Datalake File Format: Parquet

arnaud.nauwynck@gmail.com

Course Esilv 2023

This document:
https://github.com/Arnaud-Nauwynck/presentations/tree/main/pres-bigdata/Course-2022-Spark /Datalake-File-Format-Parquet.pdf

Outline

- Parquet Caracteristics
  - Structured, with Schema
  - Data – Metadata (footer)
  - Columnar ... Column Pruning
  - Splittable ... spark Dataset Partitions
  - Compression
  - Encoding
  - Statistics, Bloom ... spark Predicate-Push-Down
  - Optimize write once, read many
    ... spark dataset.repartition().sortWithinPartition().write

Parquet is a Structured Format
Strongly Typed (Schema)

File Formats

Unstructured

Text

```
text line 1 \n
text line 2 \n
```

Csv

```
Col1;Col2;Col3\n
a1;b1;c1\n
a2;b2;c2\n
```

Semi-Structured

Json

```
{"a":"a1","b":"b1"}\n
{"a":"a2","b":"b2"}\n
```

Xml

```
<elt>
  <a>a1</a>
  <b>b1</b>
```

Structured

Serialization Structured

Avro,Thrift,Protobuf

```
schema:XX,
value:0101010101
```

Columnar Structured

Orc, Parquet

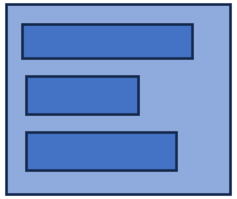
Structured : struct<>, array<>, map<>

Scalar Value
(= terminal element in grammar)
= primitive data-type

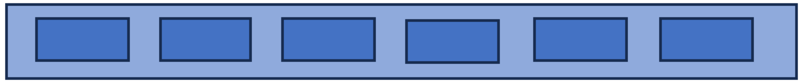
String
boolean
int
double
Date
...

Composite Value

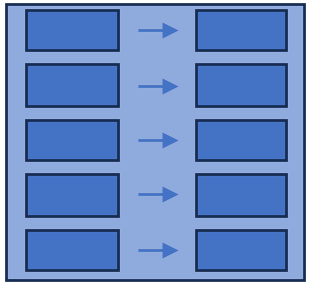
struct<a:Type1, b:Type2, ...>



array<ElementType>



map<KeyType,ValueType>



Type constraint = Schema

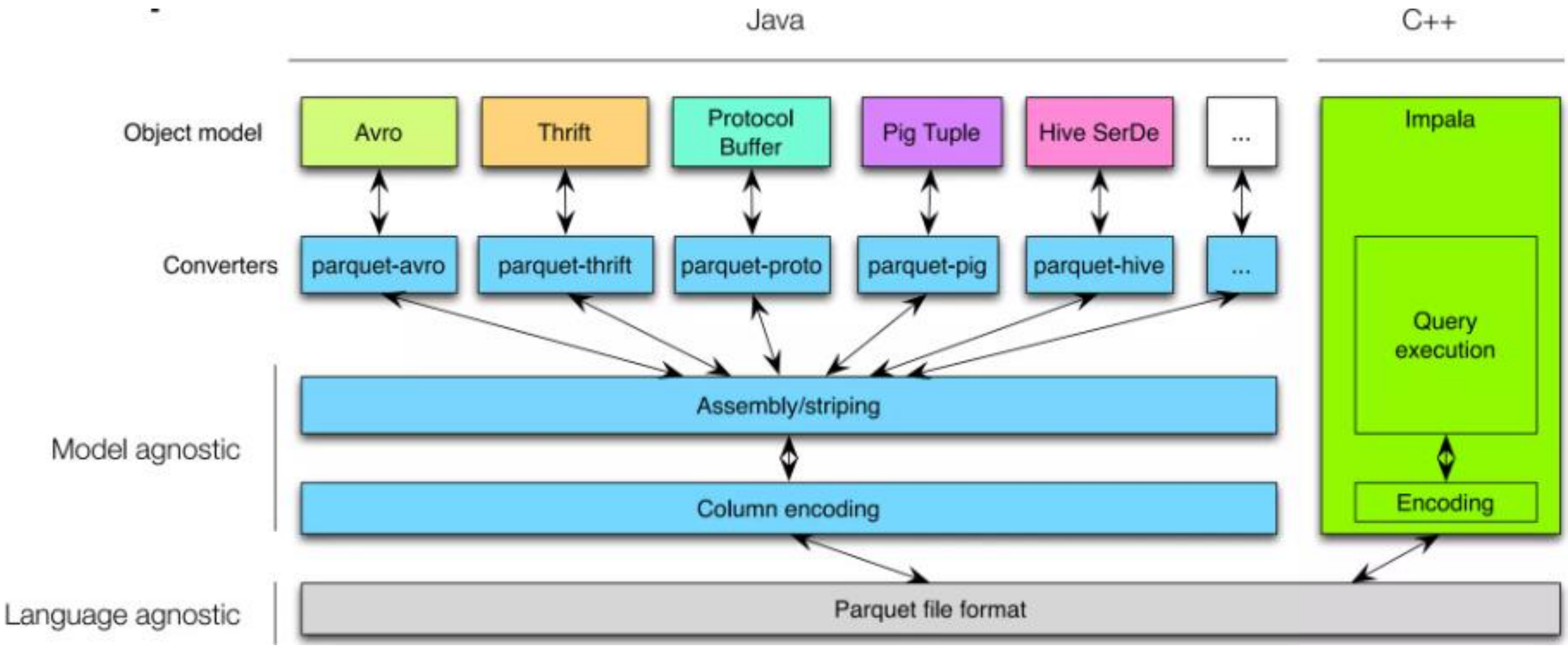
semi-structured
schema is optional

- { JSON } → JSON Schema (optional)
- <xml> → XSD Schema (optional)

structured
schema is strict

- .avro → .avsc = avro schema
  included in avro datafile format header
  implicit in avro client-server exchanges
- thrift
- protobuf
- .parquet → included in parquet file footer (= metadata)

Parquet SDK / Converter / ObjectModel

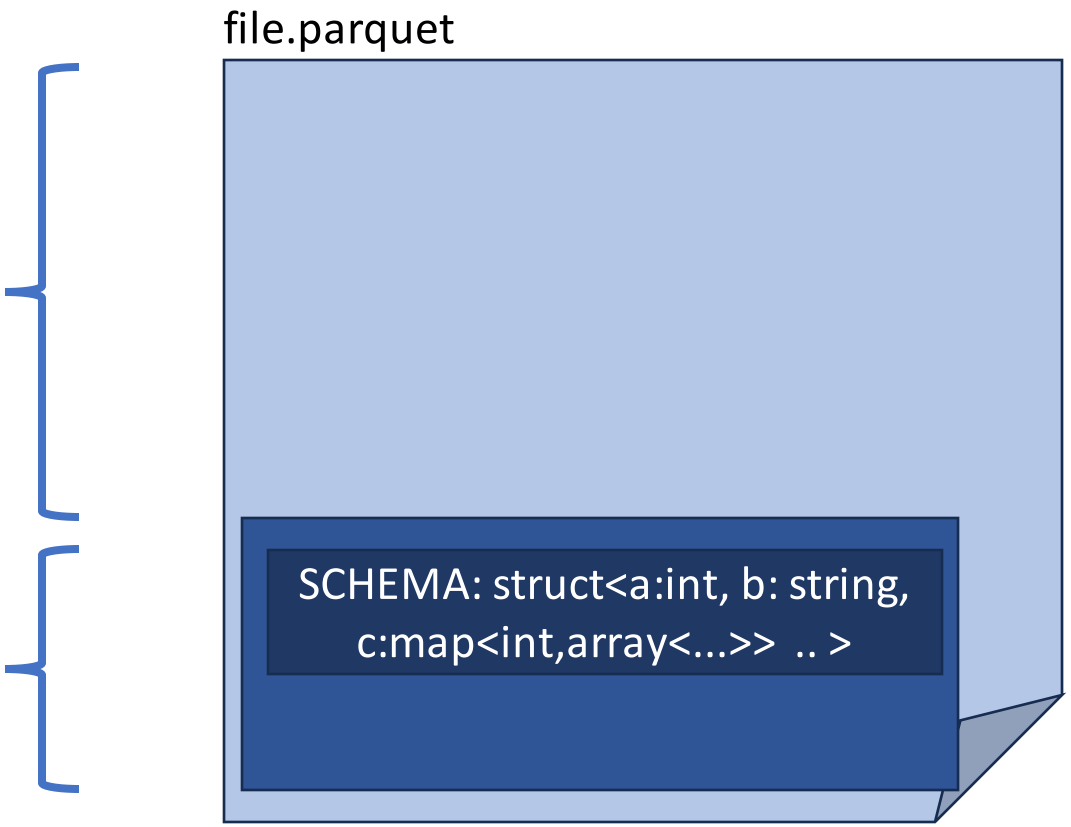


Parquet separates Data and MetaData

Parquet Schema : in metadata footer

data part
(usually N x blocks of 128 Mo)

footer: metadata part
(usually <= 50 ko)

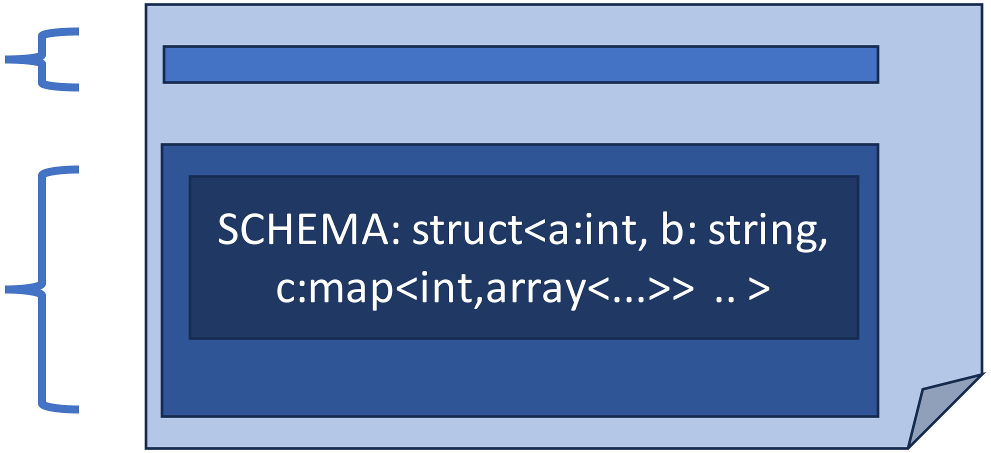


Parquet for Small Files ?

small file.parquet

data part can be small
(even empty)

parquet files
ALWAYS have an
uncompressable footer,
containing SCHEMA



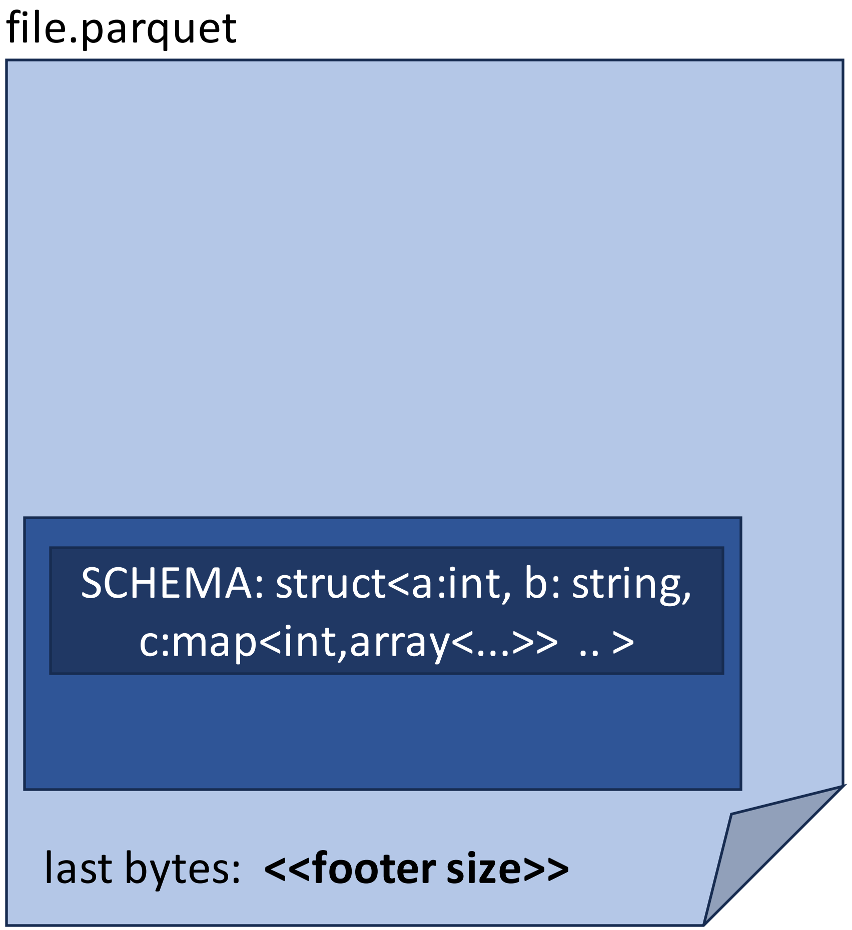
Parquet for small data is NOT efficient
bad ratio of "data / metadata" size

Read Parquet footer only

1/ query File length => len

3/ read footer [len-F, len]

2/ read last bytes [len-4, len] => F



Why Footer instead of Header?

For reader : not a big overhead, ONLY 3 calls to read

For writer : MUCH more practical to "stream" write Nx rows,
keep in-memory only few metadata,
to flush write at end

Parquet is Columnar

Parquet : Columnar File Format

Logical view: rows - columns

a1	b1	c1	d1
a2	b2	c2	d2
a3	b3	c3	d3
a4	b4	c4	d4

On Disk Row Serialization: like CSV, JSON, AVRO,

a1	b1	c1	d1	a2	b2	c2	d2	a3	b3	c3	d3	a4	b4	c4	d4

On Disk: **Columnar**

a1	a2	a3	a4	b1	b2	b3	b4	c1	c2	c3	c4	d1	d2	d3	d4

Columnar
=> better memory aligned,
Vectorized CPU pipeline

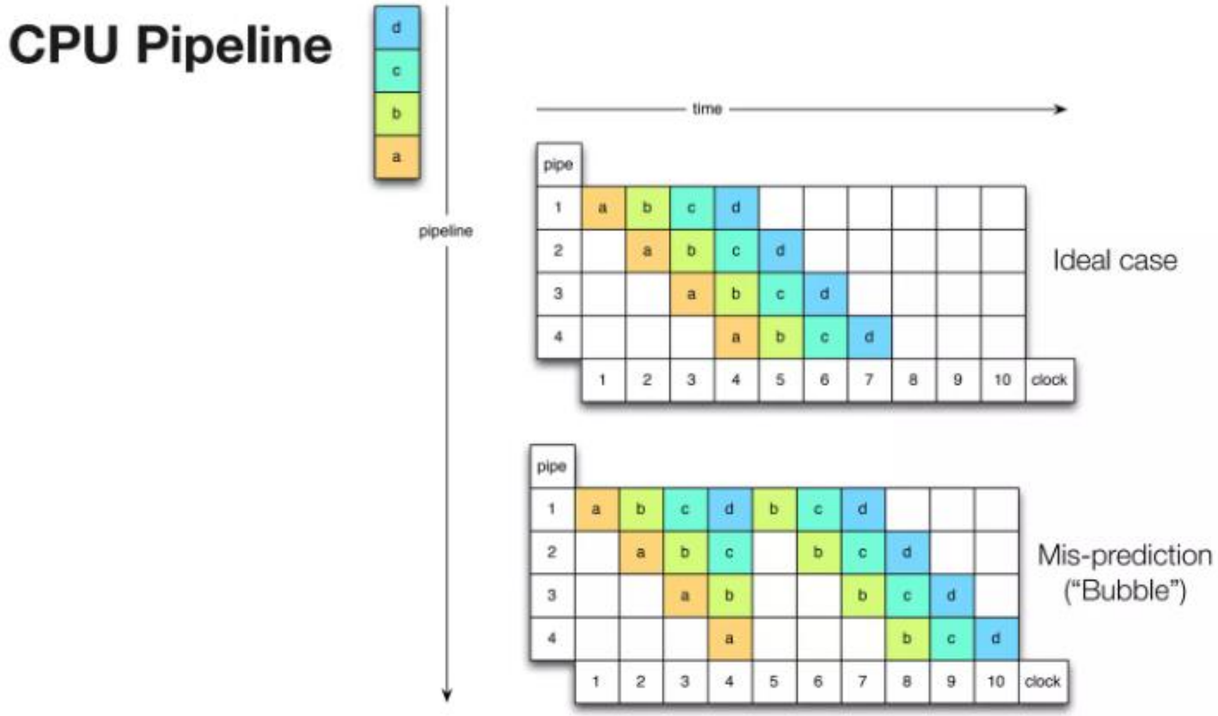
```
struct A {

  boolean f1;   // <= 1 bit (on 1 byte)
                // in-memory padding 3 bytes
  int     f2;  // 4 bytes, aligned on multiple of 4
               //  in-memory padding 4 bytes
  long    f3  // 8 bytes
}
```

array[struct<..>] <=> **array[boolean]**
array[int]
array[long]

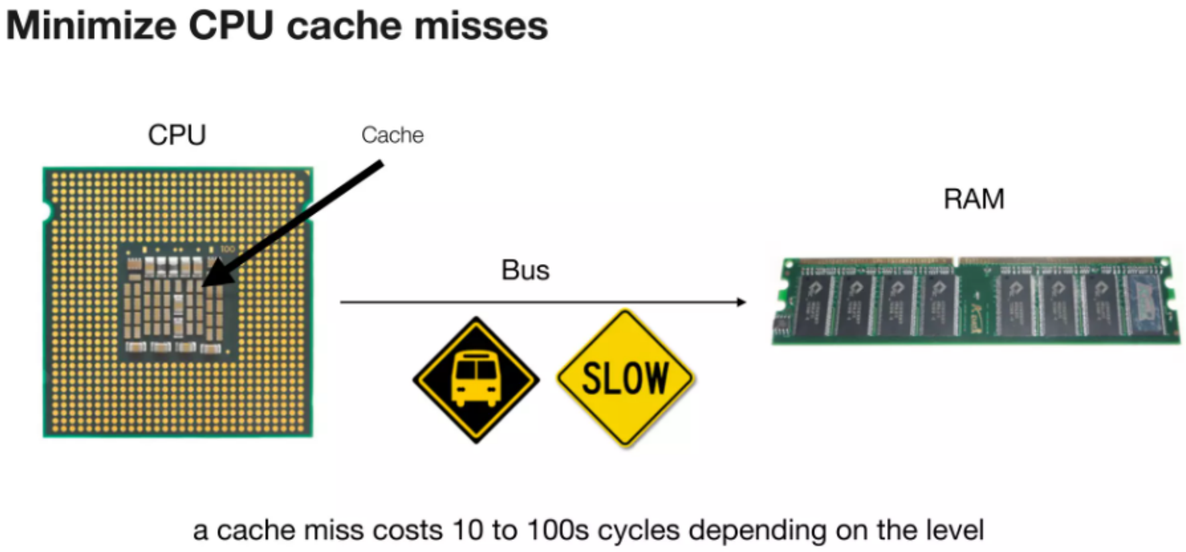
Vectorized Reader **~9x Faster**

Vectorized code ... fewer "If", "Loop", "calls"

Better CPU Pipeline



Better Memory Cache



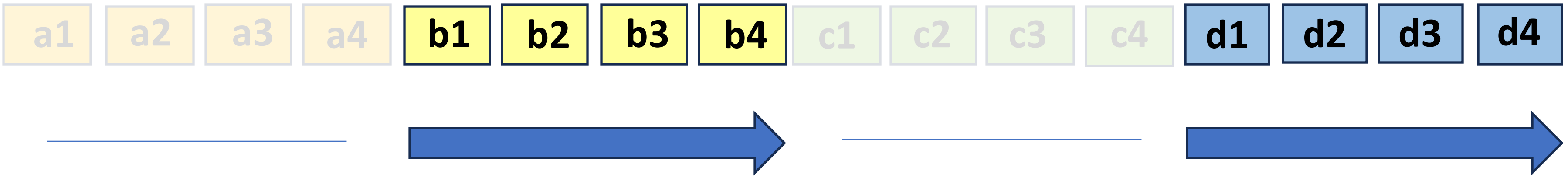
Column Pruning Optimization

SELECT b, d -- ONLY 2 columns
FROM table
WHERE ..

Logical view: rows - columns

a1	b1	c1	d1
a2	b2	c2	d2
a3	b3	c3	d3
a4	b4	c4	d4

On Disk READ: columnar

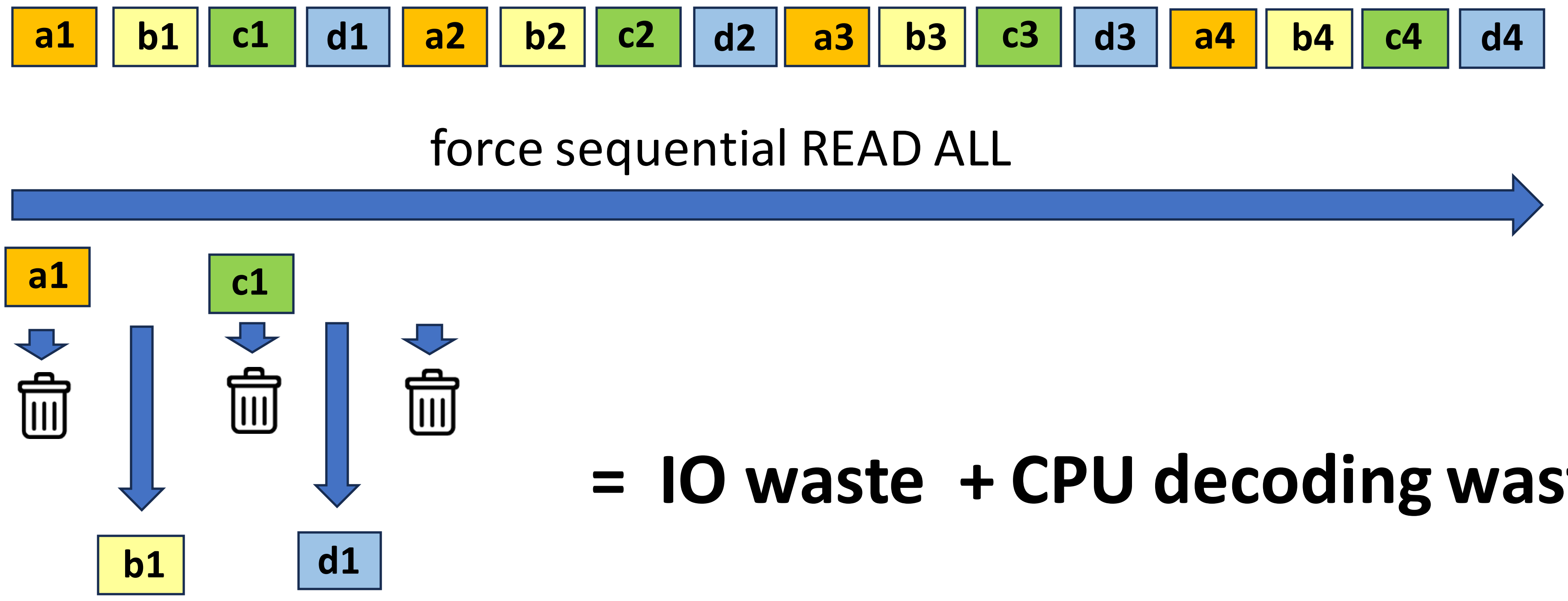


IO read for **b** values

IO read for **d(s)**

Compare IO Reads with JSON, CSV, Avro, ..

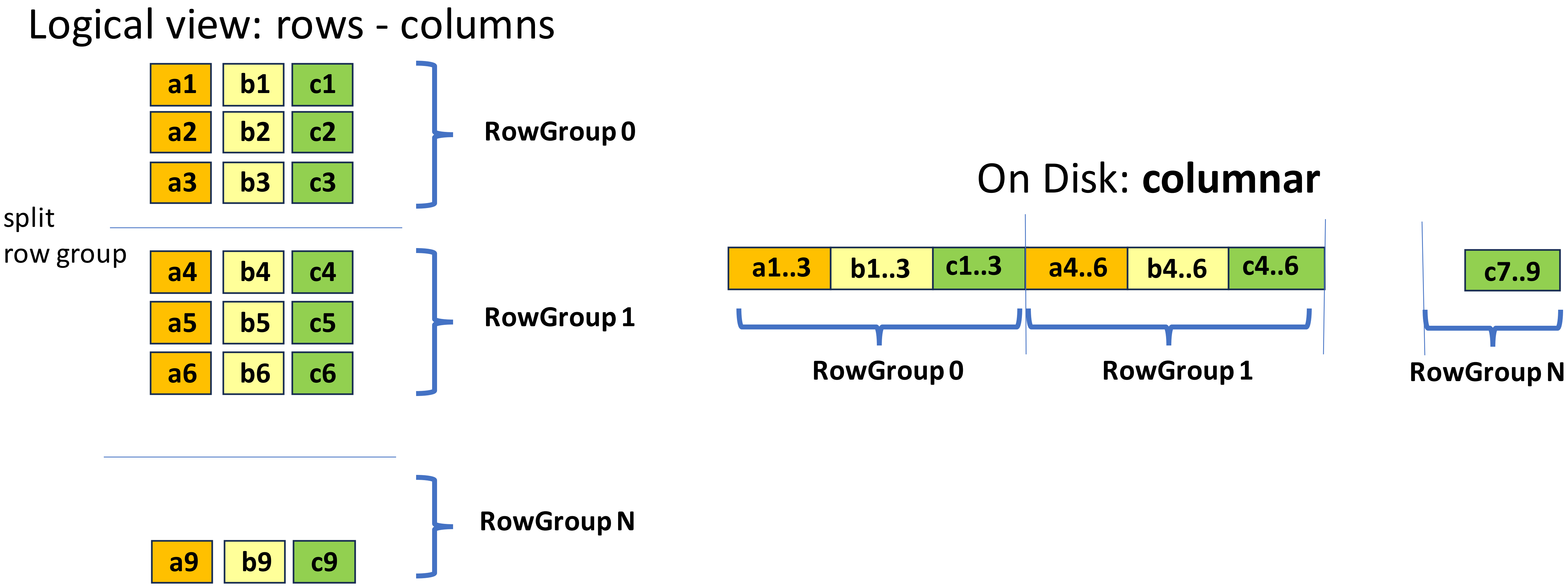
SELECT b, d -- ONLY 2 columns
FROM table
WHERE ..



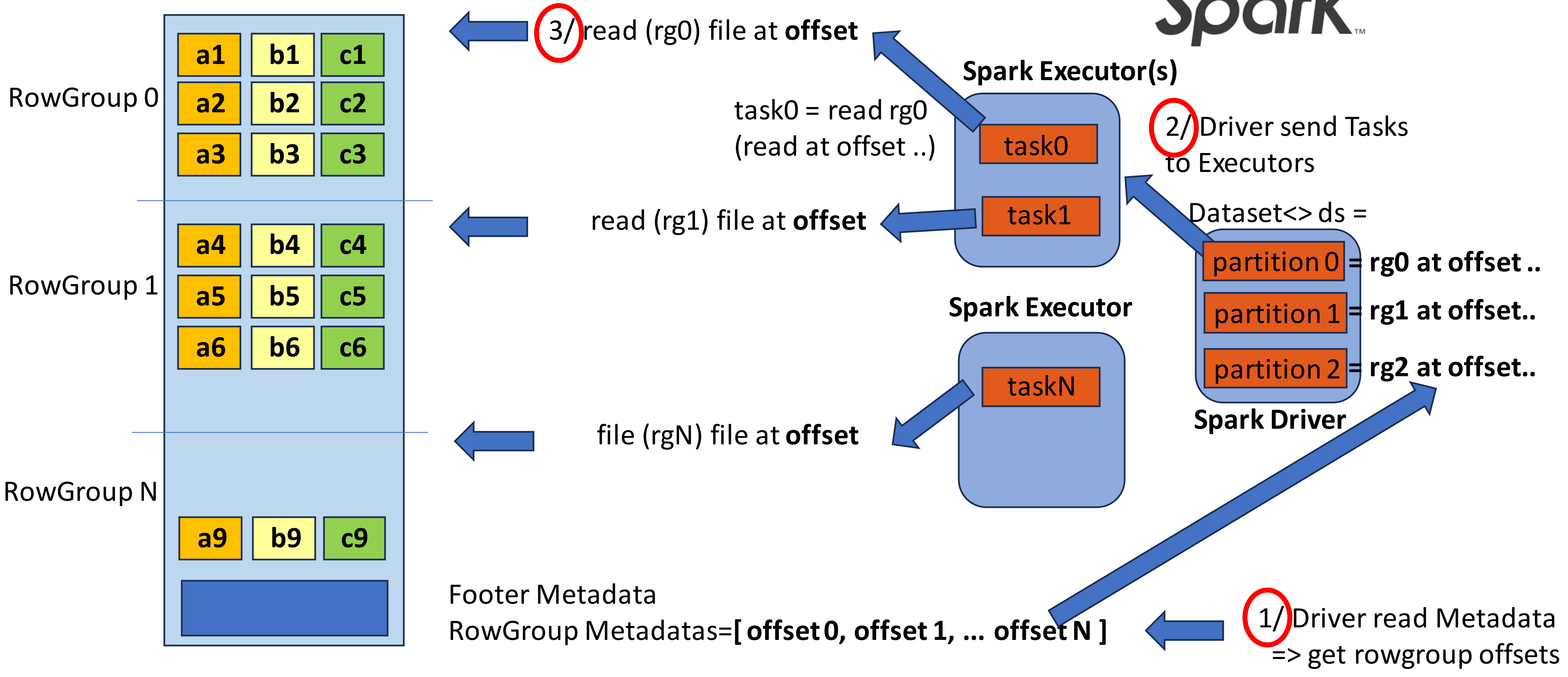
= IO waste + CPU decoding waste

Parquet is Splitteable

Parquet split rows by RowGroups



RowGroup ~ Spark Partition ~ Executor Thread



1 Parquet RowGroup(s)
-> default to 1 Spark DataSet partition

by default,

parquet.block.size = 128 Mo

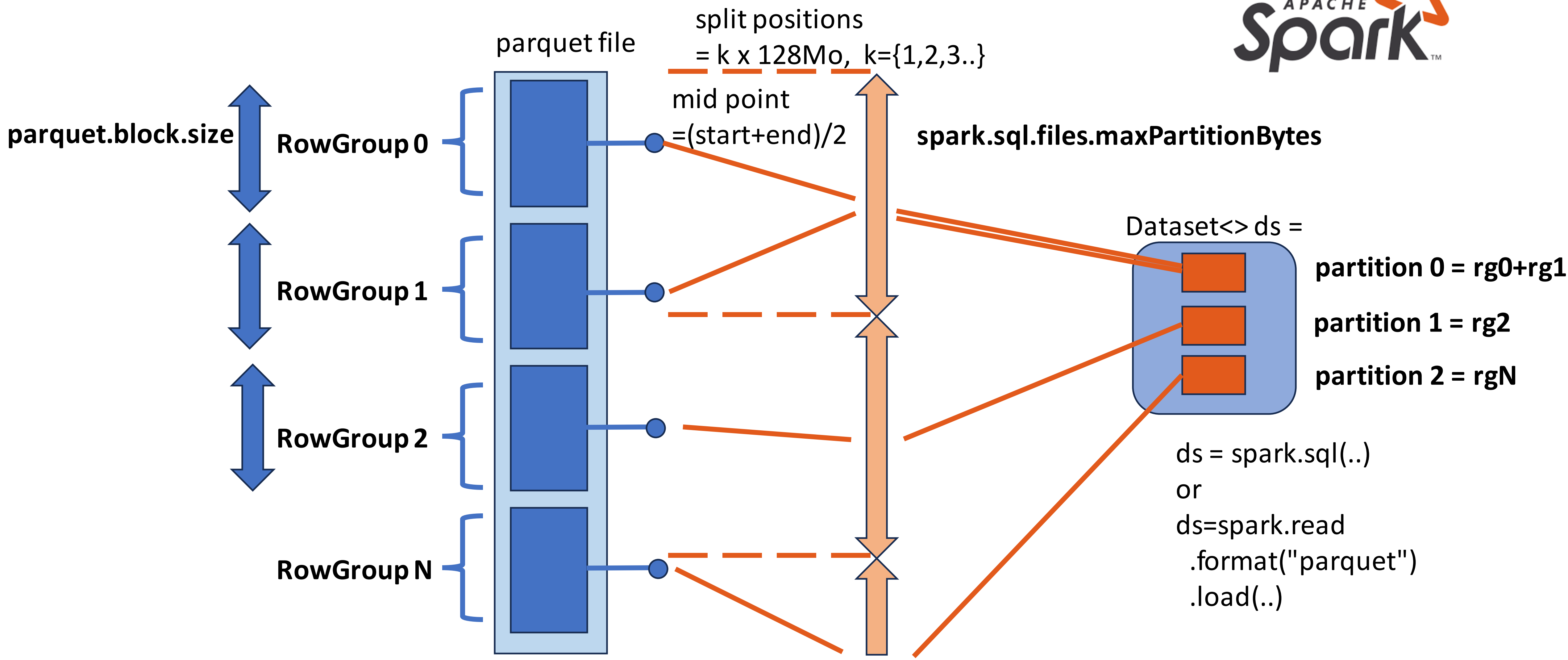
=

spark.files.maxPartitionBytes = 128 Mo

=

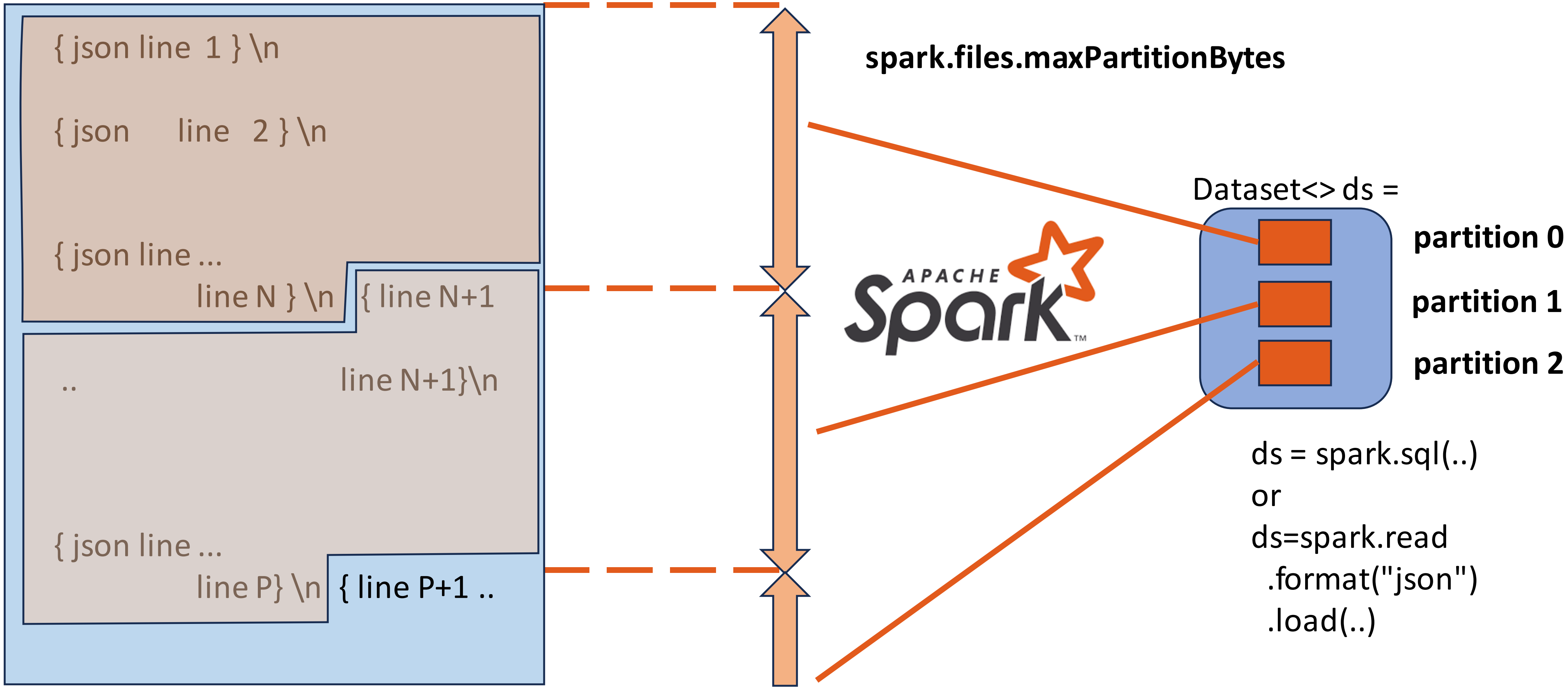
spark.sql.files.maxPartitionBytes = 128 Mo

N Parquet RowGroups -> FileSplit to P (<=N) Spark Partitions



{Texts,CSV,Json} formats FileSplit

each spark executor reader Thread
  at split start => ignore first chars until '\n'
  at split end => read extra chars until '\n



Example : Reading 1 CSV of 3.2 Go
=> 26 splits = 25 (~128 Mo) + 1 very small

Windows-SSD (C:) › data › OpenData-gouv.fr › bal

Nom	Modifié le	Type	Taille
adresses-france.csv	08/09/2022 13:49	Fichier CSV Microsoft Excel	3 280 937 Ko

3 280 937 / (128*1024) = 25.03

```
scala> val ds = spark.read.format("csv").option("delimiter",";").load("C:/data/OpenData-gouv.fr/bal")
val ds: org.apache.spark.sql.DataFrame = [_c0: string, _c1: string ... 17 more fields]

scala> ds.toJavaRDD.getNumPartitions
val res0: Int = 26
```

Parquet is Compressable { snappy | gz }

*.gz   is NOT Splitteable !

Windows-SSD (C:) › data › OpenData-gouv.fr › bal-gz

Nom	Modifié le	Type	Taille
adresses-france.csv.gz	08/09/2022 13:49	Dossier d'archive compressé	659 936 Ko

CSV file was 3.2 Go, now 660 Mo in .gz
but NOT Splitteable => 1 spark partition, reading (CPU intensive) by 1 Thread only !
~8 times slower on a 8 cores PC

```
scala> val csvGzDs = spark.read.format("csv").option("delimiter",";").load("C:/data/OpenData-gouv.fr/bal-gz")
23/11/05 20:17:12 WARN ZlibFactory: Failed to load/initialize native-zlib library
val csvGzDs: org.apache.spark.sql.DataFrame = [_c0: string, _c1: string ... 17 more fields]

scala> csvGzDs.count()
[Stage 2:>                                                      (0 + 1) / 1]
```

Compressions Algorithm

data

0101010101001

.snappy

010101

fast compression/decompression
(focus on speed)

.gz (gzip)

01101

slower-compression,
better compression ratio
(focus on size)

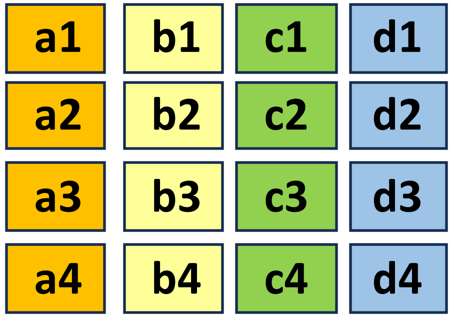
.lz4

010101

compromise between
fast / compression ratio

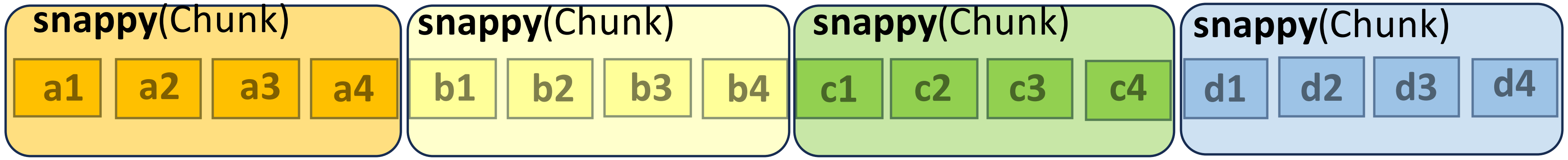
PARQUET.{snappy | gz}

Logical view: rows - columns



On Disk: **Columnar NO compression**

On Disk: **Columnar parquet.{snappy | gz }**



every "Chunk" of column data are compressed INDEPENDENTLY
=> file is STILL Splitteable

Parquet use Encodings

Run-length encoding (RLE)

a
a
a

repeated **x3** times

bb
bb
bb
bb
bb
bb

repeated **x6** times

c
c
c
c

repeated **x4** times

Var Length Encoder:

3 a
6 bb
4 c

Size of Adding "Constant" Column

adding a column with only "0" for Billions of rows

=> take only ~100 bytes per RowGroup

Dictionary Encoding

Manchester City,
Arsenal,
Manchester City,
FC Barcelone,
Arsenal,
Newcastle,
Manchester United,
Newcastle,
FC Barcelone,
Arsenal,
Manchester United,
...

Dictionary Encoder:

Distinct Dictionary Values:

1=Manchester City
2=Arsenal
3=FC Barcelone
4=Newcastle
5..

Value Indexes:

1, 2, 1, 3, 4, 5 ..

Dictionary Size Limit

By default, Dictionary size = max 1 Mega (per RowGroup - Column Chunk)

When using
- Huge RowGroup (> 128Mo) => less Dictionaries used
- Small RowGroup (32M, 64Mo) => more Dictionaries

Parameters:

parquet.enable.dictionary=true (default)

parquet.dictionary.page.size=1M

Delta Encoding

BaseValue=10007 **MinDelta=-7**

	Delta	**Delta-MinDelta (>=0)**
10007		
10003	-4	3
10005	+2	9
10002	-3	4
10009	+7	14
10002	-7	0
10008	+6	13
	..	..

+7

+7

Delta Encoding

10007, -7, 3,9,4,14,0,13

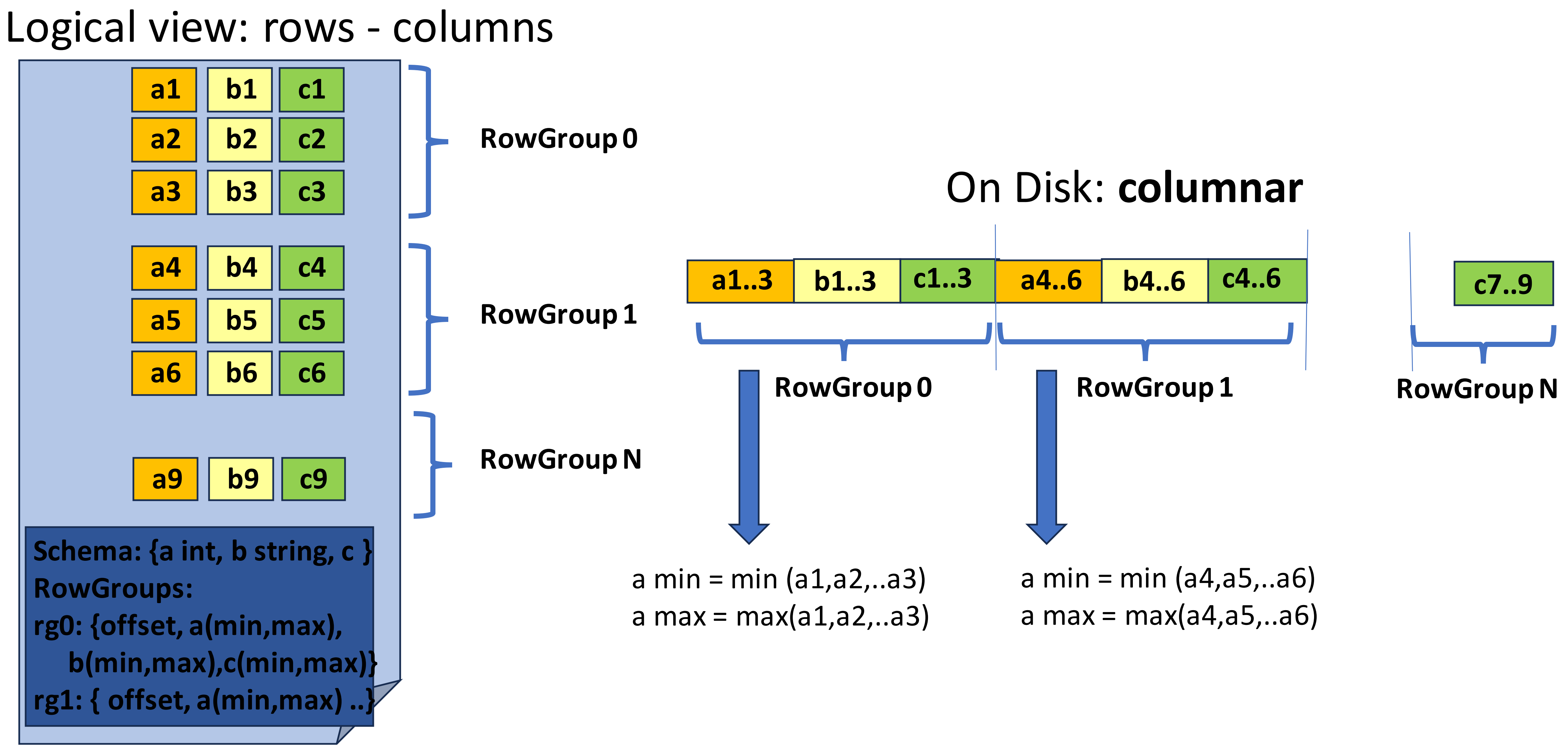
example size when fitting "int"(4 bytes)
7 x 4 = 28 bytes

when fitting 2 bytes
4 + 6 x 2 = 28 bytes

when fitting 1 bytes
4 + 2 + 6 x 1 = 12 bytes

Parquet use Statistics

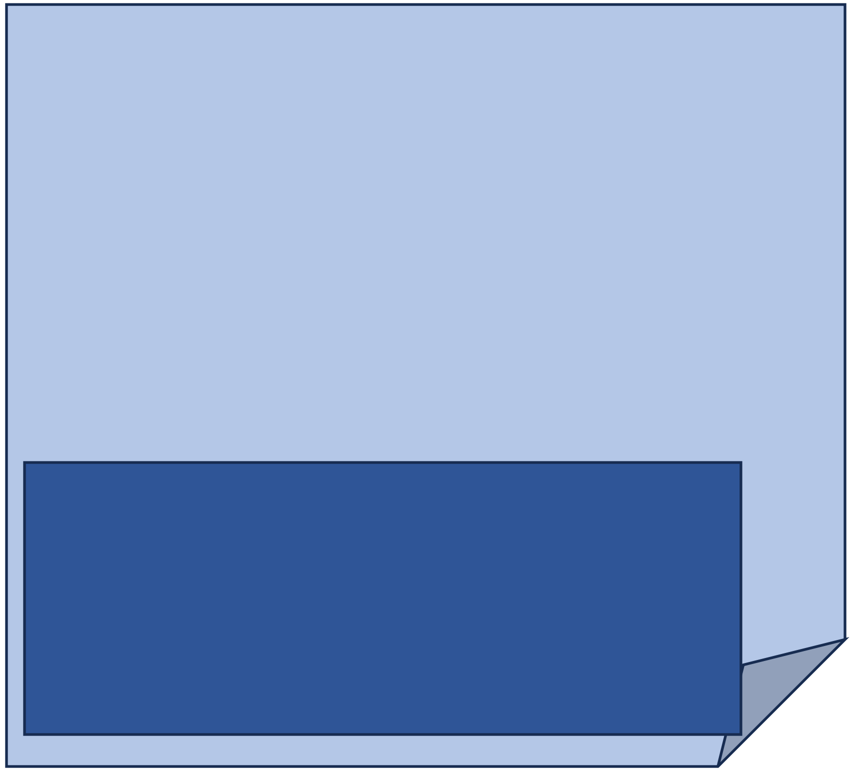
Column Statistics: min/max Value per RowGroup



Reading Metadata
=> read schema + offset + statistics

data part
(usually N x blocks of 128 Mo)

footer: metadata part
(usually <= 50 ko)

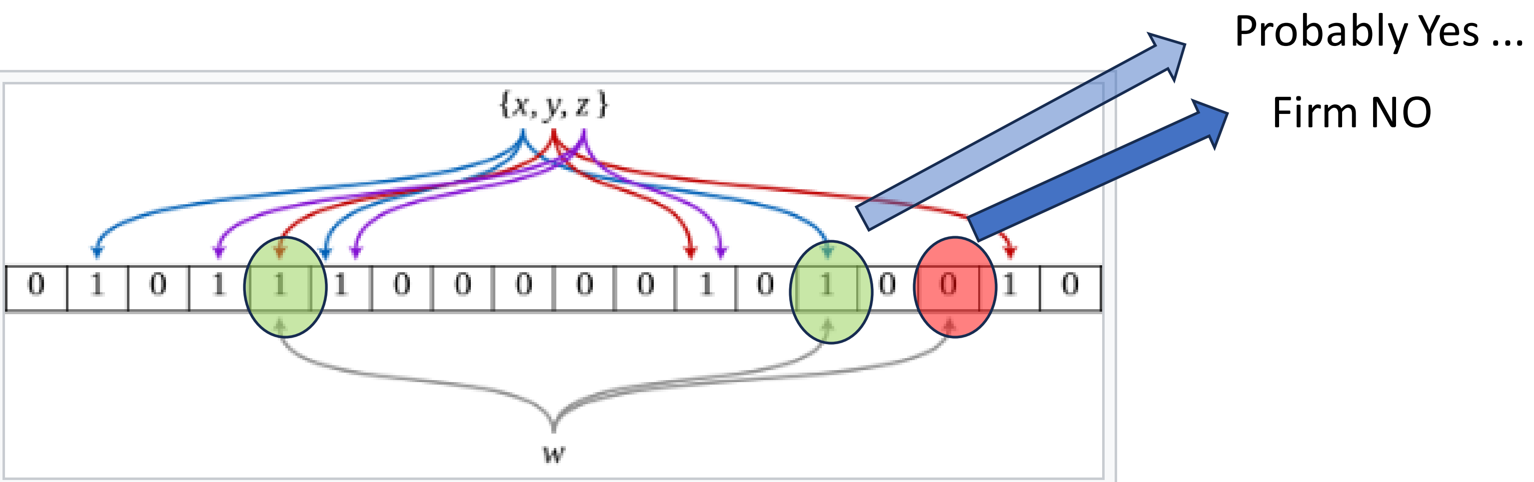


Parquet use Bloom Filter

Bloom Filter

is W in the set {x,y,z} ?

hash(W) = 0000100...10100

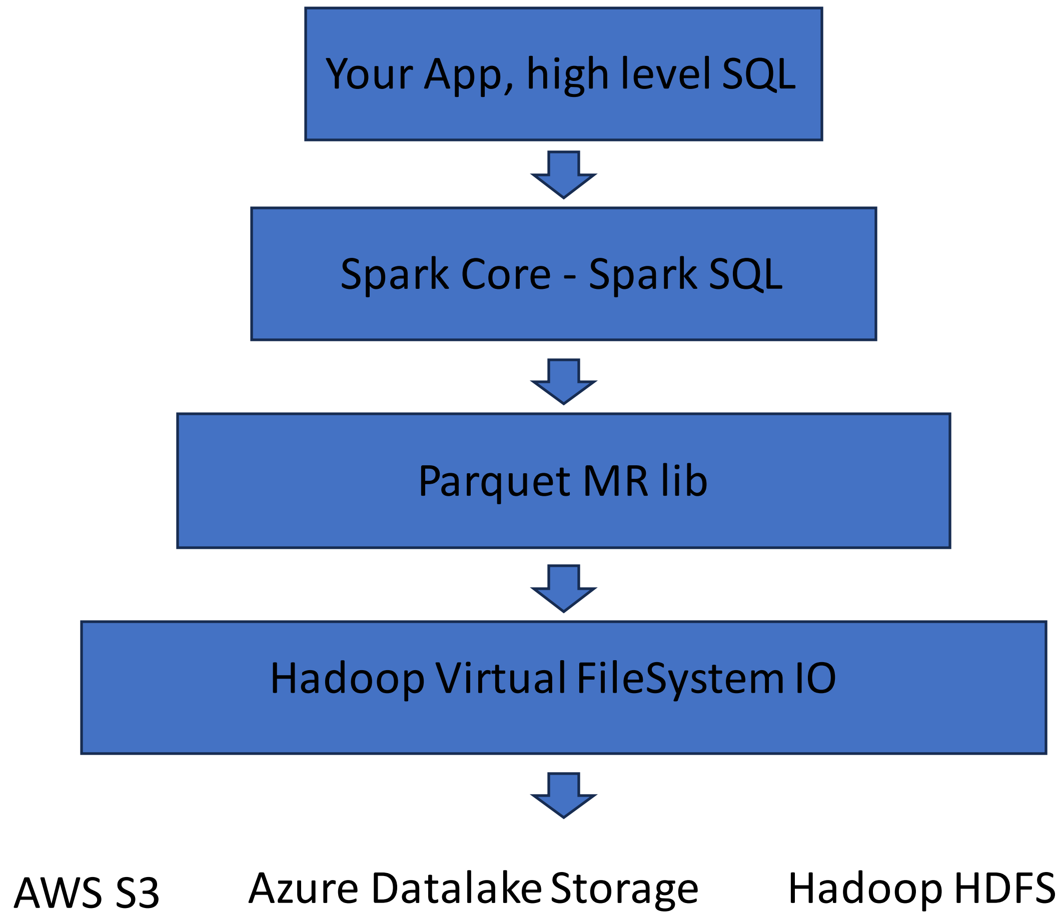


An example of a Bloom filter, representing the set $\{x, y, z\}$. The colored arrows show the positions in the bit array that each set element is mapped to. The element w is not in the set $\{x, y, z\}$, because it hashes to one bit-array position containing 0. For this figure, $m = 18$ and $k = 3$.

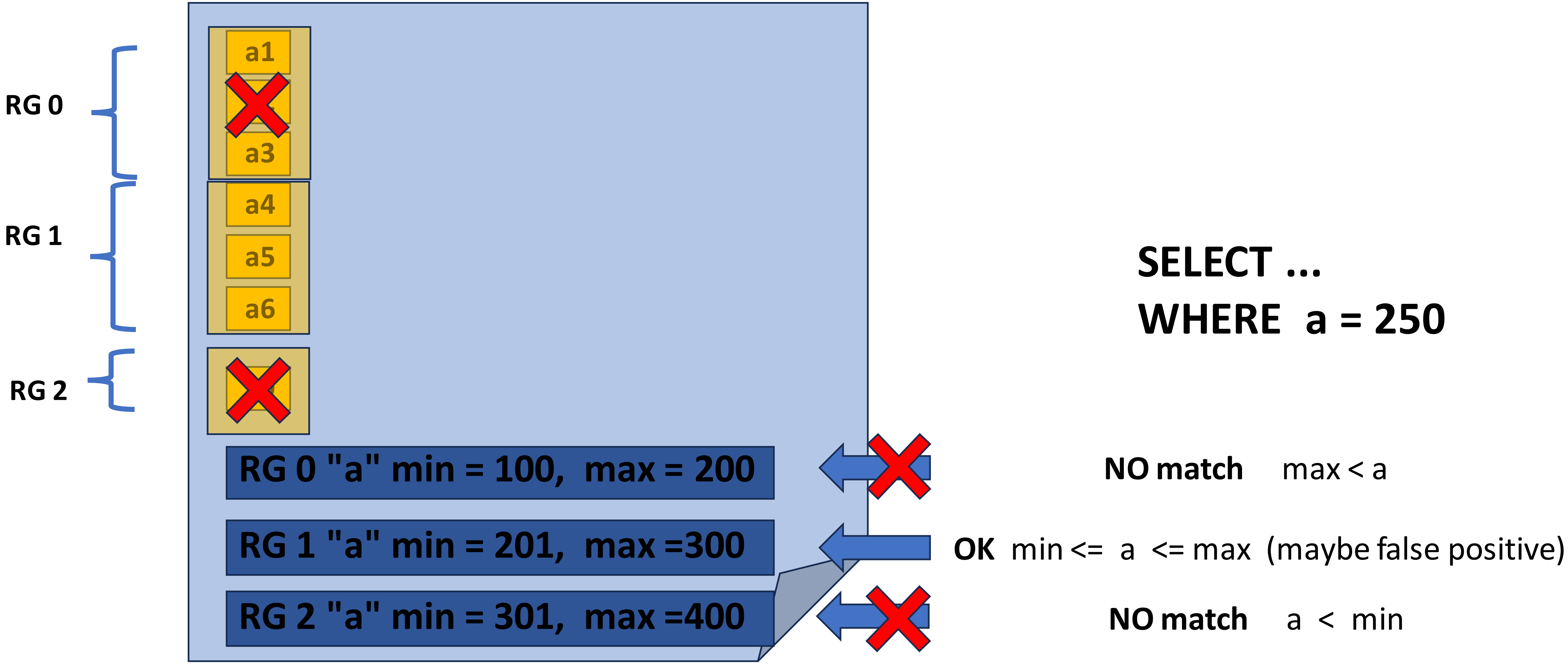
https://en.wikipedia.org/wiki/Bloom_filter

Parquet Predicate-Push-Down

Push-Down :
from Spark -> Parquet Lib -> IO Storage

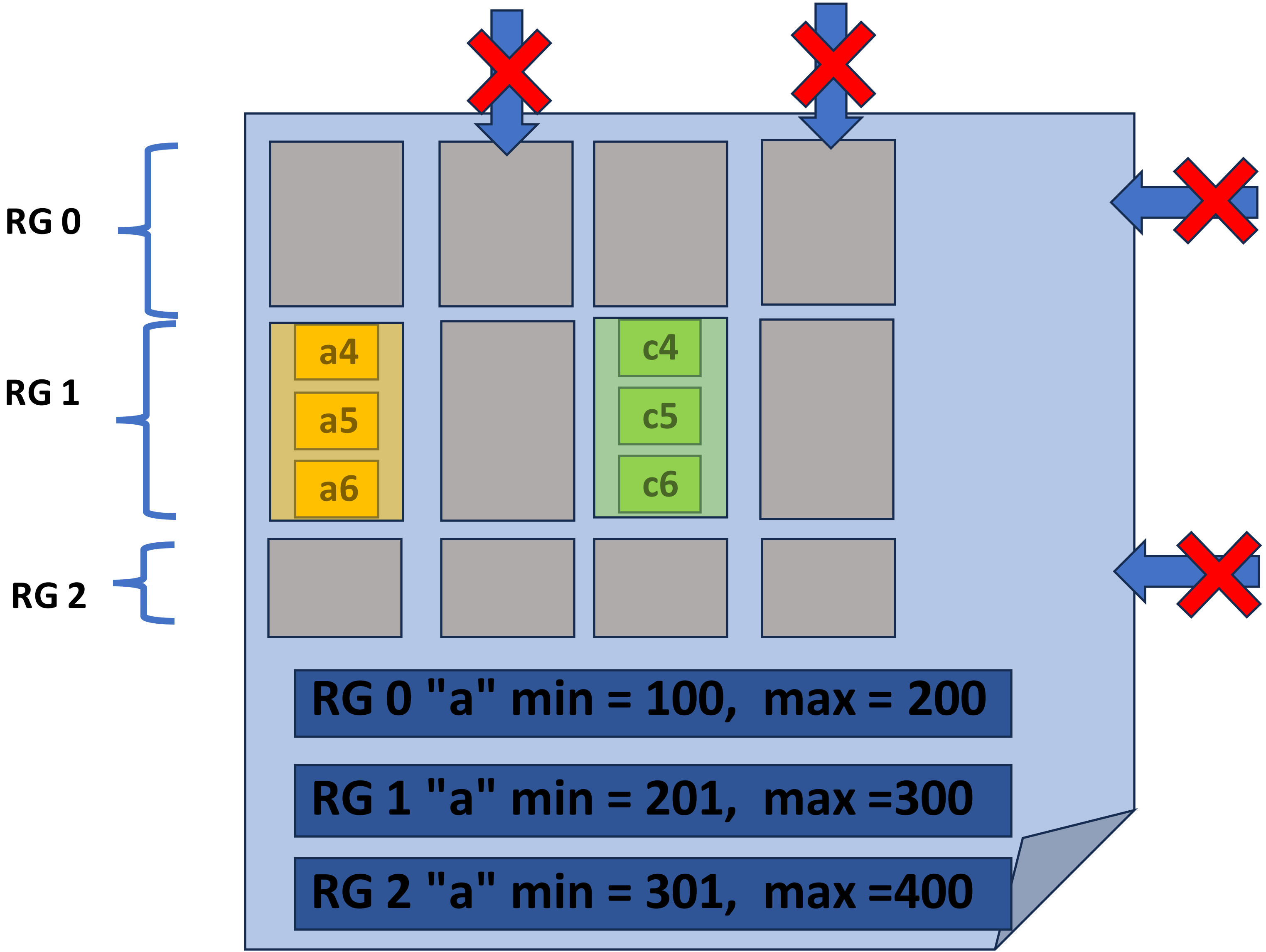


Statistics for skip read RowGroup
SELECT .. WHERE column=value



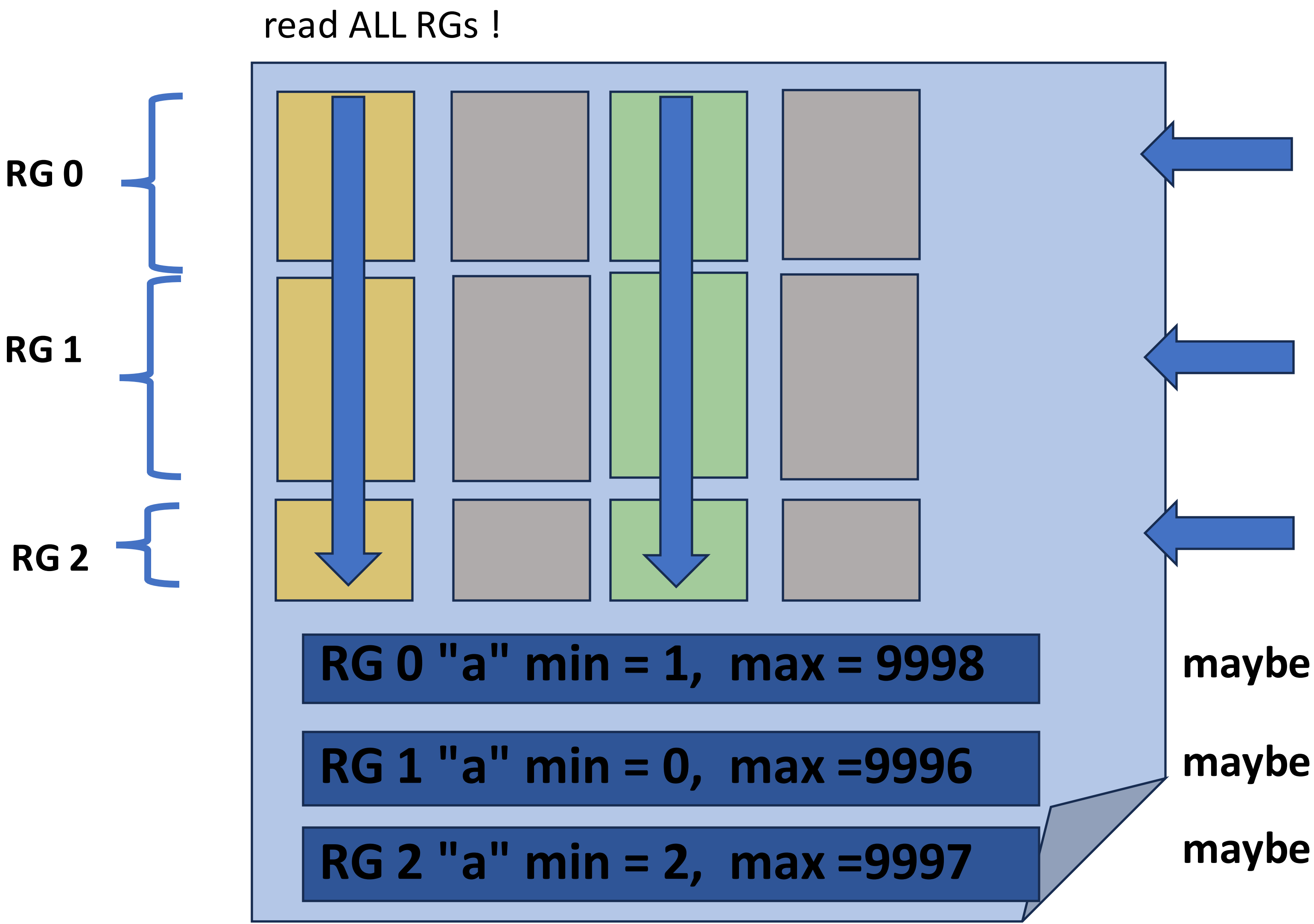
SELECT ...
WHERE a = 250

Column Pruning
+ RowGroup Pruning(PPD) = minimal Reads



SELECT c
WHERE a = 250

Badly sorted files
=> Bad min/max statistics => False positives



SELECT c
WHERE a = 250

maybe false positive min <= a <= max

maybe false positive min <= a <= max

maybe false positive min <= a <= max

Optim   Write Once /   Read Many

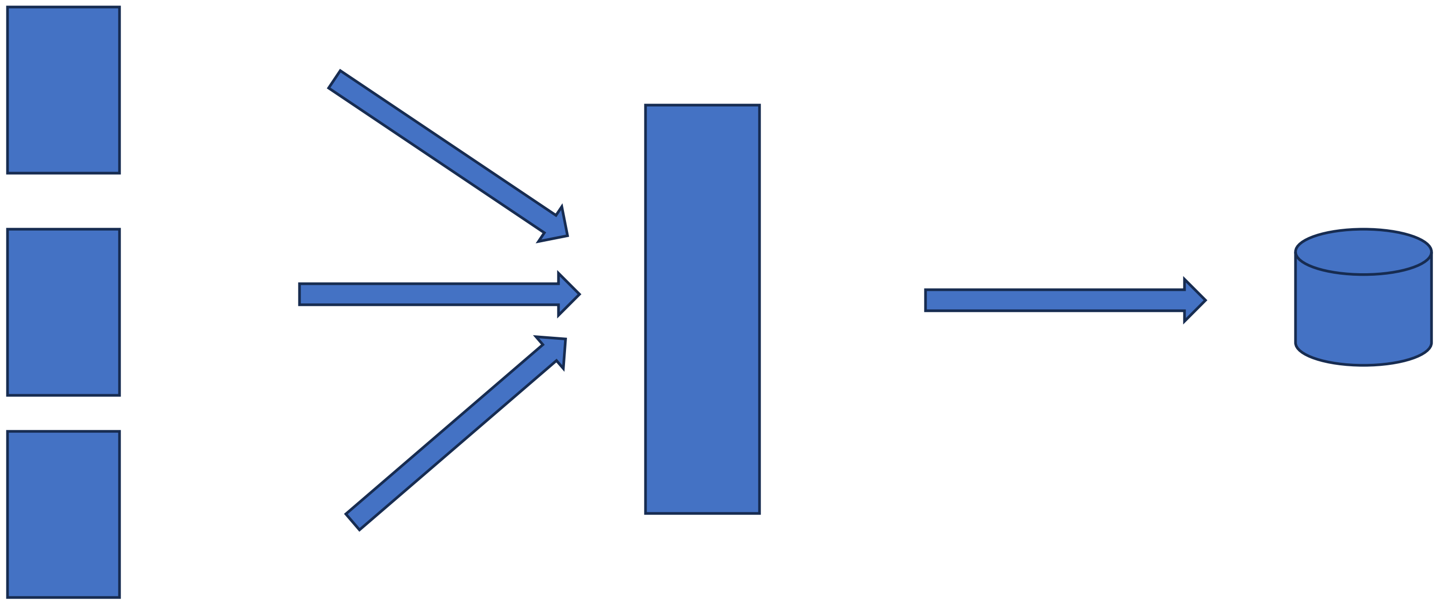
when writing Parquet files ... think how file might be read later !

spend CPU when writing   to save CPU / IO later reading

```
dataset
	.repartition(nRepartitionCount)
	// or  .repartition("col1", nRepartitionHash)

	.sortWithinPartitions("colA", "colB")

	.write
	.option("parquet.block.size", 32*MEGA)

	.format("parquet")
	.save("file://some-dir")
	// or
	// .format("hive").insertInto("hiveDb.hiveTableName")
```

avoid many small files dataset.repartition(1) or .coalesce(1)

```
dataset
     .repartition(nRepartitionCount)  // or  .coalesce(nRepartitionCount)
```

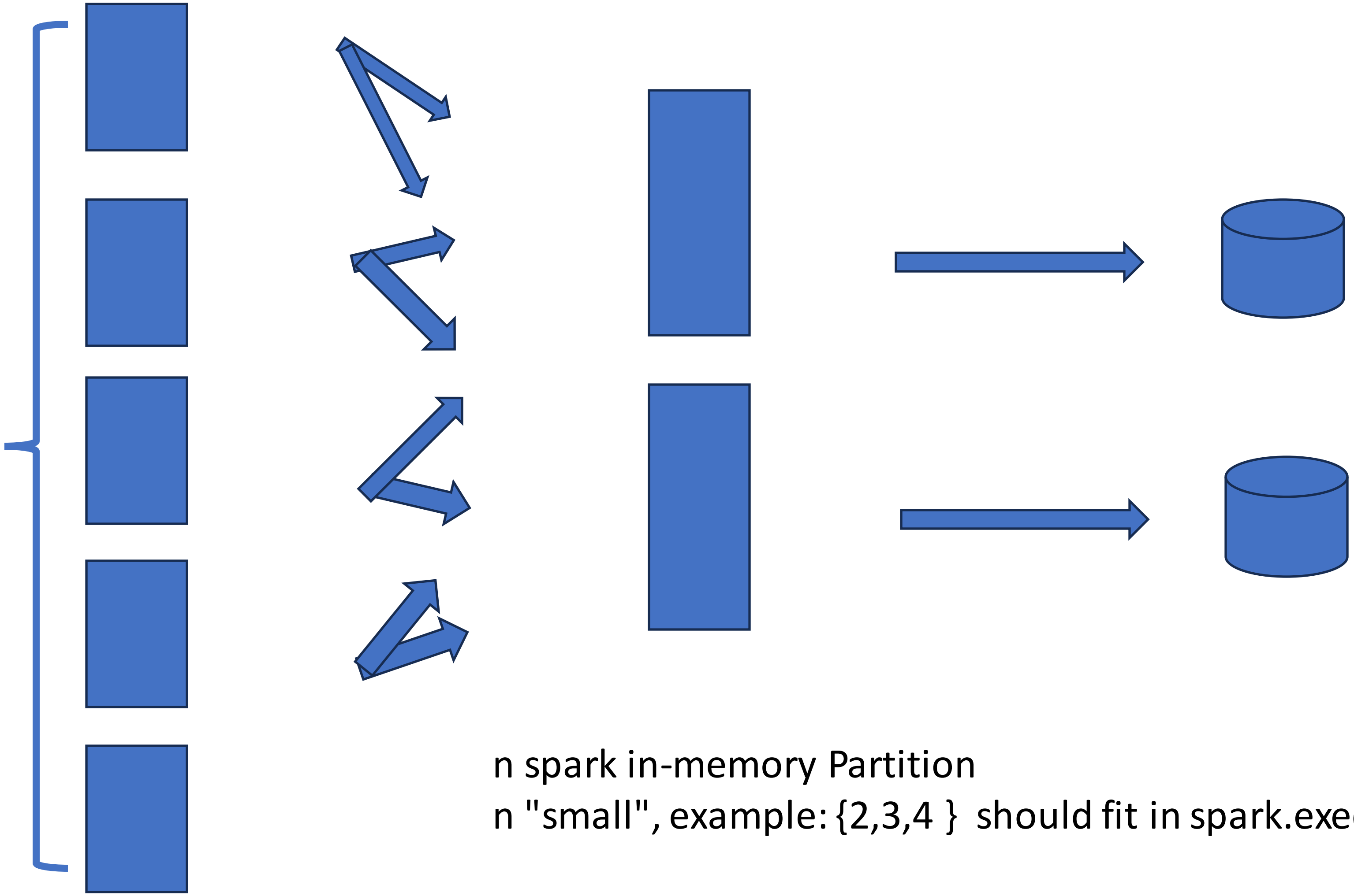


N spark in-memory Partitions
distributed over N executors

1 spark in-memory Partition
(should fit in spark.executor.memory !!)

write 1 parquet file

Does not fit in memory.. compromise to .repartition(smallN)



typically 200 partitions

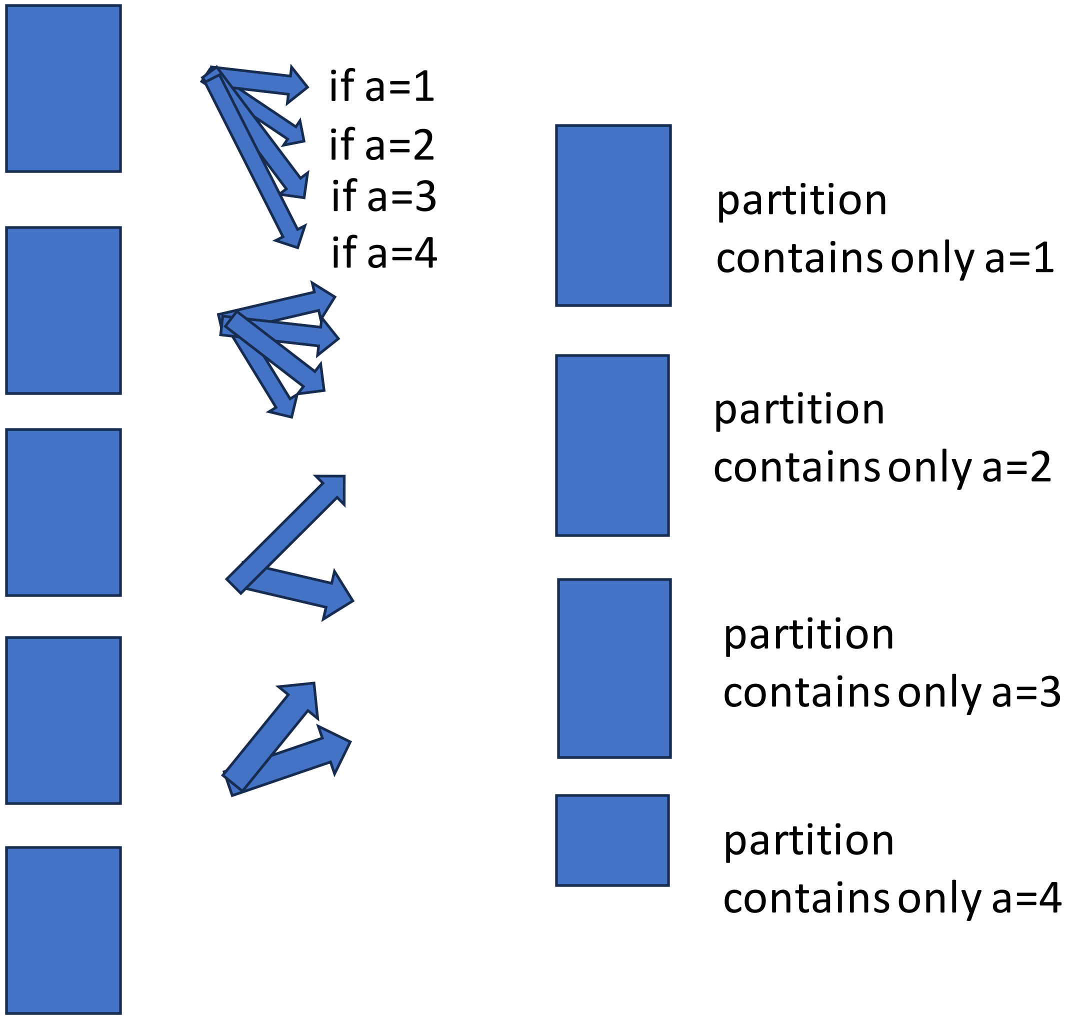
default
spark.shuffle.size=200

n spark in-memory Partition
n "small", example: {2,3,4 } should fit in spark.executor.memory !!

.repartition("column")

typical usage:
column "a" has FEW distinct values { 1, 2, 3, 4 }

ds = dataset.repartition("a")

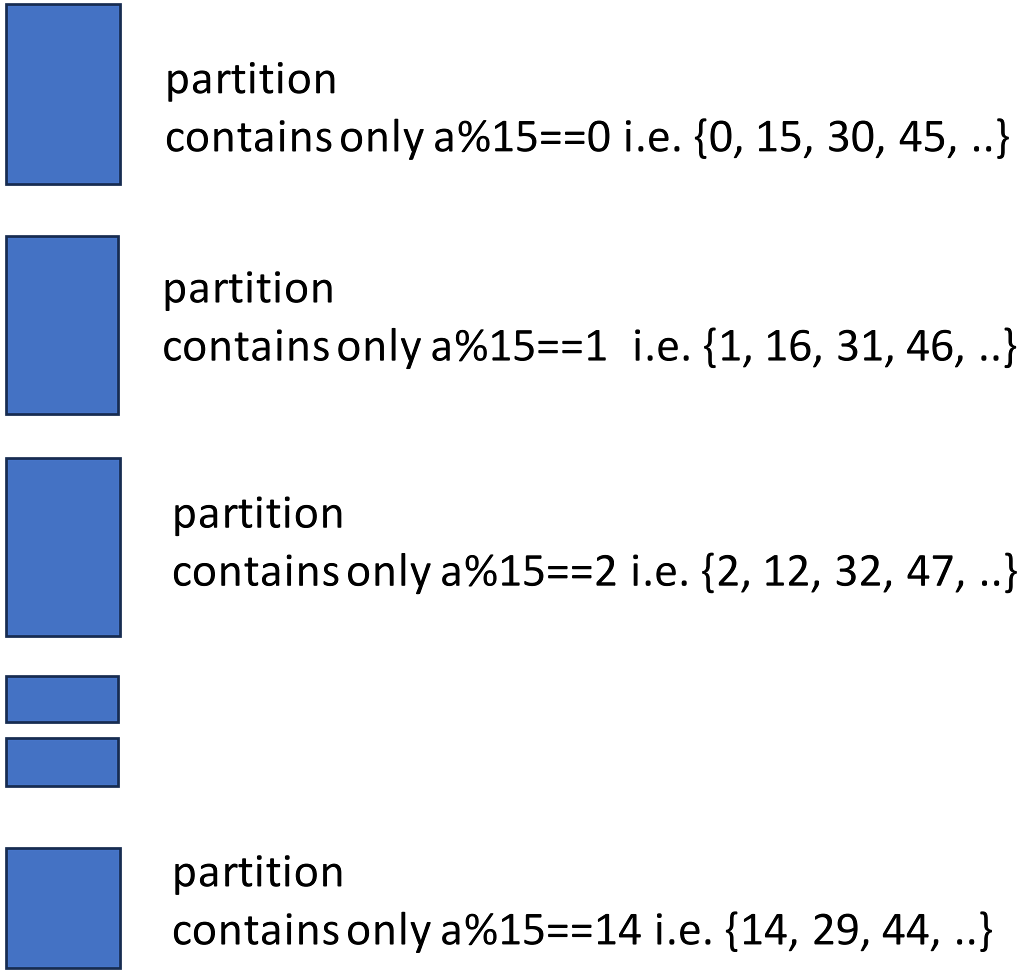


.repartition("column", nHashCount)

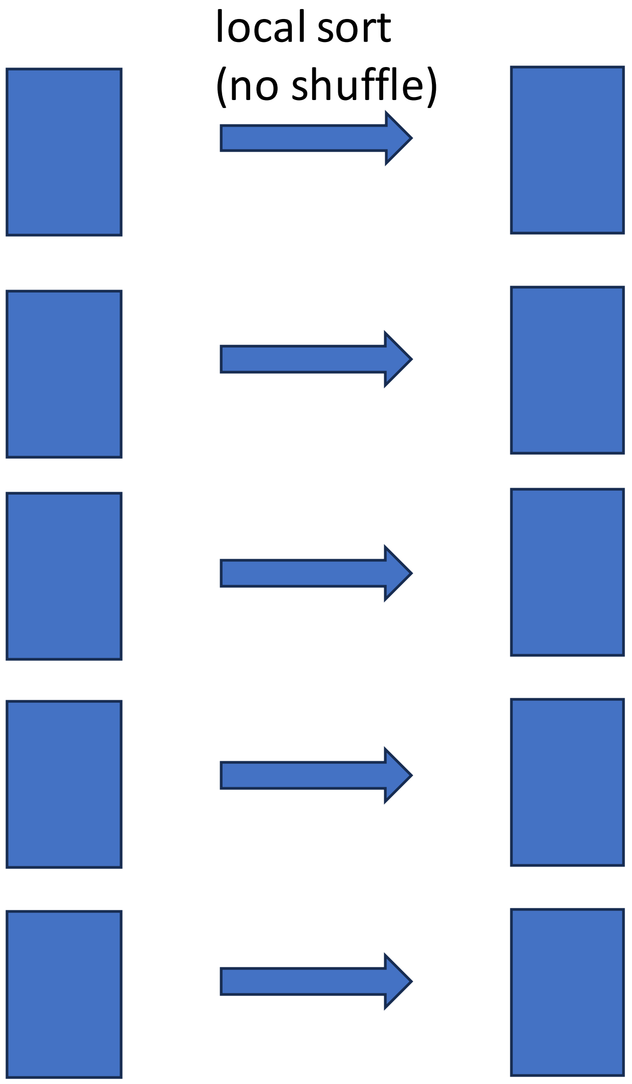
example ... having many distinct values

col "a" values in [1, 2, 3, ... 500000]

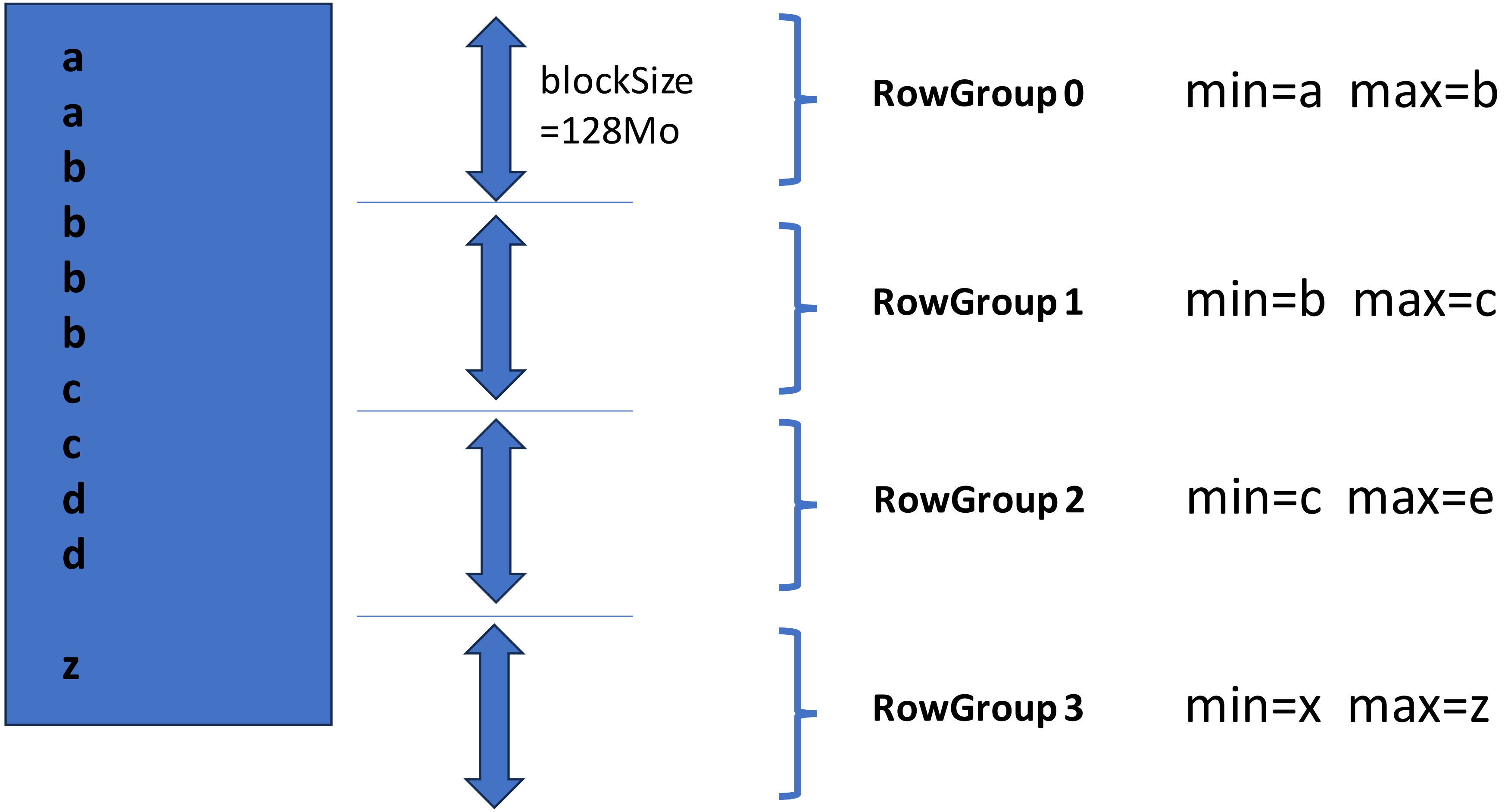
=>   h = hash(a) % 15   in [0, 1, .. 14]



.sortWithinPartition("a", "b", ...)



.sortWithinPartition
=> RowGroups stats more "compact"



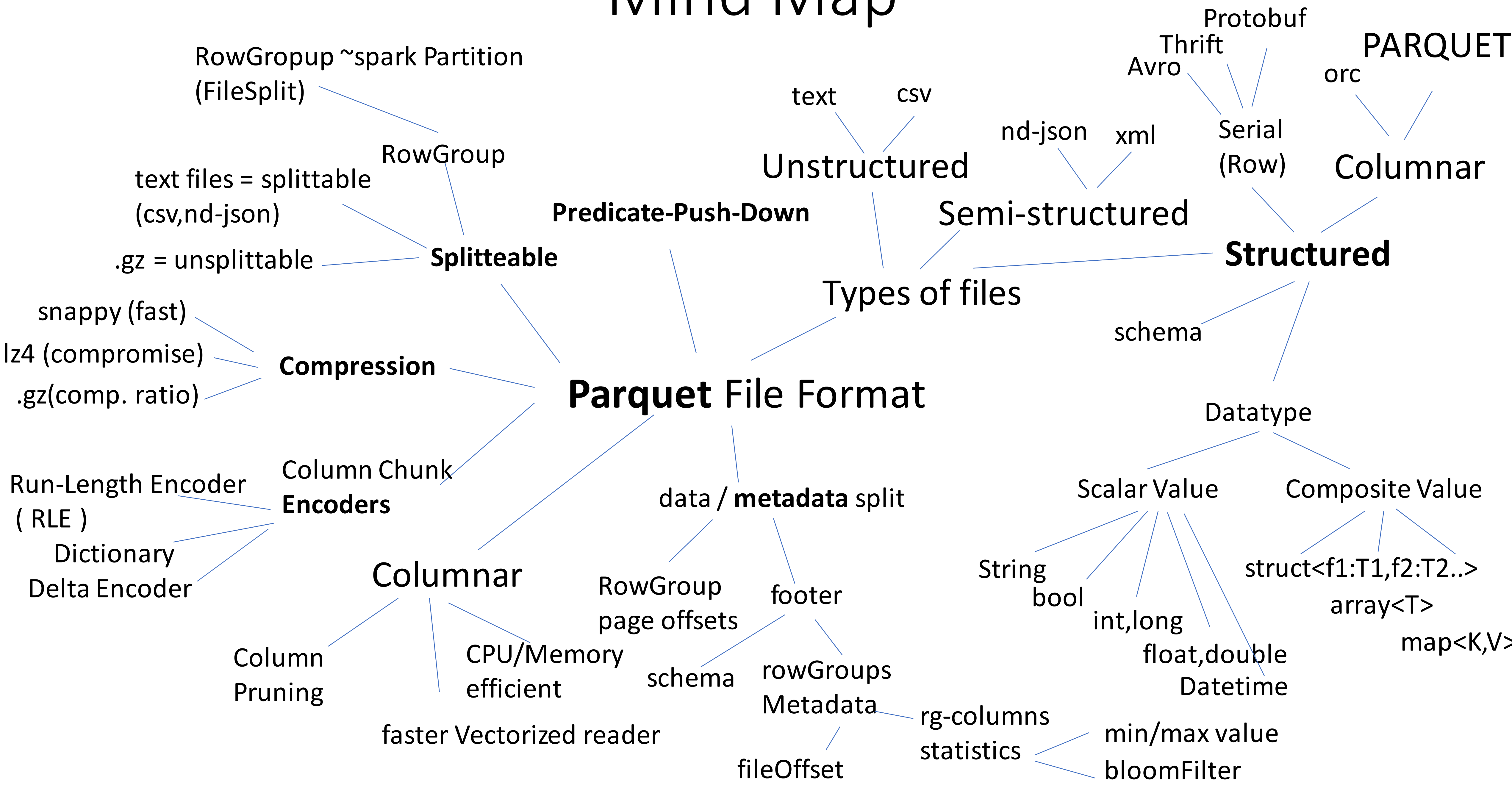
Conclusion

Parquet File Format is AMAZING

Spark is great using Parquet

Doing BigData processing = doing Spark + Parquet
with good .repartition().sortWithinPartition() ...

Mind Map


Parquet File Format
Types of files
Unstructured
text
csv
Semi-structured
nd-json
xml
Structured
Serial
(Row)
Avro
Thrift
Protobuf
Columnar
orc
PARQUET
schema
Datatype
Scalar Value
String
bool
int,long
float,double
Datetime
Composite Value
struct<f1:T1,f2:T2..>
array<T>
map<K,V>
Predicate-Push-Down
Splitteable
RowGroup
RowGropup ~spark Partition
(FileSplit)
text files = splittable
(csv,nd-json)
.gz = unsplittable
Compression
snappy (fast)
lz4 (compromise)
.gz(comp. ratio)
Column Chunk
Encoders
Run-Length Encoder
(RLE)
Dictionary
Delta Encoder
Columnar
Column
Pruning
faster Vectorized reader
CPU/Memory
efficient
data / metadata split
RowGroup
page offsets
footer
schema
rowGroups
Metadata
fileOffset
rg-columns
statistics
min/max value
bloomFilter

Questions?

arnaud.nauwynck@gmail.com