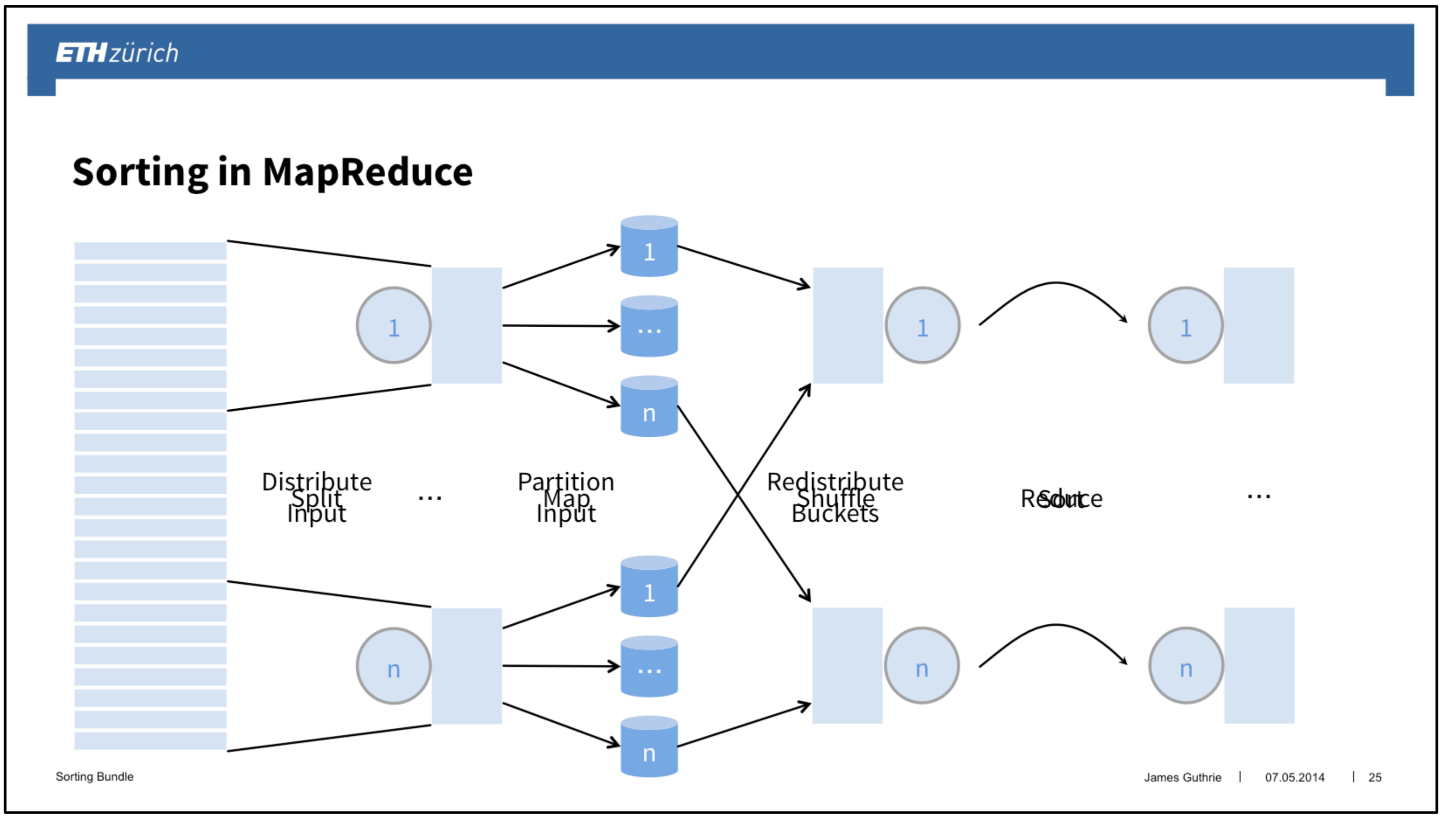
Sorting Bundle

James Guthrie | 07.05.2014 | 24

One of the problems that comes with any compute job is how we schedule the computation. In this case the question is more where than how. Specifically, where should the Map and Reduce operations be scheduled?
In single-node approaches this isn't a problem. The data is locally-attached, so we tell the OS to fetch the data, run the computation and then write the values back.
In a cluster of nodes the data may not be local to the node that is performing the calculation, MapReduce solves this by taking advantage of the fact that the Distributed File System which MapReduce runs on top of gives information as to the machine on which a given piece of data resides. This allows for Map and Reduce tasks to be scheduled on the machines that have the data stored locally.

Another problem that arose during the development of MapReduce is the problem of stragglers. In an ideal world, the work to be done could be split evenly amongst all nodes in the cluster and all nodes would finish at approximately the same time. When developing MapReduce the authors noticed that some jobs took much longer to complete because of defective hardware. The whole computation would have to wait for these few slow jobs to complete.
The authors solution to this is to schedule backup tasks for all in-progress tasks when the MapReduce task is close to completion.

(***) The affect that this has can be seen in this diagram. The left-hand column is run with backup tasks, the right-hand without. Adding backup tasks makes the computation complete in 850 instead of 1200 seconds.



Sorting in MapReduce

Sorting Bundle

James Guthrie | 07.05.2014 | 25

So we've had a look at MapReduce, we've seen an example of how it works for word frequency and we've seen a couple of technicalities – but one question still remains: how does MapReduce sort?

Lets take another look at the general distributed sort approach that I presented earlier.

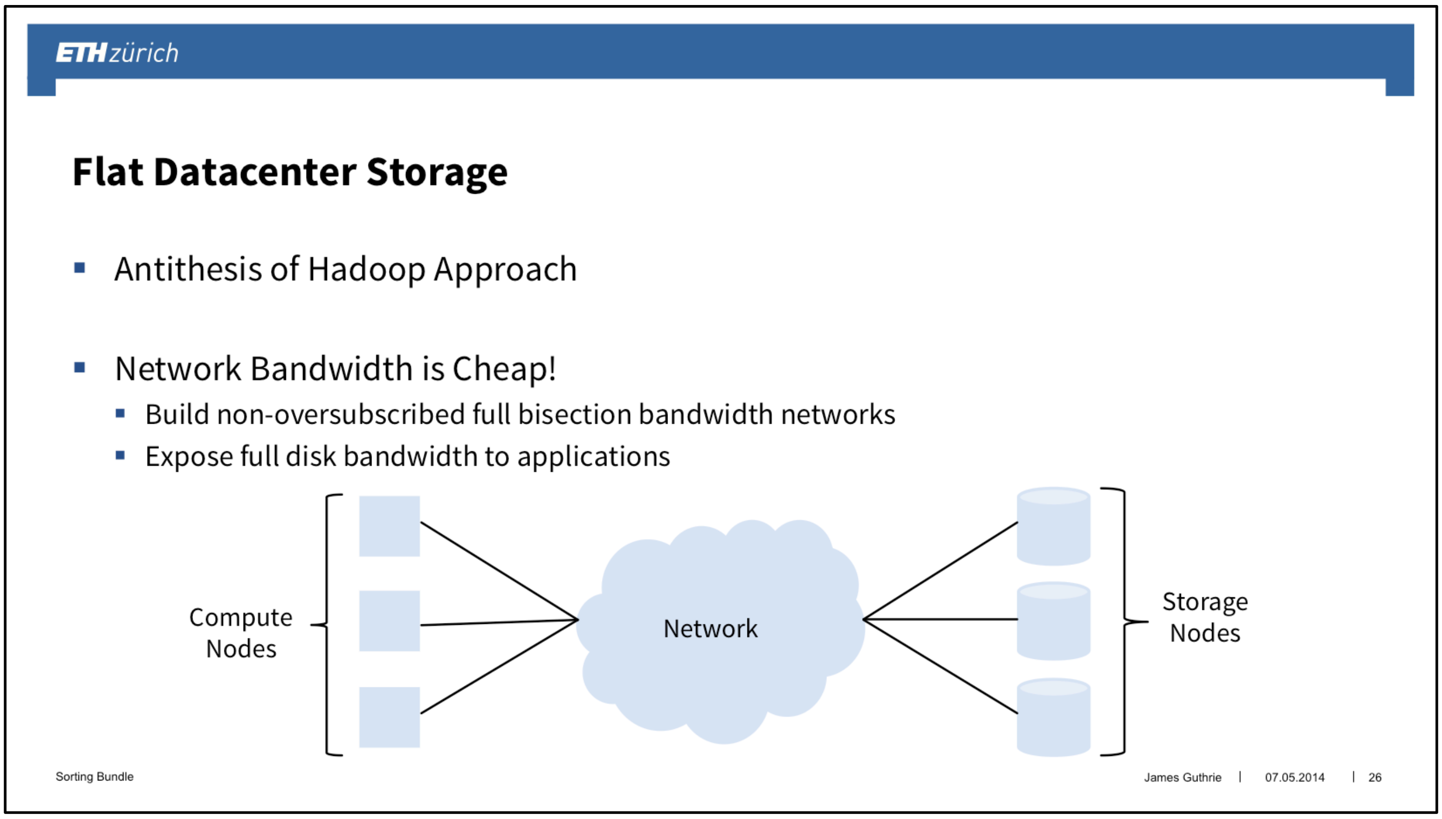
(***) Now that we've seen how MapReduce works, we realise that this looks surprisingly similar to the MapReduce example I just showed you.

In fact, we can just rename things a little and we'll see that we essentially have the same structure as two slides ago.

(***) We'll rename "Distribute input" to "Split"
(***) And we can rename "Partition input" to "Map"
(***) And we'll call "Redistribute Buckets" "Shuffle"
(***) And finally, we'll call "Sort" "Reduce"

Now, this isn't the complete truth. In fact, in MapReduce we can conceptually drop off the Reduce function.

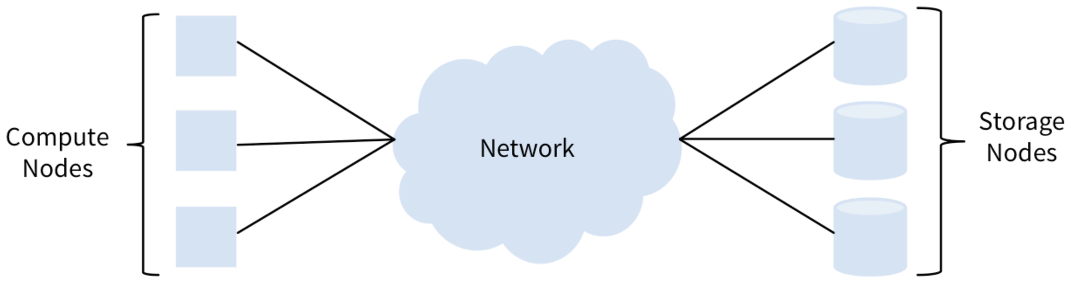
(***) The reason that we can do this is that MapReduce does the sorting for us in the map and shuffle stages. The Map operations produce sorted outputs which are merged before being passed to the Reduce function.
This means that for a MapReduce sort implementation the Reduce function is an



ETH zürich

Flat Datacenter Storage

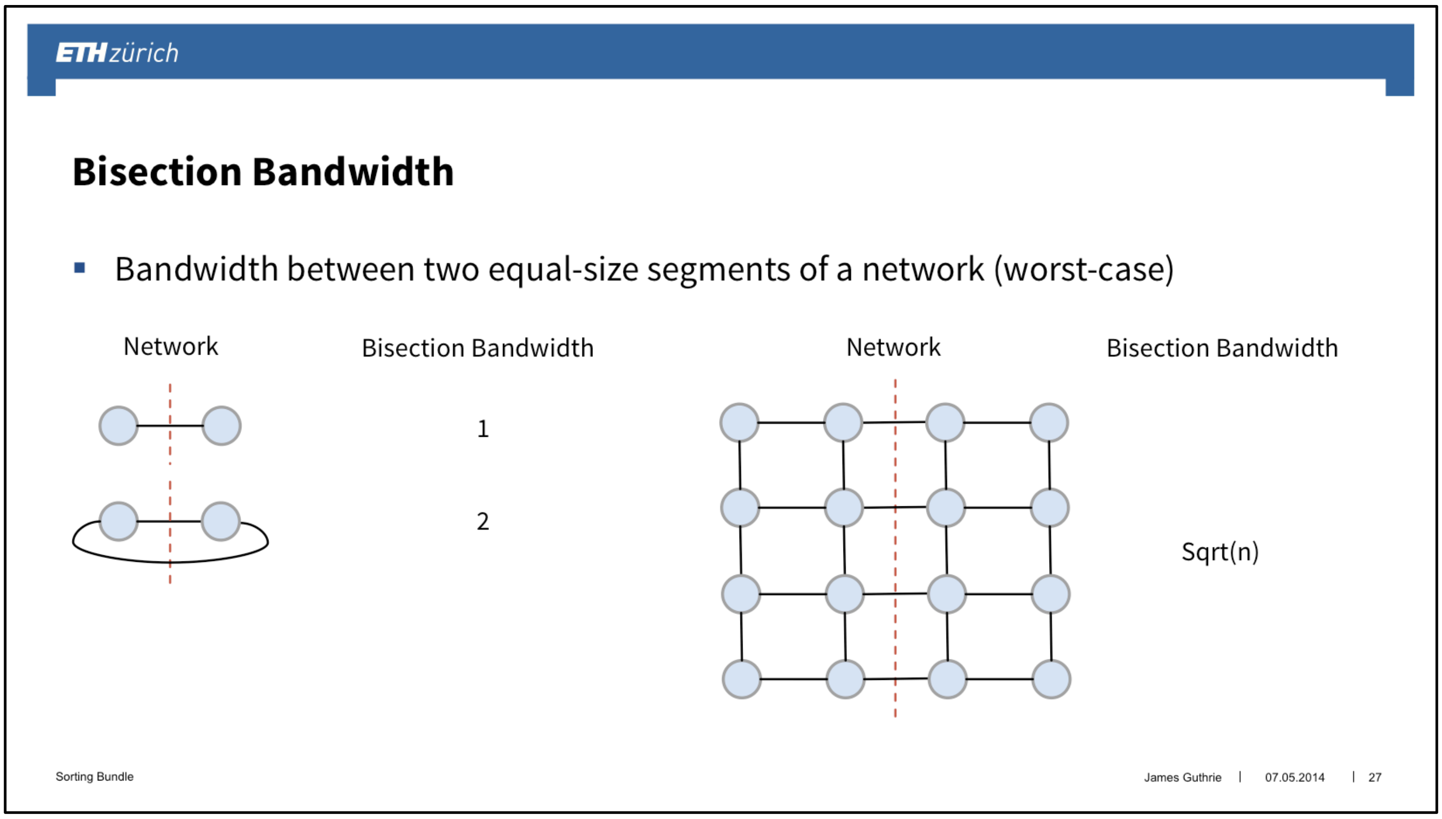
- Antithesis of Hadoop Approach
- Network Bandwidth is Cheap!
  - Build non-oversubscribed full bisection bandwidth networks
  - Expose full disk bandwidth to applications



Sorting Bundle

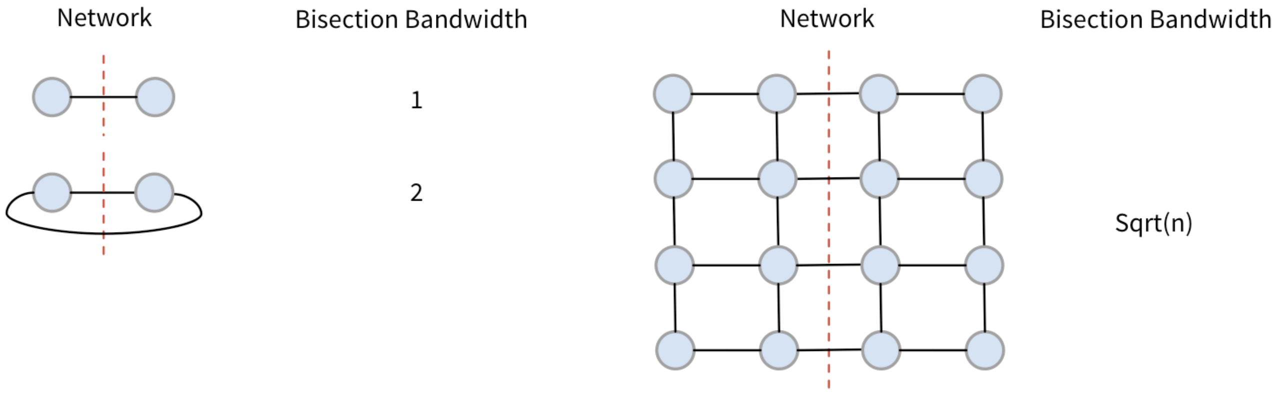
James Guthrie | 07.05.2014 | 26

Now we're going to look at a completely different take on how to solve the sorting problem.
In fact, this is pretty much the antithesis of the Hadoop approach.
As we just saw, Hadoop relies on the map/reduce paradigm (in particular the scheduling) to solve the problem of constrained network bandwidth in the datacenter.
This works incredibly well for applications which have high reduction factors but is not as effective for applications such as sort, matrix-matrix multiply and distributed database joins which have low reduction factors.
(***) FDS scales out the network such that it's not a bottleneck, allowing application developers to write their applications as though the storage was locally-attached.



Bisection Bandwidth

- Bandwidth between two equal-size segments of a network (worst-case)



Sorting Bundle James Guthrie | 07.05.2014 | 27

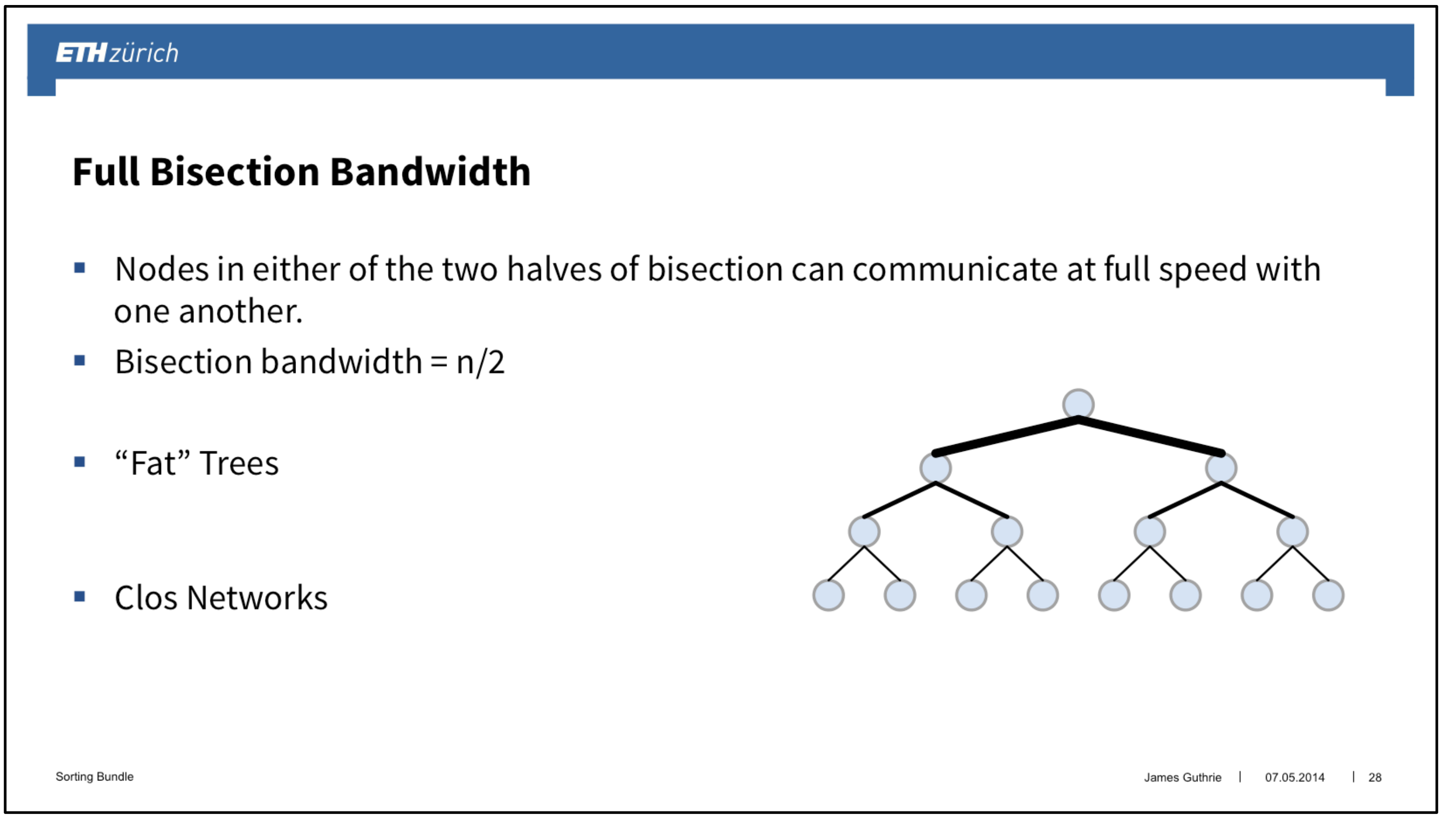
The bisection bandwidth of a network is determined by dividing the network into two equal-sized segments and measuring the bandwidth between the two segments. For any network there are a number of different partitions which result in equal-sized segments, what we are interested in is the worst-case bisection.

In the simplest case, we have a linked-list type connection of n nodes. We can divide this network into two segments with n/2 nodes each, with one link crossing between the two segments.
(***) Thus the bisection bandwidth is one.

(***) In the second case we have a ring topology. Once again we can divide this into two segments with n/2 nodes each but this time with two links crossing between the two segments.
(***) Thus the bisection bandwidth is two.

In the next case we have a 2d mesh of n nodes. We can divide this into two segments of n/2 nodes, with sqrt(n) links crossing between the two segments.
(***) Thus the bisection bandwidth is sqrt(n).

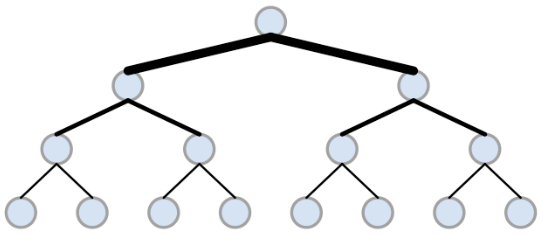
All in all, the bisection bandwidth tells us what the worst-case capacity of the network is. Lower bisection bandwidth means that the network is less capable of providing uncongested communication to all nodes.



ETH zürich

Full Bisection Bandwidth

- Nodes in either of the two halves of bisection can communicate at full speed with one another.
- Bisection bandwidth = n/2
- “Fat” Trees
- Clos Networks



Sorting Bundle

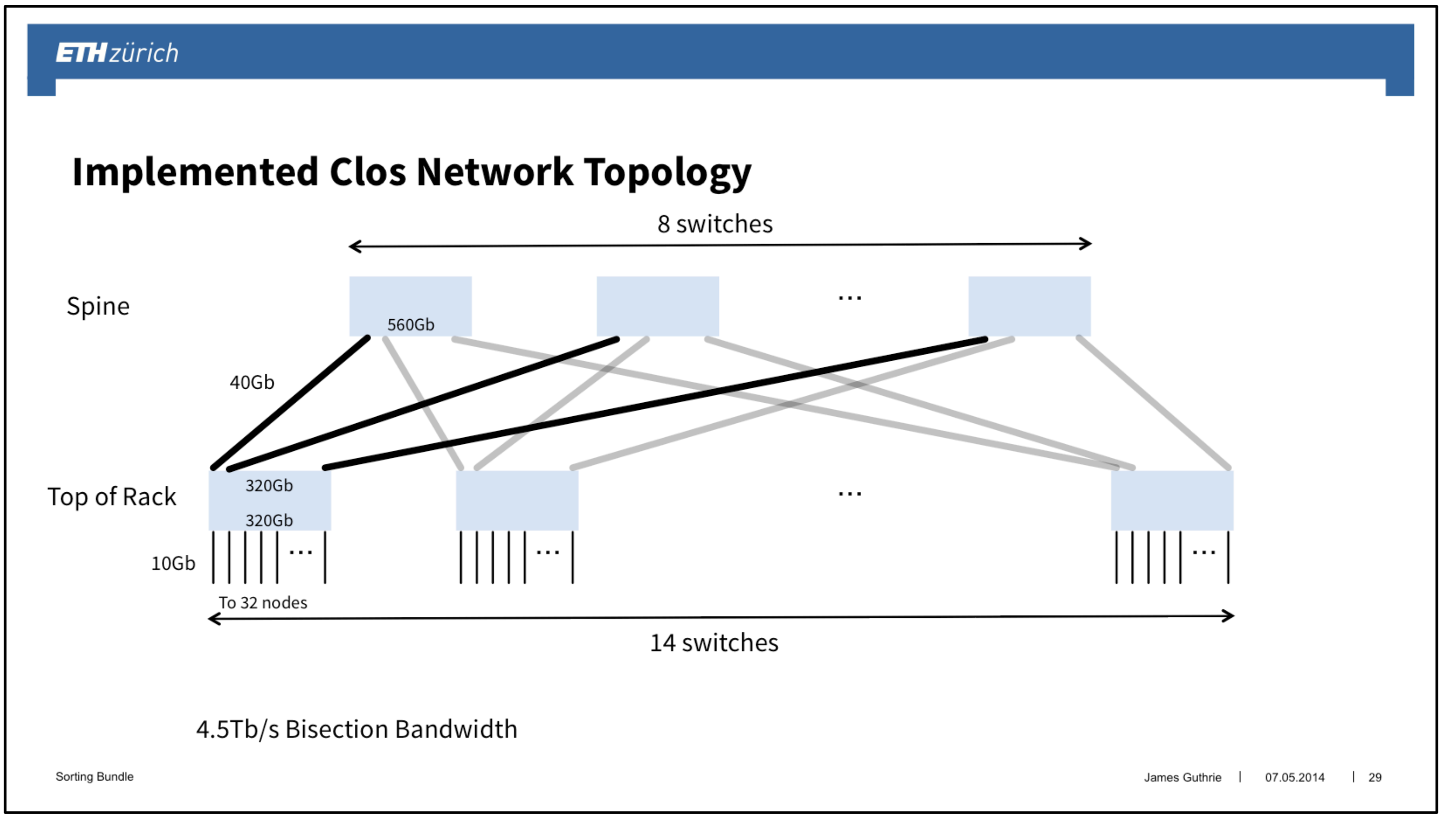
James Guthrie | 07.05.2014 | 28

So now that we have a rough understanding of the concept of bisection bandwidth, what exactly does “full bisection bandwidth” mean?

Full bisection bandwidth is a bisection bandwidth which allows all nodes in one side of the bisection to communicate with all nodes in the other side without any congestion.

(***) One approach to achieve full bisection bandwidth is so-called “fat trees”, in which the bandwidth increases from the leaves to the root.

(***) Another approach, and the approach taken by the designers of FDS are Clos networks.



Implemented Clos Network Topology

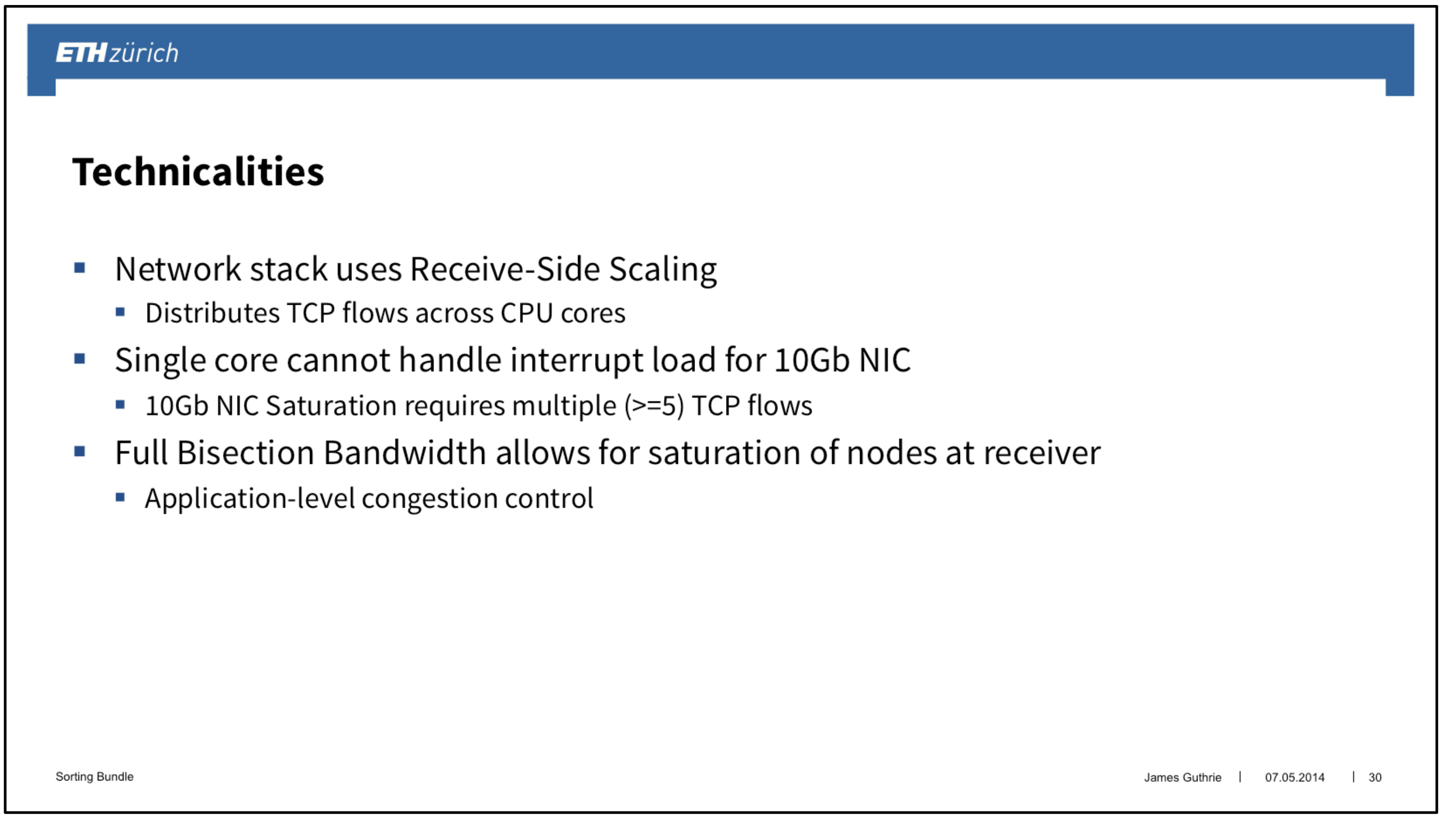
4.5Tb/s Bisection Bandwidth

Sorting Bundle James Guthrie | 07.05.2014 | 29

The network used in the FDS setup consists of 14 “Top of Rack” switches, each with 64 10Gb links.
(***) Half of the links (those pointing downwards) connect to compute/storage nodes.
(***) The other half are connected to the 8 “spine” switches which form the backbone of the network.
Each ToR switch has a 4 bonded 10Gb links (for 40Gb bandwidth) connected to each of the Spine switches. This provides each ToR with 320Gb of bandwidth to the Spine switches.
The whole network provides a bisection bandwidth of 4.5Tbps (10Gbps* 32 links per ToR * 14 ToR switches)
All of this for the small sum of about $250k.

A spanning-tree based protocol won’t work with this type of network topology as it would prune the duplicate links between ToR and Spine switches. What is required is a Multipath Routing Protocol.
The ToR switches use Equal Cost Multipath Routing to decide to which Spine switch a packet should be forwarded, the decision is made based on the hash of the destination IP which distributed flow across all spine switches and ensures that packets in a flow are not reordered.

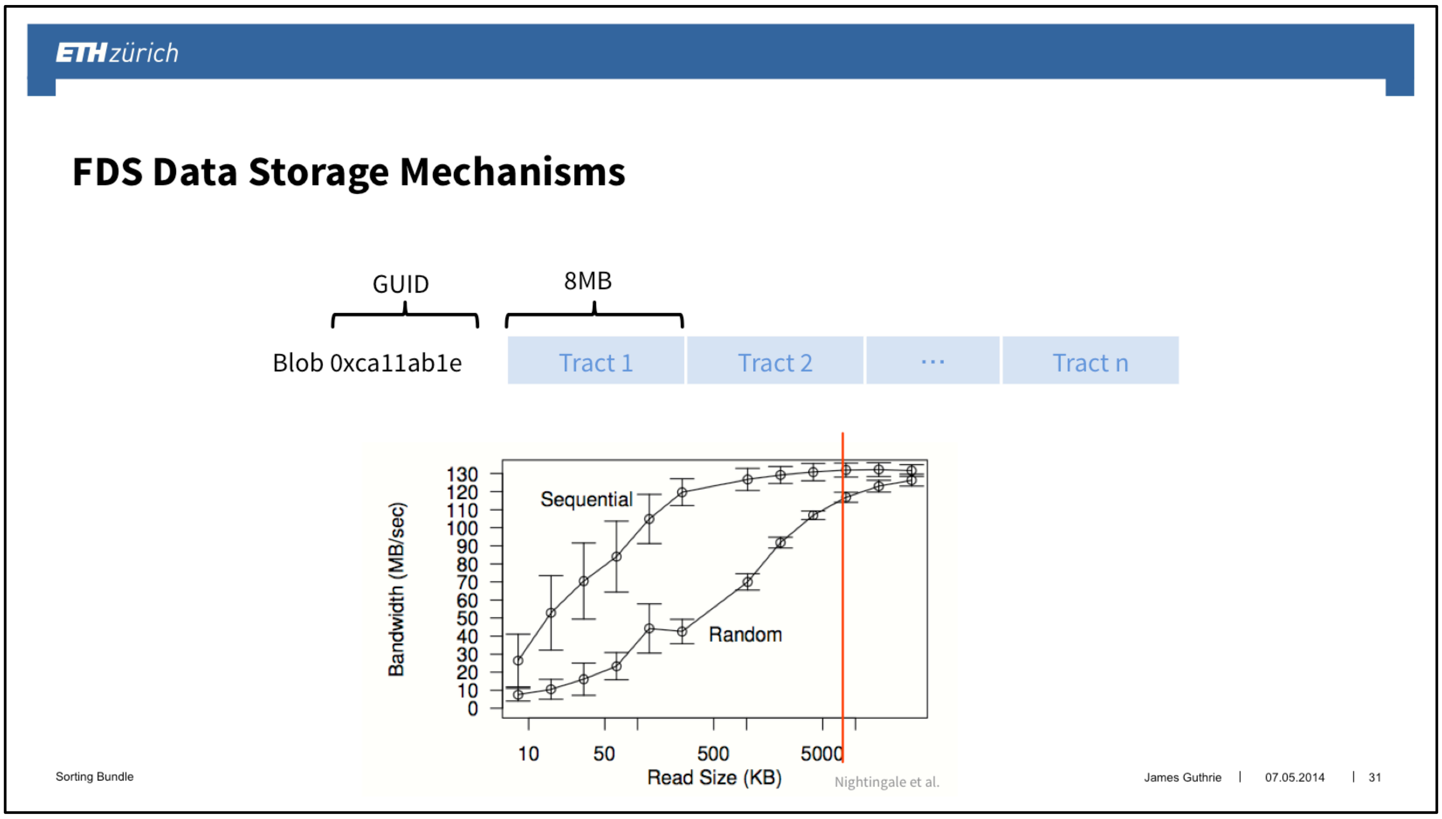
As equal-cost multipath routing is a static routing protocol, it only stochastically guarantees Full Bisection Bandwidth. Long-living high-volume flows can result in collisions which reduce overall performance of the network.



ETH zürich

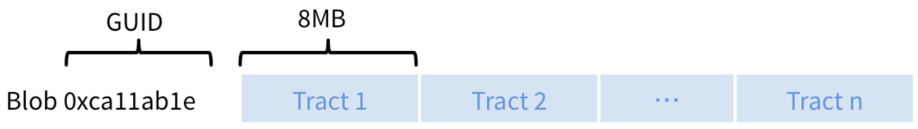
Technicalities

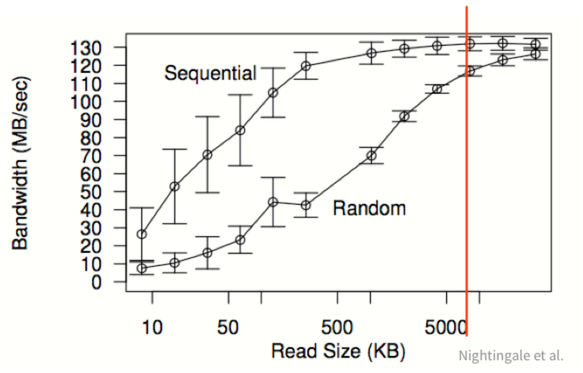
- Network stack uses Receive-Side Scaling
  - Distributes TCP flows across CPU cores
- Single core cannot handle interrupt load for 10Gb NIC
  - 10Gb NIC Saturation requires multiple (>=5) TCP flows
- Full Bisection Bandwidth allows for saturation of nodes at receiver
  - Application-level congestion control



ETH zürich

FDS Data Storage Mechanisms





Nightingale et al.

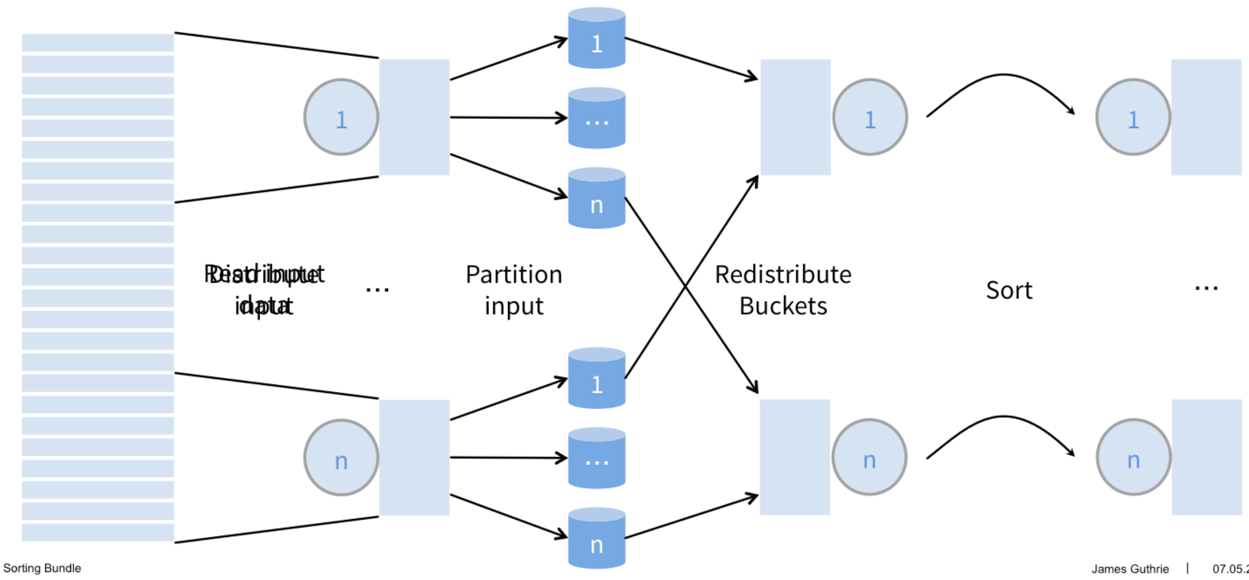
Sorting Bundle | James Guthrie | 07.05.2014 | 31

Flat Datacenter Sotrage stores data stored as a byte sequence in so-called *blobs* which are further subdivided into units called *tracts* which are the minimum read/write size. The size of tracts can be chosen arbitrarily but was chosen to be 8MB to increase random read performance with HDDs.

(***) This graph supports why 8MB tract size is a good choice. What we see is read size in KB on the x-axis (logarithmic) and read bandwidth in MB/s on the y axis. Shown are the read throughput for sequential and random access patterns. As the read size increases, the two lines merge.
(***) The red line is and around the 8MB mark – we see that the bandwidth of Random and Sequential reads is approximately equivalent.

ETH zürich

Sorting in Flat Datacenter Storage



Sorting Bundle | James Guthrie | 07.05.2014 | 33

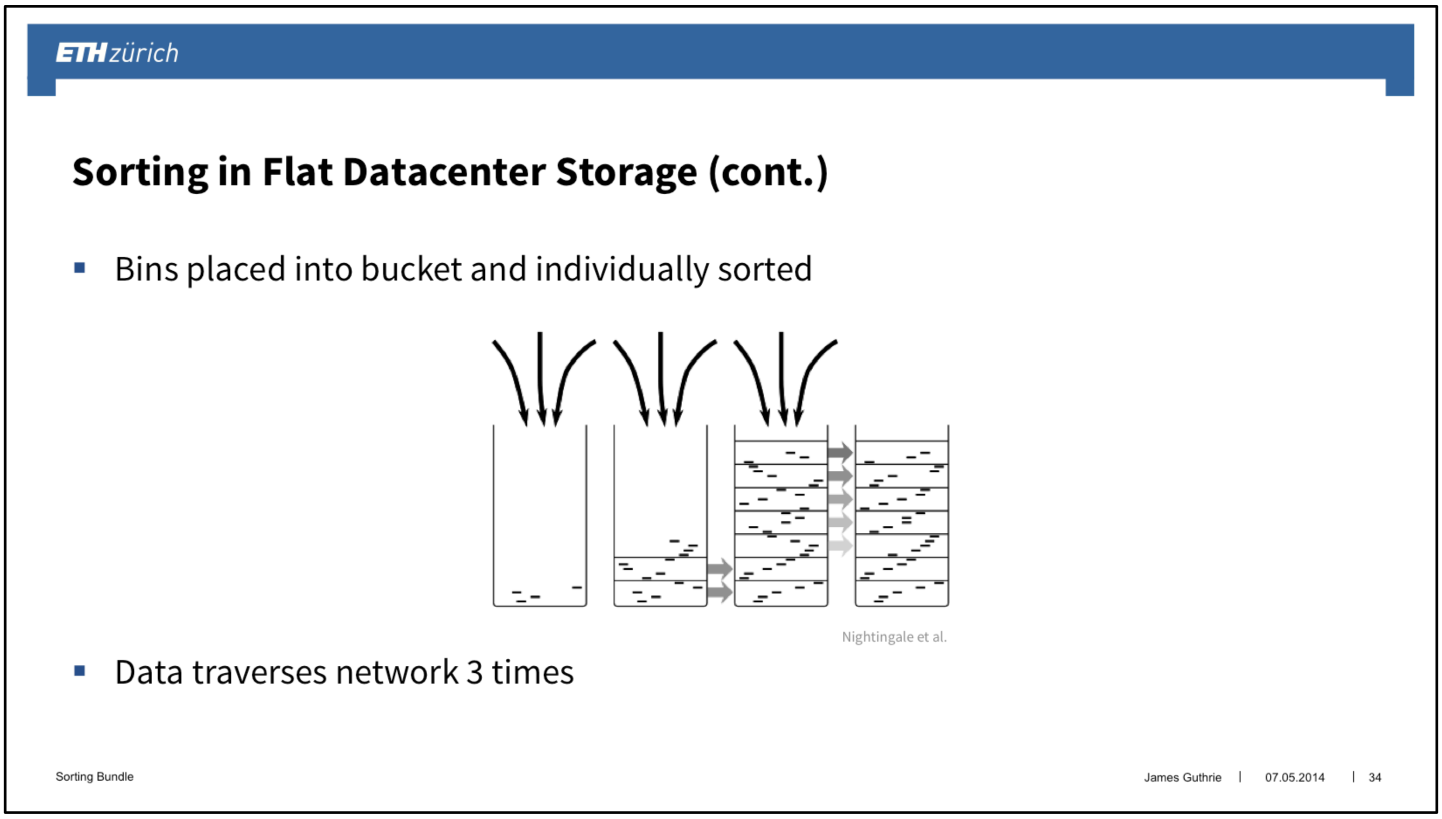
So we’ve seen quite a lot regarding network capacity and how data is stored in this file system, but how is it sorted?

(***) Lets refer back to our trusty distributed sort diagram.

As I mentioned before, the computation and storage are done by separate compute and storage nodes.
(***) As the sort is spun up, the compute nodes begin to read data from the storage nodes.

During the partitioning stage, instead of completely partitioning the output and then redistributing, the bucket is broken up into bins. As each bin is filled, it is passed to the receiving host and placed into the bucket.

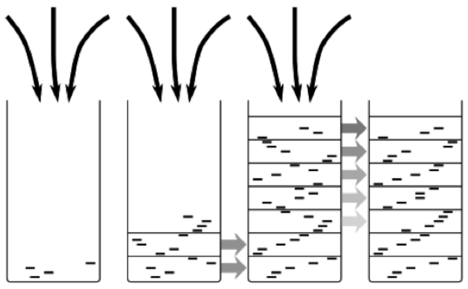
136 storage nodes, 120 compute nodes



ETH zürich

Sorting in Flat Datacenter Storage (cont.)

- Bins placed into bucket and individually sorted



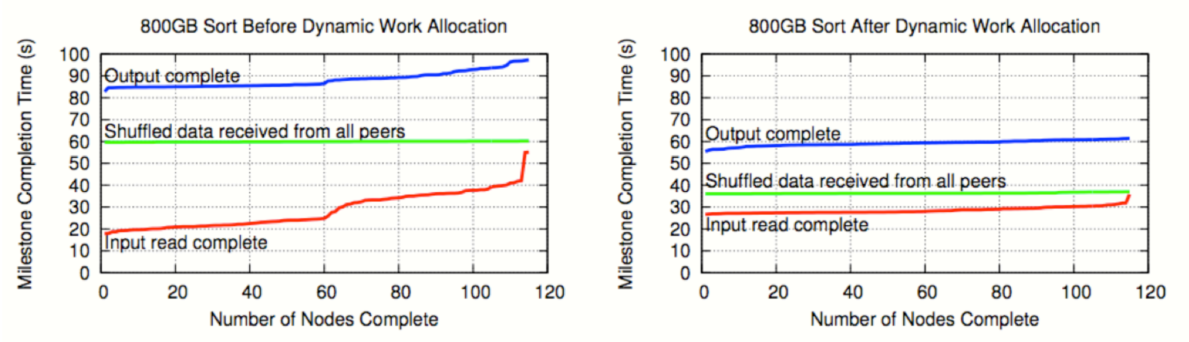
Nightingale et al.

- Data traverses network 3 times

ETH zürich

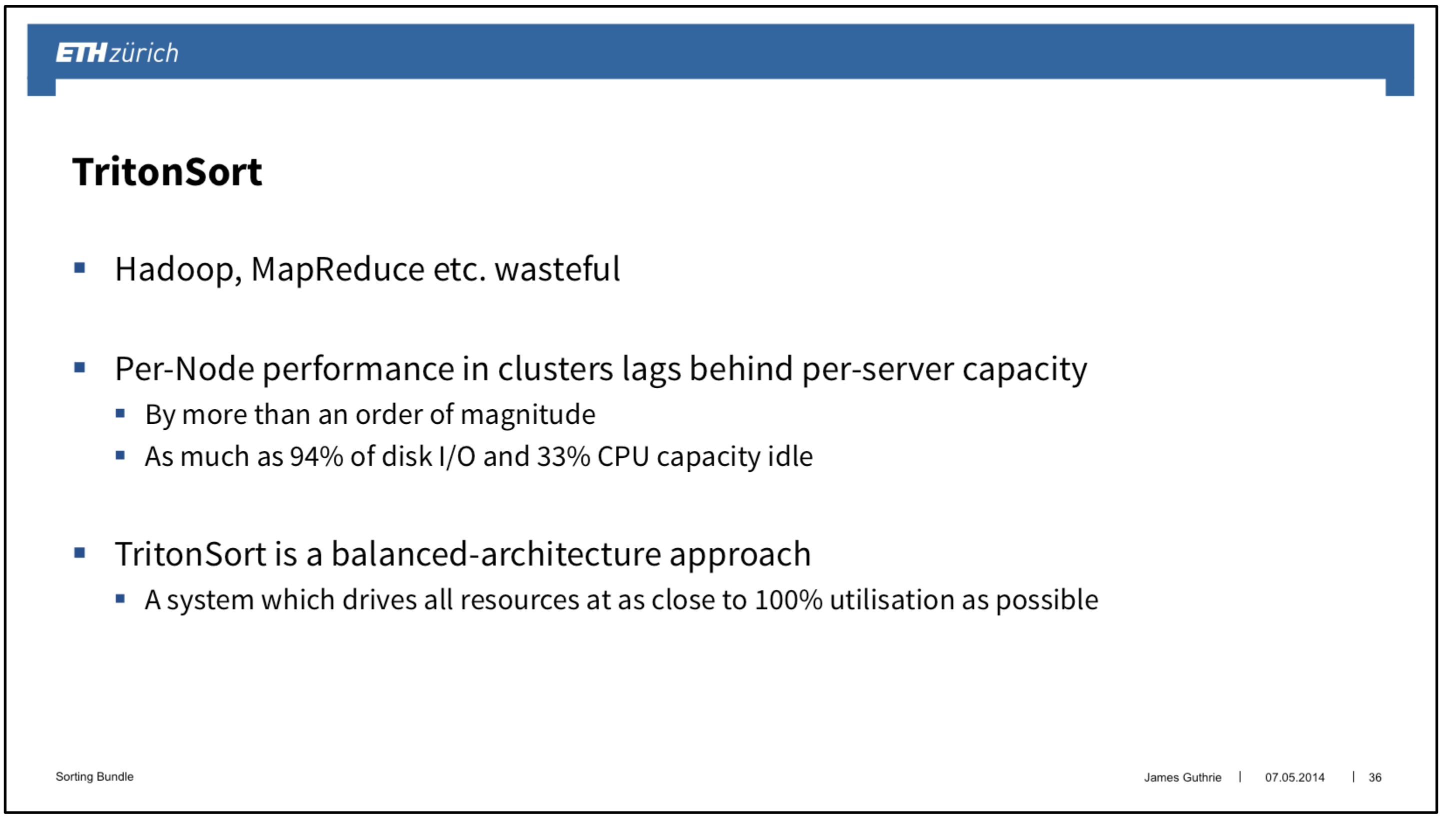
Dynamic Work Allocation

- FDS also encounters straggler problem
  - Flexibility allows for dynamic work allocation



Nightingale et al.

Flat Datacenter Storage also encounters the straggler problem that MapReduce did. The architects of Flat Datacenter Storage claim that the fact that data-locality is not a concern means that they can more dynamically schedule where and when compute jobs are run. Instead of spooling up a bunch of backup processes towards the end of the computation, FDS assigns compute jobs on demand throughout the computation.


ETH zürich

TritonSort

- Hadoop, MapReduce etc. wasteful
- Per-Node performance in clusters lags behind per-server capacity
  - By more than an order of magnitude
  - As much as 94% of disk I/O and 33% CPU capacity idle
- TritonSort is a balanced-architecture approach
  - A system which drives all resources at as close to 100% utilisation as possible

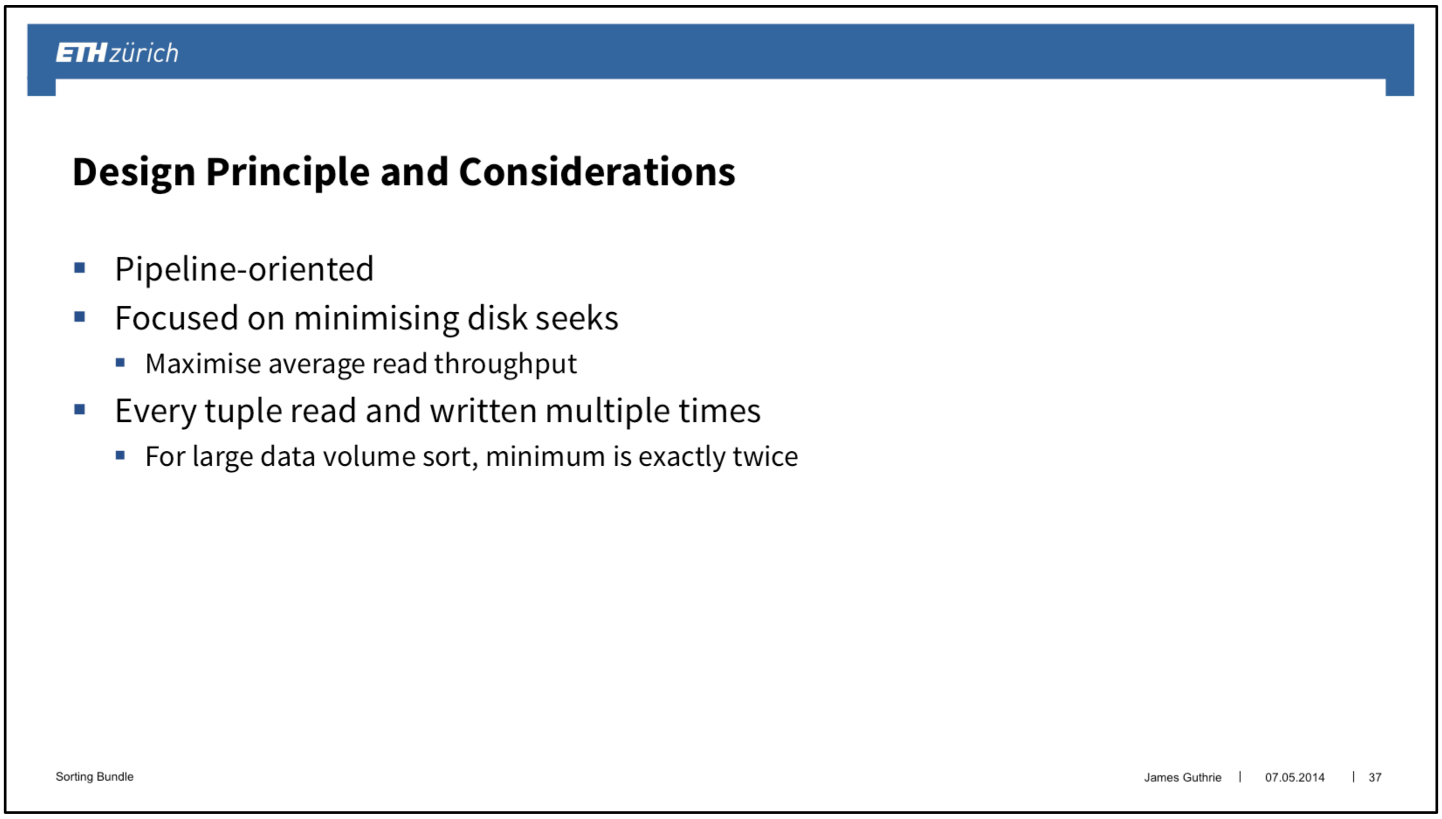
Sorting Bundle James Guthrie | 07.05.2014 | 36

TritonSort was designed as a reaction to the inefficiency of modern Data-Intensive Scalable Computing systems (DISC systems) such as MapReduce, Hadoop and Dryad. The designers of TritonSort assert that while these solutions scale linearly in the number of nodes, as much as 94% of disk I/O and 33% of CPU capacity remains idle for some computations on large clusters.

TritonSort highlights the level of efficiency which is attainable when computation, storage, memory and network are balanced.

The balanced-architecture approach of TritonSort

Hardware parameterised to load type
Software parameterised to hardware
… and to expected load



ETH zürich

Design Principle and Considerations

- Pipeline-oriented
- Focused on minimising disk seeks
  - Maximise average read throughput
- Every tuple read and written multiple times
  - For large data volume sort, minimum is exactly twice

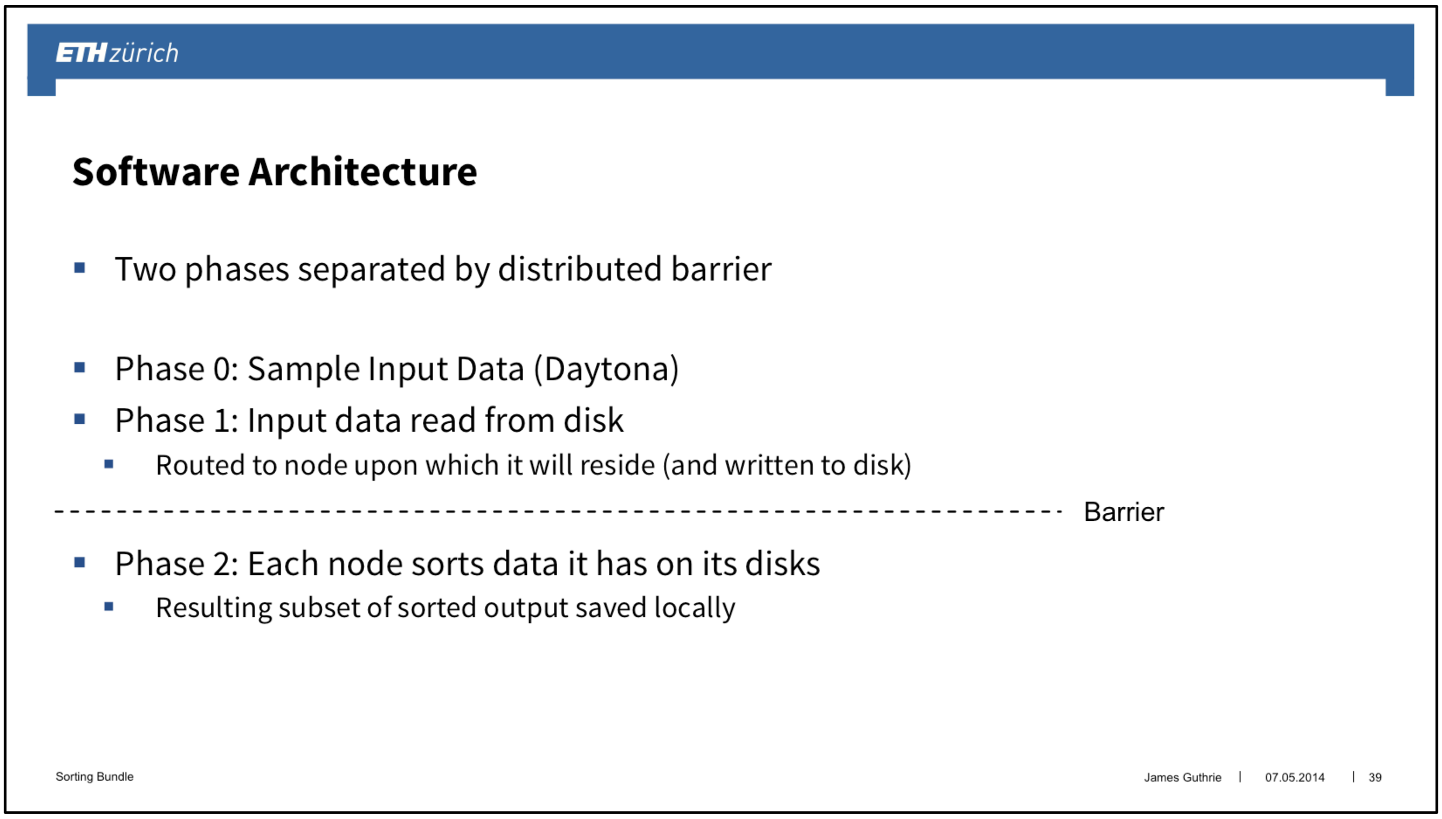
Sorting Bundle

James Guthrie | 07.05.2014 | 37

The overall design of the TritonSort is a distributed, staged, pipeline-oriented dataflow processing system

The designers identified HDD I/O bandwidth as the key bottleneck to the sorting application. In order to maintain maximum average read throughput they focus on minimising disk seeks (and reads/writes) throughout their application.

When it comes to reading and writing, the designers identify that an internal sort requires at least one read and write per tuple and that an external sort required at least two reads and writes per tuple. The designers aim to reach the minimum of two reads and writes per tuple.


ETHzürich

Software Architecture

- Two phases separated by distributed barrier

- Phase 0: Sample Input Data (Daytona)
- Phase 1: Input data read from disk
  - Routed to node upon which it will reside (and written to disk)

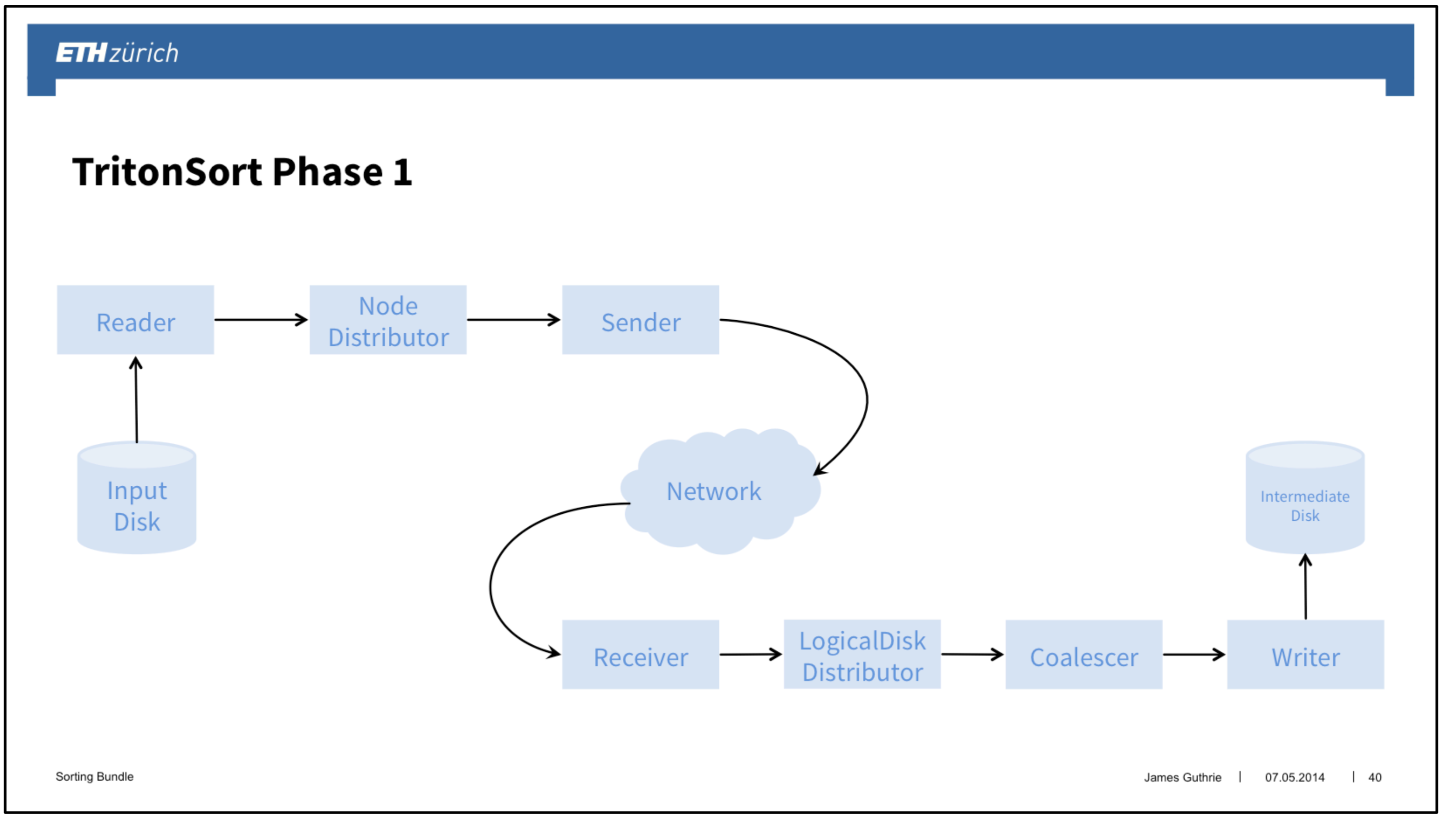
- Barrier

- Phase 2: Each node sorts data it has on its disks
  - Resulting subset of sorted output saved locally

Sorting Bundle James Guthrie | 07.05.2014 | 39

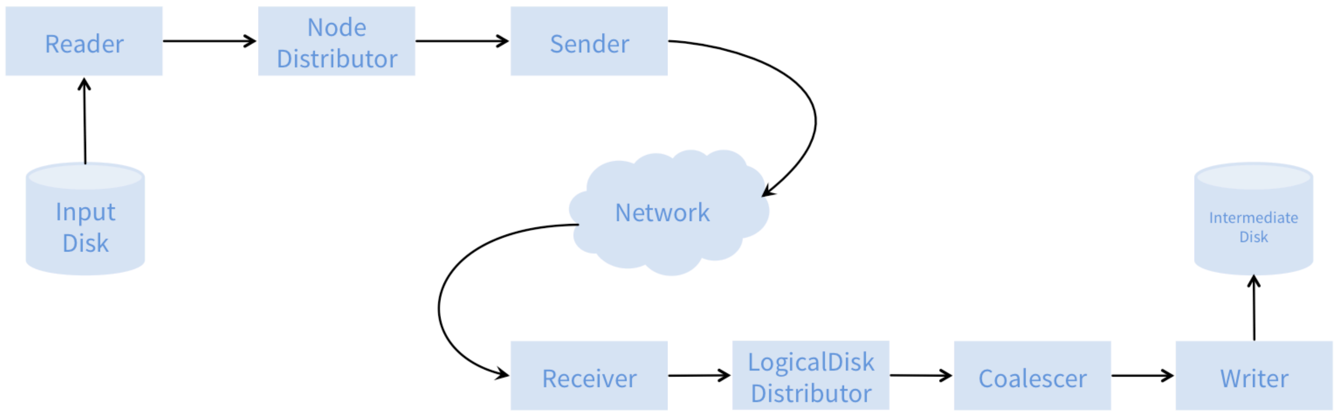
The overall software architecture of the TritonSort is two pipelined phases which are separated by a distributed barrier i.e. phase 1 must complete on all nodes before phase 2 can start.

The Daytona sort version of the TritonSort requires a Phase 0, in which the input data is sampled to determine the keyspace distribution.



ETH zürich

TritonSort Phase 1



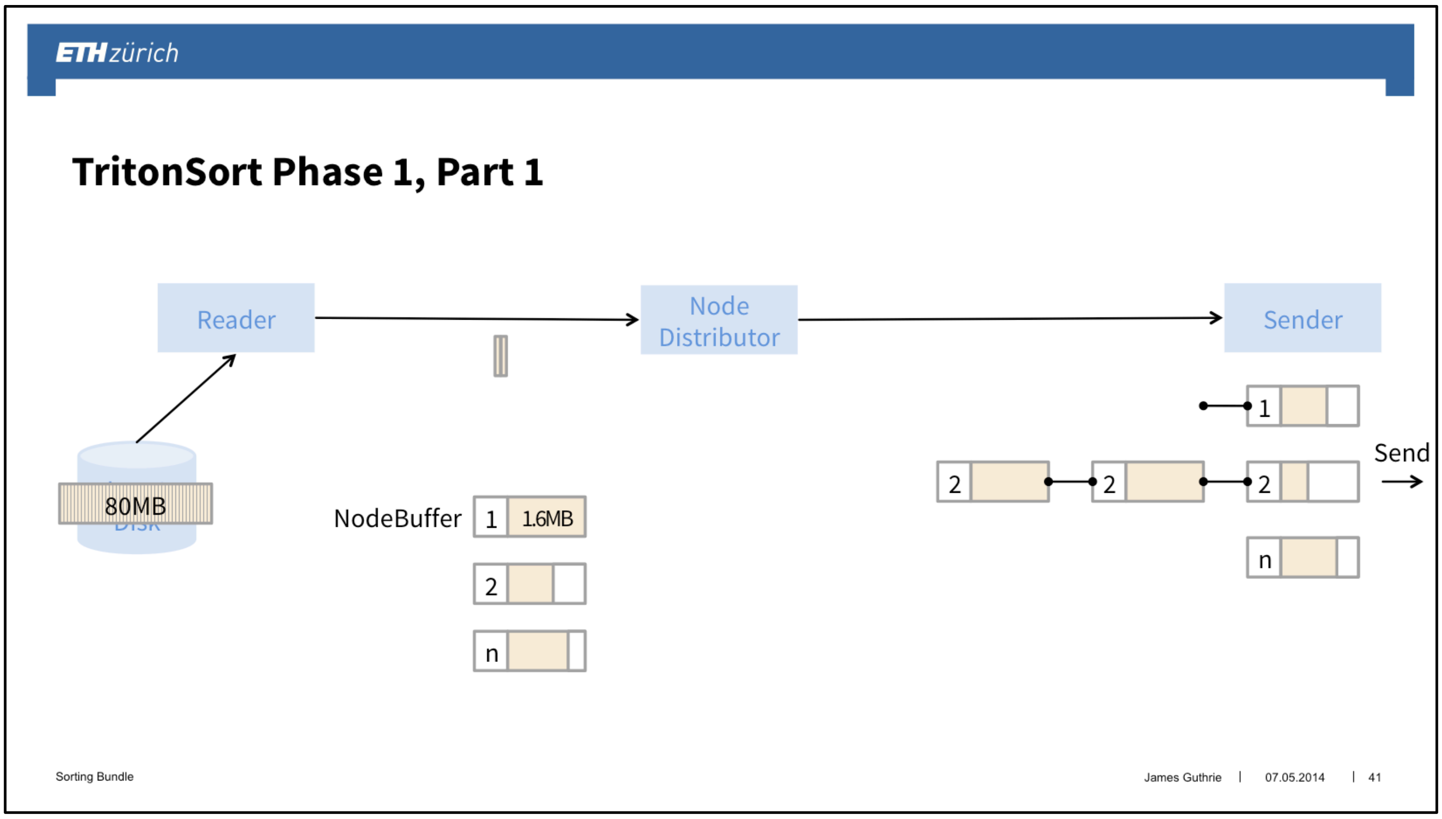
Sorting Bundle

James Guthrie | 07.05.2014 | 40

So what does phase one look like?

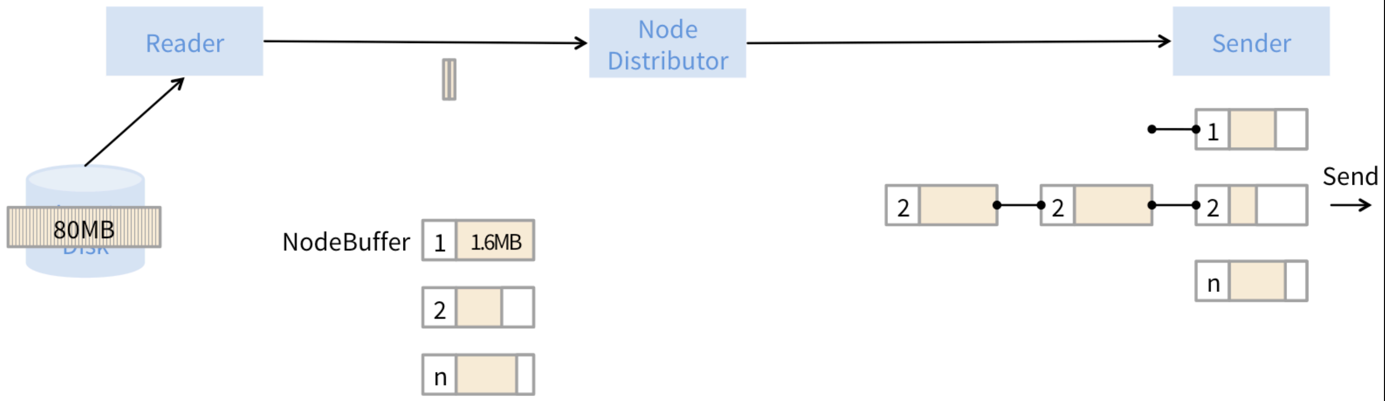
Here we see the individual stages of the pipeline used in phase one. There is really too much information to be able to go into detail about every individual step. What I will do is give you a feeling for the types of adjustments that were made to balance the system.

To start with, the 16 disks on each node are split into two halves for both phases one and two, half of the disks are read from while the other half of the disks are written to.



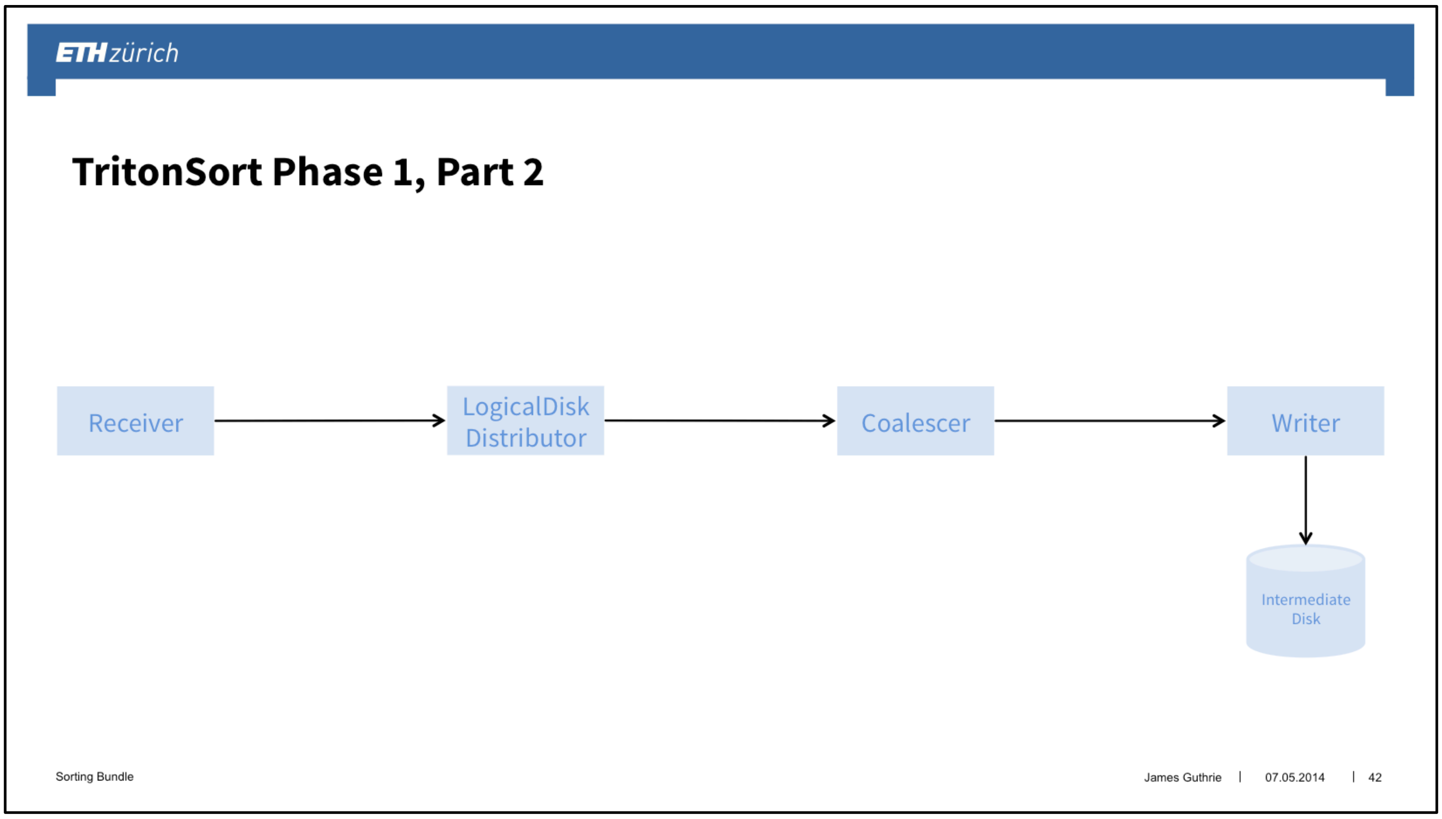
ETHzürich

TritonSort Phase 1, Part 1



Now we will take a detailed look at the parameters tuned in the first half of phase 1. This phase is made up of a Reader, Node Distributor and Sender.

(***) The reader reads 80MB chunks of data off of the disk and passes them to the Node Distributor. The chunk size is chosen to maximise read throughput.
(***) The node distributor maps each tuple to a buffer destined to a node, a “NodeBuffer”. The nodebuffer size is the 80MB chunk size divided by the number of nodes, ~1.6MB.
(***) The NodeBuffer is filled with tuples
(***) When the NodeBuffer for a specific node is full, it is passed on to the Sender and appended to a chain of outgoing NodeBuffers
(***) The Sender continually sends fixed-size chunks of the NodeBuffer across the network to the destination node. The sender is single-threaded and rate-limited by the size of the receiver’s window (16KB).

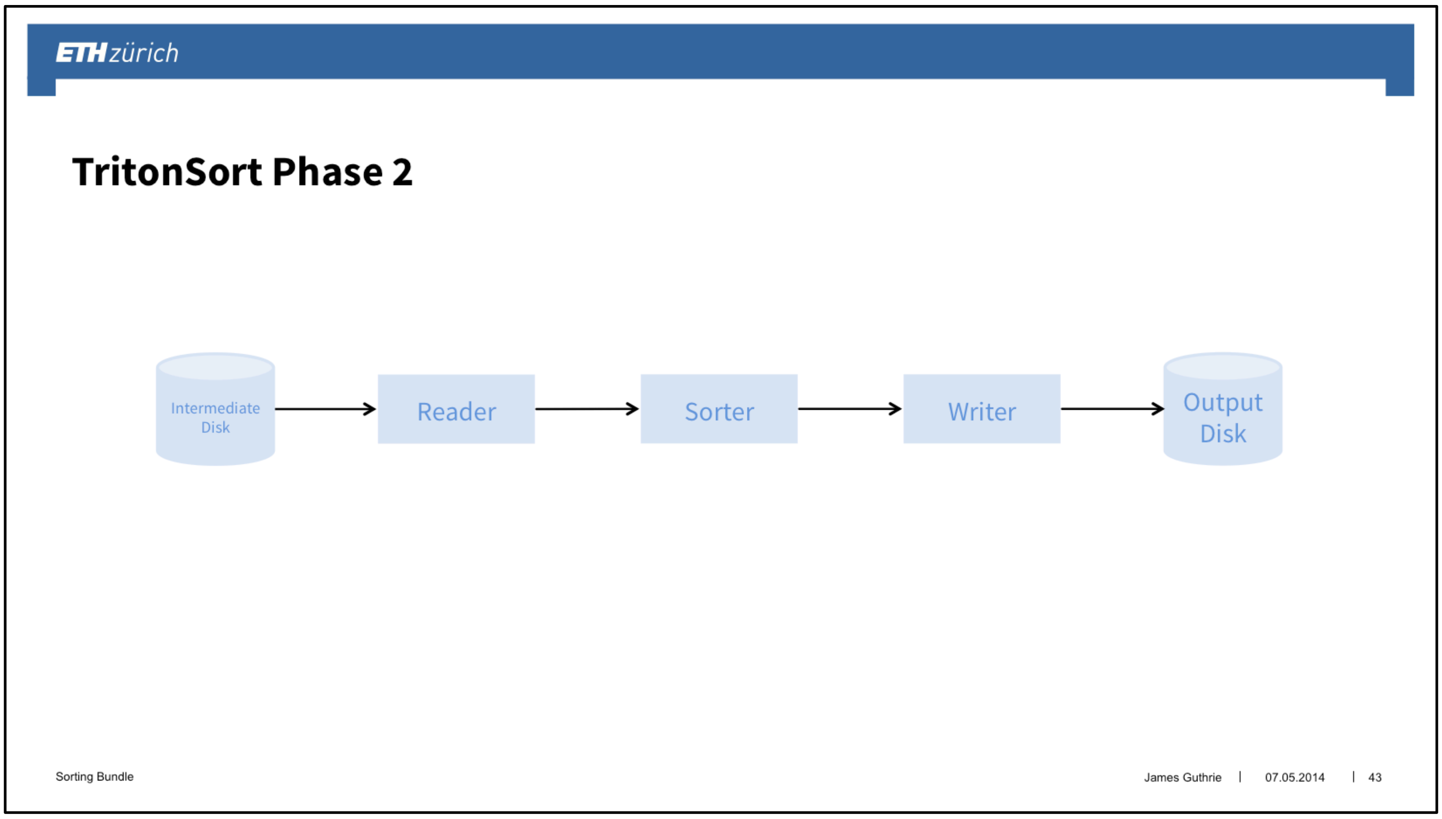

ETH zürich

TritonSort Phase 1, Part 2

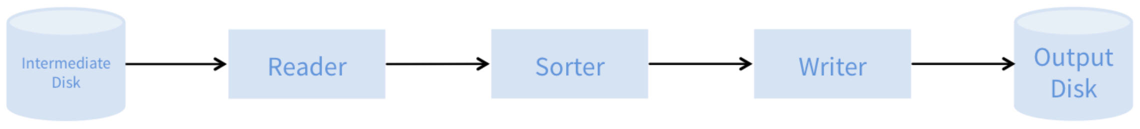
Sorting Bundle

James Guthrie | 07.05.2014 | 42

In part two of phase one the receive pipeline receives packets from the network and writes them to disk. Here again a whole bunch of tweaks are made to the sizes and numbers of buffers between the pipeline stages.


ETH zürich

TritonSort Phase 2

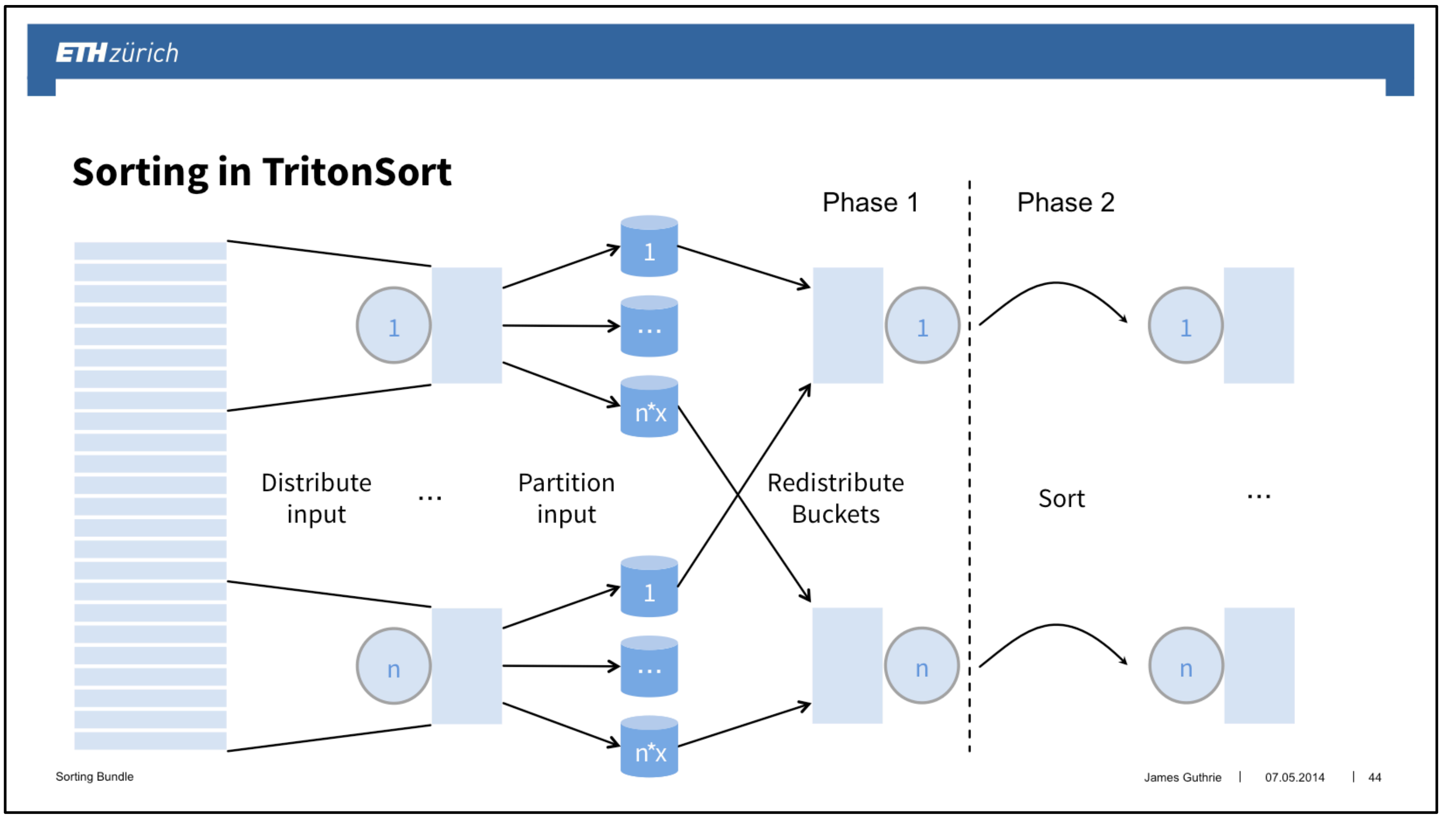


Sorting Bundle

James Guthrie | 07.05.2014 | 43

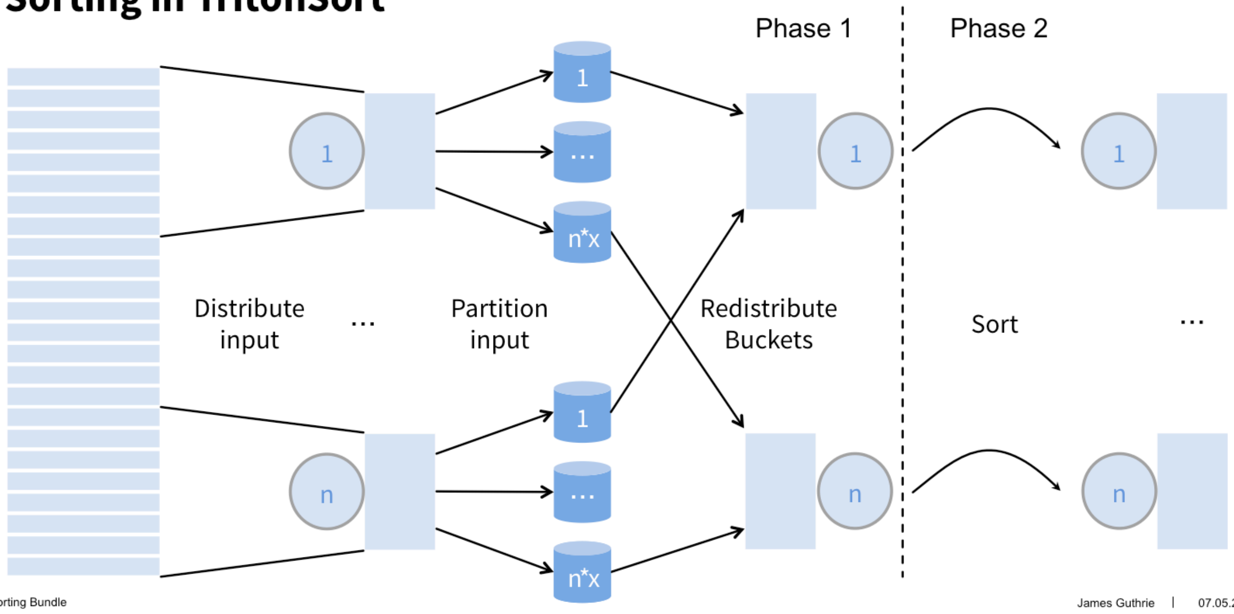
After completion of phase one, all input values are on the host on which they will reside when the data is sorted.

Phase two simply sorts the data. Again a pipelined approach is taken.



ETH zürich

Sorting in TritonSort



Sorting Bundle

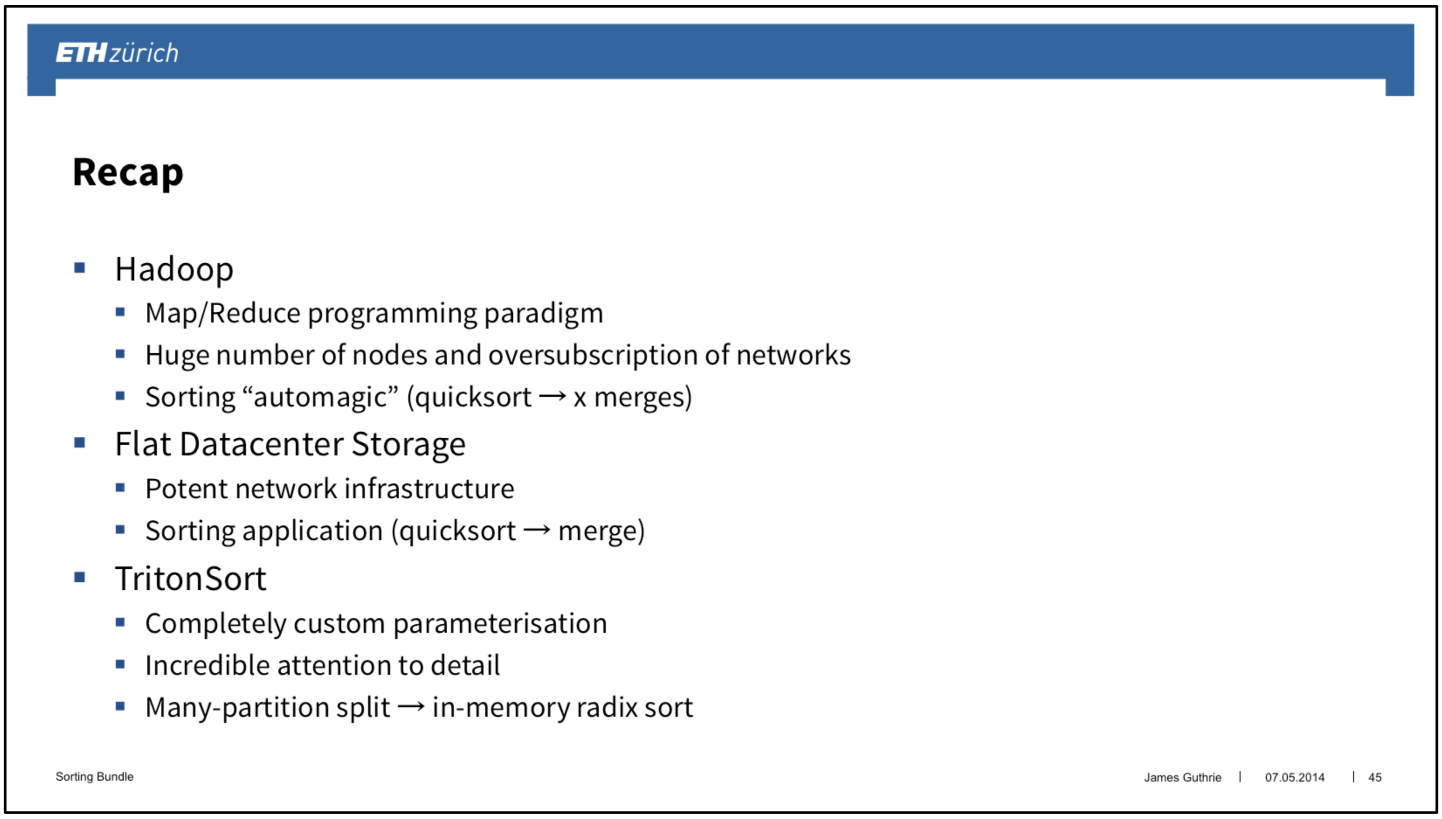
James Guthrie | 07.05.2014 | 44

We've seen that the TritonSort uses a staged pipeline to perform the sort, but how does that relate to the distributed approach I presented in the beginning?

(***) Once again it's surprising how exactly the TritonSort fits into this diagram.
(***) All we need to do is draw a vertical line separating phase 1 and phase two.

(***) In fact, the one inaccuracy still present in this diagram is that the data isn't partitioned into n partitions, but into a multiple of n.

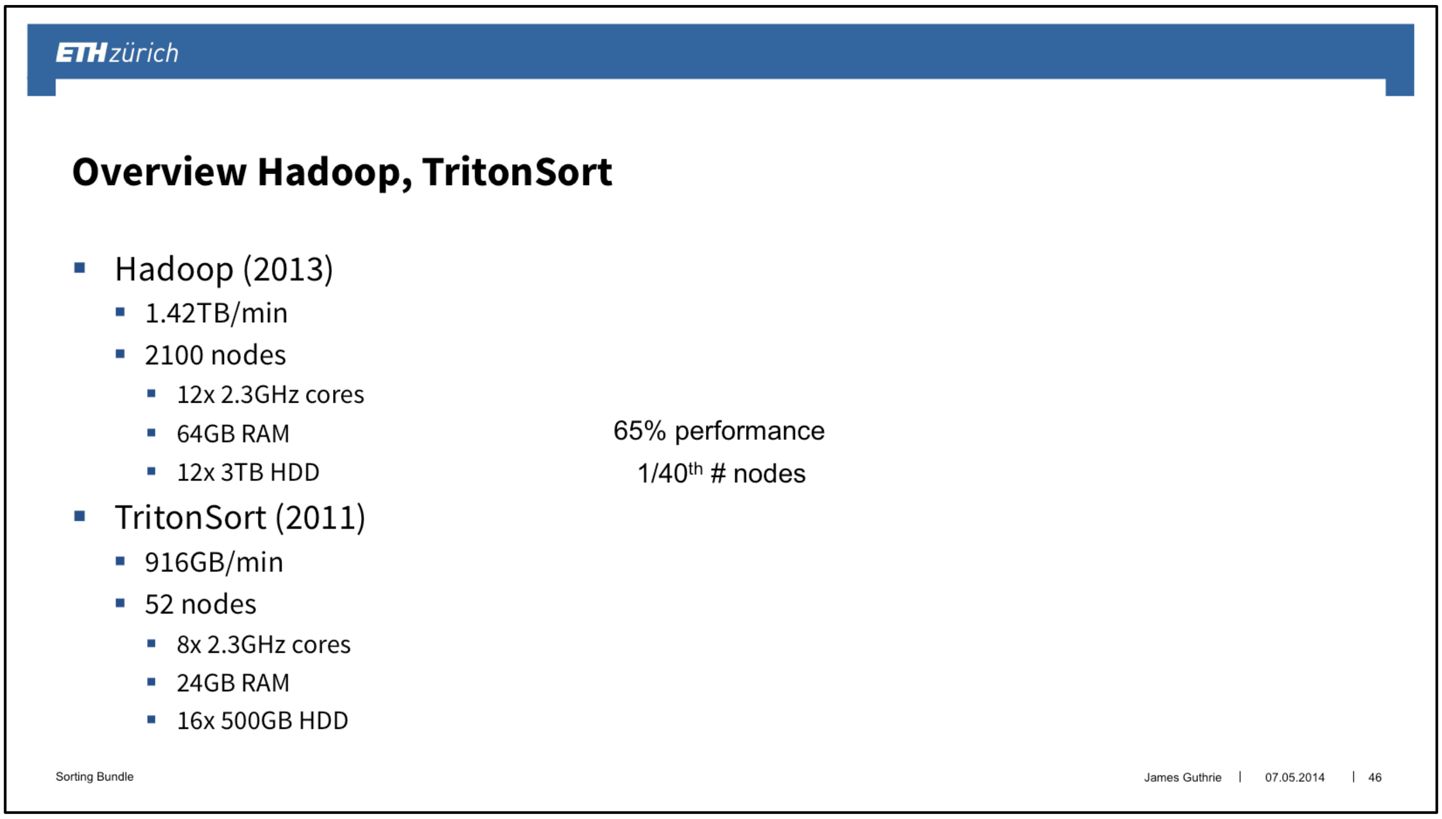
The deciding factor in the number of partitions (the size of x in the diagram) is the size of the partition relative to the machine's RAM.
In order to maximise throughput in phase 2, there is a sort pipeline running for each of the 8 input disks. Each pipeline needs enough space to hold 3 partitions, one in the reading stage, one in the sorting stage and one in the writing stage. The authors chose 850MB partition sizes in order to have 24 in the 24GB of ram with some memory left for the sort.



ETH zürich

Recap

- Hadoop
  - Map/Reduce programming paradigm
  - Huge number of nodes and oversubscription of networks
  - Sorting “automagic” (quicksort → x merges)
- Flat Datacenter Storage
  - Potent network infrastructure
  - Sorting application (quicksort → merge)
- TritonSort
  - Completely custom parameterisation
  - Incredible attention to detail
  - Many-partition split → in-memory radix sort


ETH zürich

Overview Hadoop, TritonSort

- Hadoop (2013)
  - 1.42TB/min
  - 2100 nodes
    - 12x 2.3GHz cores
    - 64GB RAM
    - 12x 3TB HDD
- TritonSort (2011)
  - 916GB/min
  - 52 nodes
    - 8x 2.3GHz cores
    - 24GB RAM
    - 16x 500GB HDD

65% performance

1/40th # nodes

Sorting Bundle James Guthrie | 07.05.2014 | 46

Lets take a look at Hadoop vs. TritonSort. Fortunately the GraySort and JouleSort with 10^12 Records are very comparable. The metric measured is different, but the volume of data to be processed is essentially the same.

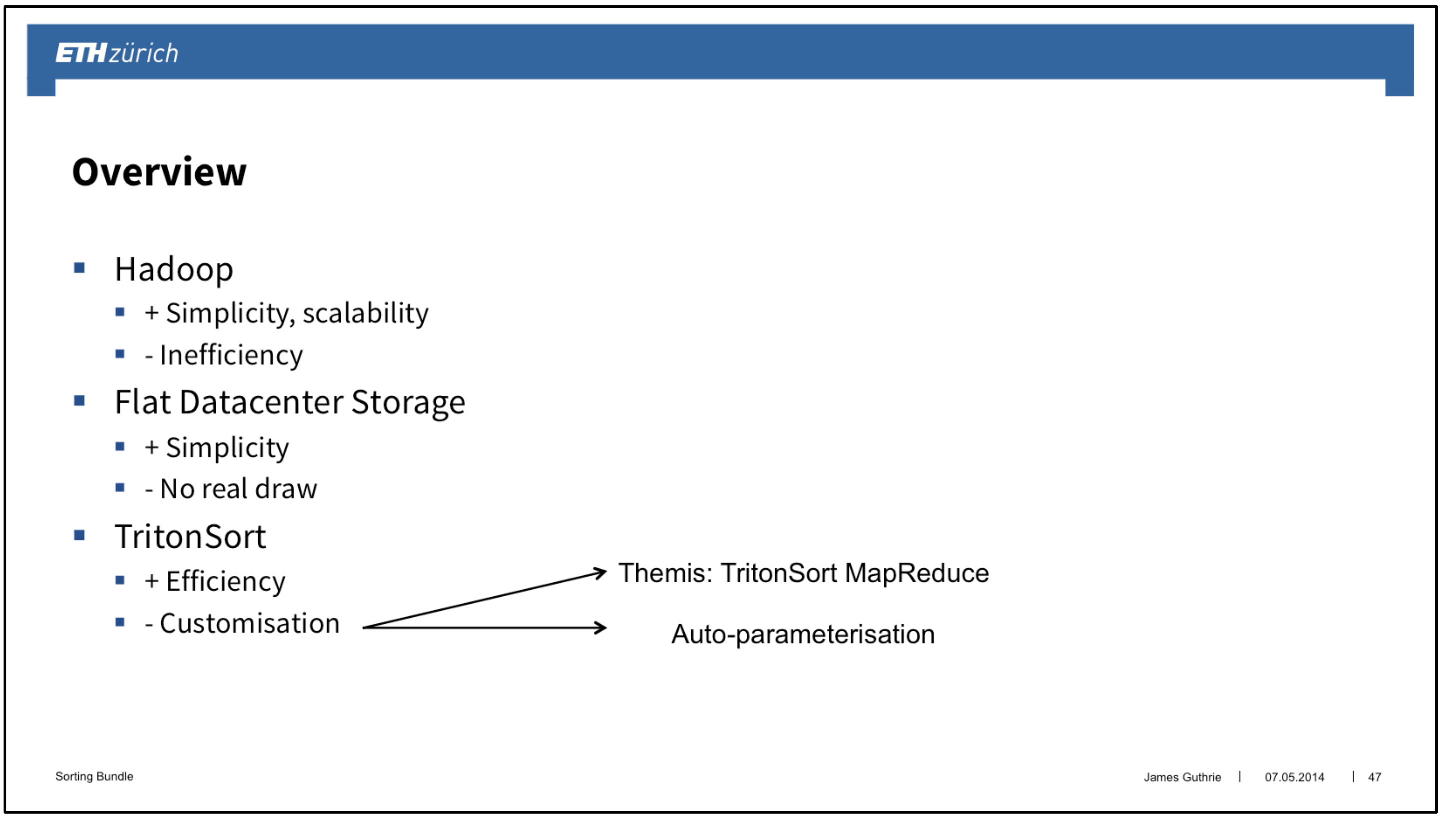
(***) So Hadoop has a performance of 1.42TB/min

(***) Versus TritonSort's 916GB/min

(***) But the Hadoop system consisted of 2100 nodes

(***) Versus TritonSorts 52 nodes.

(***) Looking at the individual nodes specs we see that they're very similar. The Hadoop cluster has more cores, more RAM and more disk space. The TritonSort Cluster has slightly more disks per node



ETH zürich

Overview

- Hadoop
  - + Simplicity, scalability
  - - Inefficiency
- Flat Datacenter Storage
  - + Simplicity
  - - No real draw
- TritonSort
  - + Efficiency
  - - Customisation $\rightarrow$ Themis: TritonSort MapReduce; $\rightarrow$ Auto-parameterisation

Sorting Bundle   James Guthrie | 07.05.2014 | 47

So to finish up today I'd like to have a brief look at the pros and cons of each of the approaches to sorting we saw today.

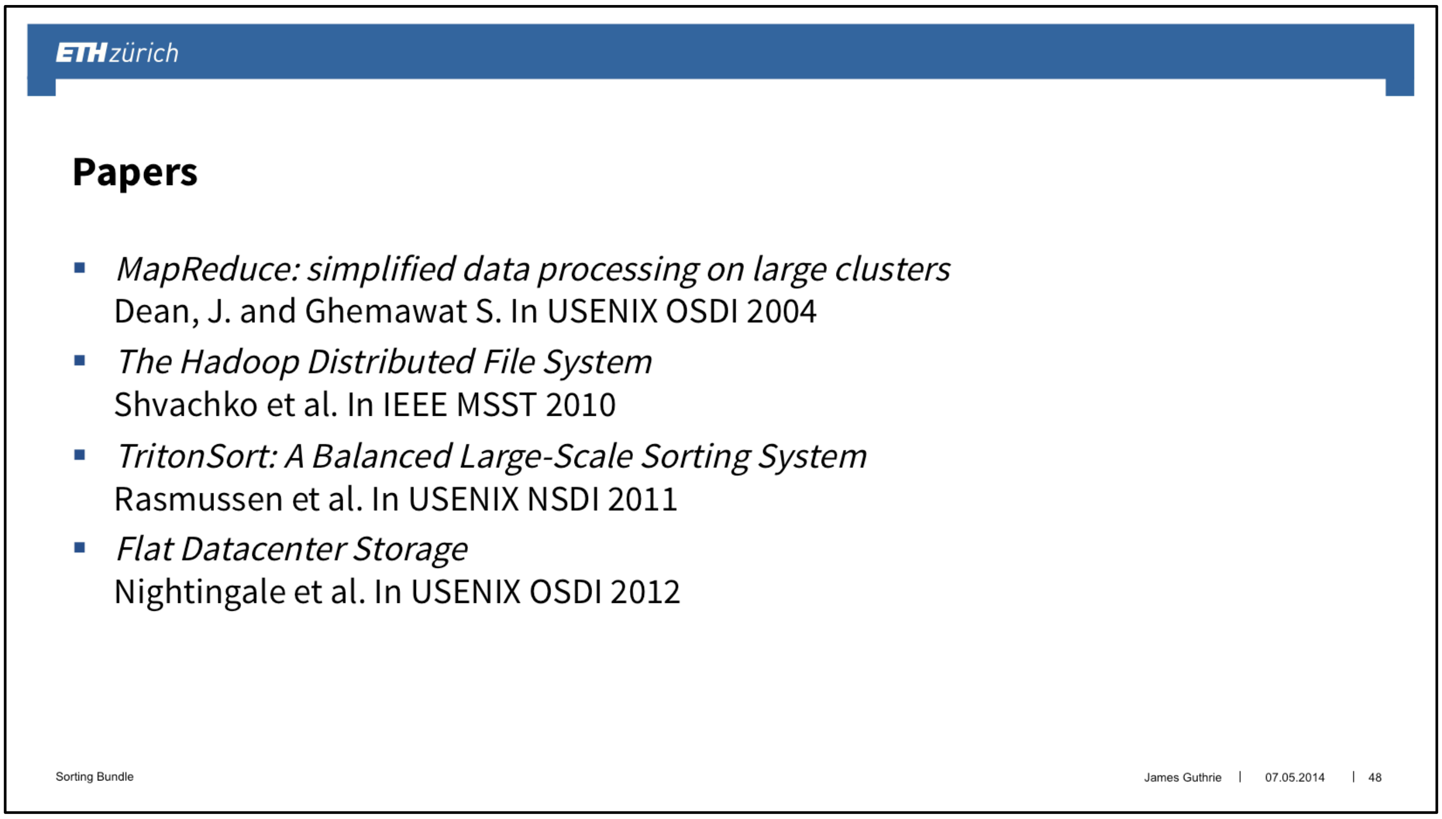
At first we saw Hadoop which offers the MapReduce programming paradigm which is both simple to implement and linearly scalable in the number of nodes. What we also saw is that although there is a lot of raw power available, it is inefficiently utilised.

Next we saw Flat Datacenter Storage. The authors of FDS claim that the 'directly-attached storage' approach makes application programming simpler without the need for taking data locality into account. Although FDS had very good performance, when compared to that of the tritonsort, FDS seems to be too much of a toy that somebody had fun building but that maybe isn't all that necessary.

Finally we saw TritonSort, which through incredible attention to detail manages to achieve astounding per-node efficiency (especially when compared with Hadoop). The downside is the amount of customisation that went into the platform in order to achieve these results.

(***) The authors of TritonSort went on to produce Themis, a MapReduce implementation based on the TritonSort system. There was not much mention of Themis, and it appears as though the research efforts into this system have stopped at UCSD.

(***) At the time the TritonSort paper was published one of the authors also mentioned



ETH zürich

Papers

- *MapReduce: simplified data processing on large clusters*
  Dean, J. and Ghemawat S. In USENIX OSDI 2004
- *The Hadoop Distributed File System*
  Shvachko et al. In IEEE MSST 2010
- *TritonSort: A Balanced Large-Scale Sorting System*
  Rasmussen et al. In USENIX NSDI 2011
- *Flat Datacenter Storage*
  Nightingale et al. In USENIX OSDI 2012

Sorting Bundle

James Guthrie | 07.05.2014 | 48