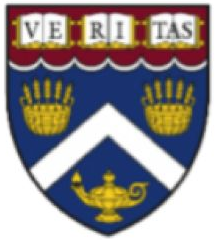
CSCI E-88 Principles Of Big Data Processing

Harvard University Extension, Fall 2019

Marina Popova



Lecture 5 Master Datasets and Batch Processing with Spark

Agenda

- admin stuff
- Master Datasets storage
  - formats and serialization
- Batch Processing with Spark

Why is workload so high?

While the focus of the class is not on learning specific frameworks (like Spark/ Google DataFlow or such) - but on understanding those common core concepts and techniques of building such systems - you still have to understand what it takes to build those systems ...

Why, you might ask?

Let's consider two common scenarios:

First: You pick up some framework or language, say Mongo, take an online class, learn how to use it, go deeper and really master it. Next, most likely, when you are given a problem to solve - you'll automatically turn to the framework you have mastered so well to solve it.

Another scenario: you get a task to solve a problem by writing a software module using some language X (say Python) and some framework ABC (for example Spark). You happily dive into the given task and produce the required solution.

What is this class about?

Yes, it is great, and nothing is wrong with either of the paths. In fact, this is what any engineer should start with.

However, this is what I call a "**commodity programming**" - a level of engineering where you have to have clear directions of how and with what tools to solve a problem.

It is also called a "tutorial hell" in this great post :) :

https://codeburst.io/digging-my-way-out-of-tutorial-hell-6dd5f9927384

My favorite quotes from the post:

What is "tutorial hell" ??

"... they [tutorials] will ... instate a false sense of knowledge in you .."

*"...**There is something comforting about being told what to do and how to do it**. It feels good. … You work on projects, and you get to complete them. You learn new technologies. You do stuff. You gain experience. Or so you think. Or so I thought.*

*Until I decided to **ditch the tutorials** and work on a real project.*

*And suddenly **I was lost** … "*

*"...**Creating something from scratch requires the knowledge of technologies, and also the knowledge of putting the pieces together**. And it is very easy to slip back to the warm and cozy comfort of another tutorial! Do I use CDN or do I download frameworks and libraries via npm? Should I use npm or yarn? What are the benefits of one over the other? And the difference between a framework and a library? I'm sure I read about it somewhere.*

*Off to do a quick Google search, and, before I know it, I'm neck deep in another tutorial. The story of my life. **I cannot resist the subtle, yet enticing calling of a new tutorial** "*

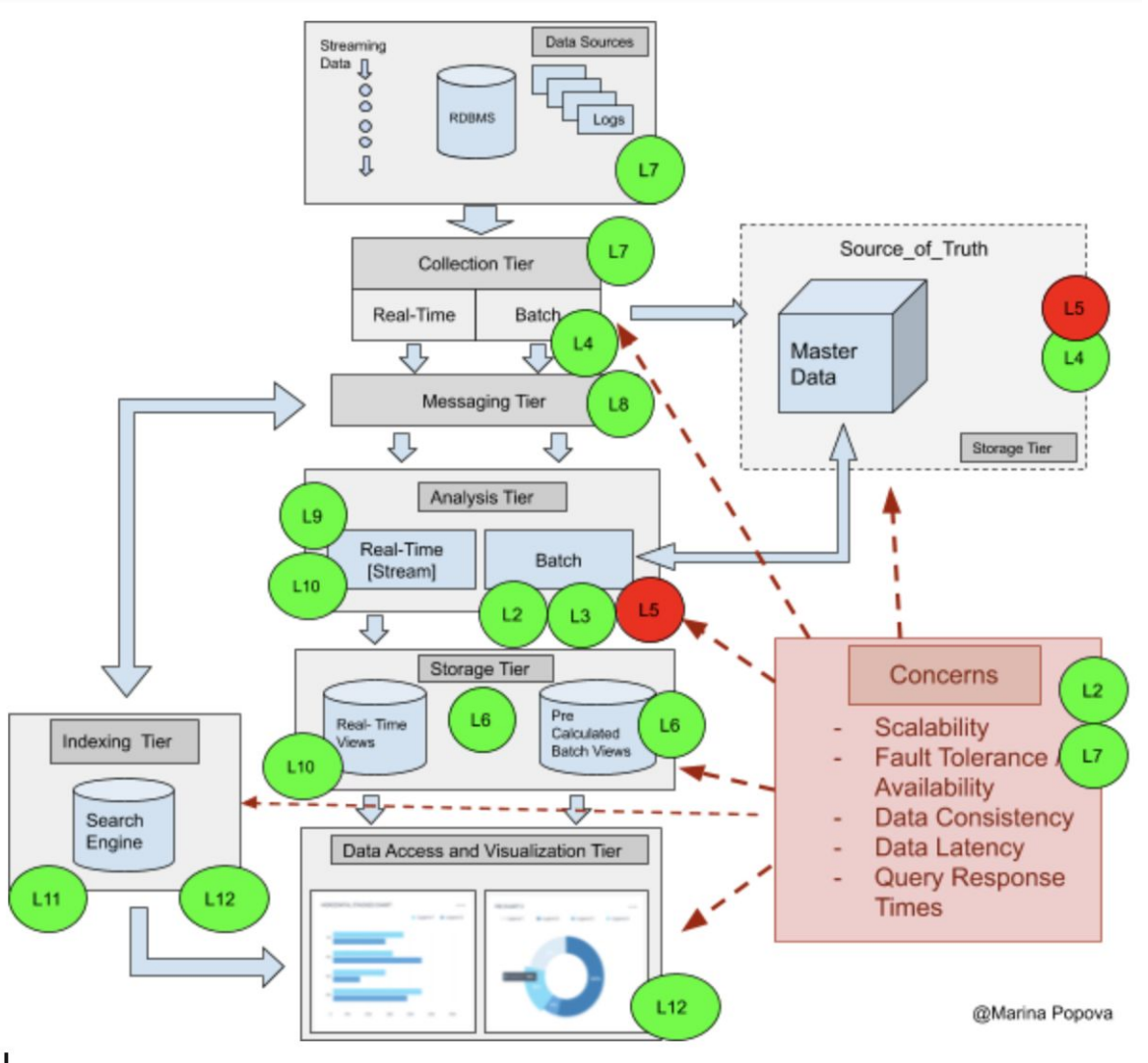
Class Goals

The goal of this class is to take you a few levels up - and **to help you dig your way out of the 'tutorial hell'** :-) How?

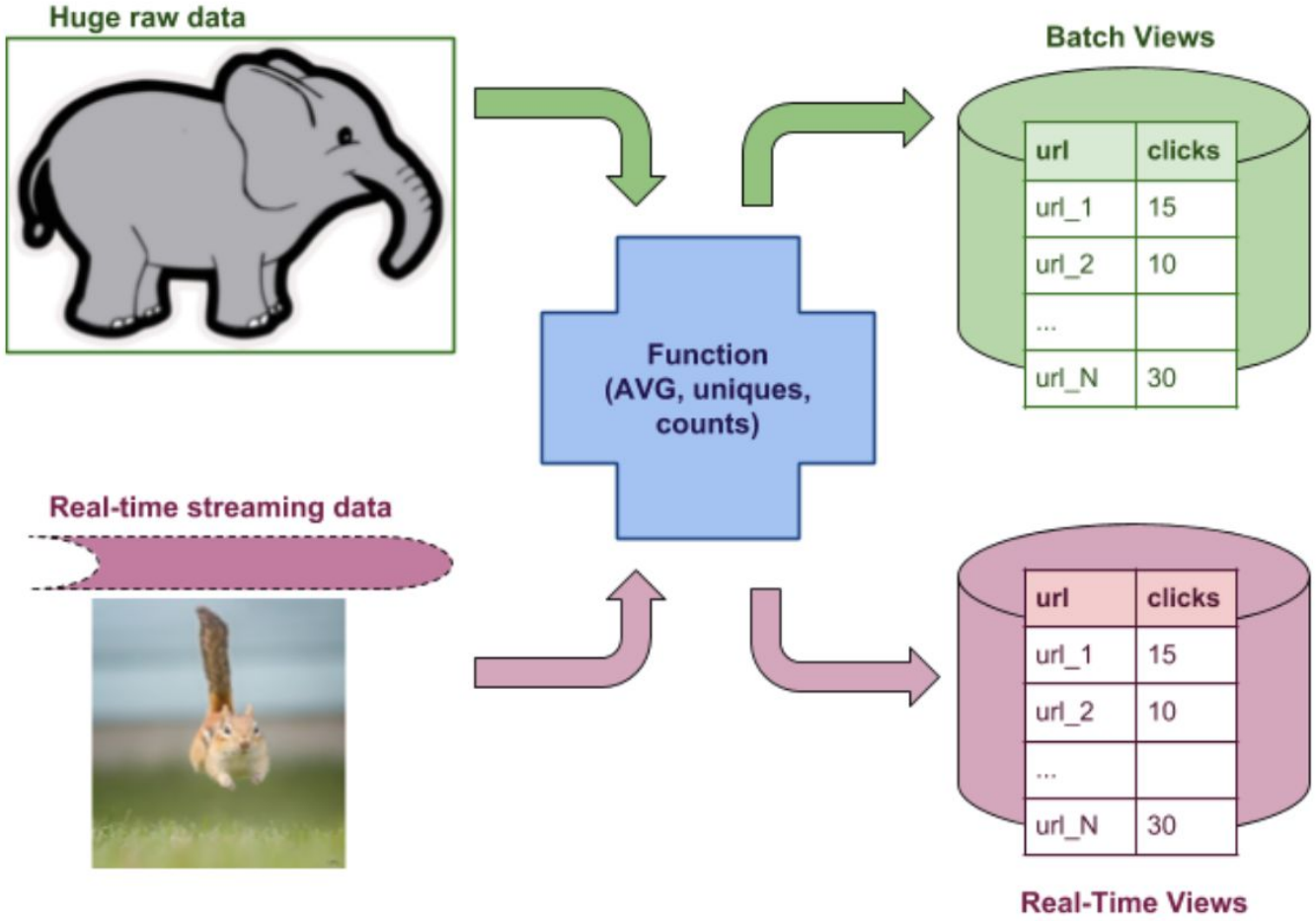
-- by giving you enough knowledge to make informed decisions
-- by teaching you common concepts and concerns of any distributed system aiming to process/analyse Big Data - so that, when given requirements to build a system X, you will be able to ask the right questions to identify correct technical requirements and system architecture
-- by teaching you common techniques/ patterns to solve common challenges - and identify suitable data processing pipelines and frameworks to do that
-- by getting you to work hands-on with some of those frameworks in your homeworks, to further reinforce those common concepts in practice - and to give invaluable practical experience and skills that are in high demand in the modern engineering teams

In short: **the goal of this class is to help you become a technology leader/ architect/ [whatever you name that :)] who understands a bigger picture of the Big Data world and is able to recommend and build real-life data processing pipelines**

Where Are We?


Streaming Data
Data Sources
RDBMS
Logs
L7
Collection Tier
L7
Real-Time
Batch
L4
Source_of_Truth
Master Data
L5
L4
Storage Tier
Messaging Tier
L8
Analysis Tier
L9
Real-Time [Stream]
Batch
L10
L2
L3
L5
Storage Tier
Real- Time Views
L6
Pre Calculated Batch Views
L6
L10
Indexing Tier
Search Engine
L11
L12
Data Access and Visualization Tier
L12
Concerns
L2
- Scalability
- Fault Tolerance / Availability
L7
- Data Consistency
- Data Latency
- Query Response Times
@Marina Popova

Where are we?


Huge raw data
Batch Views
url
clicks
url_1
15
url_2
10
...
url_N
30
Function
(AVG, uniques,
counts)
Real-time streaming data
url
clicks
url_1
15
url_2
10
...
url_N
30
Real-Time Views

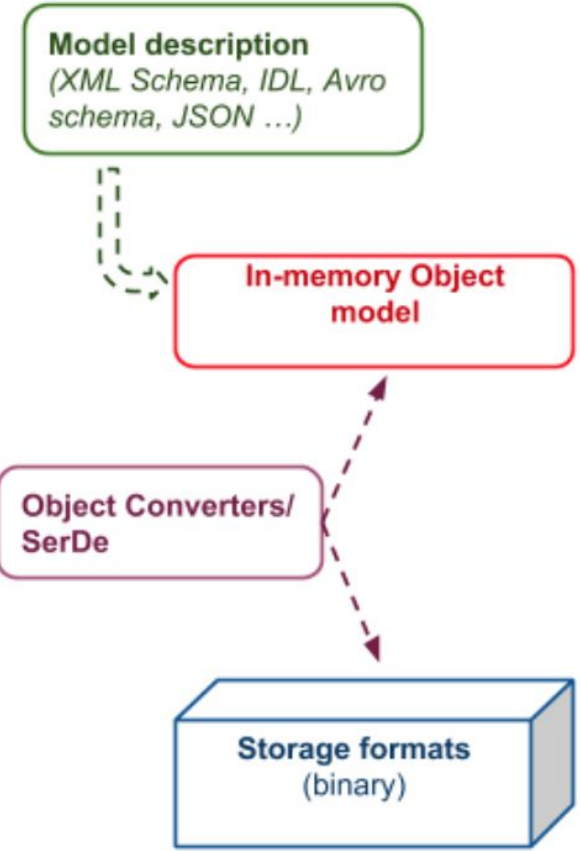
Master Datasets

Reminder for Master Data Storage - the important aspects to address are:

- ✓ What to store (content and model) ?
- ✓ Where to store to (the actual data storage systems) ?
- ○ How to store - storage formats and serialization ?
- ○ How to store - algorithms and operations ?

We discussed how to choose a data model for you master data, and how to choose the data storage system; now lets consider how to persist the data into such system

What is Serialization again?



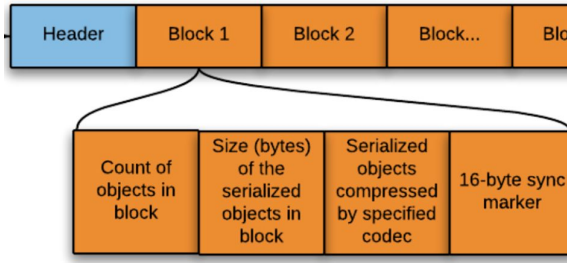

```
{"namespace": "example.avro",
 "type": "record",
 "name": "User",
 "fields": [
     {"name": "name", "type": "string"},
     {"name": "favorite_number",  "type": ["int", "null"]},
     {"name": "favorite_color", "type": ["string", "null"]}
 ]
}
```

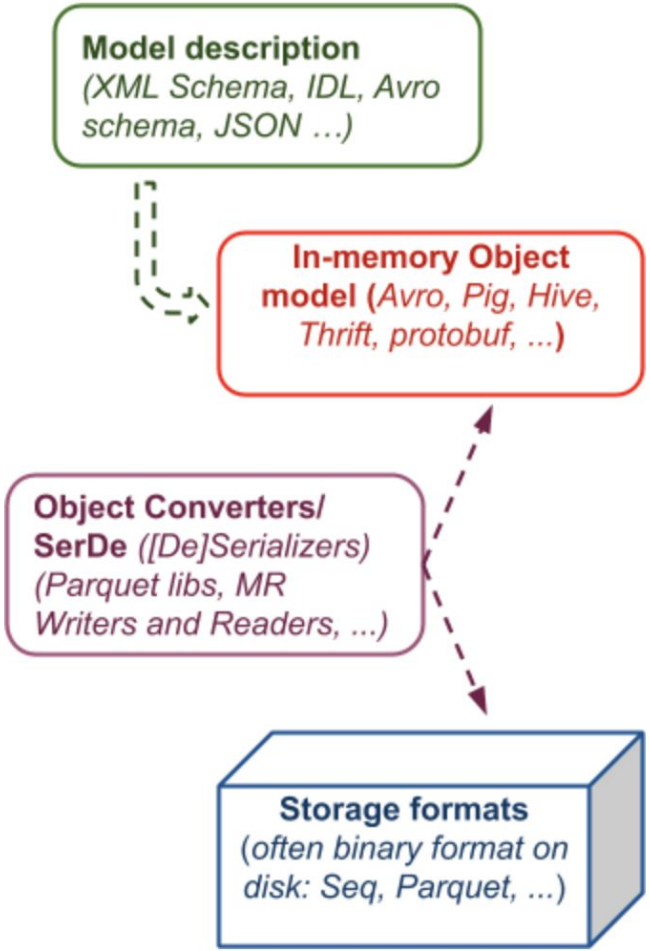
```
public class User {

    private String name;
    private int favoriteNumber;
    private String favoriteColor;
```

user.avro

```
1  4f62 6a01 0414 6176 726f 2e63 6f64 6563
2  086e 756c 6c16 6176 726f 2e73 6368 656d
3  61f2 057b 2274 7970 6522 3a22 7265 636f
4  7264 222c 226e 616d 6522 3a22 7477 6974
5  7465 725f 7363 6865 6d61 222c 226e 616d
6  6573 7061 6365 223a 2263 6f6d 2e6d 6967
```

AVRO FILE:





Important Notes

If at all possible, **data should have an enforceable schema**! Why?

- no missing or inconsistent data
- all values are of expected types

This means some SerDe framework is required!

(to serialize and deserialize your data objects when storing and retrieving them from disk)

We will focus on:

- Input data File Formats
- Serialization Formats
- Storage Formats (Columnar vs. Row-oriented)
- Compression

HDFS Storage Notes:

Important points:

- there is no “standard” data storage format for HDFS - like with any file system, HDFS allows storage of data in any format: text, binary, images, etc.
- there are formats that are optimized for storage and processing in HDFS though
- **Hadoop does not store indexes on data** - this enables very fast data ingestion. But this means that the data has to be read **all in** even if you are running a query accessing only a small subset of the data (full table scan). Solution:
  - **partitioning**: organizing your data into sub-directories according to the logical meaning of the data - make sure not many small partitions are created
  - **bucketing**: uses hashing function to map data into specific buckets
  - **denormalizing**: organizing data into pre-joined (pre-aggregated) data sets to avoid joins
  - using data formats that allow **selective reads** of data (like Parquet)

Master Dataset Formats

There are many options for representing data - either collected into your system or stored in your master data storage:

- Unstructured text or binary
- structured XML
- semi structured JSON documents
- More specialized formats like Avro, Parquet

Master Dataset Formats

Ref: http://www.svds.com/dataformats/ - for more details

Text data

- common for logs and other unstructured data like emails
- the most important consideration here is organization of files in the filesystem -
  - partitioning: organizing data by folders like “date-day” or “customerId” , etc.
- make sure to not have **too many small files**!
- compression: depends on how data will be used:
  - the most compact form - saves storage and network I/O - good for long-term/less frequent access
  - less compact but splittable format - to enable parallel processing via MapReduce

Master Dataset Formats

Structured text data (XML and JSON)

- there is no built-in Hadoop InputFormat for either - and JSON usually does not have markers for start/end of records, which makes it hard to split for parallel processing. Solutions:
  - use a container format such as Avro
  - use a library:
    - for example, for XML: XMLLoader for Pig
    - for JSON: Elephant Bird project (LzoJsonInputFormat)

Binary data

- use a container format like SequenceFiles

Master Dataset Formats

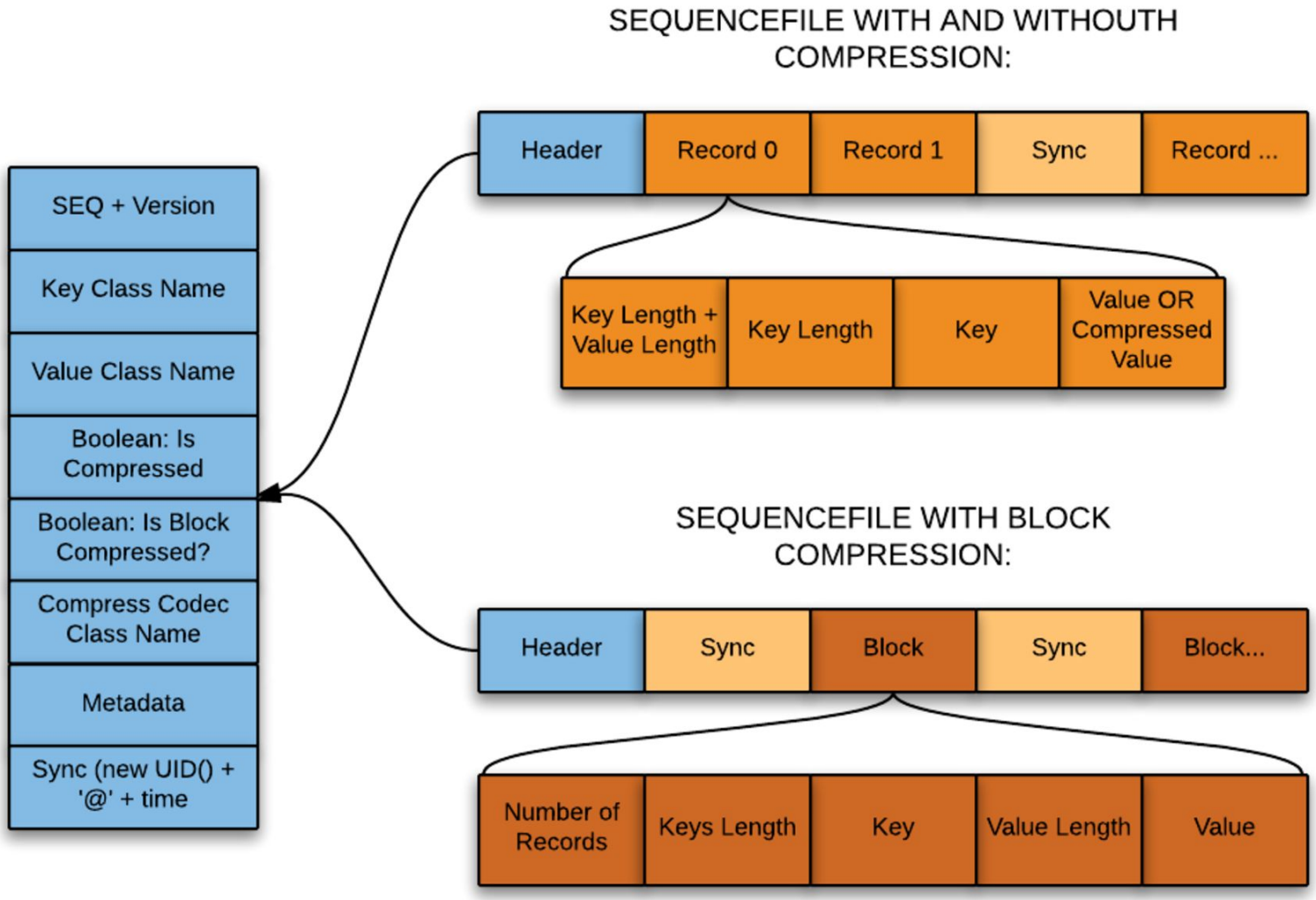
There are several **Hadoop-specific file formats** that were specifically created to work well with MapReduce.
They all support **splittable compression**.

SequenceFile format

- ***binary storage*** on disk (less space than text)
- file-based format; there are also others like MapFiles, SetFiles - but they are very specialized
- well integrated with the Hadoop frameworks (Pig, Hive, MapReduce)
- only supported in Java
- supports append only

Master Dataset Formats

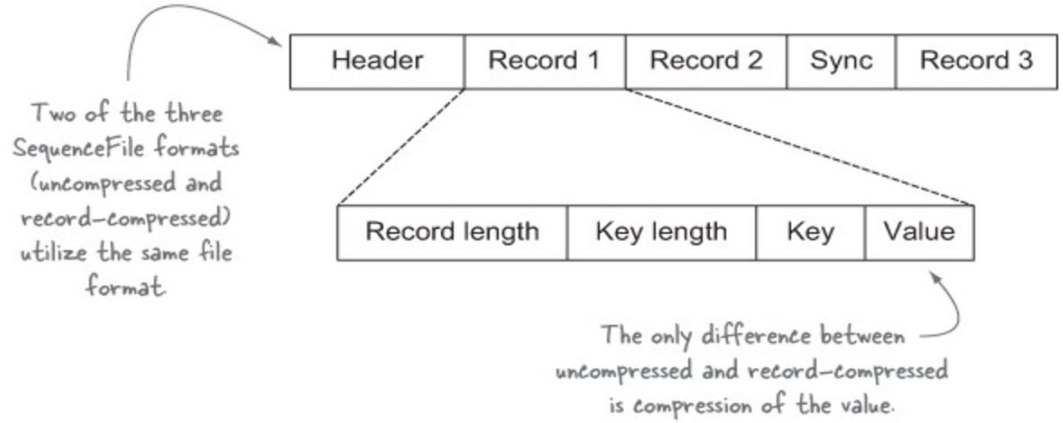
- store data as binary key-value pairs; available formats for records are:
  - uncompressed
  - record-compressed
  - block-compressed (block is a group of records compressed together, not an HDFS FS block)
    - see diagram on the next slide
- has a header with metadata:
  - compression codec used
  - key and value class names
  - user-defined metadata
  - no schema info (!!! - this means schema evolution is not supported)
- sync marker, before each block - very important, as it defines splittable points
- supports splitting even when the data inside the file is compressed


SEQUENCEFILE WITH AND WITHOUTH COMPRESSION:
SEQ + Version
Key Class Name
Value Class Name
Boolean: Is Compressed
Boolean: Is Block Compressed?
Compress Codec Class Name
Metadata
Sync (new UID() + '@' + time
Header
Record 0
Record 1
Sync
Record ...
Key Length + Value Length
Key Length
Key
Value OR Compressed Value
SEQUENCEFILE WITH BLOCK COMPRESSION:
Header
Sync
Block
Sync
Block...
Number of Records
Keys Length
Key
Value Length
Value

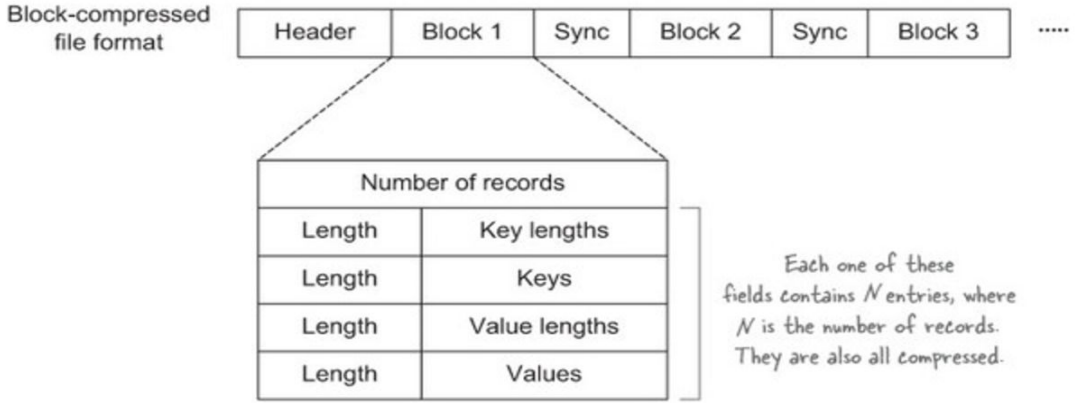
Difference between:

- Uncompressed
- Record compressed
- Block compressed

Uncompressed and Record Compression



Block Compression

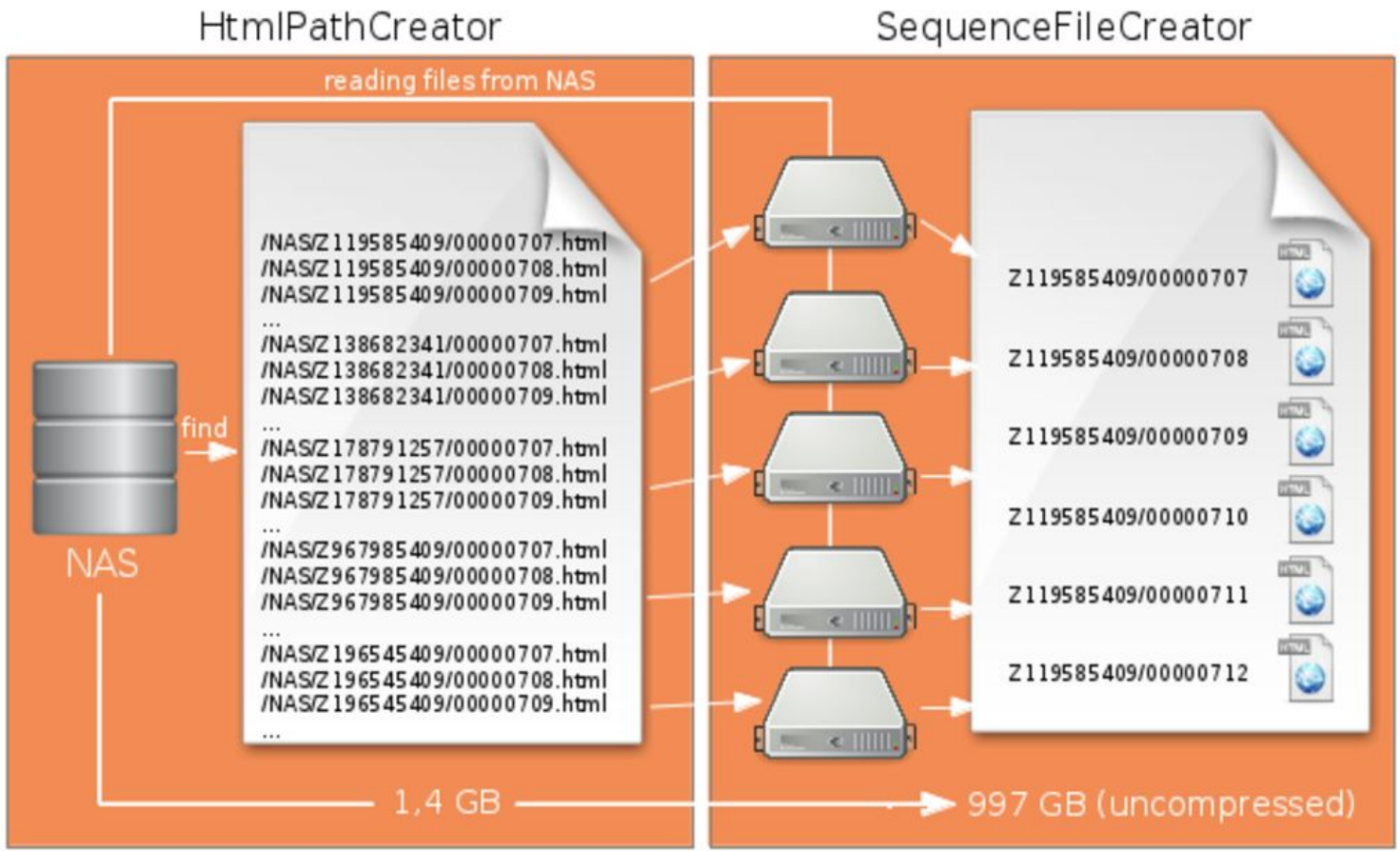


Master Dataset Formats

An interesting use case of packing small input files into big SequenceFile-formatted files for efficient MR processing:
http://openpreservation.org/blog/2012/08/07/big-data-processing-chaining-hadoop-jobs-using-taverna/

The following part is relevant:
"... dealing with lots of HTML files, means that we are facing Hadoop's "Small Files Problem". In brief, this is to say that the files we want to process are too small for taking them directly as input for the Map function. In fact, loading 1000 HTML files into HDFS – which by the way would require quite some time – in order to parse them in a Map function, would let the Hadoop JobTracker create 1000 Map tasks. Given the task creation overhead this would result in a very bad processing performance. In short, Hadoop does not like small files, but, on the contrary, the larger the better.

One approach to overcome this shortcoming is to create one large file, a so called *SequenceFile*, in a first step, and subsequently load it into the distributed file system (HDFS). This is done … as a Map function which reads HTML files directly from the file server, and stores a file identifier as 'key' and the content as *BytesWritable* 'value' (key-value-pair), as illustrated in the following figure:



@Marina Popova

Master Datasets - Serialization Frameworks

Serialization is a process of converting data structures into byte streams either **for storage or for transmission over a network**.
De-serialization - the reverse process.

The main serialization format in Hadoop is the Writables interface (for values) and **WritableComparable** (for keys). However, it is only supported in Java.
Other formats were developed to address this and other issues with Writables

Serialization Frameworks

Thrift

- developed by Facebook - to address cross-language support
- uses IDL (Interface Definition Language) to define interfaces and to generate stubs for different languages
- does not support internal compression
- not splittable
- does not support MapReduce natively (there are libs though that address that)

Protocol Buffers

- developed by Google - same goals as those of Thrift, also uses IDL
- same drawbacks

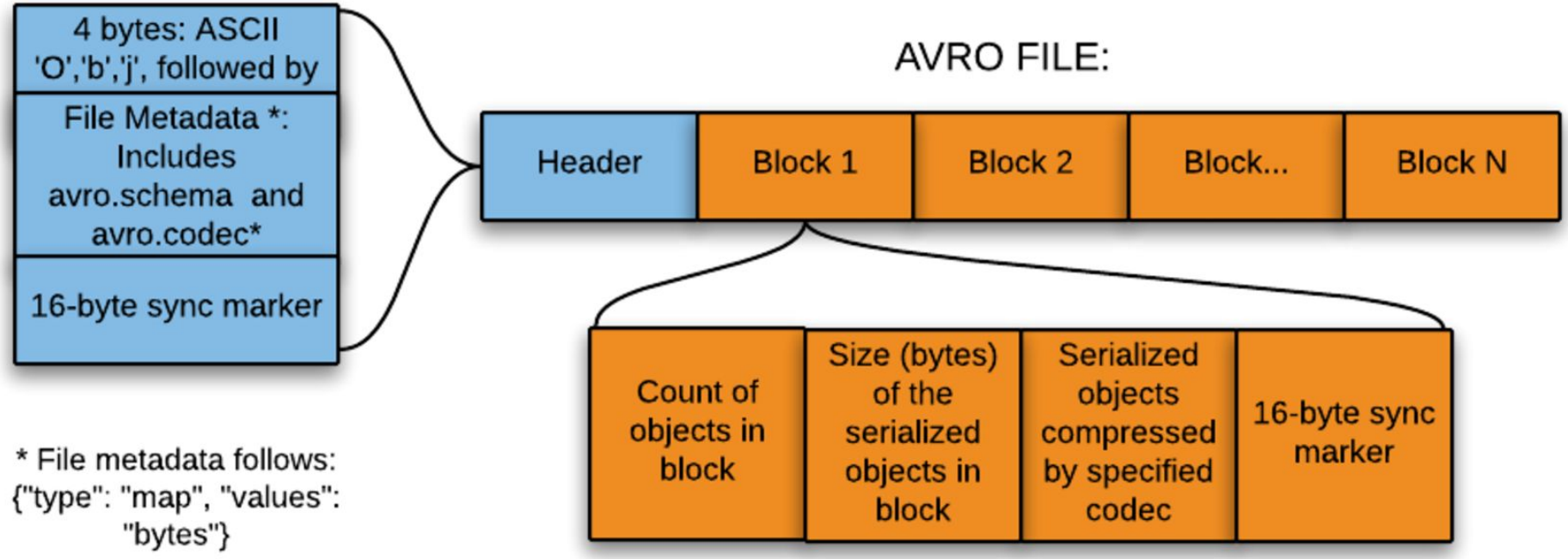
Avro as a Serialization Framework and Data format

Avro - the best choice, usually

Ref: https://avro.apache.org/docs/current/

- a language-neutral serialization framework
- uses schema to define data - either JSON or Avro IDL
- is self-describing - metadata is stored in the headers —> can easily be read from a different language at a later time
- compressible and splittable - uses sync markers
- supports schema evolution (!) - schema used to write the file does not have to match a schema used to read the file
- supports Snappy and Deflate compression
- specifies two serialization encodings: binary and JSON

AVRO File Format



AVRO Schema

Example Avro Schema (from Avro docs):

Avro schemas are defined using JSON. Schemas are composed of primitive types (null, boolean, int, long, float, double, bytes, and string) and complex types (record,enum, array, map, union, and fixed). Simple schema example, ***user.avsc*:**

```
{
    "namespace": "example.avro",
    "type": "record",
    "name": "User",
    "fields": [
        {
            "name": "name",
            "type": "string"
        },
        {
            "name": "favorite_number",
            "type": [
                "int",
                "null"
            ]
        },
        {
            "name": "favorite_color",
            "type": [
                "string",
                "null"
            ]
        }
    ]
}
```

Avro Serialization Framework

Avro provides a convenient way to represent complex data structures within a Hadoop MapReduce job. Avro data can be used as both input to and output from a MapReduce job, as well as the intermediate format.
Great post with examples:
https://blog.matthewrathbone.com/2016/09/01/a-beginners-guide-to-hadoop-storage-formats.html

For Java, the following classes can be used:

```
import org.apache.avro.mapred.AvroKey;
import org.apache.avro.mapred.AvroValue;
import org.apache.avro.mapreduce.AvroJob;
import org.apache.avro.mapreduce.AvroKeyInputFormat;
import org.apache.avro.mapreduce.AvroKeyValueOutputFormat;
```

Avro Serialization Framework

How to process and store data in Avro format?
https://avro.apache.org/docs/current/index.html

- get all dependencies
- define your schema - users.avsc file
- optionally: generate model classes,

using the commands from the tutorial:
java -jar avro-tools-1.8.2.jar compile schema user.avsc .

```
<dependency>
    <groupId>org.apache.avro</groupId>
    <artifactId>avro</artifactId>
    <version>1.8.2</version>
</dependency>
<dependency>
    <groupId>org.apache.hadoop</groupId>
    <artifactId>hadoop-client</artifactId>
    <version>2.7.3</version>
</dependency>
```

```
Y0115:avro marinapopova$ tree .
.
├── avro-tools-1.8.2.jar
├── example
│   └── avro
│       └── User.java
└── user.avsc

2 directories, 3 files
Y0115:avro marinapopova$
```

Avro Serialization Framework

Write a MR job that either uses the generated model class , examples.avro.User, to read Avro files in and process

OR do not generate model classes and use GenericRecord instead
import org.apache.avro.generic.GenericRecord;
You would want to use this type as either your mapper/reducer output type ,
or use AvroKey<GenericRecord> as input type

You would also need to load your Avro schema of type:
org.apache.avro.Schema

```
schema = new Schema.Parser().parse(new File("schemas/user.avsc"));
```

Master Datasets - File Formats

Once that is done - you can create or read values as GenericRecords

For example, you can create a record as:

```
GenericRecord myGenericRecord = new GenericData.Record(schema);
```

and use put("field_name", field_value) methods to add fields

Please read Avro APIs for more details: https://avro.apache.org/docs/current/mr.html

Master Datasets - Storage Formats - Continue

Storage Formats - Columnar vs. Row-oriented Formats

Most DB systems (especially RDBMS) store records in the row-oriented way.
This is efficient when you need to access most of the columns of the fetched rows.

Recently, some storage systems/databases started supporting columnar storage. Benefits:

- skips I/O and decompression on columns that are not part of the query
- works well for queries that access small number of columns only - for example, aggregations by specific columns over many rows
- compression within a column is more efficient (less entropy)

Master Datasets - Storage Formats

Example columnar storage formats

RCFile

- mostly used for Hive storage - not widely used
- breaks files into row splits and then uses columnar storage within each split
- not very query performant and not storage efficient

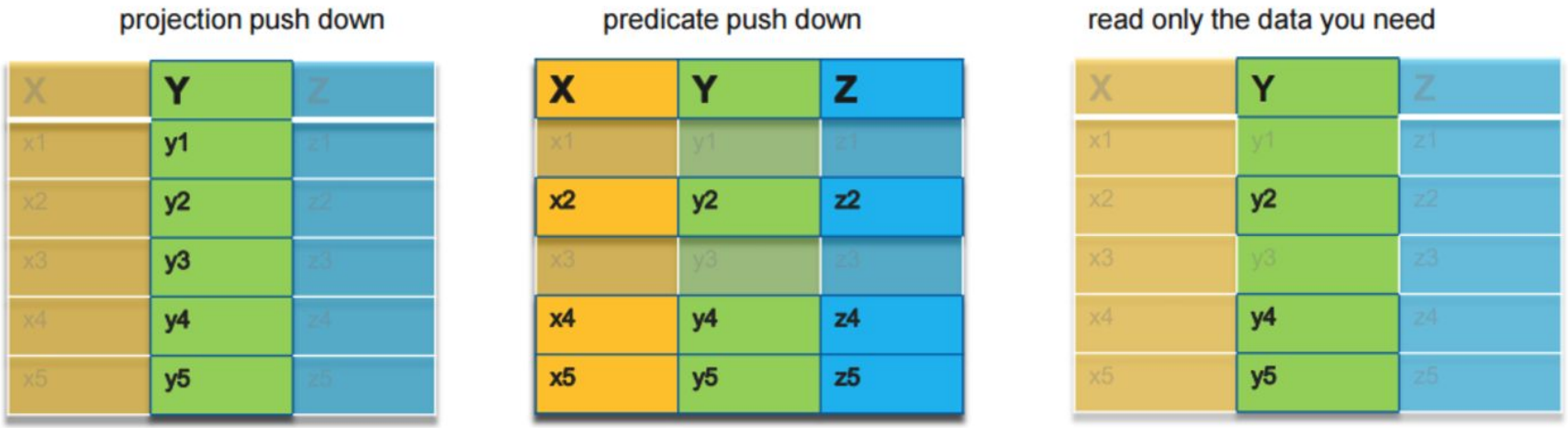
ORC

- provides light-weight , always-on compression (lib, LZO, Snappy is also supported)
- is splittable
- but mostly supported Hive type models - and as such is not a general purpose format

Master Datasets - Storage Formats

Parquet - most commonly used: [https://parquet.apache.org/documentation/latest/]

- general-purpose storage format for Hadoop - can be used by Java, Hive, Pig, Impala, Spark and others
- predicate (selection criteria) can be pushed down to the storage - so that only required fields/rows are returned



Master Datasets - Storage Formats

- efficient compression - can be specified on the per-column level
- supports complex nested data structures

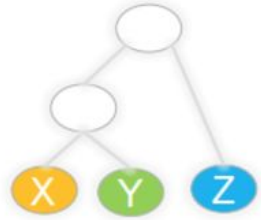
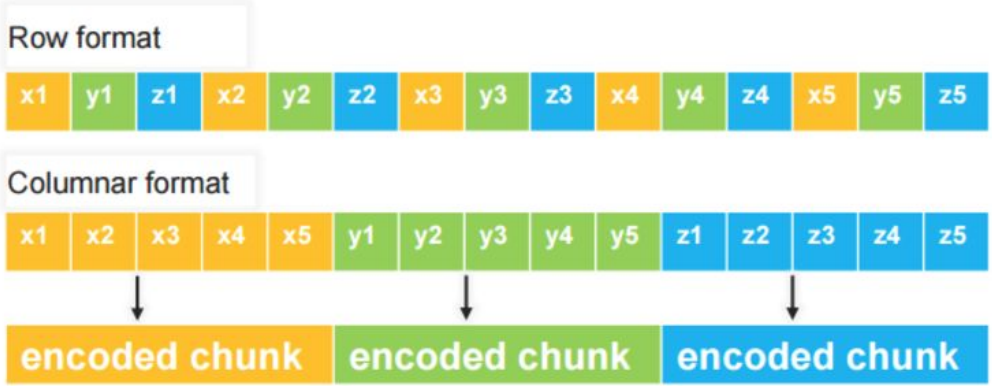


Table representation

X	Y	Z
x1	y1	z1
x2	y2	z2
x3	y3	z3
x4	y4	z4
x5	Y5	z5



Master Datasets - File Formats

Main characteristics:

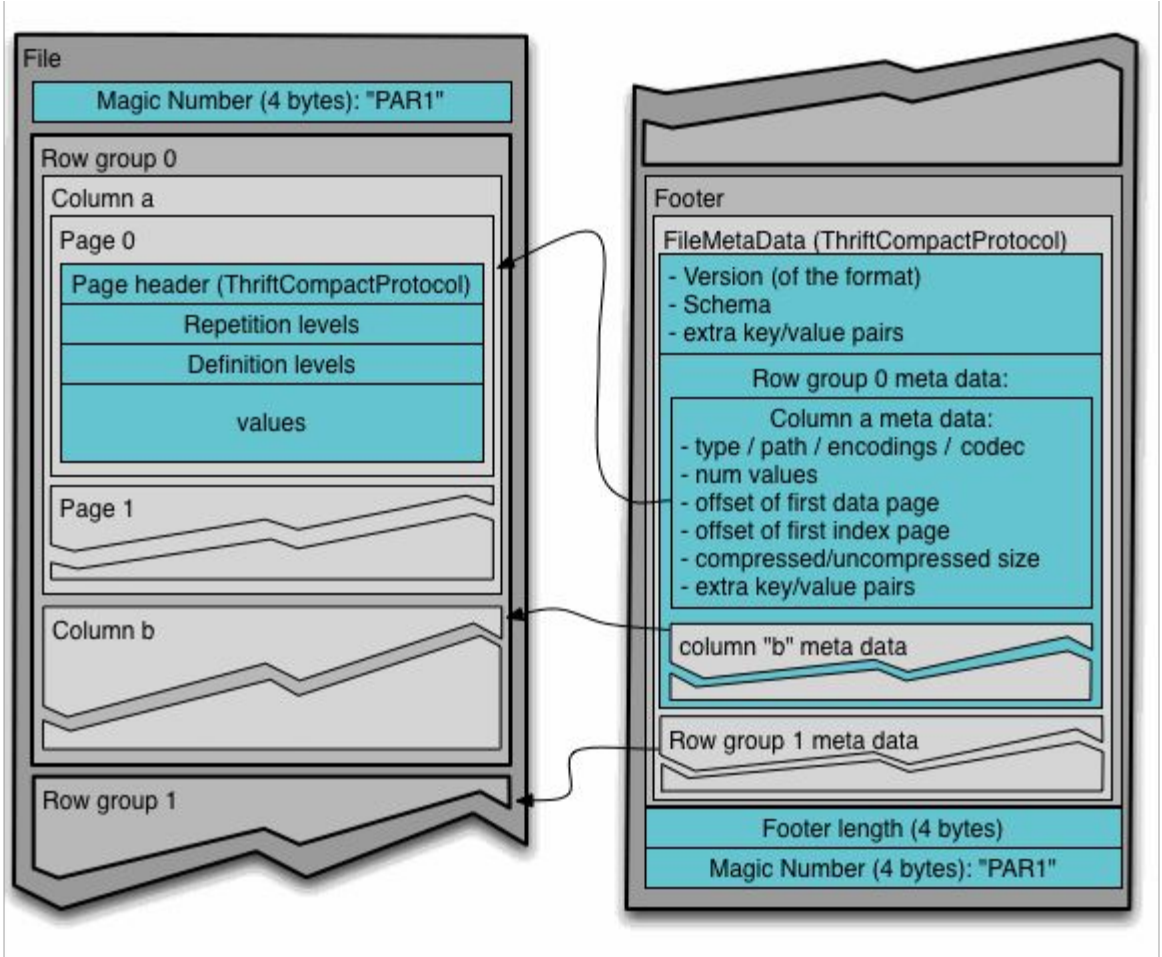
- stores metadata at the end of the file - —> self-documenting
- each data file contains the values for a set of rows (“the row group”).
- the values from each column are organized so that **they are all adjacent** → good compression
- **can be read and written with Avro (and Thrift) APIs and Avro schemas (!) - more later**

Ref: https://dzone.com/articles/understanding-how-parquet
https://techmagie.wordpress.com/category/big-data/avro-parquet/

Parquet Format

- File: A hdfs file that must include the metadata for the file
- Row group: A logical horizontal partitioning of the data into rows. A row group consists of a column chunk for each column in the dataset.
- Column chunk: A chunk of the data for a particular column. These live in a particular row group and is guaranteed to be contiguous in the file.
- Page: Column chunks are divided up into pages. A page is conceptually an indivisible unit (in terms of compression and encoding).

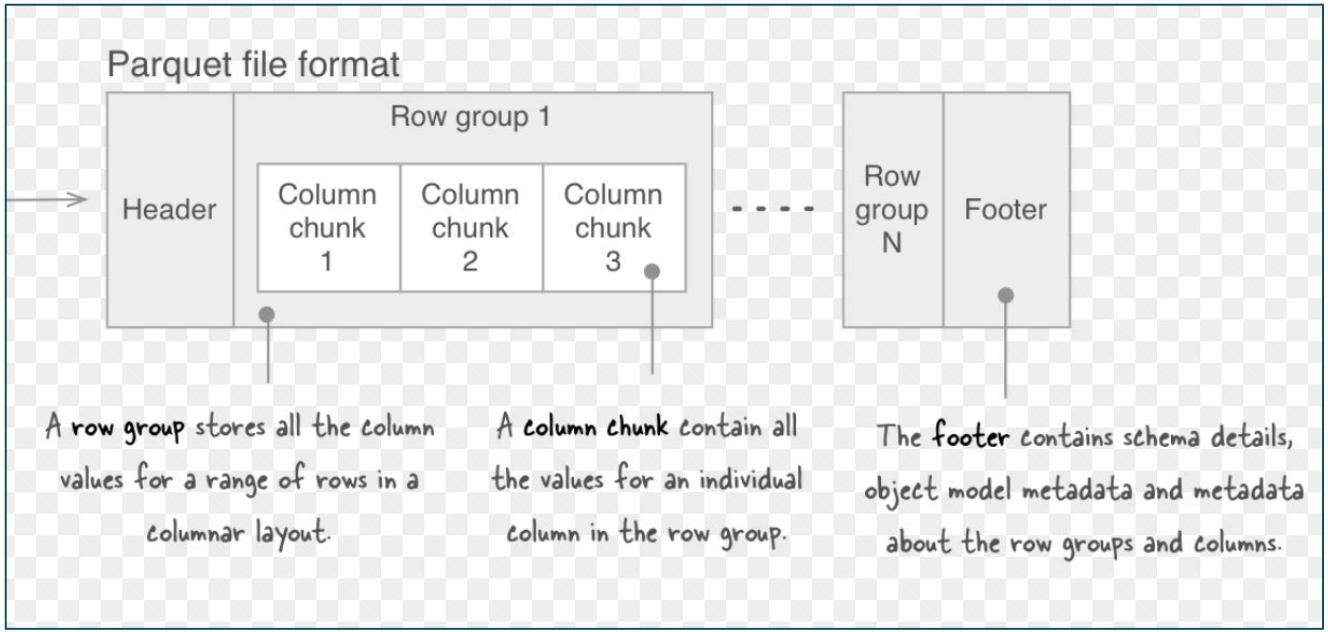
Hierarchically, a file consists of one or more row groups. A row group contains exactly one column chunk per column. Column chunks contain one or more pages.



Parquet Format

Unit of parallelization

- MapReduce - File/Row Group
- IO/ queries - Column chunk
- Encoding/Compression - Page



Master Datasets - Storage Formats

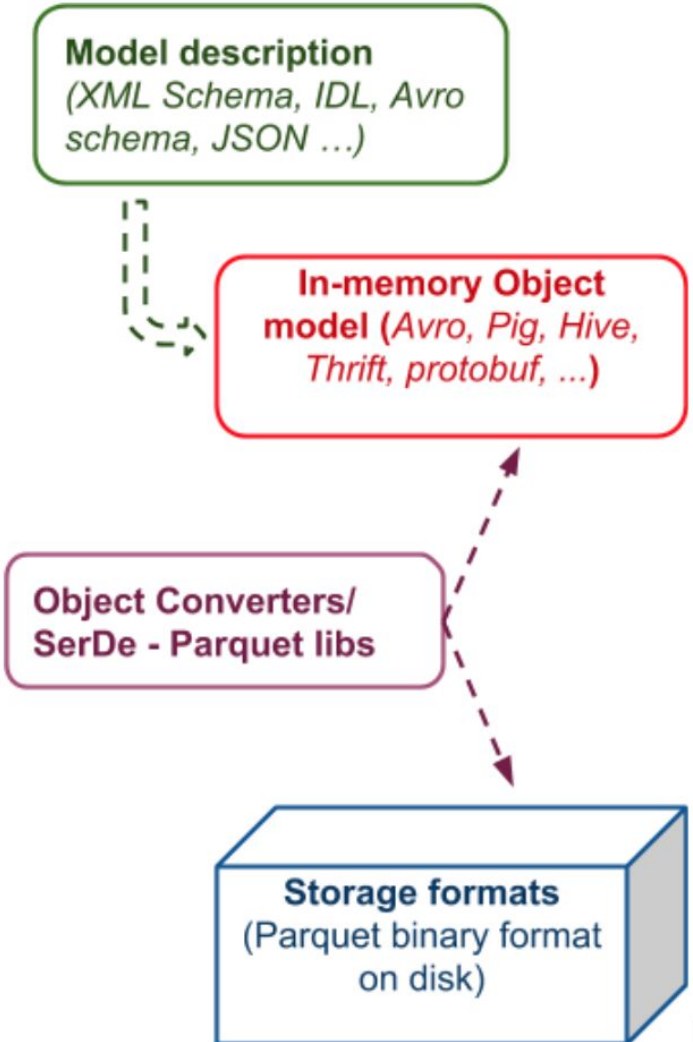
How does Parquet integrates with Avro (or other formats) ?

It supports Avro files via:

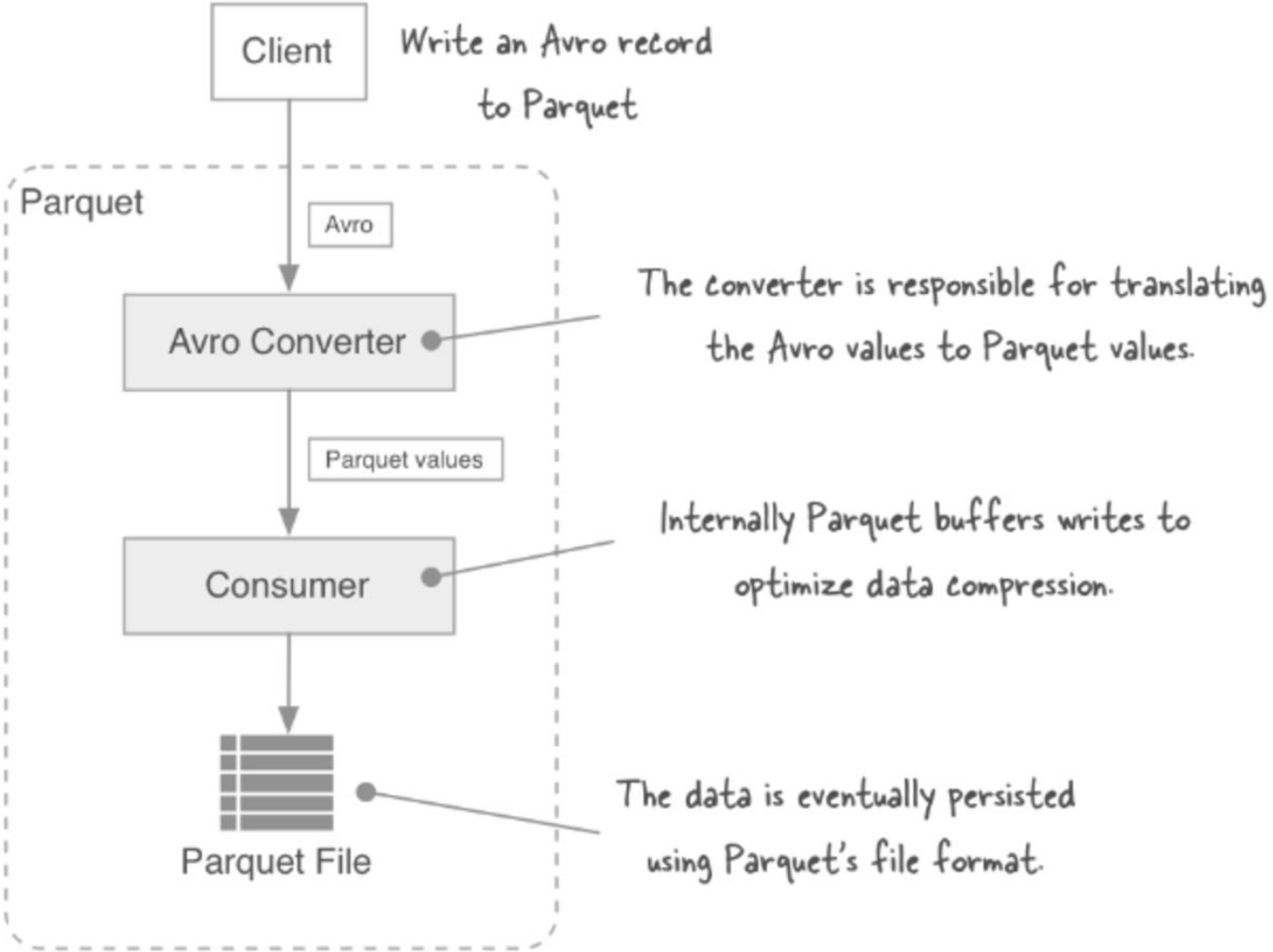
object model converters that

map an ***external object model*** to

Parquet's ***internal data types***



- ***Object model*** is in memory representation of data. In Parquet it is possible to change the Object model to Avro which gives a rich object model.
- ***Storage format*** is serialized representation of data on disk. Parquet has columnar format.
- ***Object Model converters*** are responsible for converting the Parquet's data type data into Object Model data types.


Client
Write an Avro record to Parquet
Parquet
Avro
Avro Converter
The converter is responsible for translating the Avro values to Parquet values.
Parquet values
Consumer
Internally Parquet buffers writes to optimize data compression.
Parquet File
The data is eventually persisted using Parquet's file format.

Master Datasets - Storage Formats

Conclusion: Columnar v/s Row formats OR Parquet v/s Avro

- Columnar formats are generally used where you need to query only few columns in the row
- Row formats are used where you need to access all the fields of row.

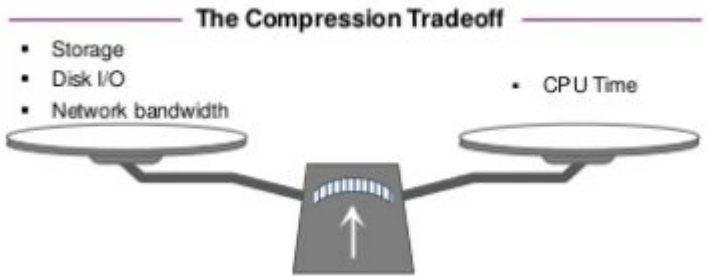
Failure behavior for different file formats:

- columnar formats - failure/corruption can cause incomplete rows
- sequence files: will be readable till the first corrupted row, but not recoverable after that

Master Datasets - Compression

Compression - very important consideration!

- reduces the amount of disk and network I/O - critical for parallel processing of large amounts of data
- the most important criteria for compression formats is whether they are splittable or not
- any compression format can be made splittable though if used with the container file formats



Master Datasets - Compression

Lets take a look at a few types of compression:

- Snappy
  - developed by Google
  - high compression speeds
  - medium compression size
  - not splittable - has to be used with container formats like SequenceFiles or Avro
- LZO
  - optimized for speed, not size
  - splittable - but this requires an additional indexing step

Master Datasets - Compression

- Gzip
  - very high compression performance
  - medium write performance
  - not splittable - has to be used with container formats like SequenceFiles or Avro
- bzip2
  - very high compression performance
  - low processing performance - makes it much less suitable for Hadoop processing
  - is splittable

Master Datasets: summary

An interesting performance overview of some serialization formats and compression was published in this article:
https://db-blog.web.cern.ch/blog/zbigniew-baranowski/2017-01-performance-comparison-different-file-formats-and-storage-engines

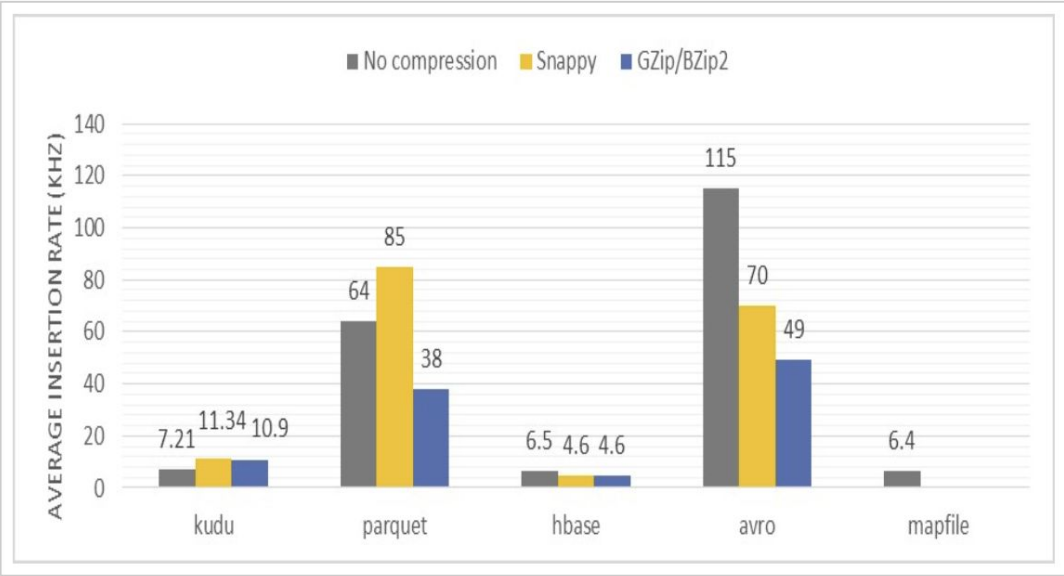


Figure reports on the average ingestion speed (10^3 records/s) per data partition for each tested format and compression type

@Marina Popova

Master Datasets: summary

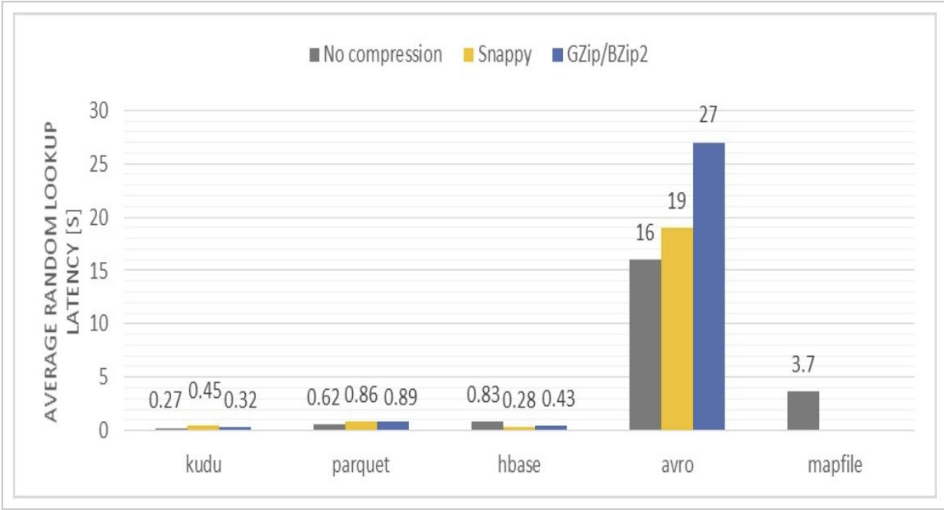


Figure reports on the average random record lookup latency [in seconds] for each tested format and compression type

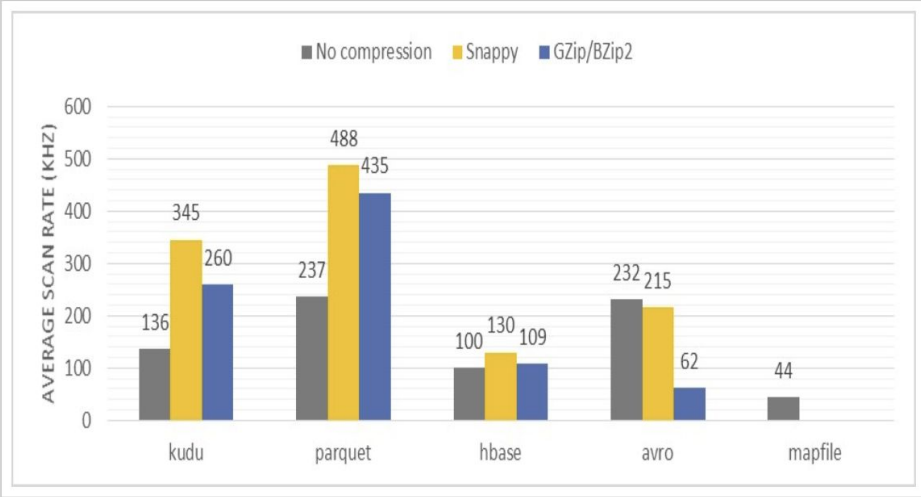
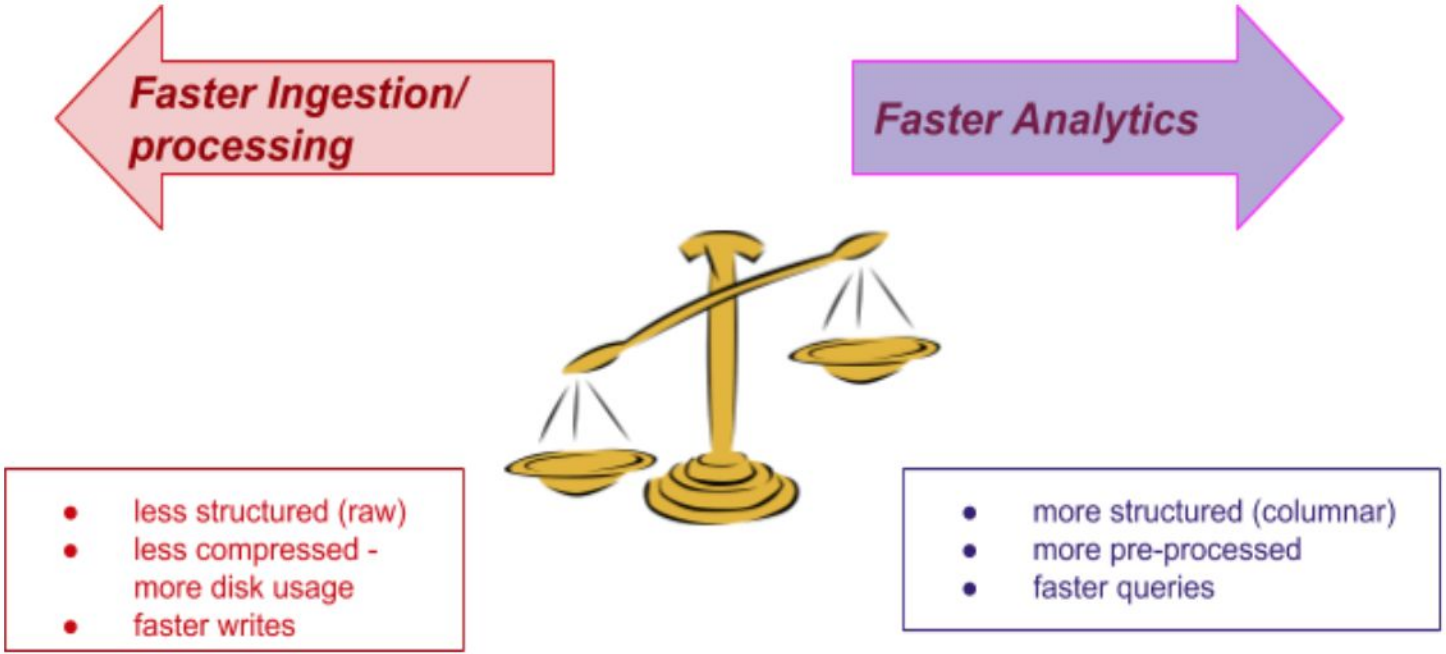


Figure reports on the average scans speed with the same predicate per core [in k records/s] for each tested format and compression type

@Marina Popova

Master Datasets - Conclusions

Good ref: https://www.svds.com/tbt-choose-data-format/



@Marina Popova

Master Datasets - Conclusions

Ref: *https://databaseline.bitbucket.io/an-overview-of-file-and-serialization-formats-in-hadoop/*

		Writables	Thrift	Protocol Buffers	Avro	RCFile	ORC	Parquet
Origin		Hadoop	Facebook	Google	Hadoop	Facebook et al.	Hortonworks	Twitter Cloudera
Rationale			Language-independent interfaces (IDL) to services	Data exchange between services through IDL	Portable replacement for Writables	Columnar replacement for SequenceFiles	Improved RCFile	Similar to ORC but portable
Applicability		Java	General	General	General	Hive	Hive	General
Storage		Row	Row	Row	Row	Columnar	Columnar	Columnar
Native MR support		Yes	No	No	Yes	Yes	Yes	Yes
Compressible		Yes	No	No	Yes	Yes	Yes	Yes
Splittable		Yes	No	No	Yes	Yes	Yes	Yes
Schema evolution		No	Yes	Yes	Yes	No	No	Yes**
Self-describing		No	No	No	Yes	Yes	Yes	Yes
Block size (MB)		-	-	-	64	4	256	128/256***
Supported data types	Simple	Null	✓		✓	✓	✓	✓	✓
		Boolean	✓	✓	✓	✓	✓	✓	✓
	Numerical	Integer (8 bits): byte	✓	✓		✓	✓	✓	✓*
		Integer (16 bits): short	✓	✓	✓*		✓	✓	✓*
		Integer (32 bits): int	✓	✓	✓*	✓	✓	✓	✓*
		Integer (64 bits): long	✓	✓	✓*	✓	✓	✓	✓*
		Float (32 bits): float	✓		✓*	✓	✓	✓	✓
		Float (64 bits): double	✓	✓	✓*	✓	✓	✓	✓
	Text	String	✓	✓	✓	✓	✓	✓	✓
		Character					✓	✓	
	Time	Date					✓	✓	✓
		Timestamp					✓	✓	✓
		Interval							✓
	Binary			✓					✓
	Collections	Array	✓			✓	✓	✓	
		List		✓				✓	✓
		Set		✓					
		Map	✓	✓	✓	✓	✓	✓	✓
		SortedMap	✓						

What we did:

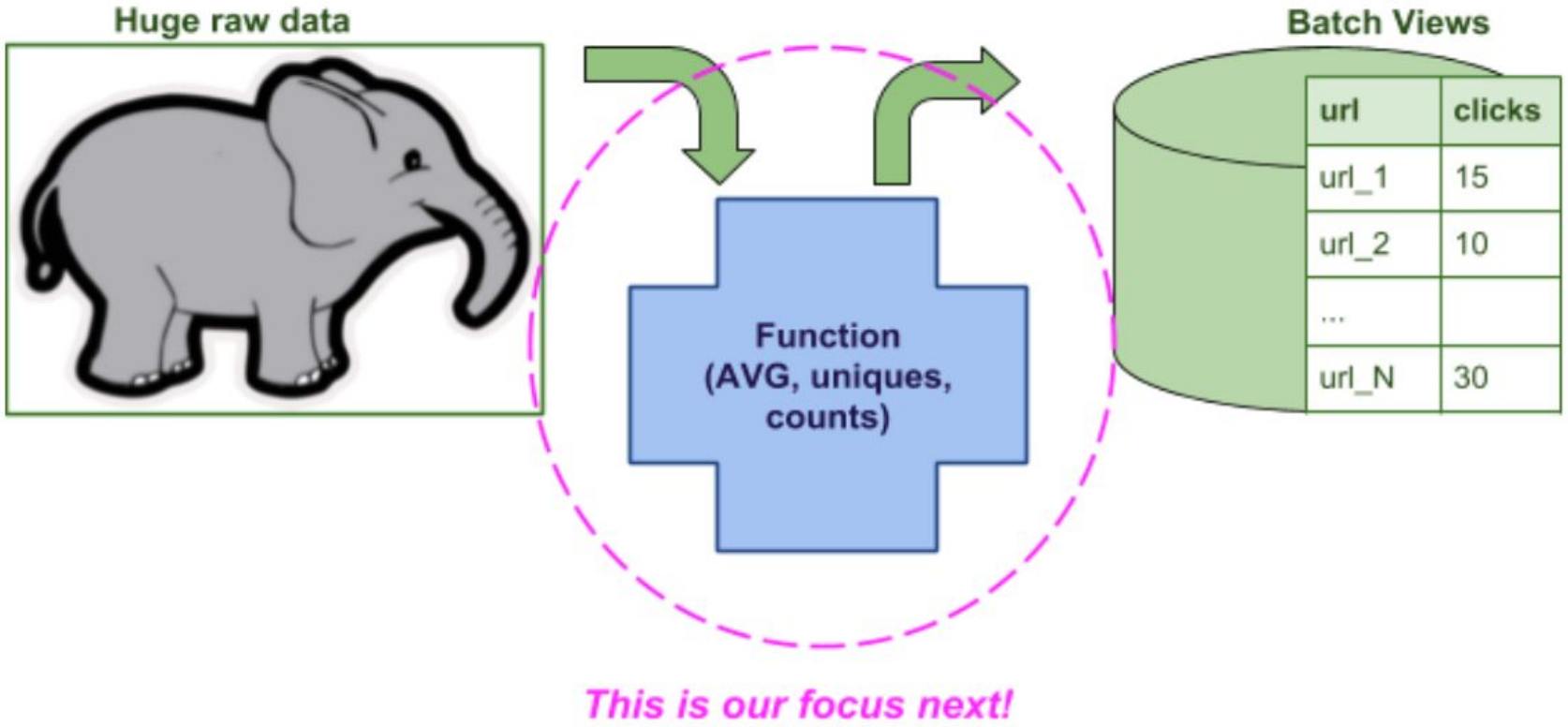
- ✓ What to store (content and model) ?
- ✓ Where to store to (the actual data storage systems) ?
- ✓ How to store - storage formats and serialization ?
- ○ How to store - algorithms and operations ?
  - ○ we will discuss this later on, in the Collection Tier lecture

What we do next:

back to having fun ...
... with Batch Processing !

@Marina Popova

Moving on ….


Huge raw data
Function
(AVG, uniques,
counts)
Batch Views
url
clicks
url_1
15
url_2
10
...
url_N
30
This is our focus next!

Batch Processing

How do you compute Batch Views?

We already know the most important principles and techniques to do processing of large amounts of data:

- parallelized operations on chunks of data
- Data immutability
- Idempotent operations
- No shared state

And the most important requirements for this kind of processing:

- fault-tolerance (nodes go down, etc.)
- horizontal scalability under increasing load
- zero data loss
- ability to process all historical data - or a partition of it
    - Importance of data partitioning again!

Batch Processing

We already learned one of the most prevalent techniques to conduct parallel operations on such large scale:
Map-Reduce programming model

We also reviewed one specific framework that implements this model:
Hadoop MR

We will consider another example framework that implements the same MapReduce paradigm - Spark

So, what is wrong with HadoopMR and why do we need to look at other frameworks?
While Apache Hadoop has become the most popular open-source framework for large-scale data storing and processing based on the MapReduce model, it has some important limitations:

- Intensive disk-usage
- Low inter-communication capability
- Inadequacy for in-memory computation
- Poor performance for online and iterative computing

Spark

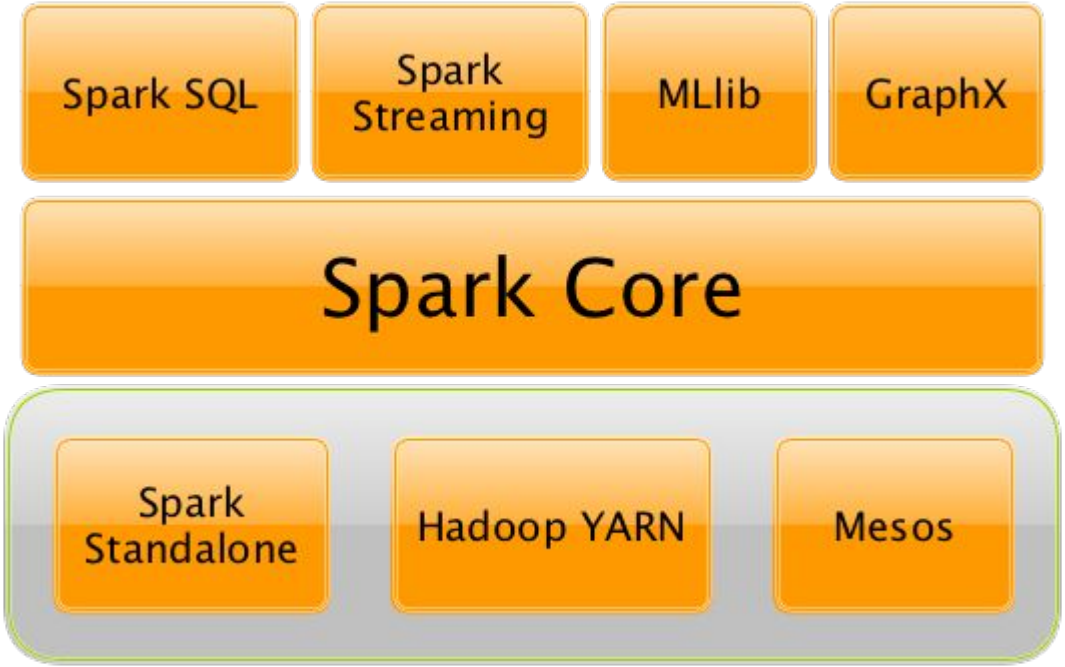
What is Spark?

Ref: https://bdataanalytics.biomedcentral.com/articles/10.1186/s41044-016-0020-2

- Apache Spark is a framework aimed at performing fast distributed computing on Big Data by using **in-memory primitives**.
- It allows user programs to load data into memory and query it repeatedly, making it a well suited tool for online and iterative processing (especially for ML algorithms)
- It was motivated by the limitations in the MapReduce/Hadoop paradigm which forces to follow a linear dataflow that make an intensive disk-usage.

Spark

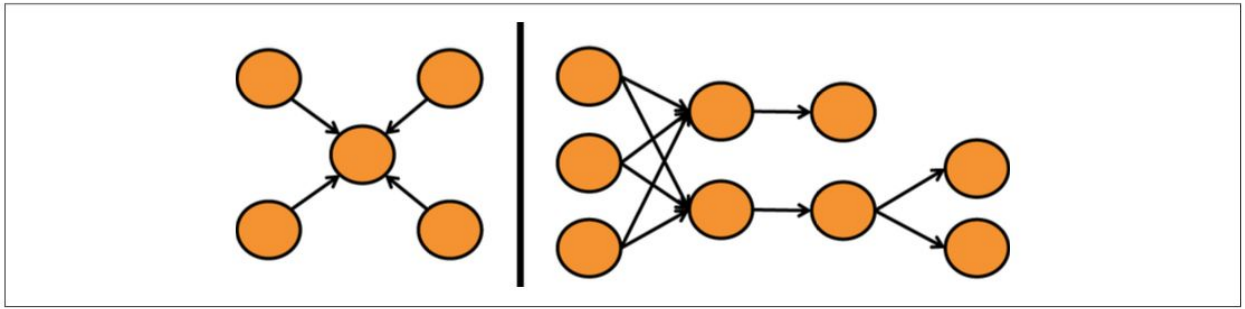
Spark Platform overview: https://jaceklaskowski.gitbooks.io/mastering-apache-spark/content/spark-overview.html



@Marina Popova

Spark

- MapReduce programming model only has two phases: map and/or reduce.
- Complex applications and data flows can be implemented by chaining these phases
- This chaining forms a **'graph' of operations** - which is known as a **"directed acyclic graphs", or DAGs**
- DAGs contain series of actions connected to each other in a workflow
- In the case of MapReduce, the DAG is a series of map and reduce tasks used to implement the application - and it is the developer's job to define each task and chain them together.



Spark

Main differences between Hadoop MR and Spark:

- With Spark, the engine itself creates those complex chains of steps from the application's logic. This allows developers to express complex algorithms and data processing pipelines within the same job and allows the framework to optimize the job as a whole, leading to improved performance.
- Memory-based computations

Common features:

- Data locality
- Staged execution (stages separated by shuffle phases)
- Reliance on distributed file system for on-disk persistence (HDFS)

Spark

Spark Core: How Spark Works

References:

https://www.slideshare.net/SigmoidHR/spark-and-spark-streaming-internals

https://www.slideshare.net/michiard/introduction-to-spark-internals

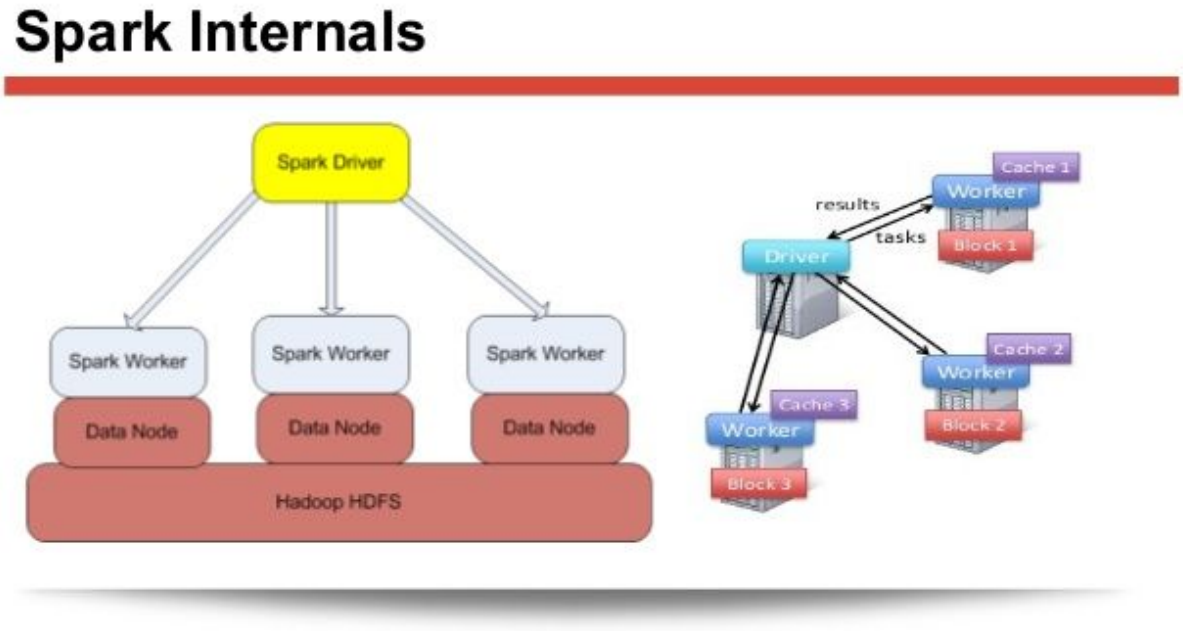
Very good reference for internal Spark workings:

http://datastrophic.io/core-concepts-architecture-and-internals-of-apache-spark/

"Hadoop Applications Architecture", by Mark Grover, Ted Malaska

Spark

- Similar to Hadoop MR Master-Slave architecture
- relying on HDFS for scalable and reliable on-disk persistence



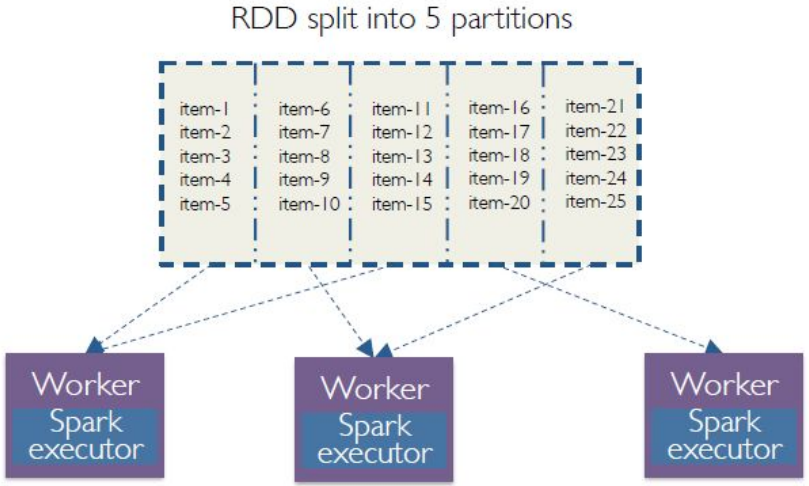
Spark

RDD: Resilient Distributed Dataset

- Spark is based on distributed data structures called Resilient Distributed Datasets (RDDs) which can be thought of as **immutable parallel data structures**
- It can persist intermediate results into memory or disk for re-usability purposes, and customize the partitioning to optimize data placement.
- RDDs are also fault-tolerant by nature. RDD stores information about its parents to optimize execution (via pipelining of operations) and recompute partition in case of failure
- RDD provides API for various transformations and materializations of data
- There are two ways to create RDDs: parallelizing an existing collection in your driver program, or referencing a dataset in an external storage system, such as a shared filesystem, HDFS, HBase, or **any data source offering a Hadoop InputFormat.**

RDDs - Partitions:

- RDDs are designed to contain huge amounts of data , that cannot fit onto one single machine → hence, the data has to be partitioned across multiple machines/nodes
- Spark automatically partitions RDDs and distributes the partitions across different nodes
- A partition in spark is an atomic chunk of data stored on a node in the cluster
- Partitions are basic units of parallelism in Apache Spark
- RDDs in Apache Spark are collection of partitions
- Operations on RDDs automatically place tasks into partitions, maintaining the locality of persisted data



Ref: https://medium.com/@thejasbabu/spark-under-the-hood-partition-d386aaaa26b7

@Marina Popova

Spark

Internally, each RDD is characterized by the following main properties which are defined on the RDD interface:

- a list of partitions
- a list of dependencies on other RDDs
- a function for computing each split
- optionally, a list of preferred locations to compute each split on (e.g. block locations for an HDFS file)
- optionally, a Partitioner for key-value RDDs (e.g. to say that the RDD is hash-partitioned)

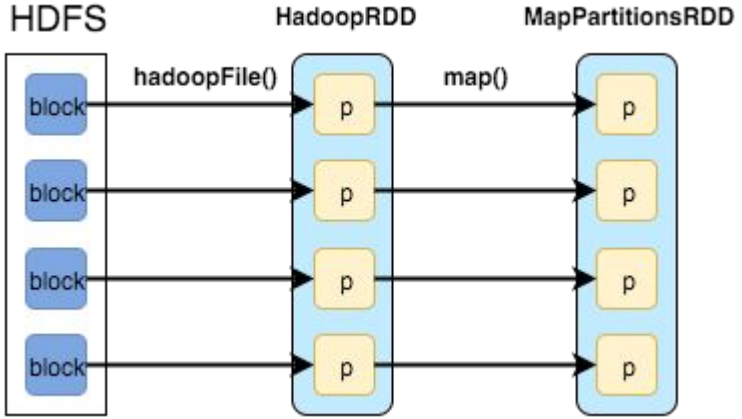
```
//a list of partitions (e.g. splits in Hadoop)
def getPartitions: Array[Partition]

//a list of dependencies on other RDDs
def getDependencies: Seq[Dependency[_]]

//a function for computing each split
def compute(split: Partition, context: TaskContext): Iterator[T]

//(optional) a list of preferred locations to compute each split on
def getPreferredLocations(split: Partition): Seq[String] = Nil

//(optional) a partitioner for key-value RDDs
val partitioner: Option[Partitioner] = None
```

Spark

Here's an example of RDDs created during a call of method

sparkContext.textFile("hdfs://...")

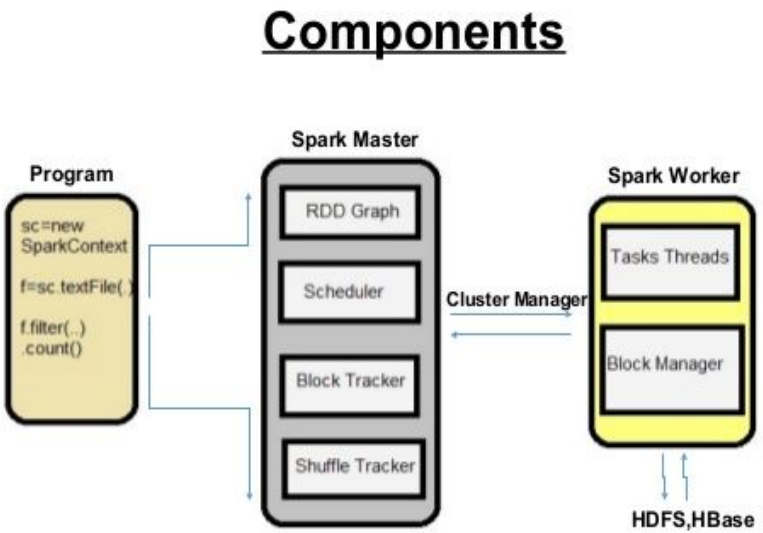
which first loads HDFS blocks in memory and then applies map() function to filter out keys creating two RDDs:



- HadoopRDD:
  - getPartitions = HDFS blocks
  - getDependencies = None
  - compute = load block in memory
  - getPrefferedLocations = HDFS block locations
  - partitioner = None

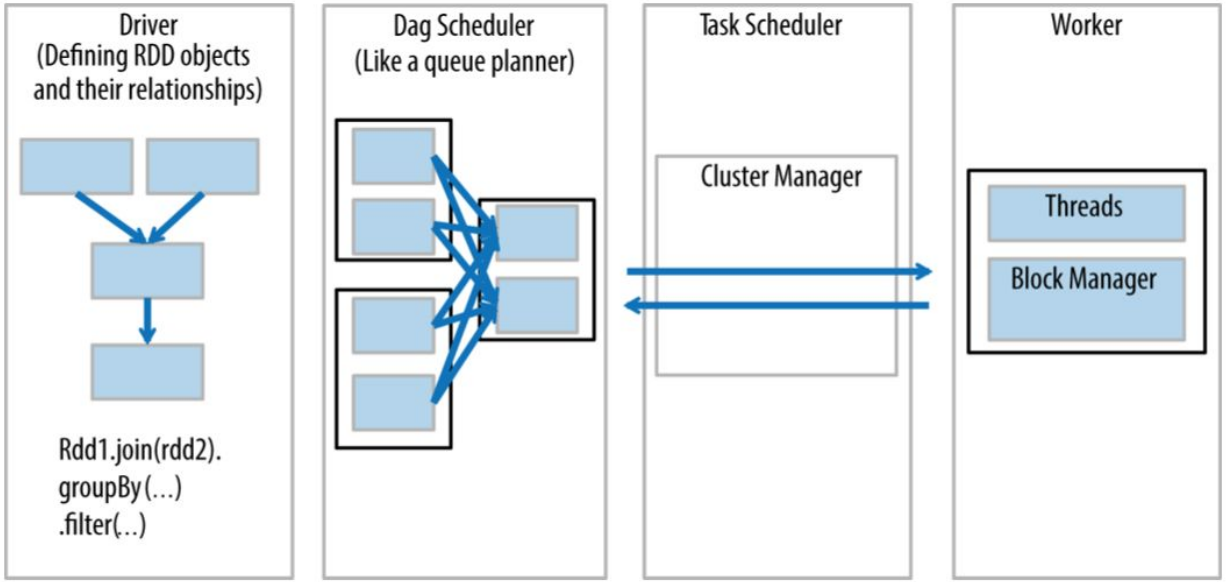
- MapPartitionsRDD
  - getPartitions = same as parent
  - getDependencies = parent RDD
  - compute = compute parent and apply map()
  - getPrefferedLocations = same as parent
  - partitioner = None

Spark - job architecture



@Marina Popova

Spark



- The **Driver** is the code that includes the “main” function and defines the RDDs
- Parallel operations on the RDDs are sent to the **DAG scheduler**, which will optimize the code and arrive at an efficient DAG that represents the data processing steps in the application.

Spark

- The resulting DAG is sent to the **ClusterManager**. The cluster manager has information about the workers, assigned threads, and location of data blocks and is responsible for assigning specific processing tasks to workers.
- The cluster manager is also the service that handles DAG play-back in the case of worker failure

- **Executors/Workers**:
  - run tasks scheduled by driver
  - executes its specific task without knowledge of the entire DAG
  - store computation results in memory, on disk or off-heap
  - interact with storage systems
  - send its results back to the Driver application

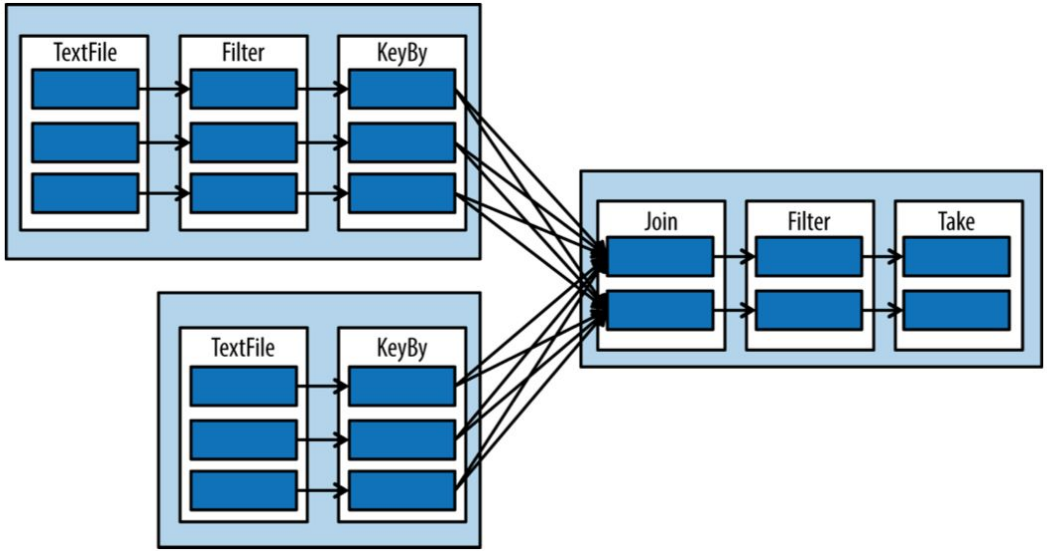
Spark - Fault Tolerance

RDDs store their **lineage**—the set of transformations that was used to create the current state, starting from the first input format that was used to create the RDD.

If the data is lost, Spark will replay the lineage to rebuild the lost RDDs so the job can continue.

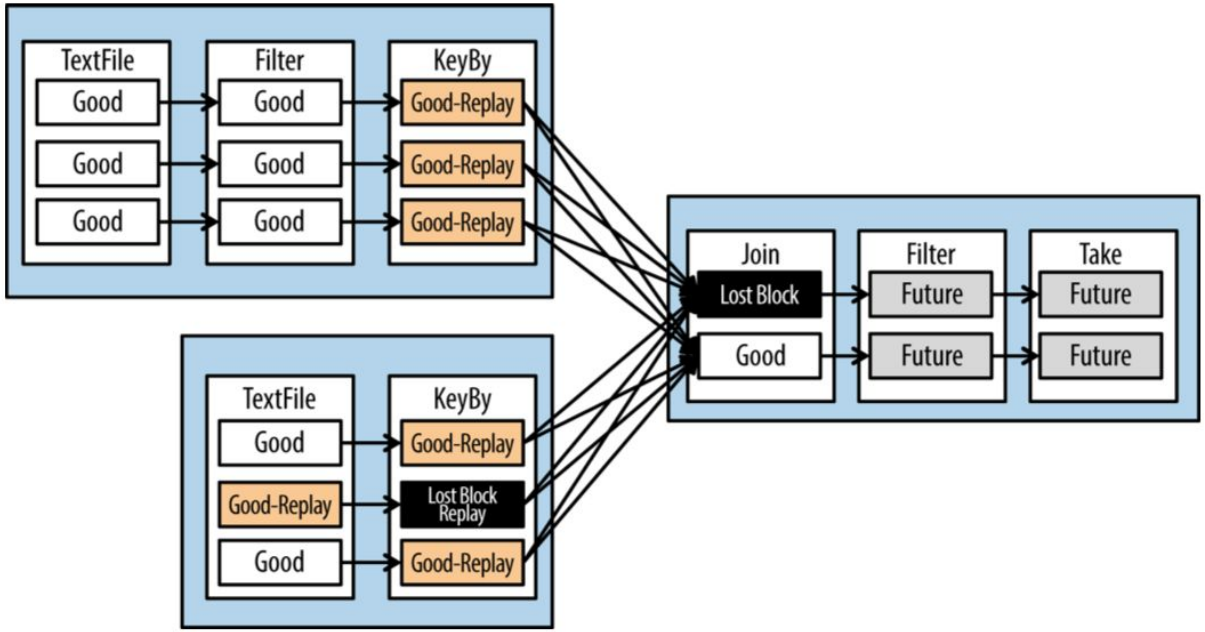
Lets see how this works:

The inner boxes are RDD partitions; the next layer is an RDD and single chained operation



Spark

Now let’s say we lose the partition denoted by the black box.
Spark would replay the “Good Replay” boxes and the “Lost Block” boxes to get the data needed to execute the final step



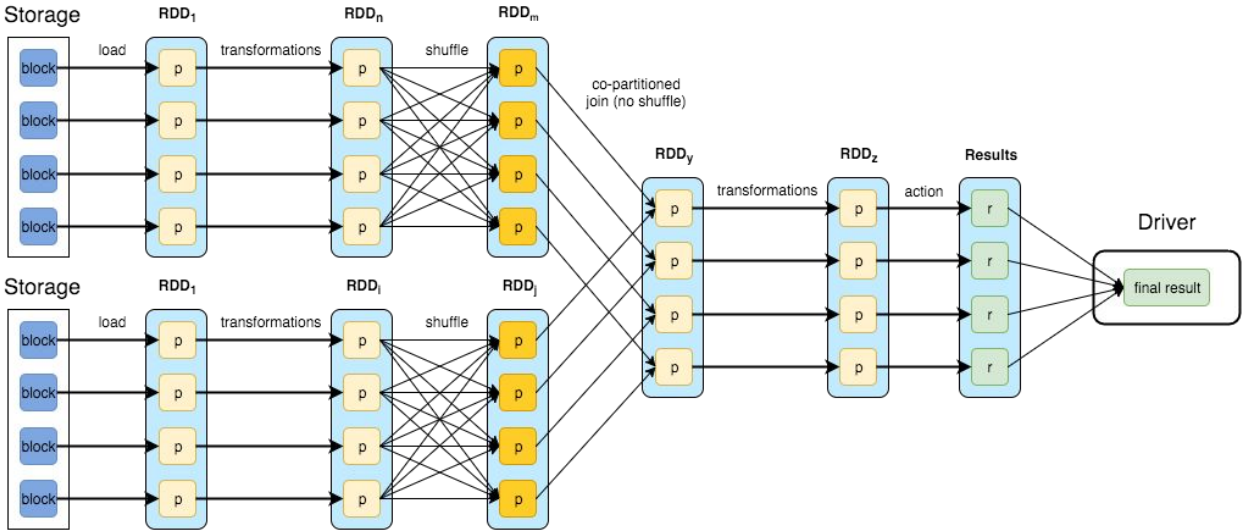
Spark

Execution Workflow details

- client/user defines the RDD transformations and actions for the input data
- DAGScheduler will form the most optimal Direct Acyclic Graph which is then split into stages of tasks.
- **Stages combine tasks which don’t require shuffling/repartitioning if the data**
- Tasks are then run on workers and results are returned to the client

Lets take a look at how stages are determined by looking at an example of a more complex job's DAG

Spark



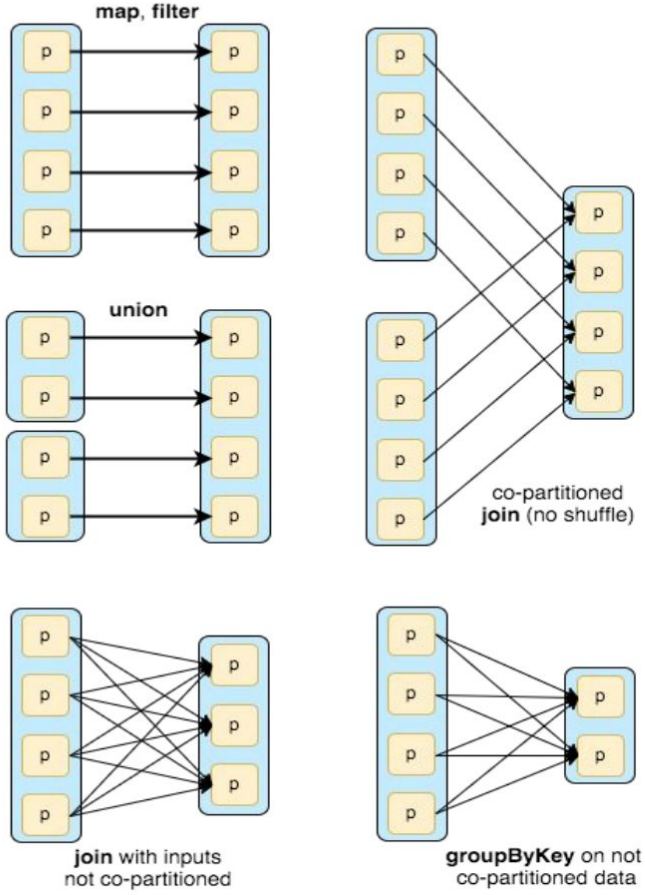
Transformations create dependencies between RDDs

There are two types of dependencies, **"narrow"** and **"wide"**

Ref: http://datastrophic.io/core-concepts-architecture-and-internals-of-apache-spark/

@Marina Popova

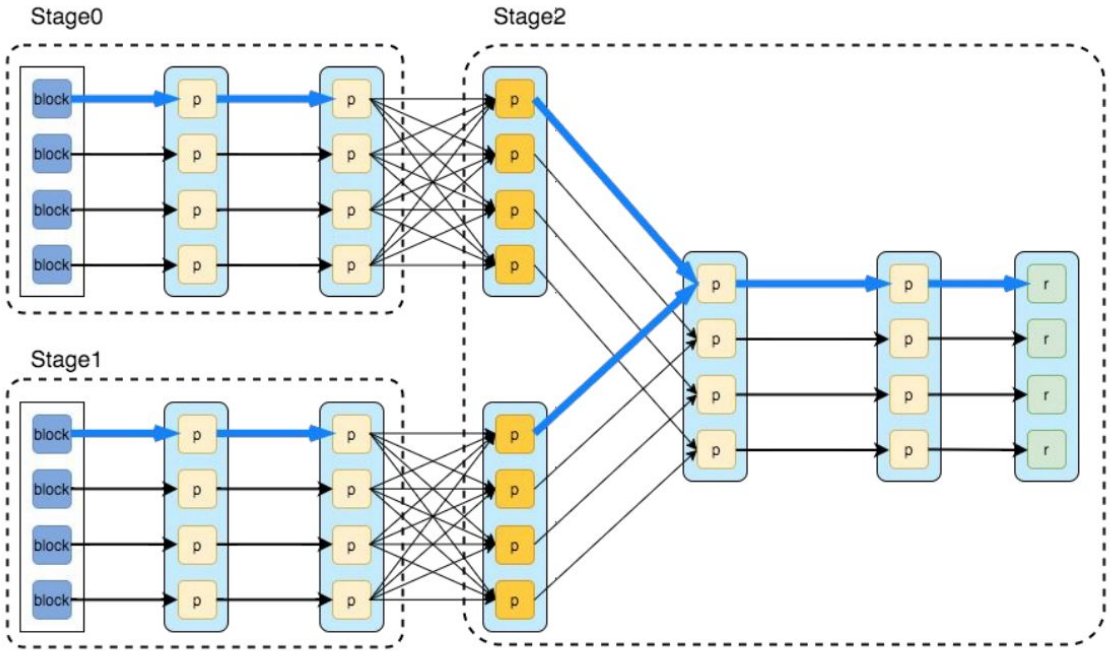
Dependency types



- Narrow (pipelineable)
  - each partition of the parent RDD is used by at most one partition of the child RDD
  - allow for pipelined execution on one cluster node
  - failure recovery is more efficient as only lost parent partitions need to be recomputed

- Wide (shuffle)
  - multiple child partitions may depend on one parent partition
  - require data from all parent partitions to be available and to be shuffled across the nodes
  - if some partition is lost from all the ancestors a complete recomputation is needed

Stages and Tasks



- Stages breakdown strategy
  - check backwards from final RDD
  - add each “narrow” dependency to the current stage
  - create new stage when there’s a shuffle dependency
- Tasks
  - *ShuffleMapTask* partitions its input for shuffle
  - *ResultTask* sends its output to the driver

Spark

Summary or the staging strategies:

- RDD operations with "narrow" dependencies, like map() and filter(), are pipelined together into one set of tasks in each stage
- operations with "wide" /shuffle dependencies require multiple stages (one to write a set of map output files, and another to read those files after a barrier).
- In the end, every stage will have only shuffle dependencies on other stages, and may compute multiple operations inside it

Spark

Spark job details

Shared variables

- Spark includes two types of variables that allow sharing information between the execution nodes:
  - ***broadcast*** variables and ***accumulator*** variables.
- **Broadcast variables** are sent to all the remote execution nodes, where they can be used for data processing.
- This is similar to the role that Configuration objects play in MapReduce.
- **Accumulators** are also sent to the remote execution nodes, but unlike broadcast variables, they can be modified by the executors, with the limitation that you only add to the accumulator variables.
- Accumulators are somewhat similar to MapReduce counters.

SparkContext

- SparkContext is an object that represents the connection to a Spark cluster.
- It is used to create RDDs, broadcast data, and initialize accumulators.

Spark

Transformations

- Transformations are functions that take one RDD and return another
- RDDs are immutable, so transformations will never modify their input, only return the modified RDD.
- Transformations in Spark are always lazy, so they don’t compute their results. Instead, calling a transformation function only creates a new RDD with this specific transformation as part of its lineage.
- The complete set of transformations is only executed when an action is called

Spark

Most common transformations

map()
Applies a function on every element of an RDD to produce a new RDD. This is similar to the way the MapReduce map() method is applied to every element in the input data. For example: lines.map(s=>s.length) takes an RDD of Strings (“lines”) and returns an RDD with the length of the strings.

filter() - Takes a Boolean function as a parameter, executes this function on every element of the RDD, and returns a new RDD containing only the elements for which the function returned true. For example, lines.filter(s=>(s.length>50)) returns an RDD containing only the lines with more than 50 characters.

keyBy()
Takes every element in an RDD and turns it into a key-value pair in a new RDD. For example, lines.keyBy(s=>s.length) return, an RDD of key-value pairs with the length of the line as the key, and the line as the value.

Spark

join()
Joins two key-value RDDs by their keys. For example, let's assume we have two RDDs: lines and more_lines. Each entry in both RDDs contains the line length as the key and the line as the value. lines.join(more_lines) will return for each line length a pair of Strings, one from the lines RDD and one from the more_lines RDD. Each resulting element looks like <length,<line,more_line>>.

groupByKey()
Performs a group-by operation on a RDD by the keys. For example: lines.group ByKey() will return an RDD where each element has a length as the key and a collection of lines with that length as the value

sort()
Performs a sort on an RDD and returns a sorted RDD.

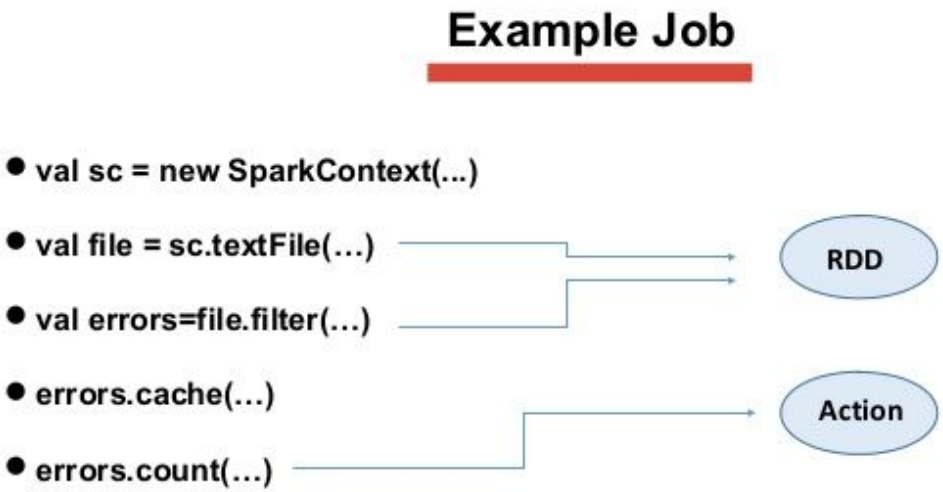
Note that transformations include functions that are similar to those that MapReduce would perform in the map phase, but also some functions, such as groupByKey(), that belong to the reduce phase.

Spark

Actions

- *Actions* are methods that take an RDD, perform a computation, and return the result to the driver application.
- Actions trigger the computation of transformations.
- The result of the computation can be a collection, values printed to the screen, values saved to file, or similar.
- However, an action will never return an RDD.

Spark


Example Job
val sc = new SparkContext(...)
val file = sc.textFile(…)
val errors=file.filter(…)
errors.cache(…)
errors.count(…)
RDD
Action

Spark SQL

Good Ref: https://jaceklaskowski.gitbooks.io/mastering-spark-sql/spark-sql.html

Spark SQL is all about distributed in-memory computations on structured data on massive scale.

The primary difference between the computation models of Spark SQL and Spark Core is the relational framework for ingesting, querying and persisting (semi)structured data using **relational queries** that can be expressed using a high-level SQL-like APIs: Dataset API

Spark SQL's `Dataset` API describes a distributed computation that will eventually be converted to a DAG of RDDs for execution.

Under the covers, structured queries are automatically compiled into corresponding RDD operations.

Spark SQL supports two "modes" to write structured queries: Dataset API and SQL.

An older interface: DataFrame - the only option for Python

Spark SQL

Examples: https://spark.apache.org/docs/latest/sql-programming-guide.html#running-sql-queries-programmatically

```
peopleDF = spark.read.json("examples/src/main/resources/people.json")

# DataFrames can be saved as Parquet files, maintaining the schema information.
peopleDF.write.parquet("people.parquet")

# Read in the Parquet file created above.
# Parquet files are self-describing so the schema is preserved.
# The result of loading a parquet file is also a DataFrame.
parquetFile = spark.read.parquet("people.parquet")

# Parquet files can also be used to create a temporary view and then used in SQL statements.
parquetFile.createOrReplaceTempView("parquetFile")
teenagers = spark.sql("SELECT name FROM parquetFile WHERE age >= 13 AND age <= 19")
teenagers.show()
# +------+
# |  name|
# +------+
# |Justin|
# +------+
```

Spark SQL and Parquet

SparkSQL can take direct advantage of the Parquet columnar format in a few important ways: (Ref: https://db-blog.web.cern.ch/blog/luca-canali/2017-06-diving-spark-and-parquet-workloads-example#comment-35274)

- **Partition pruning:** read data only from a list of partitions, based on a filter on the partitioning key, skipping the rest
- **Column projection:** read the data for columns that the query needs to process and skip the rest of the data
    - Spark and Parquet can optimize the I/O path and reduce the amount of data read from storage
- **Predicate push down:** is another feature of Spark and Parquet that can improve query performance by reducing the amount of data read from Parquet files. Predicate push down works by evaluating filtering predicates in the query against metadata stored in the Parquet files. Parquet can optionally store statistics (in particular the minimum and maximum value for a column chunk) in the relevant metadata section of its files and can use that information to take decisions, for example, to skip reading chunks of data if the provided filter predicate value in the query is outside the range of values stored for a given column.

Spark - Job Deployment and Execution

The best way to learn Spark programming is by following Spark official tutorials:
https://spark.apache.org/docs/latest/rdd-programming-guide.html

Code Examples:
https://spark.apache.org/examples.html

Deployment Options:

- Local development and testing - local install and debugging via Eclipse or another IDE
- Cluster installation based on Hadoop
- Cloudera VMs
- AWs EMR with Spark:

https://aws.amazon.com/blogs/aws/new-apache-spark-on-amazon-emr/

Spark

How to run

Execution Modes

spark-shell --master [local | spark | yarn-client | mesos]

- launches REPL connected to specified cluster manager
- always runs in client mode

spark-submit --master [local | spark:// | mesos:// | yarn] spark-job.jar

- launches assembly jar on the cluster

Spark

How to run - continue

Masters

- local[k] - run Spark locally with K worker threads
- spark - launches driver app on Spark Standalone installation
- mesos - driver will spawn executors on Mesos cluster (deploy-mode: client | cluster)
- yarn - same idea as with Mesos (deploy-mode: client | cluster)

Deploy Modes

- client - driver executed as a separate process on the machine where it has been launched and spawns executors
- cluster - driver launched as a container using underlying cluster manager

Batch Processing - other options

Other options ?

Is Spark the only framework that does the in-memory optimizations for MR processing model?

- **No!**
- there are many…. (too many :))

Some flavors are:

- pure batch/stream processing frameworks that work with data from multiple input sources (Spark, Flink, Storm)
- "improved" storage frameworks that also provide MR-type operations on their data (Presto, MongoDB, …)

Which one is better? The answer, as always, is "it depends" :-)