Big Data Tech for

Analytics,
Geo,
Data science,
Machine Learning,
Stuff….

N LAB: *Data at Scale*

Dr. Evgeniya Lukinova

Big Data Technology covers both storage and processing

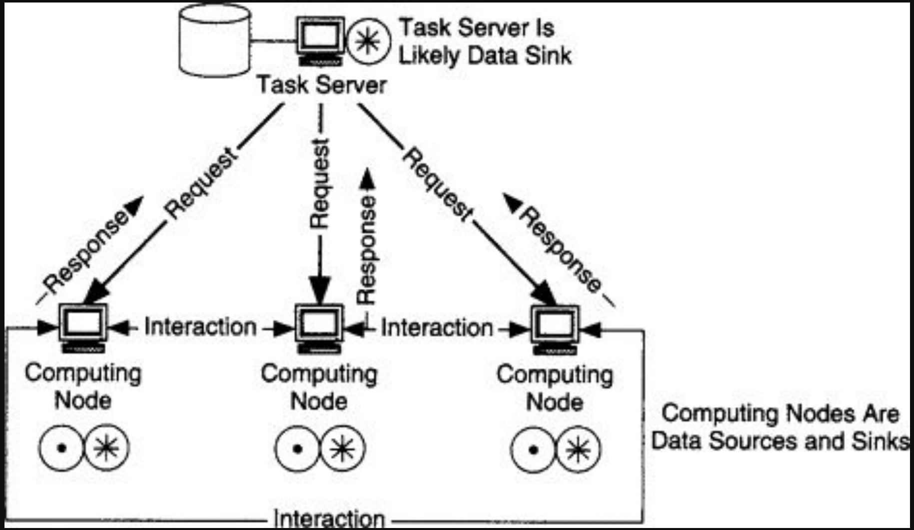
Lots of workers? Need a manager(s)
Need a communication protocol.

Need to know how to subdivide tasks
Need to send code to workers
Need to transfer data between workers
→ unless it is already correctly
subdivided and located

Need to collate results of the workers

What if a worker is lazy? Quits?
Makes an error?

Faster to "just do it yourself"?



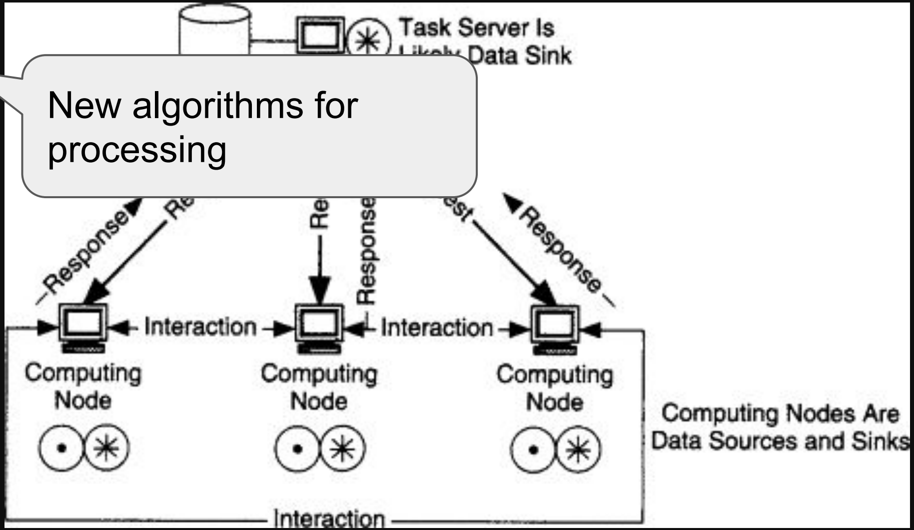
Big Data Technology covers both storage and processing

Lots of workers? Need a manager(s)
Need a communication protocol.

New programing paradigms to write algorithms

Need to know how to subdivide tasks
Need to send code to workers
Need to transfer data between workers
→ unless it is already correctly subdivided and located

New algorithms for processing



Need to collate results of the workers

Extra logic and complexity within the algorithms

What if a worker is lazy? Quits? Makes an error?

Faster to "just do it yourself"?

It might run slower, or not that much faster (cost outweigh the benefit)

Big Data Technology covers both storage and processing

Lots of workers? Need a manager(s)
Need a communication protocol.

Either because we use a database that "guesses" or we write code to pre-distribute (replicated?) data or we have an algorithm that does multiple passes of data

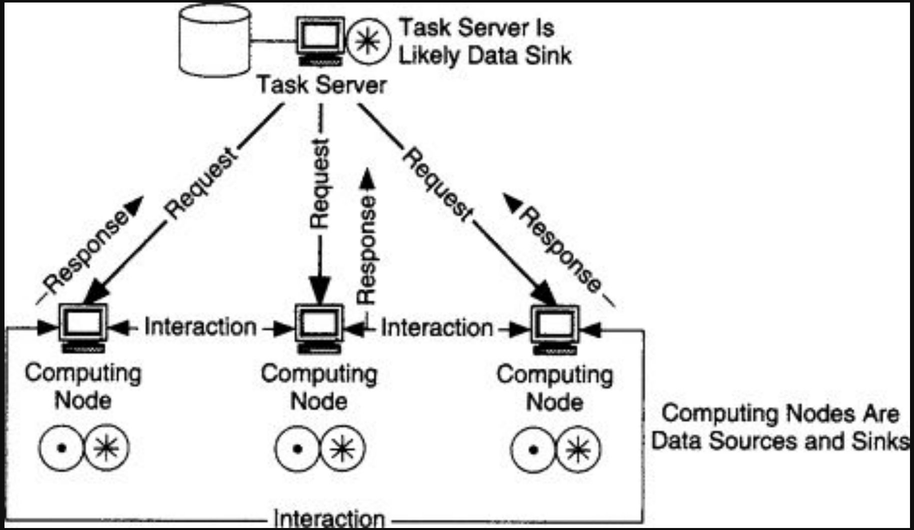
Need to know how to subdivide tasks
Need to send code to workers
Need to transfer data between workers
→ unless it is already correctly subdivided and located

Easy option:
→ Each table is stored on a different compute node
→ Fast single table reads
→ Slow joins….

Need to collate results of the workers

What if a worker is lazy? Quits? Makes an error?

Faster to "just do it yourself"?



Big Data Technology covers both storage and processing

Lots of workers? Need a manager(s)
Need a communication protocol.

Either because we use a database that "guesses" or we write code to pre-distribute (replicated?) data or we have an algorithm that does multiple passes of data

Need to know how to subdivide tasks
Need to send code to workers
Need to transfer data between workers
→ unless it is already correctly subdivided and located

Holy Grail of Big data tech - hide all this complexity.

Hard to use the original procedural programming - new paradigm required.
→ parfor, MapReduce, Spark

Each paradigm enables certain parallel tasks to be written easily. But not all. Work is ongoing and highly efficient solutions remain hard.

Easy option:
→ Each table is stored on a different compute node
→ Fast single table reads
→ Slow joins….

Need to collate results of the workers

What if a worker is lazy? Quits? Makes an error?

Faster to "just do it yourself"?

SQL… since this doesn't contain instructions, one just lists the task… huge potential.

Also since this includes the specification of the data storage can potentially leverage benefits of jointly optimising storage and processing!

Big Data Technology covers both storage and processing

Before we talk about these more user friendly big data technology solutions (spoiler, it's SQL)
let's summarize what "big data tech" buys us when we distribute storage and/or processing.

A summary of "big data" technologies use cases

Specifically, we are talking about enabling distributed storage and processing.

Centralized data and processing

→ Limited central processing
→ Limited data storage

Traditional systems (e.g. RDBMs)

→ Simple availability
(all in the same location, either all up/down)

→ Centrally controlled algorithms
(easy concurrency, ACID, etc)
→ No time costs for moving data around
→ No data duplication

A summary of "big data" technologies use cases

Specifically, we are talking about enabling distributed storage and processing.

Centralized data and processing

→ Limited central processing
→ Limited data storage

Traditional systems (e.g. RDBMs)

→ Simple availability
(all in the same location, either all up/down)

→ Centrally controlled algorithms
(easy concurrency, ACID, etc)
→ No time costs for moving data around
→ No data duplication

Distributed data, centralized processing

→ Limited processing
→ Unlimited storage

Analytics when processing data locally from a data lake/relational database.

→ Complexity in availability
(could have partial data access due to outages, need logic to deal with this)
→ Centrally controlled algorithms

→ Cost (complexity and transfer time) to bring the data to you

"Big Data" technology use cases (for analytics)

Specifically, we are talking about enabling distributed storage and processing.

Centralized data and processing

→ Limited central processing
→ Limited data storage

Traditional systems (e.g. RDBMs)

→ Simple availability
(all in the same location, either all up/down)

→ Centrally controlled algorithms
(easy concurrency, ACID, etc)
→ No time costs for moving data around
→ No data duplication

Distributed data, centralized processing

→ Limited processing.
→ Unlimited storage

Analytics when processing data locally from a data lake/relational database.

→ Complexity in availability
(could have partial data access due to outages, need logic to deal with this)
→ Centrally controlled algorithms

→ Cost (complexity and transfer time) to bring the data to you

Centralized data, distributed processing

→ Unlimited processing capability
→ Limited storage capability

Simple, ad-hoc distributed computing tasks.

→ Complexity in availability (can easily avoid nodes not available at start, but what about failure partially through processing?)
→ Need a new programming paradigm to split up computation

→ Cost to send the data to the processing. **Data is still distributed, just in memory. Still need distributed data structures.**

"Big Data" technology use cases (for analytics)

Specifically, we are talking about enabling distributed storage and processing.

Centralized data and processing

→ Limited processing
→ Limited data storage

Traditional systems (e.g. RDBMs)

→ Simple availability
(all in the same location, either all up/down)

→ Centrally controlled algorithms
(easy concurrency, ACID, etc)
→ No time costs for moving data around
→ No data duplication

Distributed data, centralized processing

→ Limited processing.
→ Unlimited storage

Analytics when processing data locally from a data lake/relational database.

→ Complexity in availability
(could have partial data access due to outages, need logic to deal with this)
→ Centrally controlled algorithms

→ Cost (complexity and transfer time) to bring the data to you

Centralized data, distributed processing

→ Unlimited processing capability
→ Limited storage capability

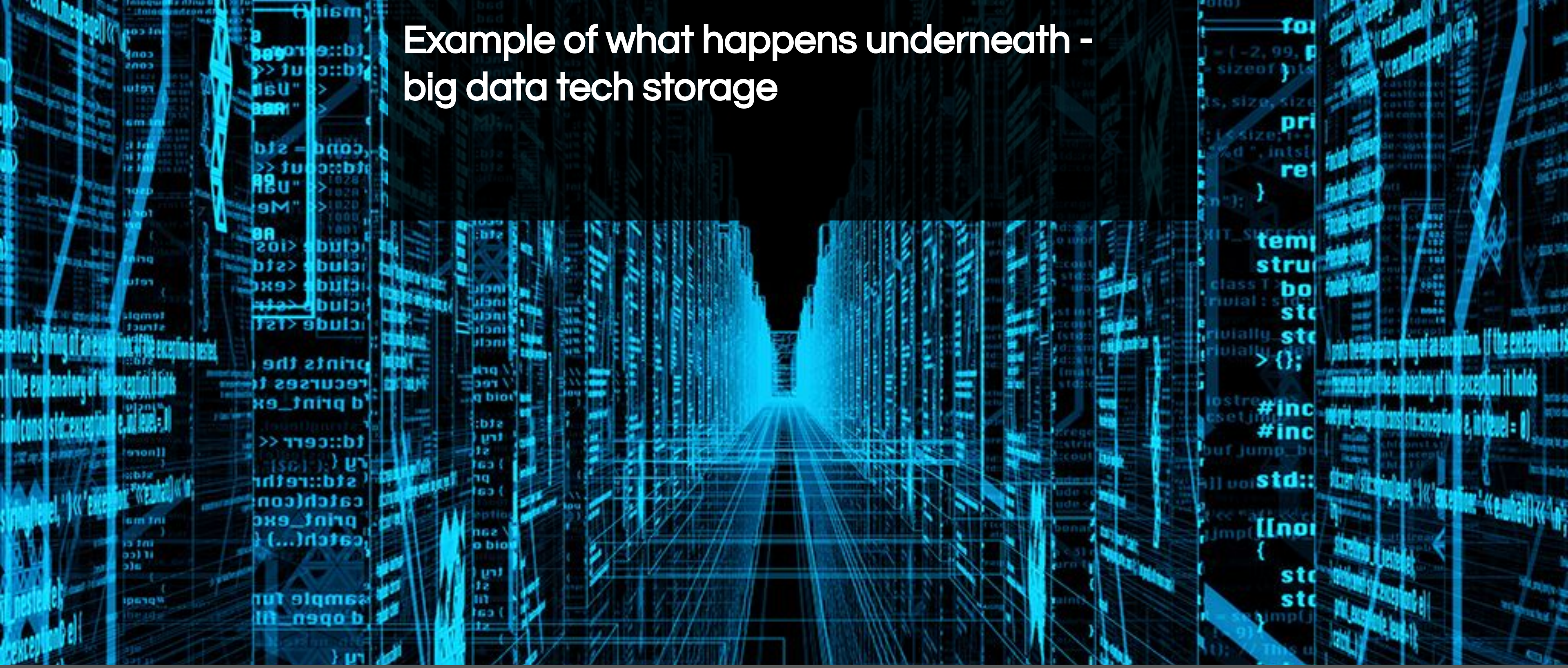
Simple, ad-hoc distributed computing tasks.

→ Complexity in availability (can easily avoid nodes not available at start, but what about failure partially through processing?)
→ **Need a new programming paradigm to split up computation.**
→ Cost to send the data to the processing. **Data is still distributed, just in memory. Still need distributed data structures.**

Distributed data, distributed processing

→ Reduced data transfer costs
Ability to scale to many compute nodes and arbitrary storage.
Premeditated, repeated "big data" processing.
→ Both sets of complexity in availability
→ **Need a new programming paradigm to split up computation**
→ **Need ways of manually/auto splitting data (optimally)**
→ Costs for moving data around when processing if required and back to you (may be a negligible cost).

Example of what happens underneath - big data tech storage



An example of a Big Data Storage solution and why it is hard.... (so you know what you're getting into if you try it....)

NoSQL data stores / processing paradigms

Simple strategies for distribution make some access tasks hard.

→ Replication makes updates hard
→ Partitioning data can lead to long access times

New parallel programming paradigms (more in a minute) aim to allow easy partitioning and replication of data.

Enables programmers (or algorithms on our behalf) to easily dynamically replicate and partition.

For analytics the above is often less of an issue.

More of an issue is the extra costs: when big data technology makes things slower

Key-value stores as a data structure (rather than binary blobs - files).

- A very large dictionary.
- **Provide a way to break data into logical groups**
  - Operations can then be specified to run on these groups in parallel
    - Distribute + Computate

Key-value stores as a data structure.

- A very large dictionary.
- **Provide a way to break data into logical groups**
  - Operations can then be specified to run on these groups in parallel
    - Distribute + Computate

Additionally keys allow indirect mapping to distributed data.

key → value
key → location → value

Keeping this mapping up-to-date and resilient to failure introduces a lot of complexity and extra processing steps.

Key-value stores as a data structure.

- A very large dictionary.
- **Provide a way to break data into logical groups**
  - Operations can then be specified to run on these groups in parallel
    - Distribute + Computate
- **IMPORTANT:** Not all tasks can be completed by a divide and conquer (in parallel) approach! Not all problems are suitable for parallel computation.
  - Eg. the overall mean student performance computed as the average of mean module performance
    - here there are two steps that must be done sequentially (1st compute the per module mean and then take these results and compute the overall mean)

FBA: 76, 62, 52
D@S: 55, 66

AVG(FBA)=63.3
AVG(D@S)=60.5

(63.3+60.5)/2 = 61.9
vs.
(76+62+52+55+66)/5 = 62.2

Key-value stores as a data structure.

- A very large dictionary.
- **Provide a way to break data into logical groups**
  - Operations can then be specified to run on these groups in parallel
    - Distribute + Computate
- **IMPORTANT:** Not all tasks can be completed by a divide and conquer (in parallel) approach! Not all problems are suitable for parallel computation.
  - Eg. the overall mean student performance computed as the average of mean module performance
- How we think when programing distributed computing (map-reduce paradigm, spark) will be based on this data structure
- Levels of abstraction are now built on top to hide this, but understanding is still important when things break or go slowly (technology is new, happens more than we'd like)

- Tables can be logically constructed
  - Keys = columns
  - Values = data in the column

    OR
  - Keys = row IDs
  - Values = rows
- Key value pairs can conceptually form tables.
- So you can see how we start to get SQL built on top….. however…. not all SQL tasks are easy (quick) to do in parallel.

Summary.

Key → Value pairs can be easily distributed. Basis for distributed processing paradigm coming up.

Fast access.

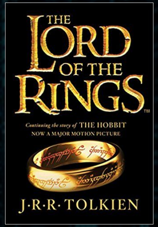
There is still significant complexity. It is just hidden from you.

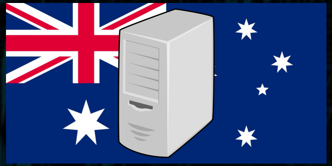
[setup and maintenance can be harder]

Trade-off between consistency and availability.

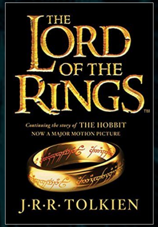
Data replicated and distributed for speed (**high availability**) and in case of network failures **(partition tolerance)**

Global stock: 1

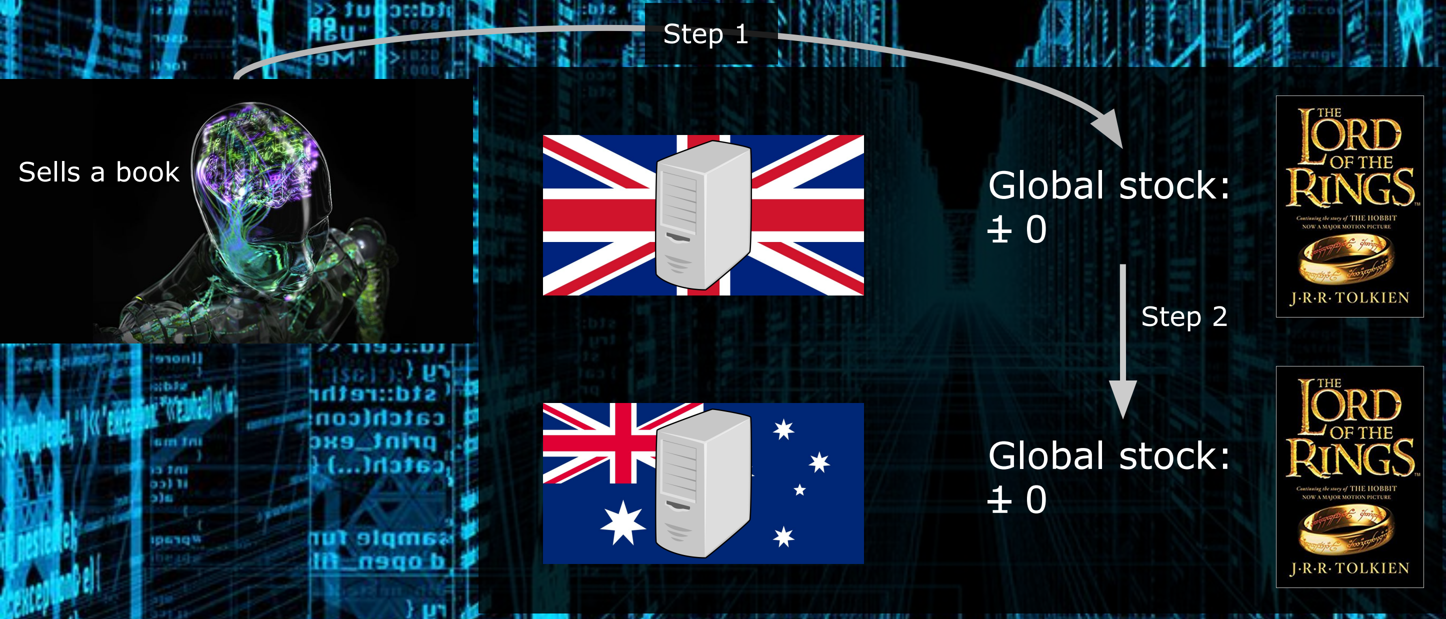




Global stock: 1



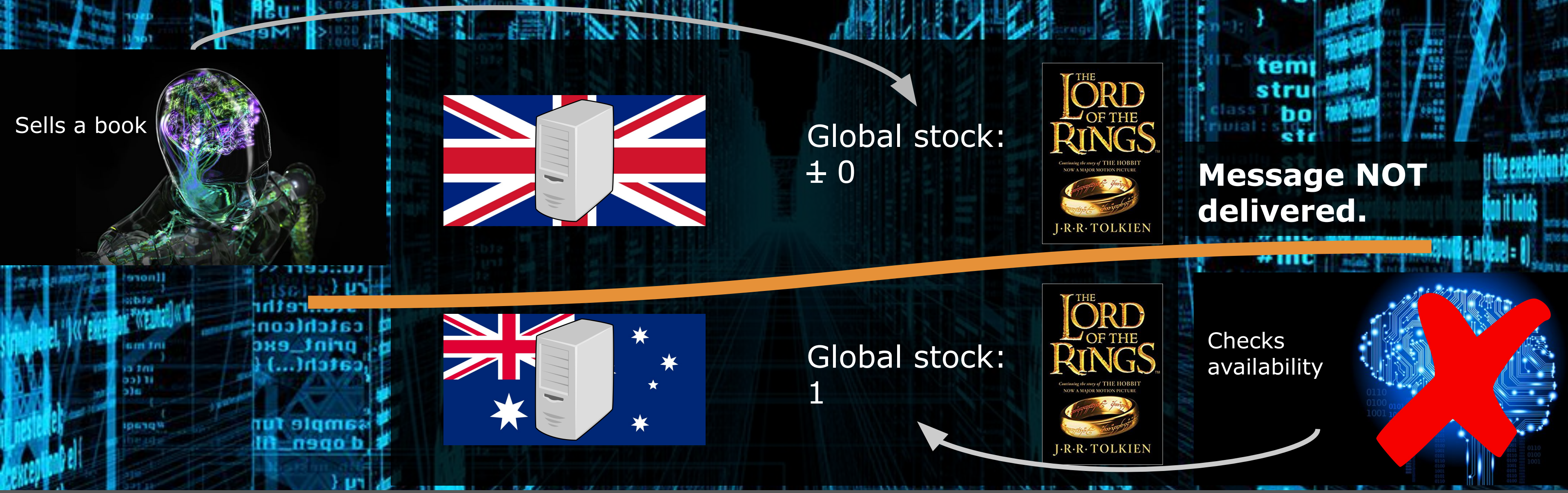

Step 1
Sells a book
Global stock:
± 0
Global stock:
1
THE LORD OF THE RINGS
J·R·R· TOLKIEN
THE LORD OF THE RINGS
J·R·R· TOLKIEN



Step 1
Sells a book
Global stock:
± 0
Step 2
Global stock:
± 0

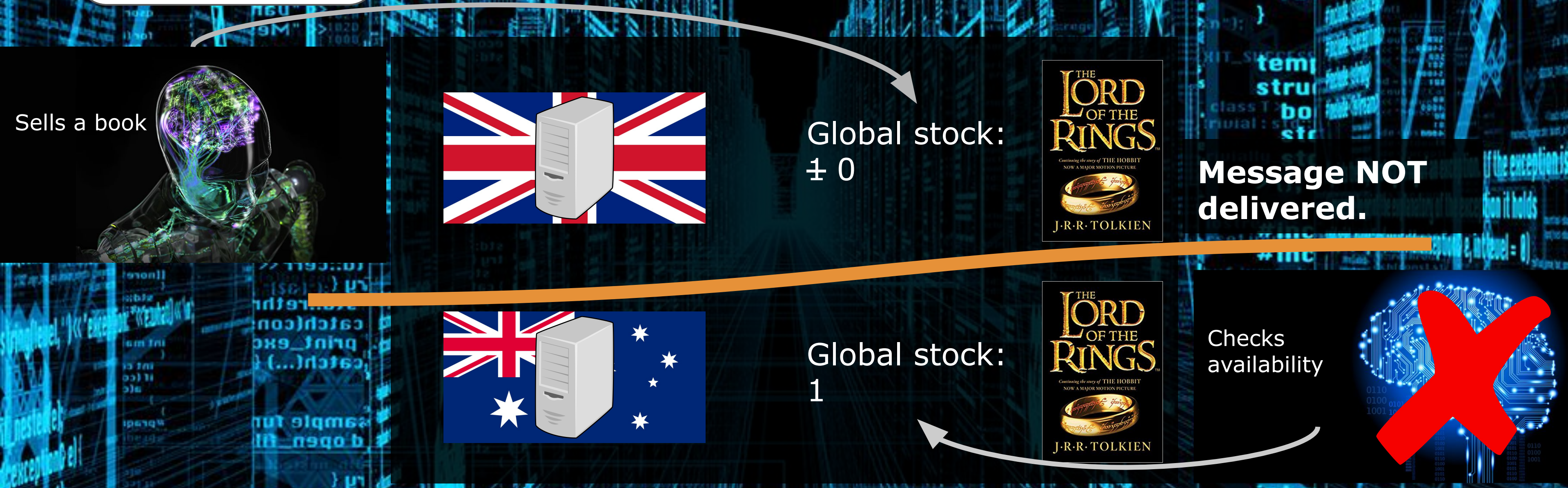

Step 1
Sells a book
Global stock:
± 0
Step 2
Global stock:
± 0
Checks availability
Step 3
THE LORD OF THE RINGS
J·R·R·TOLKIEN
THE LORD OF THE RINGS
J·R·R·TOLKIEN

NETWORK Failure: Database can become inconsistent.

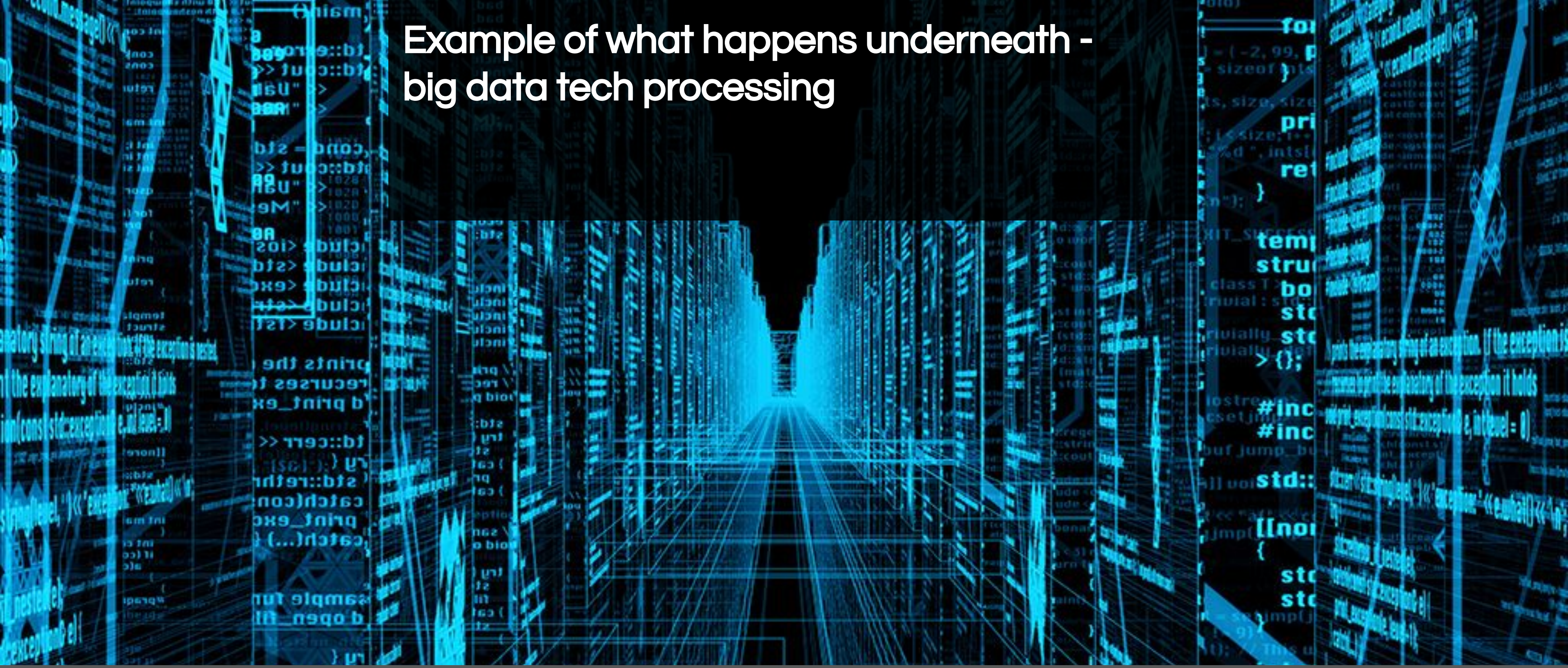


Trade-off, consistency vs. availability

Could monitor link and messages (via receipts), if down throw error.
Don't sell. Or **don't list book availability.**



Example of what happens underneath - big data tech processing



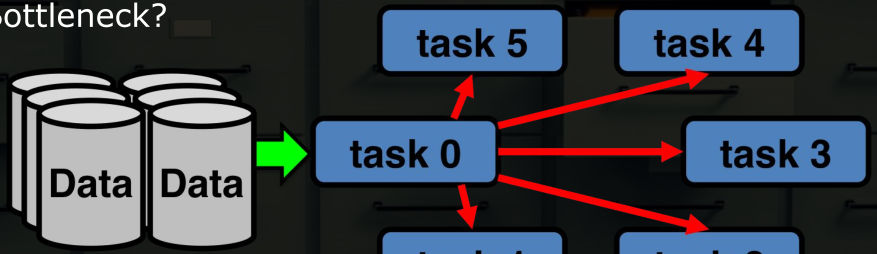
From storage to processing...

Map-Reduce

Traditional parallelism.

Data is brought to the compute.

Bottleneck?



From storage to processing...

Map-Reduce

Map-Reduce parallelism.

Compute is (already) moved to the data!

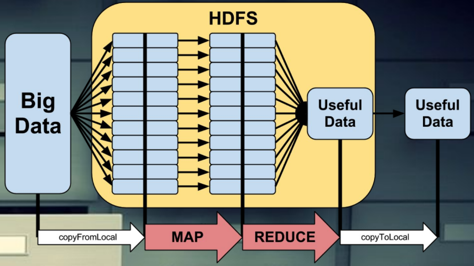
Assume data is already stored in a distributed way in a key-value store*.

Assume compute exists on each storage node.

Task: Find the number of unique 1 character words, 2 character words…
in 50,000 blog posts.

Map-Reduce

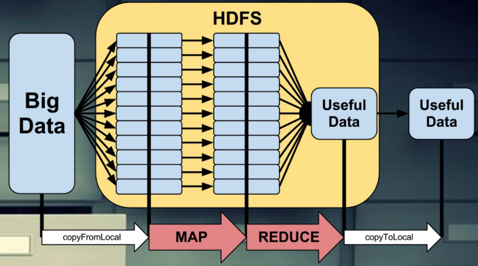
Multi-stage compute paradigm



Task: Find the number of unique 1 character words, 2 character words…
in 50,000 blog posts.

Map-Reduce

Multi-stage compute paradigm



Map stage (**distributed**).

Takes data **already on the computer,** maps it into **appropriate** (key, values) pairs.

Per document (independently, in parallel) place all words of same length in different buckets.

Each worker gets 50 blog posts (id, blog).

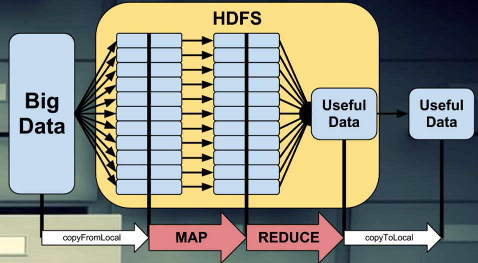
For each word in each blog list: (char_ct, word)

m1: [(1,"a"),(5, "hello"),(2,"if"), (1,"I")]
m2: [(2,"an"),(5, "break"),(3,"the"),(1,"a")]

Task: Find the number of unique 1 character words, 2 character words…
in 50,000 blog posts.

Map-Reduce

Multi-stage compute paradigm



Map stage (**distributed**).

Takes data **already on the computer,** maps it into **appropriate** key, values pairs.

Per document (independently, in parallel) place all words of same length in different buckets.

Each worker gets 50 blog posts (id, blog).

For each word in each blog list: (char_ct, word)

m1: [(1,"a"),(5, "hello"),(2,"if"), (1,"I")]
m2: [(2,"an"),(5, "break"),(3,"the"),(1,"a")]

Group stage.

All values with the same key grouped and sent to the same node for compute.

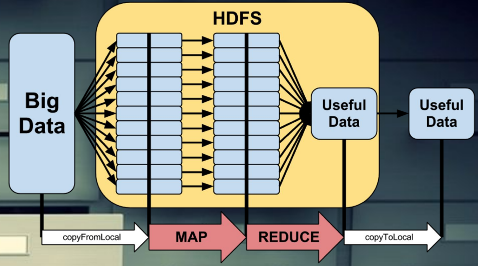
Merge all buckets with the same labels. Redistribute the buckets amongst the workers.

r1: (1, ["a", "I", "a",.....])
r2: (2, ["if" ,"an", ….])
r3: (3, ["the", ….])
r4: (5, ["hello","break", ….])

Task: Find the number of unique 1 character words, 2 character words… in 50,000 blog posts.

Map-Reduce

Multi-stage compute paradigm



Map stage (**distributed**).

Takes data, maps it into **appropriate** key, values pairs.

Per document (independently, in parallel) place all words of same length in different buckets.

Each worker gets 50 blog posts (id, blog).

For each word in each blog list: (char_ct, word)

m1: [(1,"a"),(5, "hello"),(2,"if"), (1,"I")]
m2: [(2,"an"),(5, "break"),(3,"the"),(1,"a")]

Group stage.

All values with the same key grouped and sent to the same node for compute.

Merge all buckets with the same labels. Redistribute the buckets amongst the workers.

r1: (1, ["a", "I", "a",.....])
r2: (2, ["if" ,"an", ….])
r3: (3, ["the", ….])
r4: (5, ["hello","break", ….])

Reduce stage.

Takes all key-value pairs with a given key. Performs some compute to get a value.

Per bucket (independently and in parallel) count the number of distinct words.

(if there really was only this little data)

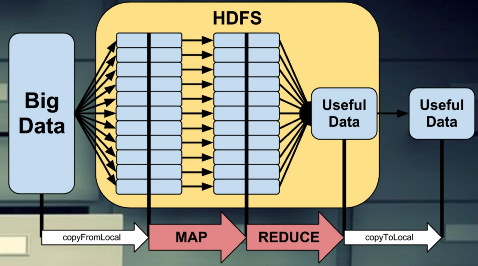
r1: (1, ["a", "I", "a"]) → 2
r2: (2, ["if" ,"an"]) → 2
r3: (3, ["the"]) → 1
r4: (5, ["hello","break"]) → 2

DONE!
(fetch the counts, make a list and return it)

Task: Find the number of unique 1 character words, 2 character words…
in 50,000 blog posts.

Map-Reduce

Multi-stage compute paradigm



Map stage (**distributed**).

Takes data, maps it into **appropriate** key, values pairs.

Per document (independently, in parallel) place all words of same length in different buckets.

DISTRIBUTED, say 1,000 nodes (wherever the data is)

Independently processing input data chunks of data that can be processed independently.

Programmer must define.

Group stage.

All values with the same key grouped and sent to the same node for compute.

Merge all buckets with the same labels. Redistribute the buckets amongst the workers.

NOT DISTRIBUTED

Fast non-independent task (system)

Reduce stage.

Takes all key-value pairs with a given key. Performs some compute to get a value.

Per bucket (independently and in parallel) count the number of distinct words.

DISTRIBUTED, say 100 nodes

Independently process pre-defined independent tasks

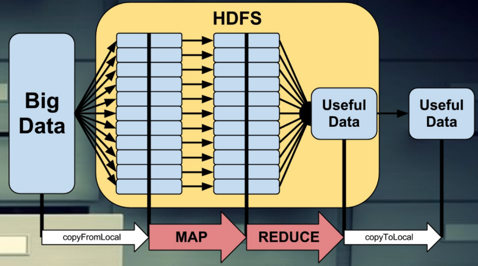
Programmer must define.

DONE!
(fetch the counts, make a list and return it)

Task: Find the number of unique 1 character words, 2 character words…
in 50,000 blog posts.

Map-Reduce

Multi-stage compute paradigm



Map stage (**distributed**).

Takes data, maps it into **appropriate** key, values pairs.

Per document (independently, in parallel) place all words of same length in different buckets.

DISTRIBUTED, say 1,000 nodes (wherever the data is)

Independently processing input data chunks that can be processed independently.

Programmer must define.

Group stage.

All values with the same key grouped and sent to the same node for compute.

Merge all buckets with the same labels. Redistribute the buckets amongst the workers.

NOT DISTRIBUTED

Fast non-independent task (system)

Reduce stage.

Takes all key-value pairs with a given key. Performs some compute to get a value.

Per bucket (independently and in parallel) count the number of distinct words.

DISTRIBUTED, say 100 nodes

Independently process pre-defined independent tasks

Programmer must define.

Paradigm forces **you** to re-cast your problem into chunks of **independent computation** in a **given structure**.

If you can do this, it can be done fast!

Not all problems can be. **And we're at algorithm design….**

Map-Reduce

How we can use it...

Map-Reduce programming.

Chain of independent (Map, group, reduce) operations **on set of key-value data.**

Cannot do everything, still need to be embedded in sequential programming.

Clean data, convert data to correct format → Call someone elses mapreduce function → Call someone elses mapreduce function → Post-process & visualize the results

Map-Reduce

How we can use it...

Map-Reduce programming.

Chain of independent (Map, group, reduce) operations **on set of key-value data.**

Cannot do everything, still need to be embedded in sequential programming.

Significantly more complex than Python.

→ Cannot do everything.

→ Low level interface **(other's MapReduce functions do not do the high-level tasks we might want to)**

Probably not for us.

Clean data, convert data to correct format → Call someone elses mapreduce function → Call someone elses mapreduce function → Post-process & visualize the results

We'll be having an introduction to this way of programming via a Demo.

Using Spark, which is built on Map-Reduce but better!

We're not algorithm designers...

But as people build on top of this and provide better libraries and abstractions, why not!

(but since these are harder to design, techniques availability may lag)

However, there is a lot of extra complexity.

Data movement. Small data, poor implementations = worse performance!!!

Distributed linear regression? Why not.

Distributed deep learning. Almost required.

Premature optimization is the root of all evil*.

(evil = frustration + bugs + time lost)

APACHE Spark™

Builds on Map-Reduce but faster (**10-100x speedup).**

Spark MLlib

OK, so what good libraries do we have?

For Data processing:
- SparkSQL.
- Cockroachdb
- Google Spanner

For Analytics:
- Spark MLib
- Spark
- MapReduce

Can be sped up by correct data storage:
- Delta lake (from Databricks - SparkSQL)
- Cockroachdb (PostgreSQL compatible)
- Google Spanner (own SQL variant)

OK, so what good libraries do we have?

For Data processing:
- SparkSQL.
- Cockroachdb
- Google Spanner

For Analytics:
- Spark MLib
- Spark
- MapReduce

Can be sped up by correct data storage:
- Delta lake (from Databricks - SparkSQL)
- Cockroachdb (PostgreSQL compatible)
- Google Spanner (own SQL variant)

The implementation of distributed SQL is getting quite good.

Distributed ML algorithms is harder.

Silver lining - often, after preprocessing / feature engineering, data is no long "big data".

APACHE Spark

Builds on Map-Reduce but faster (**10-100x speedup).**

Slightly higher level than MapReduce. Required to be aware of because Spark ML doesn't quite hide everything from us yet.

Spark MLlib

Higher level of abstraction for doing machine learning & data analytics

What we want to use.

In MapReduce, after each (map, group, reduce) data is written to disk and reloaded.

Rather than write things to disk, Spark provides a graph/chain processing paradigm. Moves away from key-value pairs to Resilient Distributed Datasets (RDDs)

APACHE Spark™

Builds on Map-Reduce but faster (**10-100x speedup).**

Slightly higher level than MapReduce. Required to be aware of because Spark ML doesn't quite hide everything from us yet.

Spark MLlib

Higher level of abstraction for doing machine learning & data analytics

What we want to use.

In MapReduce, after each (map, group, reduce) data is written to disk and reloaded.

Rather than write things to disk, Spark provides a graph/chain processing paradigm. Moves away from key-value pairs to Resilient Distributed Datasets (RDDs)

Intuitively, programs are written as a set of consecutive (from your point of view) number of transforms on a common data structure.

Somewhat like SQL!

Chain length is arbitrary, execution order is globally optimised.

Data is kept in **local memory or redistributed** based on automatic analysis of the chain. **10-100x faster!**

Builds on Map-Reduce but faster (**10-100x speedup).**

Slightly higher level than MapReduce. Required to be aware of because Spark ML doesn't quite hide everything from us yet.

Higher level of abstraction for doing machine learning & data analytics

What we want to use.

Ok, so we are not using key-value pairs, what now!

RDDs are built on key-value pairs.

Collections (lists) of data with either explicit partitioning for parallelizing
→ list is a list of key-value pairs

or implicit partitioning if not
→ if the list is something else

More operations and low level control of parallelization if it is a list of key-value pairs

Ok, so we are not using key-value pairs, what now!

RDDs are built on key-value pairs.

Collections (lists) of data with either explicit partitioning for parallelizing
→ list is a list of key-value pairs

or implicit partitioning if not
→ if the list is something else

More operations and low level control of parallelization if it is a list of key-value pairs

Resilient Distributed Datasets (RDDs):

A collection of data.

E.g:

a collection of numbers:
[1,2,3,4]

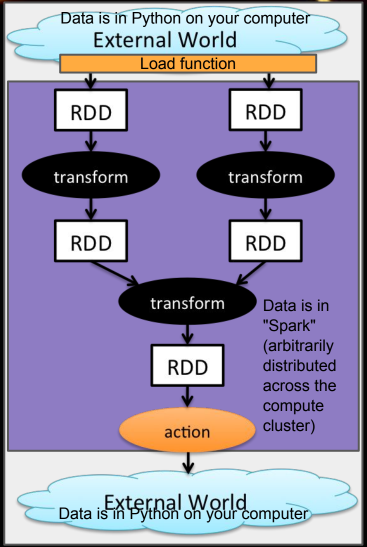
a collection of key value pairs:
[('a',7),('a',2),('b',2)]

a collection of tuples:
[(10, [0.5,0.1]), (4, [0.5,0.4])]

Introducing…

When you load data into spark you load it in a special format that can be automatically distributed.

RDDs (Resilient Distributed Datasets)
→ A collection numbers, tuples, key-value pairs….



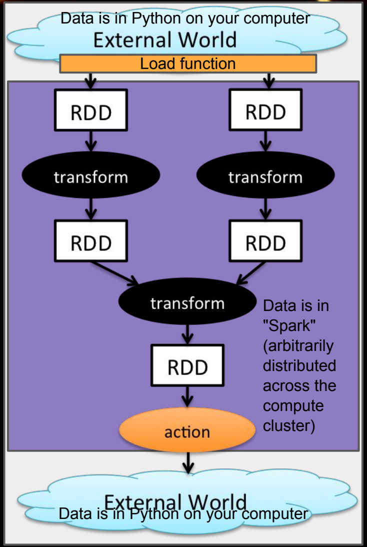
RDDs live in the spark cluster NOT your computer.

Introducing... Apache Spark™

When you load data into spark you load it in a special format that can be automatically distributed.

RDDs (Resilient Distributed Datasets)
→ A collection numbers, tuples, key-value pairs....

RDDs live in the spark cluster NOT your computer.



We load data from Python (our computer) into the spark cluster in to RDDs via special functions.

We process data via **transforms** in Spark (away from our computer).

Transforms: Actions on RDD(s) that return RDD(s).

We move the data back to our computer (Python) via an **action.**

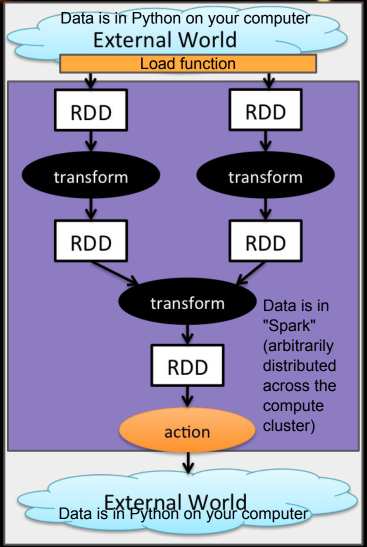
* Most common. Some exceptions exist but these are well beyond the scope of this course.

Introducing… Apache Spark

When you load data into spark you load it in a special format that can be automatically distributed.

RDDs (Resilient Distributed Datasets)
→ A collection numbers, tuples, key-value pairs….

RDDs live in the spark cluster NOT your computer.



We load data from Python (our computer) into the spark cluster in to RDDs via special functions.

We process data via **transforms** in Spark (away from our computer).

Transforms: Actions on RDD(s) that return RDD(s).

We move the data back to our computer (Python) via an **action.**

Transforms are lazy!

They form a chain of processing that are not done until an action is requested.

Why? The computer will optimize the way it processes based on the set of actions.

Adding a new transform could drastically alter the best way of doing things.

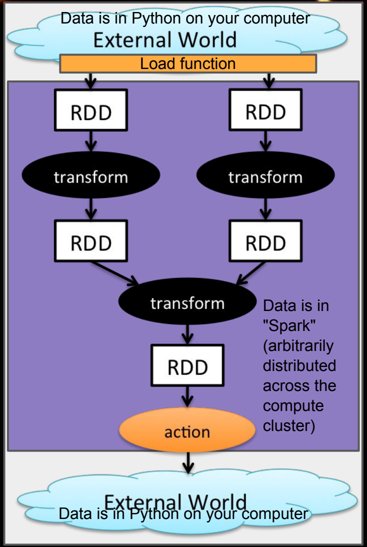
Also: If no one needs (requests) the output, why do the processing?

* Most common. Some exceptions exist but these are well beyond the scope of this course.

Introducing...

A common transform is reduceByKey

→ is a method of an RDD
→ takes a **compatible function** as
a parameter



A **compatible function** here is one
of the form:

```
def fn(a, b):
    <some processing>
    return  value
```

```
e.g.
def fn(a, b):
    return a + b
```

Introducing... Apache Spark

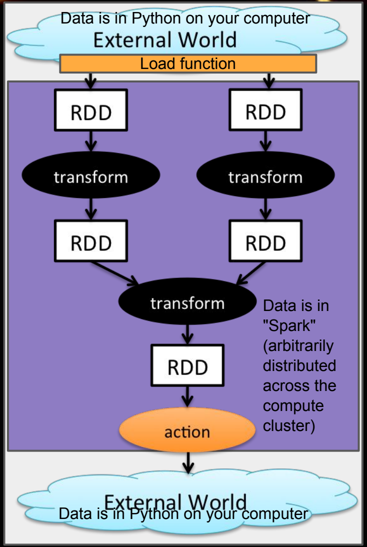
A common transform is reduceByKey

→ is a method of an RDD
→ takes a **compatible function** as a parameter

→ all item pairs in the RDD are grouped (put into buckets) based on the key value

→ for each bucket the function is repeatedly applied to pairs of values until a single value is computed

→ returns a new RDD (one key→ value pair per bucket)



A **compatible function** here is one of the form:

```
def fn(a, b):
    <some processing>
    return  value
```

```
e.g.
def fn(a, b):
    return a + b
```

Introducing... Apache Spark

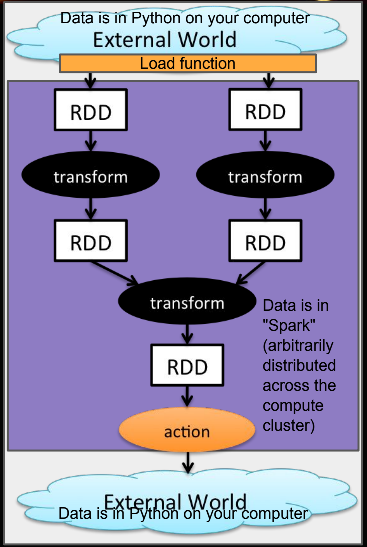
A common transform is reduceByKey

→ is a method of an RDD
→ takes a **compatible function** as a parameter

→ all item pairs in the RDD are grouped (put into buckets) based on the key value

→ for each bucket the function is repeatedly applied to pairs of values until a single value is computed

→ returns a new RDD (one key→ value pair per bucket)



A **compatible function** here is one of the form:

```
def fn(a, b):
    <some processing>
    return  value
```

```
e.g.
def fn(a, b):
    return a + b
```

Like SQL, a finite number of transformations exist.

The way you combine them and the functions you pass enable a wide (**but not complete**) range of processing.

Unfortunately, **not all problems can be cast in this** (map/reduce) **way.**

This paradigm is not complete. Also, some tasks are very hard to convert to this way of thinking.

Still need languages like Python.

Introducing... Apache Spark

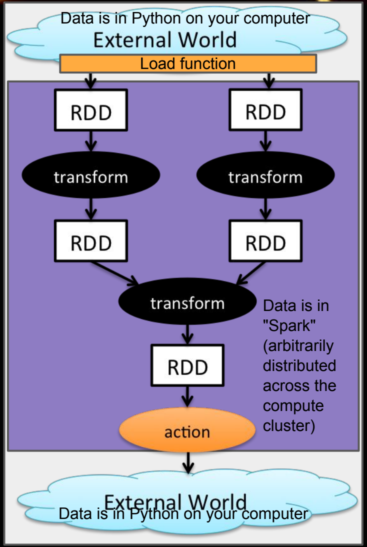
A common transform is reduceByKey

→ is a method of an RDD
→ takes a **compatible function** as a parameter

→ all item pairs in the RDD are grouped (put into buckets) based on the key value

→ for each bucket the function is repeatedly applied to pairs of values until a single value is computed

→ returns a new RDD (one key→ value pair per bucket)



A **compatible function** here is one of the form:

```
def fn(a, b):
    <some processing>
    return  value
```

In SQL we are abstracting navigational access.

Here we are **abstracting away** how to access and schedule across a **distributed system**.

Still, Spark is a lower level of abstraction than SQL.

Like SQL, a finite number of transformations exist.

The way you combine them and the functions you pass enable a wide (**but not complete**) range of processing.

Unfortunately, **not all problems can be cast in this** (map/reduce) **way.**

This paradigm is not complete. Also, some tasks are very hard to convert to this way of thinking.

Still need languages like Python.

* Most common. Some exceptions exist but these are well beyond the scope of this course.

Introducing...

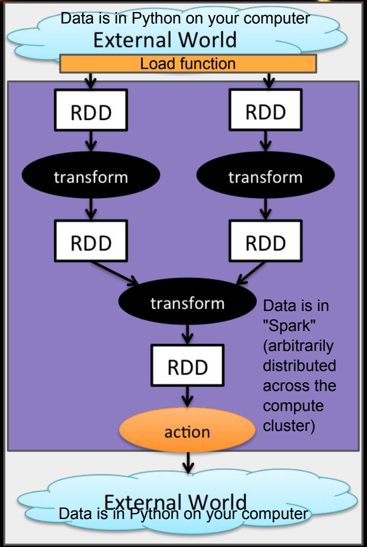
A common transform is reduceByKey

→ is a method of an RDD
→ takes a **compatible function** as a parameter

→ all item pairs in the RDD are grouped (put into buckets) based on the key value

→ for each bucket the function is repeatedly applied to pairs of values until a single value is computed

→ returns a new RDD (one key→ value pair per bucket)



Still saying what to do. Can't do everything. **Still encapsulated in e.g. Python.**

Spark **does** provide an SQL like transforms and actions! **Not complete. Not guaranteeing ACID etc. Not necessarily faster.**

More broadly this is what **NewSQL databases** are looking at! Abstraction on top of this abstraction...

This **level can be good for analytics** though… let's look at this more!

In SQL we are abstracting navigational access.

Here we are **abstracting away** how to access and schedule across a **distributed system**.

Still, Spark is a lower level of abstraction than SQL.

Like SQL, a finite number of transformations exist.

The way you combine them and the functions you pass enable a wide (**but not complete**) range of processing.

Unfortunately, **not all problems can be cast in this** (map/reduce) **way.**

This paradigm is not complete. Also, some tasks are very hard to convert to this way of thinking.

Still need languages like Python.

* Most common. Some exceptions exist but these are well beyond the scope of this course.

Introducing…

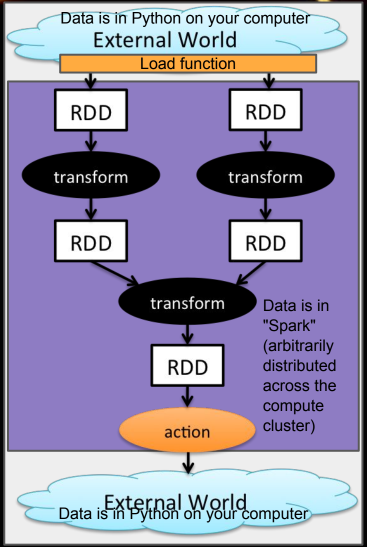
Spark MLlib

Transforming RDDs is still quite low level.

Machine learning (building supervised/unsupervised models) has a fixed common set of steps.

Someone else should work out a shorthand version of the process that parallelize automatically.

→ Less general (suitable for ML only), but higher level of abstraction. Yes please.

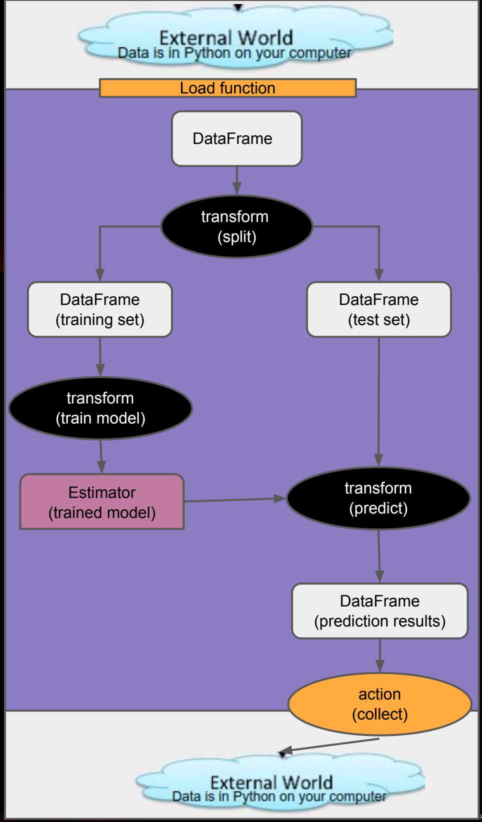


Rather than RDDs → DataFrames.

Spark DataFrames:
RDDs with extra information.

Spark ML we transform DataFrames.

Unlike Spark, in Spark ML we also have **estimators** (models)**.** We can fit these to end up with **trained models**.



Solutions part-way between

What you need to remember from today:

→ What a key-value data structure is.

→ Accept that key-value pairs can be easily distributed and form the basis of most parallel processing paradigms.

→ Processing key-value pairs in parallel can not always be done: **Task and data dependent.**

Current programming languages either do not give the computer enough freedom to decide when to do things in parallel and when not to (Python)
OR
They are not smart enough yet to work it out (SQL for data manipulation, unknown for full data analytics)

→ Requires us to think / learn a new programming paradigm OR use libraries so we still think in linear steps but each step involves (hidden) parallel processing.

Map-Reduce is a parallel programing paradigm.

Spark is a parallel programming framework.

Can't do everything.

Embedded in a traditional programming language.

Typically backed by a **distributed file system.**

+

A distributed resource manager.

The open version of this ecosystem is known as **Hadoop.**

Spark runs on top of Hadoop. Hadoop incorporates MapReduce.

With Hadoop
Distributed data, distributed processing

Without Hadoop
Centralized data, distributed processing

Hadoop **tries** to provide a distributed file system that **looks like a traditional file system.**

The Hadoop file system.

Arbitrary files can be stored.

Hadoop **tries** to provide a distributed file system that **looks like a traditional file system.**

The Hadoop file system.

Arbitrary files can be stored.

However, to ensure correct automatic distribution and use by algorithms you must use specific formats.

For Spark we load into Resilient Distributed Dataset (RDD) structures or DataFrames for processing.

Save DataFrames to Columnar file format!

These are ones that Hadoop knows how to cut into key-value pairs*!!

A common format is tab-delimited text files.

Others:
Sequence files (basically key→ value pairs)
Columnar file formats (towards tables as an abstraction over key-value pairs). These become **DataFrames** in Spark!

Spark Demo
Almost.

FUNCTIONS

```
def add5( num_in ):
    return num_in + 5
```

Functions comprise of a name (human readable versions of memory address where they are stored) and a definition.

Just like variables comprise of a name and a value (definition).

e.g. age = 3
my_book =

FUNCTIONS

```
def add5( num_in ):
    return num_in + 5
```

```
def add5( num_in ):
    return num_in + 5

def run_fn( fn, num_in ):
    return fn( num_in )


>> run_fn( add5, 2)
>> 7
```

Passing Functions

Functions comprise of a name (human readable versions of memory address where they are stored) and a definition.

Just like variables comprise of a name and a value (definition).

e.g. age = 3
my_book =

As such functions can be passed to other functions by name, just like variables.

[THIS IS WHAT YOU NEED TO REMEMBER]

FUNCTIONS

```
def add5( num_in ):
    return num_in + 5
```

```
def add5( num_in ):
    return num_in + 5

def run_fn( fn, num_in ):
    return = fn( num_in )


>> run_fn( add5, 2)
>> 7
```

Passing Functions

Functions comprise of a name (human readable versions of memory address where they are stored) and a definition.

Just like variables comprise of a name and a value (definition).

e.g.
age = 3
my_book =

As such functions can be passed to other functions by name, just like variables.

[THIS IS WHAT YOU NEED TO REMEMBER]

Anonymous Functions

Sometimes we do not declare variables but directly use them in functions:

```
print('at')
```

rather than

```
my_str = 'hi'
print(my_str)
```

FUNCTIONS

```
def add5( num_in ):
    return num_in + 5
```

```
def add5( num_in ):
    return num_in + 5

def run_fn( fn, num_in ):
    return fn( num_in )


>> run_fn( add5, 2)
>> 7
```

Passing Functions

Functions comprise of a name (human readable versions of memory address where they are stored) and a definition.

Just like variables comprise of a name and a value (definition).

e.g. age = 3
my_book = 

As such functions can be passed to other functions by name, just like variables.

[THIS IS WHAT YOU NEED TO REMEMBER]

Anonymous Functions

Sometimes we do not declare variables but directly use them in functions:

```
print('at')
```

rather than

```
my_str = 'hi'
print(my_str)
```

```
def add5( num_in ):
    return num_in + 5

Is the same as:

lambda num_in: num_in + 5
```

We can do the same with functions via a keyword **lambda**.

```
def run_fn( fn, num_in ):
    return fn( num_in )

>> run_fn( lambda num_in: num_in + 5, 2)

>> 7
```

FUNCTIONS

```
def add5( num_in ):
    return num_in + 5
```

```
def add5( num_in ):
    return num_in + 5

def run_fn( fn, num_in ):
    return fn( num_in )


>> run_fn( add5, 2)
>> 7
```

Passing Functions

Passing function definitions is important in parallel processing as the code to run needs to be distributed to each compute node as well as the data.

As such functions can be passed to other functions by name, just like variables.

[THIS IS WHAT YOU NEED TO REMEMBER]

Anonymous Functions

```
def add5( num_in ):
    return num_in + 5

Is the same as:

lambda num_in: num_in + 5
```

We can do the same with functions via a keyword **lambda**.

```
def run_fn( fn, num_in ):
    return fn( num_in )

>> run_fn( lambda num_in: num_in + 5, 2)

>> 7
```

Spark Demo