http://arnaud-nauwynck.github.io

Big Data – Part 4

Hadoop Ecosystem
HiveMetaStore, Parquet, IO Optims

arnaud.nauwynck@gmail.com

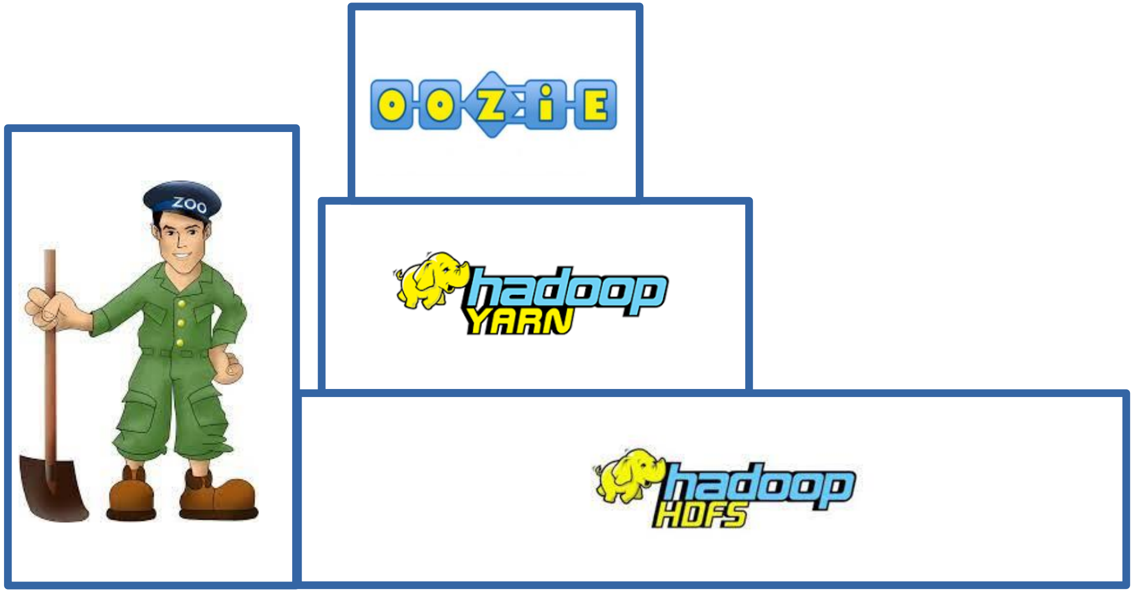
Outline

- Prev Part3: Low-Level Hadoop components
  - ZooKeeper, Hdfs, Yarn, Oozie
- Hive MetaStore
- Parquet
- IO Optims

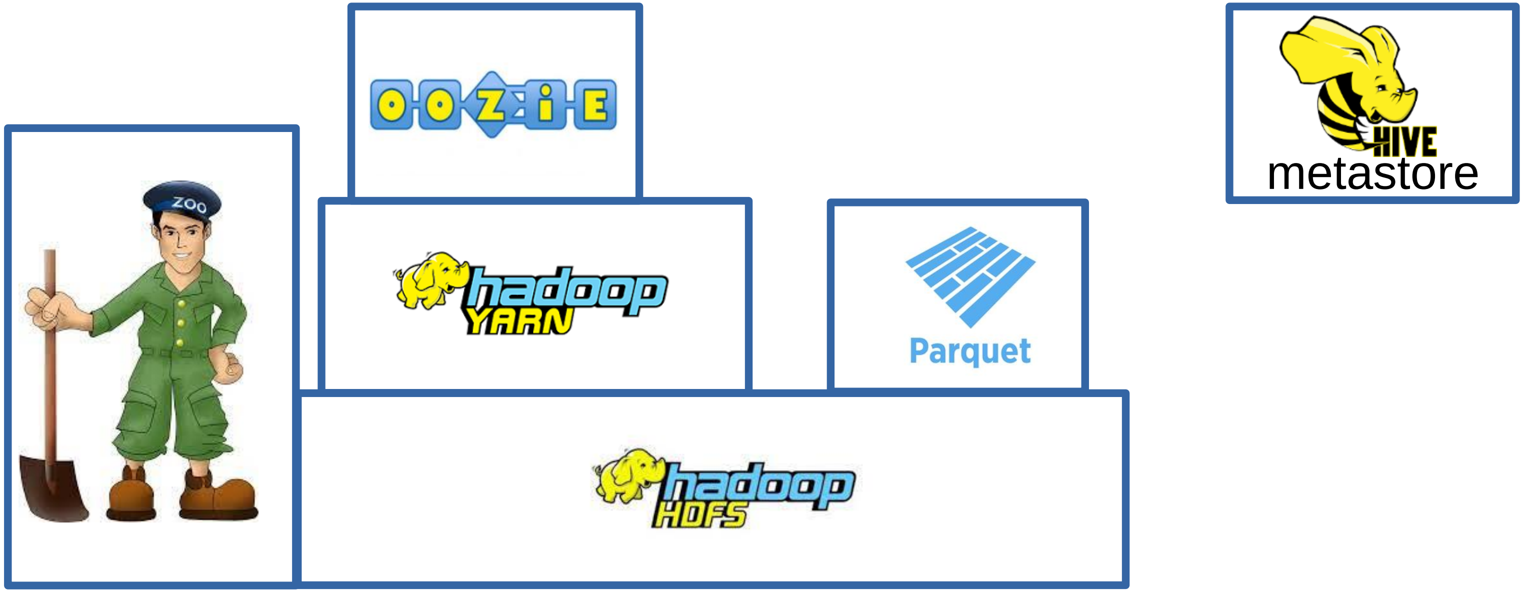
  Schema, Splittable blocks format, Partitions Pruning, Columns Pruning, PPD

Prev Part3: Low-Level Focus

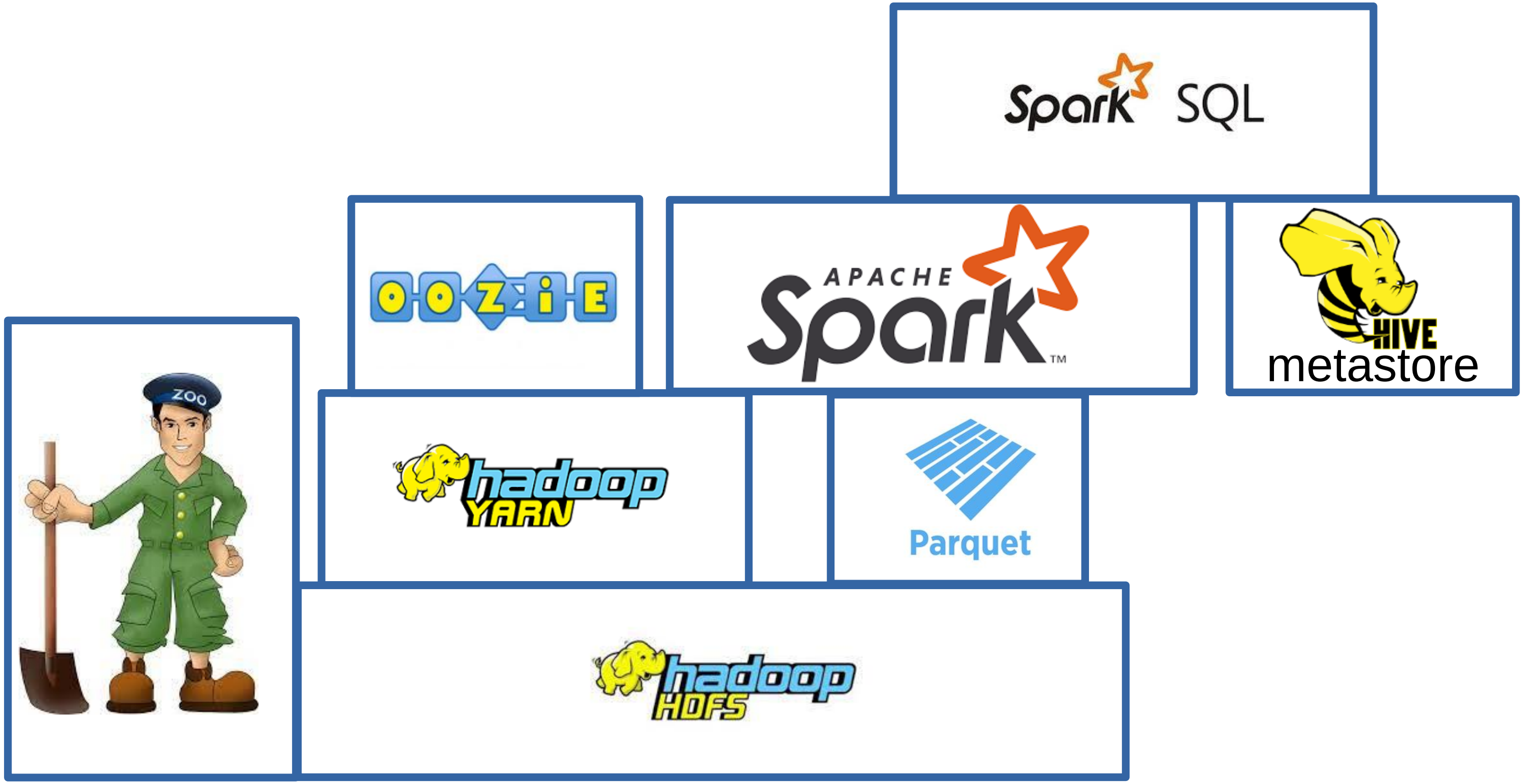
ZooKeeper, HDFS, Yarn, Oozie



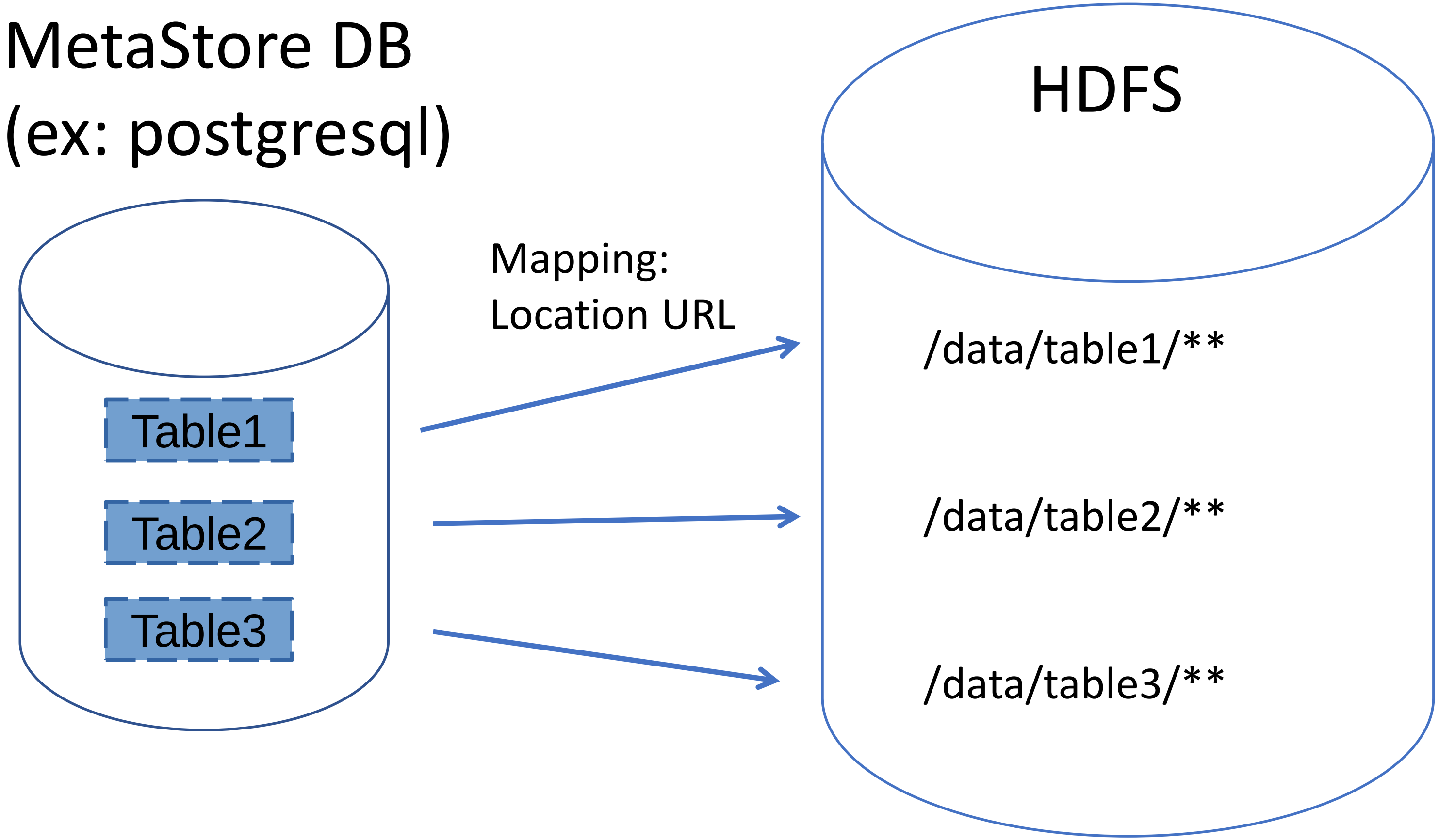
This Part: 4… Technical Focus
MetaStore, Parquet, IO Optims



Next Part 5 … High-Level Focus
Spark, Spark SQL



(Hive) MetaStore



MetaStore

Contains only **DDL** (Data Definition Langage)
metadata (no HDFS data)

Logical view mapping : **name in SQL ⇔ location in HDFS**

File format encoding: parquet, orc, avro, csv, json, …

Schema : **column types**

Sample CREATE EXTERNAL TABLE

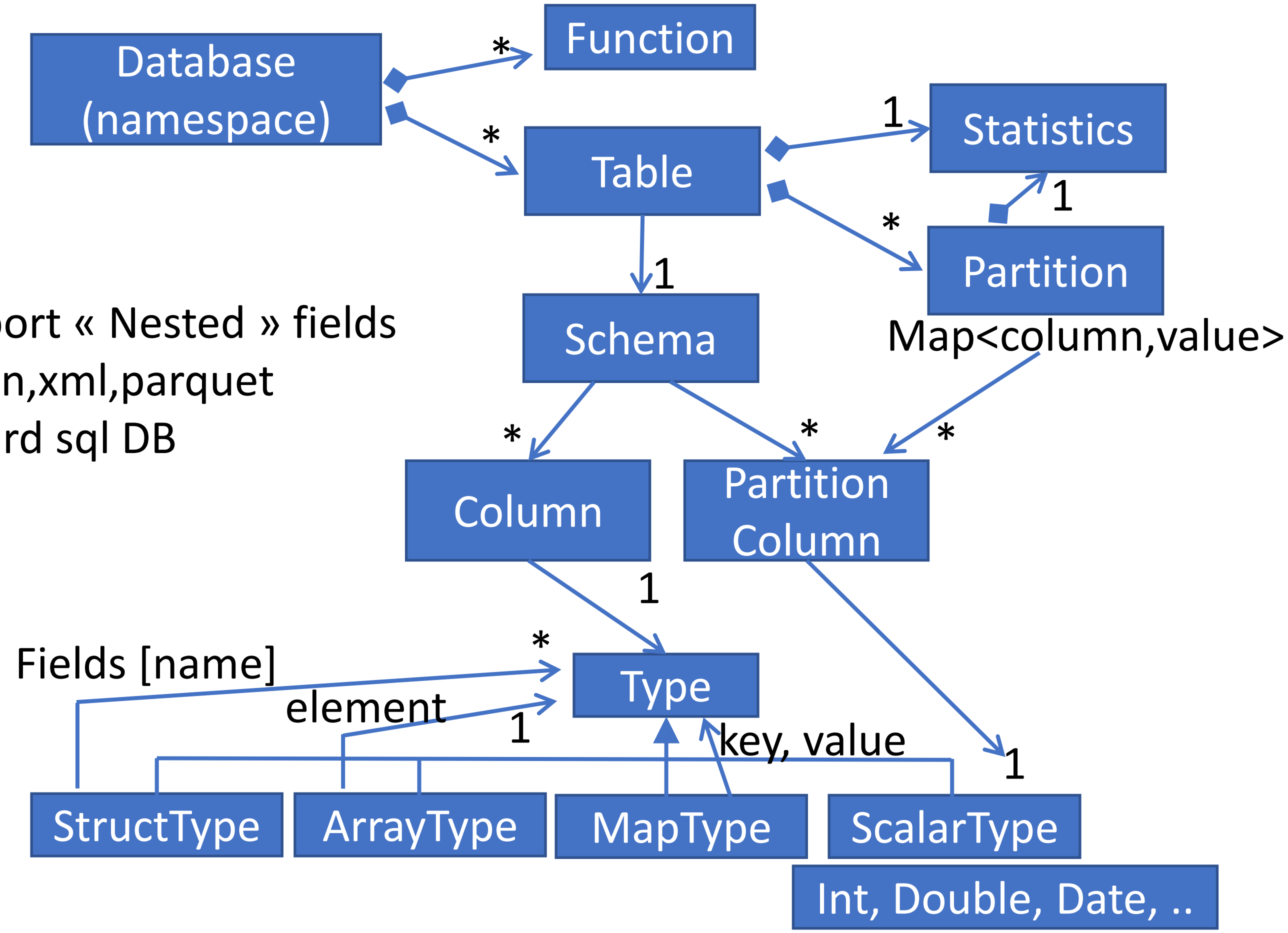
```
CREATE EXTERNAL TABLE db.student (
   id int,
   firstName string,
   lastName string
)
PARTITIONED BY (
  promo int
)
STORED AS parquet
LOCATION '/data/student'
```

Advanced CREATE EXTERNAL TABLE

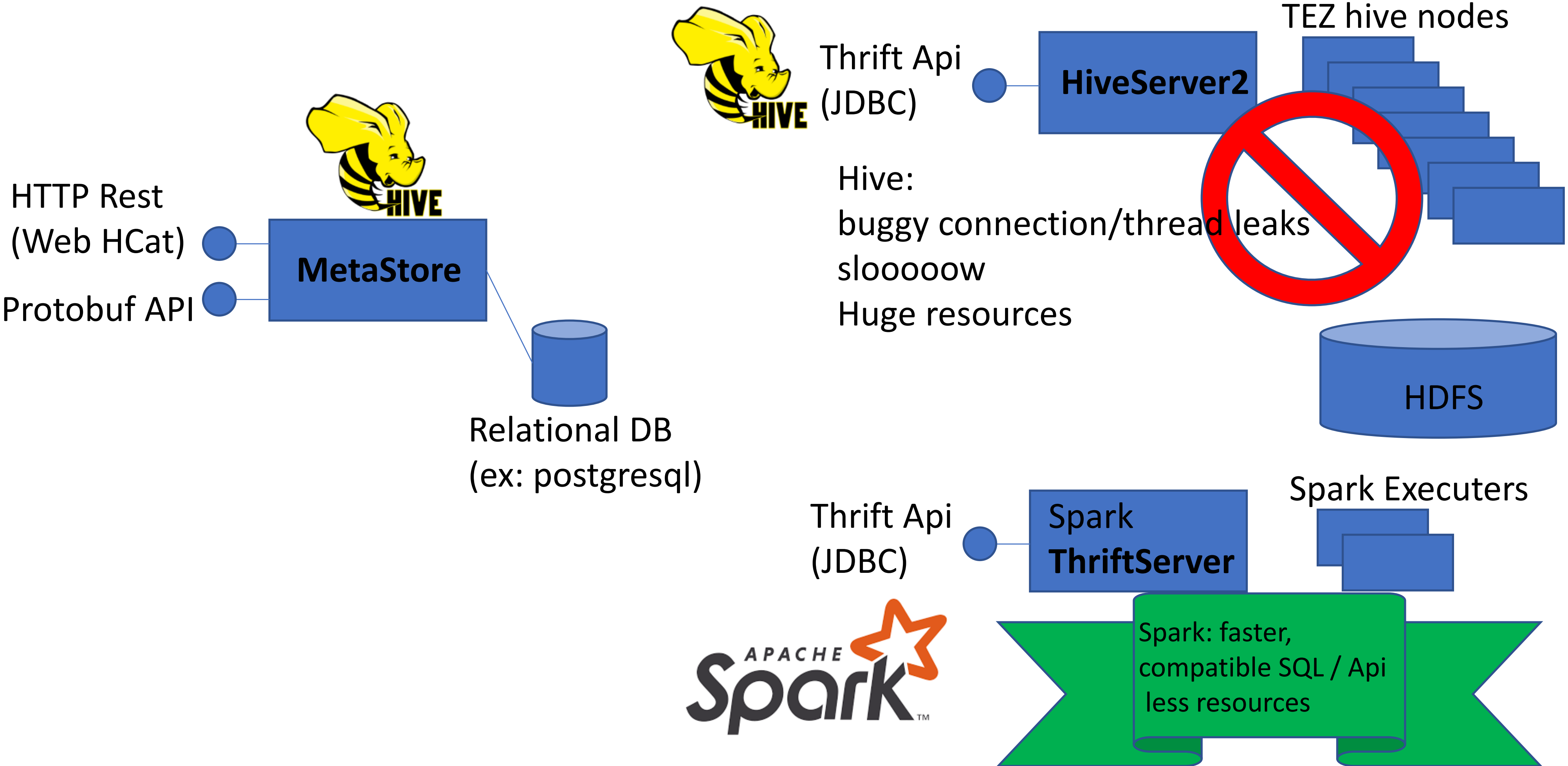
```
CREATE EXTERNAL TABLE db.student (
   id int, firstName string, lastName string,
   address struct< street string,number int,zipcode int >,
   graduations array< struct< name string, obtentionDate date > >,
   extraData map< string,string >
)
PARTITIONED BY ( promo int )
CLUSTERED BY ( id, ...)  SORTED BY (lastName, firstName )
STORED AS parquet
LOCATION '/data/student'
```

MetaStore Model

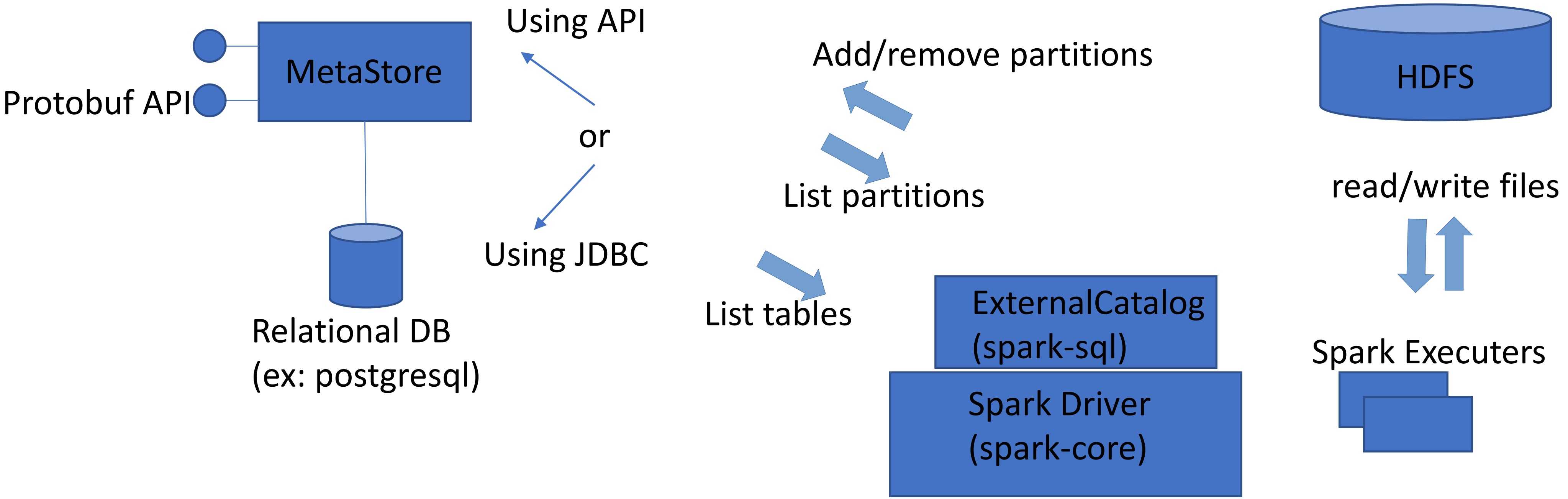
Schema support « Nested » fields
like typed json,xml,parquet
unlike standard sql DB



Hive MetaStore Architecture



Spark supports Hive MetaStore



Sql> DDL

```
Sql>
show databases;
use ‘db’;

show tables in ‘db’;
show tables in ‘db’ like ‘s*’;
describe table db.student;
show create table db.student;

alter table db.student set location ‘/data/student2’;
drop table db.student;
```

DDL.. EXTERNAL table

« EXTERNAL TABLE » : data exists independently of metastore

when creating table ... Schema must be compatible with existing files
Non-sense to « alter table » for column
When dropping … files are not deleted

Do not use opposite « MANAGED TABLE »
When creating => create empty dir, location= « {db.location}/{table} »
When dropping => delete all files !

Sql> DML

Sql>

INSERT INTO table values(..)

=> save to new file(s) !!

preserve existing ones

(also preserve partially uncommited ones..)

INSERT OVERWRITE / DELETE

=> reload all files

+ save all to new files

+ delete old files

Sql> Update? DML

by default Spark 3.x does NOT support UPDATE
(nor UPSERT, MERGE)

Only with extensions of « DeltaLake », « Iceberg », ..

DELTA LAKE

ICEBERG

Spark> Update?
read().map().write()

Full Scan ALL files
Load ALL
in-memory

```
spark
    .read().format(« PARQUET »).load(« /data/table1 »)

    .map( x -> {   …transform row to 'update' values; return newRow } )

    .write().format(« PARQUET »).mode(SaveMode.Overwrite).save(« /data/table2 »)
```

Process ALL
In-memory

Delete ALL files
+ save ALL
in-memory

Sql> … NO « ACID »

~~A~~ tomic

~~C~~ onsistent

~~I~~ solated

D urable

Granularity of insert (append / overwrite)

Write a single ROW ➡ in 1 new **File**

HDFS hates Small Files

(Too many files) !!

Write from shuffled RDD (several executors) ➡ in 200 **Files**

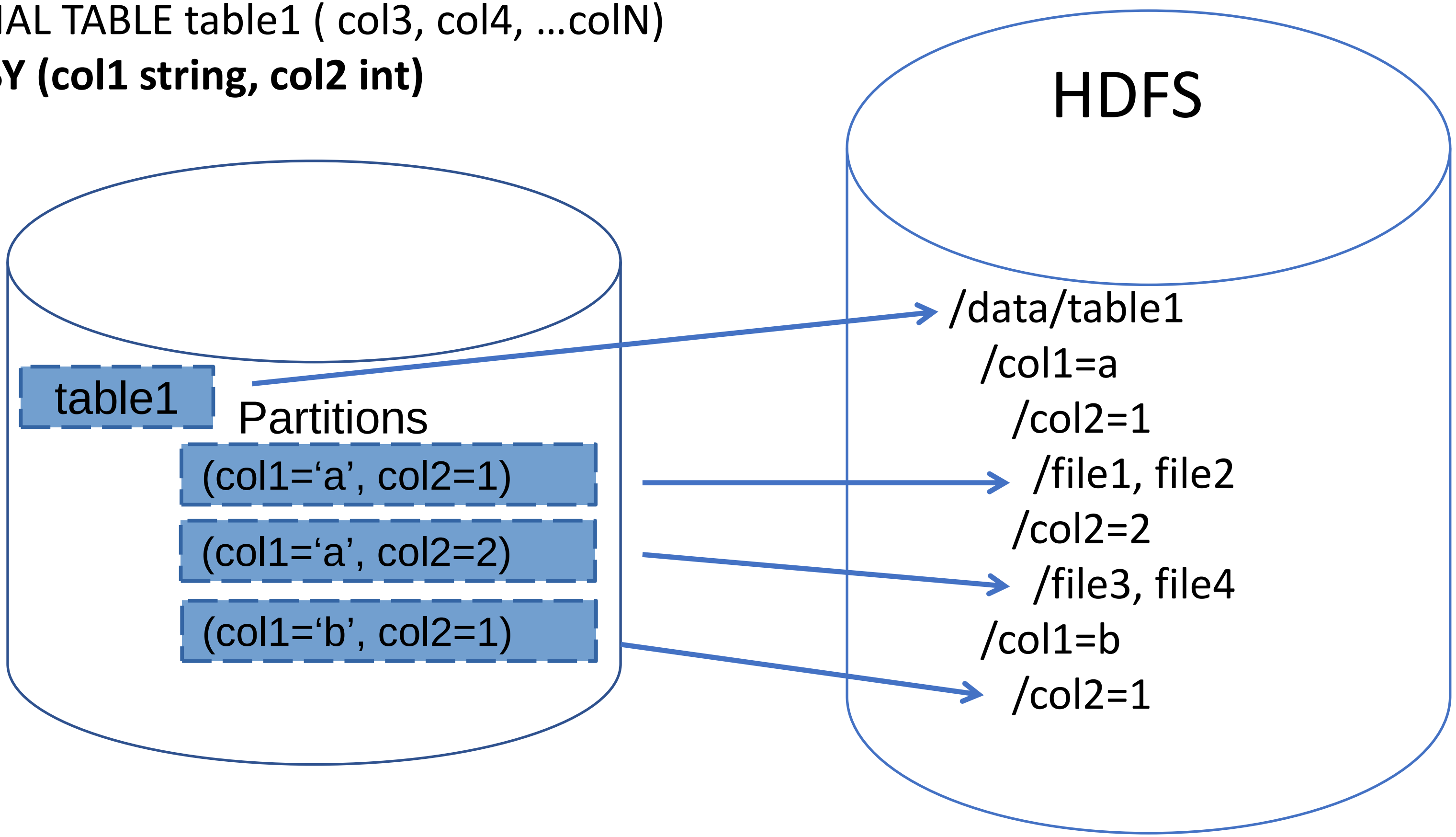
by default

spark.sql.shuffle.partitions=200 !!

Overwrite some files, and no touch others ➡ Possible only by **partition**

PARTITIONED BY (col1, col2)

CREATE EXTERNAL TABLE table1 (col3, col4, …colN)
PARTITIONED BY (col1 string, col2 int)



Alter table ADD PARTITION / MSCK REPAIR TABLE

Need EXPLICIT add !!
Otherwise dir/files not scanned => 0 result

Sql>
ALTER TABLE .. **ADD PARTITION** (col1='a', col2=1);
… Or
MSCK REPAIR TABLE ..; -- (inneficient rescan all)

Discover.partitions ??
… False good idea

ALTER TABLE … SET TBLPROPERTIES ('discover.partitions' = 'true')

hive-site.xml
  metastore.partition.management.task.frequency=600

  … => INNEFICIENT : Polling metastore thread every 10mn to scan HDFS, and alter
  + Spark still using explicit partitions

What if you have Peta bytes, with millions of dirs?

Optim: Partitions Pruning

Sql> select … from db.student where promo=2020 and …

Condition on partitioned column

Scan only files in

/data/student/promo=2020/**

Skip others

/data/student/promo=2019/

/data/student/promo=2018/

…

Partition: what for ?

NOT (Not-only) for searching faster !!!

(worst than parquet Predicate-Push-Down)

Granularity of Save mode Overwrite

… adapt to your batch scope

DO NOT define too (>2) many partition levels

Example Batch – Partitioned save

/data/table1/date=2021-12-25/

Partition for today

batch today scope=x

/sub-batch-scope=x/

/sub-batch-scope=y/

(optional) sub-partition
for sub-batch scope

batch today scope=y

/data/table1/date=2021-12-24/

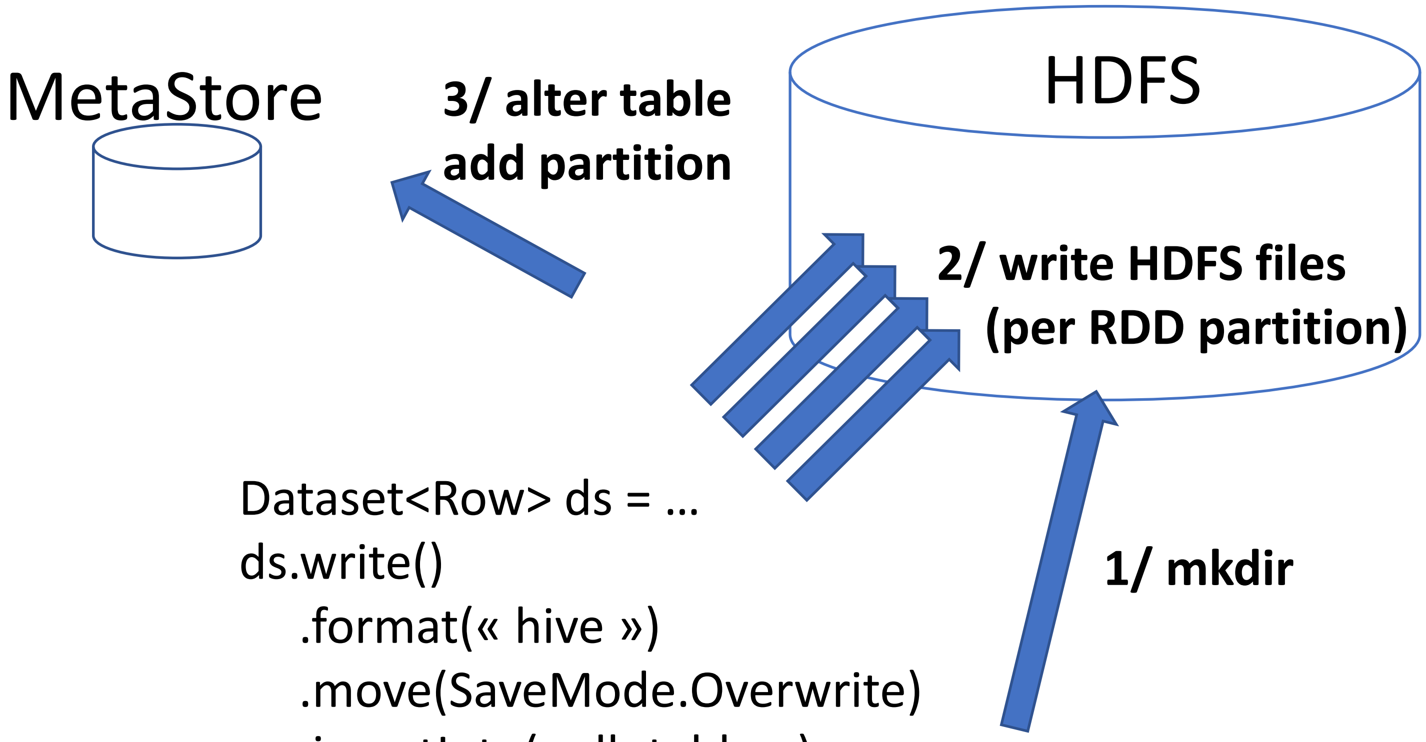
Already computed
from yesterday's batch
(do not update)

/data/table1/date=2021-12-23/

…older history

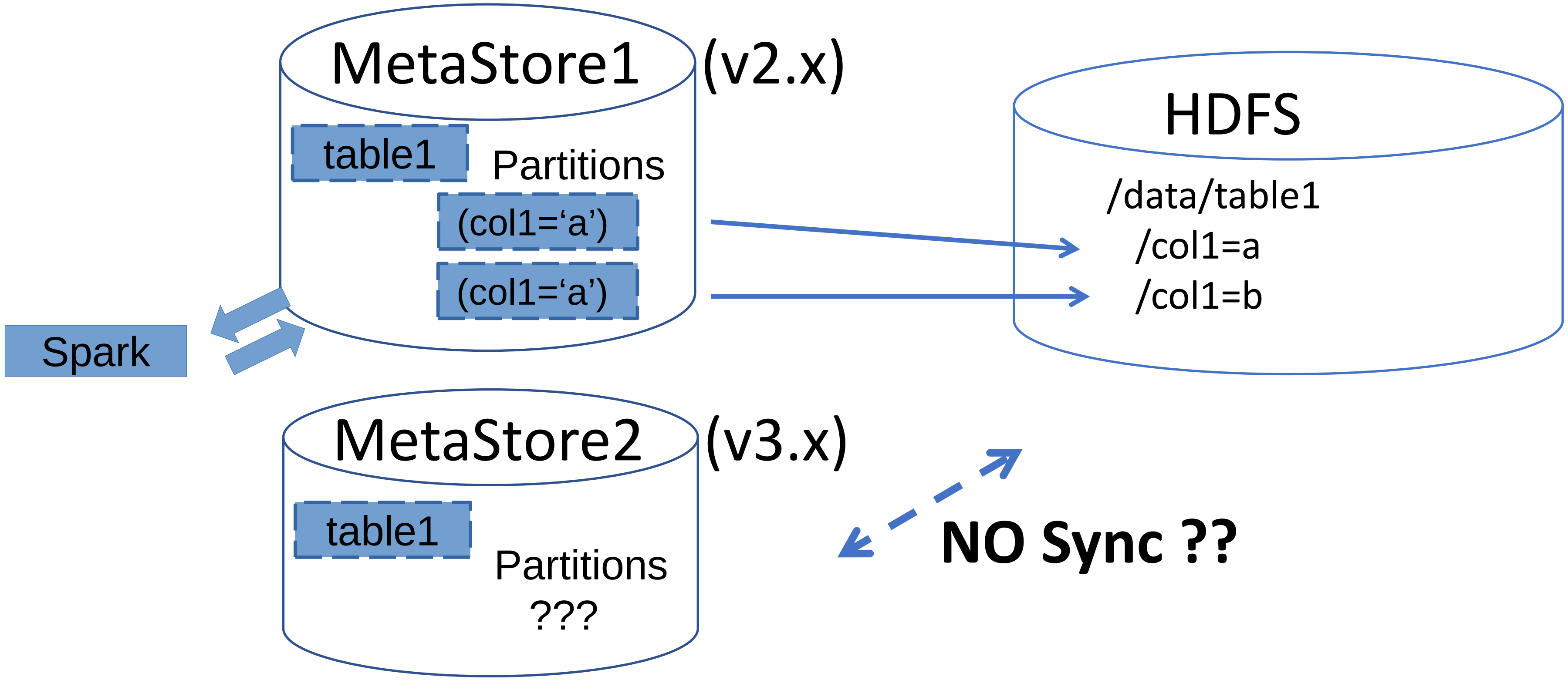
Immutable history

Spark .save()
=> mkdir + write Files + add partition

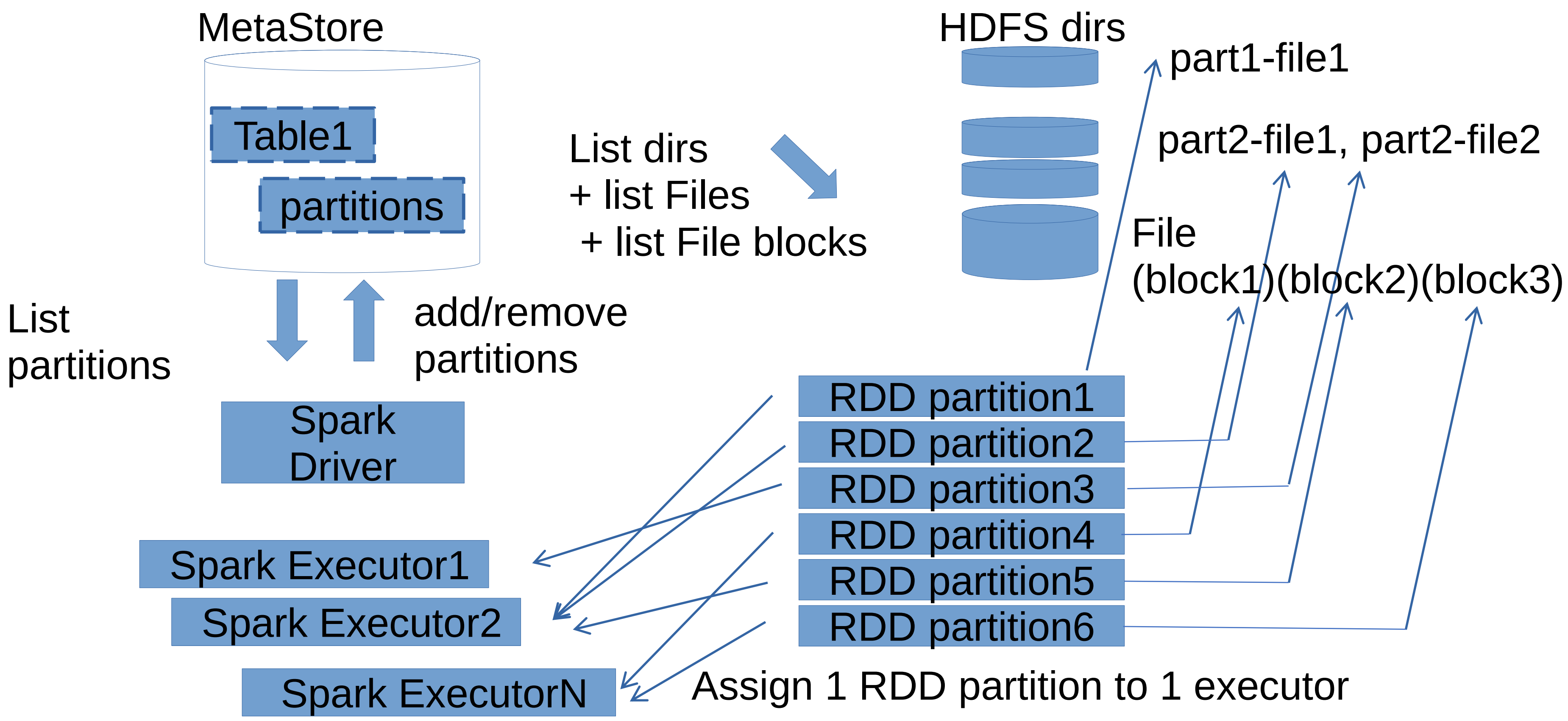



```
Dataset<Row> ds = …
ds.write()
    .format(« hive »)
    .move(SaveMode.Overwrite)
    .insertInto(« db.table »);
```

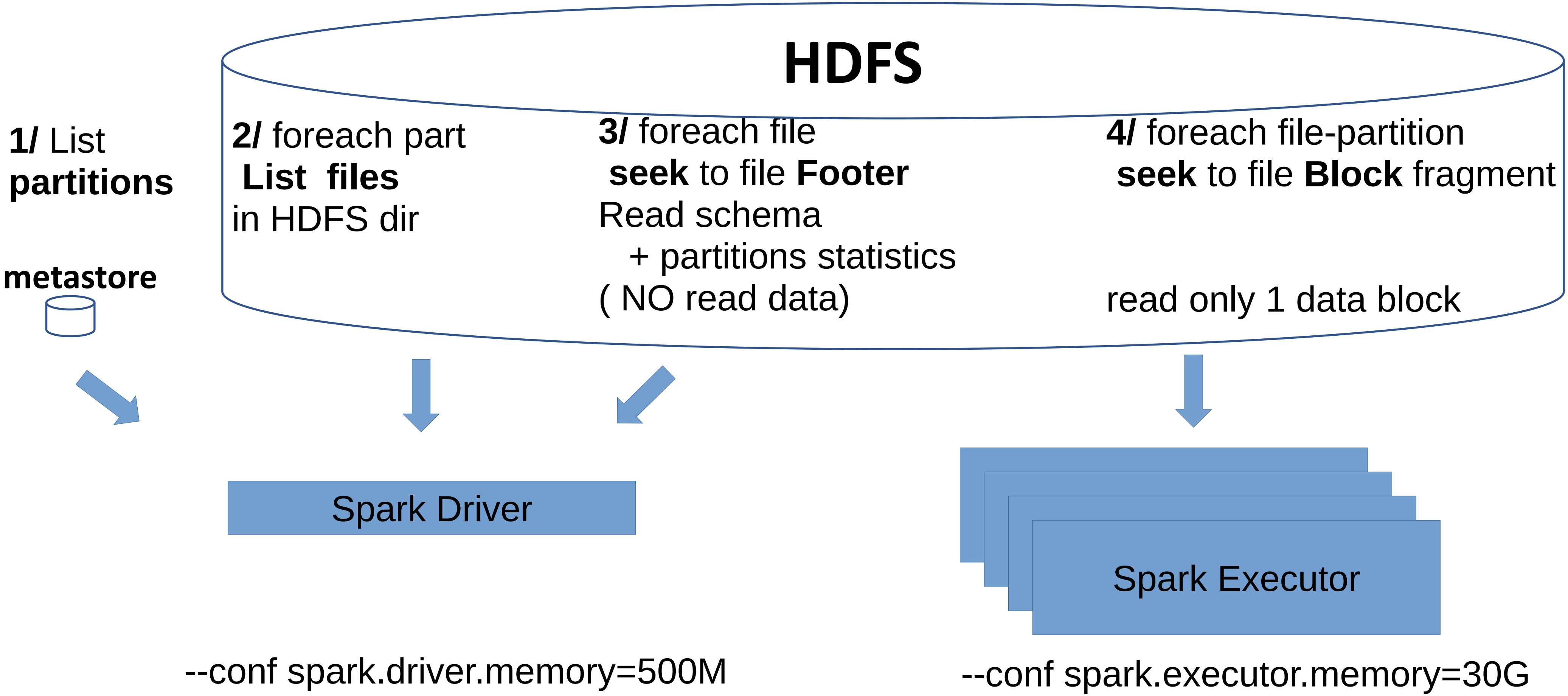
Synchronize HDFS with several MetaStores?



Spark RDD Partitions >> MetaStore Partitions

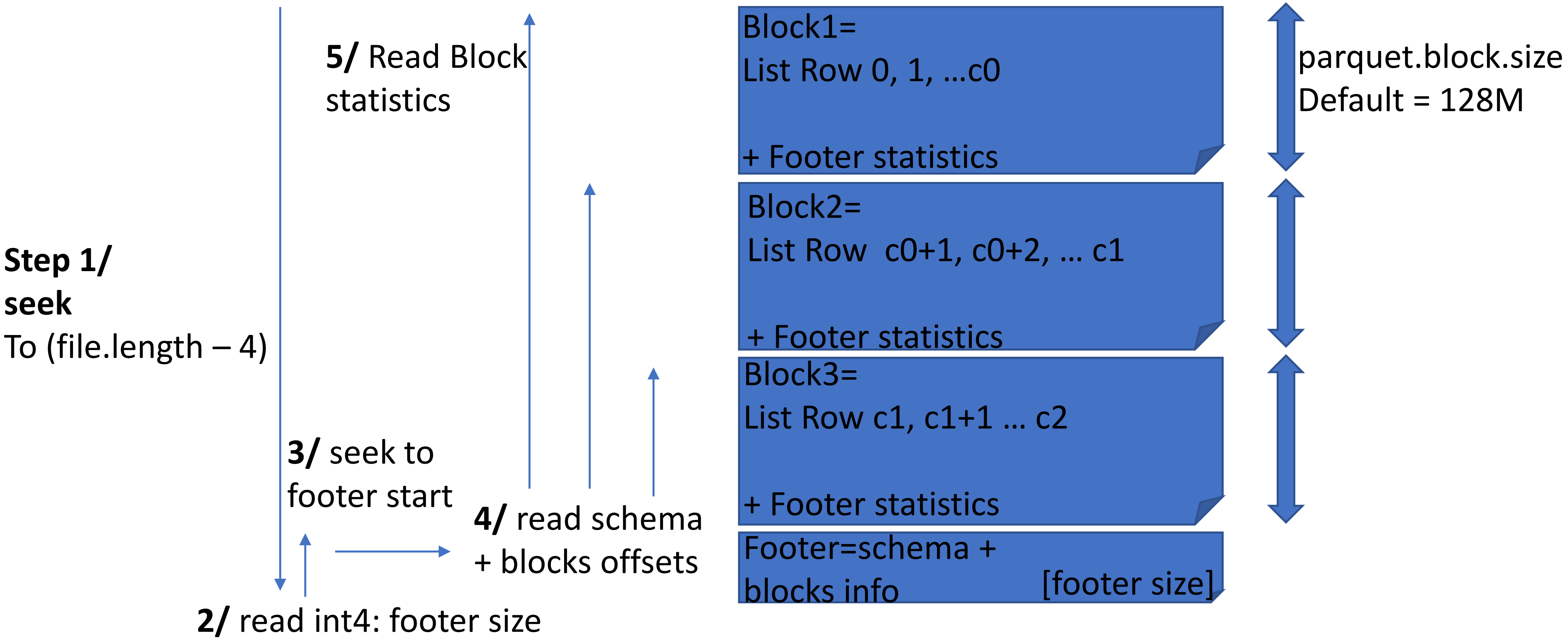


Spark RDD Partitions
= MetaStore Partition * Files * Blocks



PARQUET File Format

Splitteable File Format

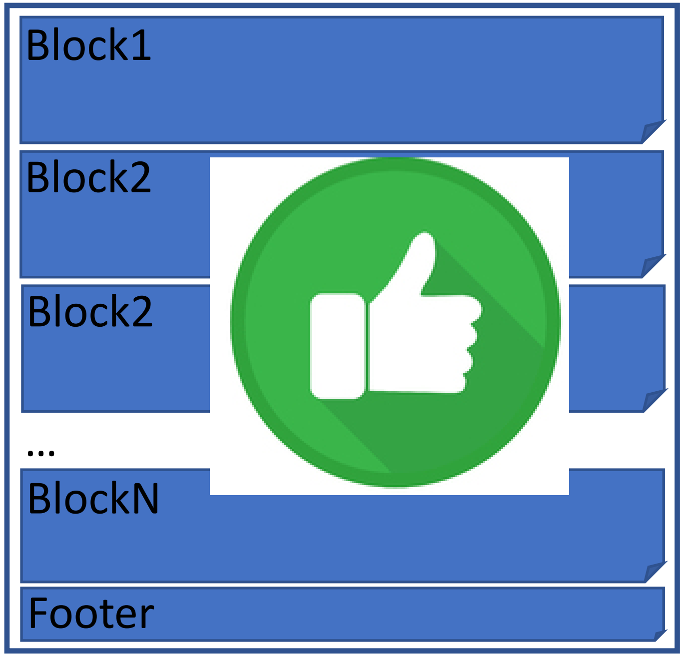


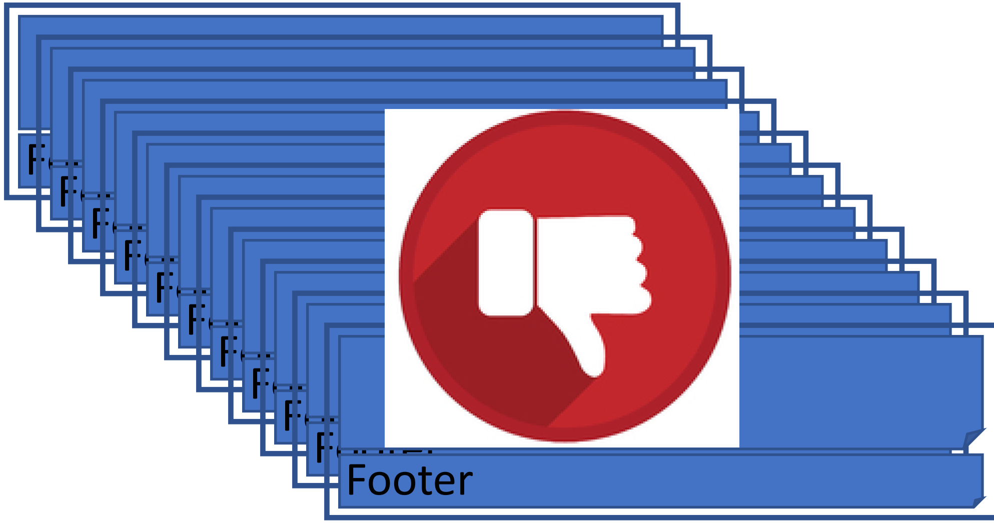
Performances
File Blocks >> MetaStore + HDFS Dir + Files

Better to
have 1 Huge HDFS file
(several Go)

than

Too MANY
Too Small files
(few 128+1 Mo)





Typical Partition / Files Volumes

For daily batch

1 partition per day … 5 year of data = ~1500 partitions OK

1 file per partition … OK, even if strange to have 1 file per directory

(maybe 2,3 files per partition … if no fit in spark executor mem)

File may be >= several Giga bytes …. OK great

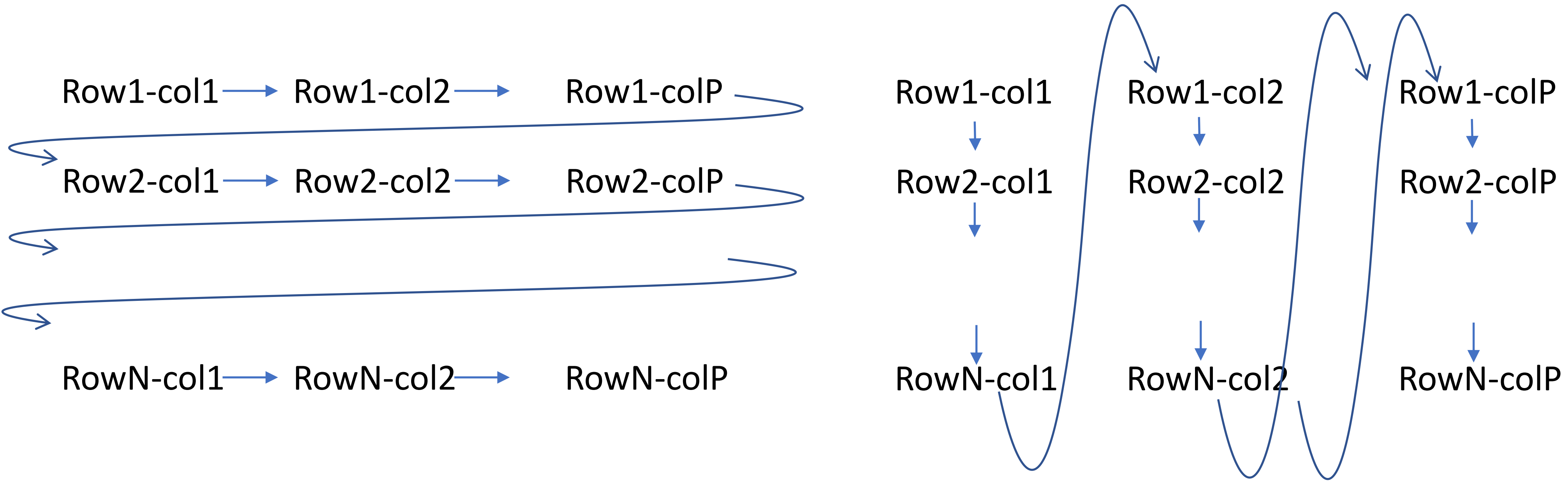
File parquet.block.size = 16M, 32M (? overwrite default 128M)

compromise:
Smaller => more dictionary encoding,
better PPD, maybe less compression
Bigger => less partitions

« Columnar » Storage File

Content = List<Row> = row1, row2, .. rowN   *   Row=col1, col2, … colP

Classic (row-storage) file   Columnar-storage file



Why columnar ?

Read only needed columns data

Seek to skip unneeded ones

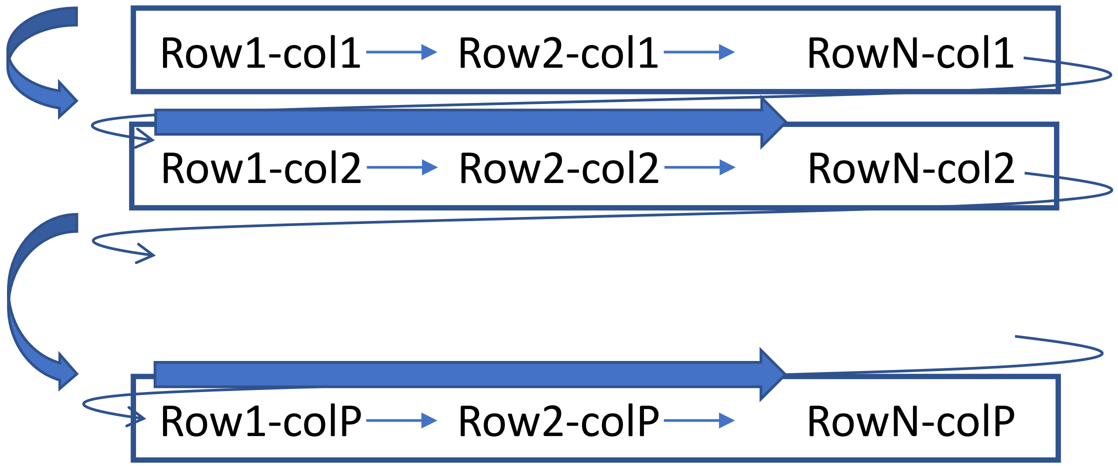
Example: SELECT col2, colP from …

1/ seek() to col2 offset
(Skip sequential bytes for col1)

2/ Full read col2

3/ seek to colP offset
(Skip bytes for col3, col4, … colP-1)

4/ Full read colP

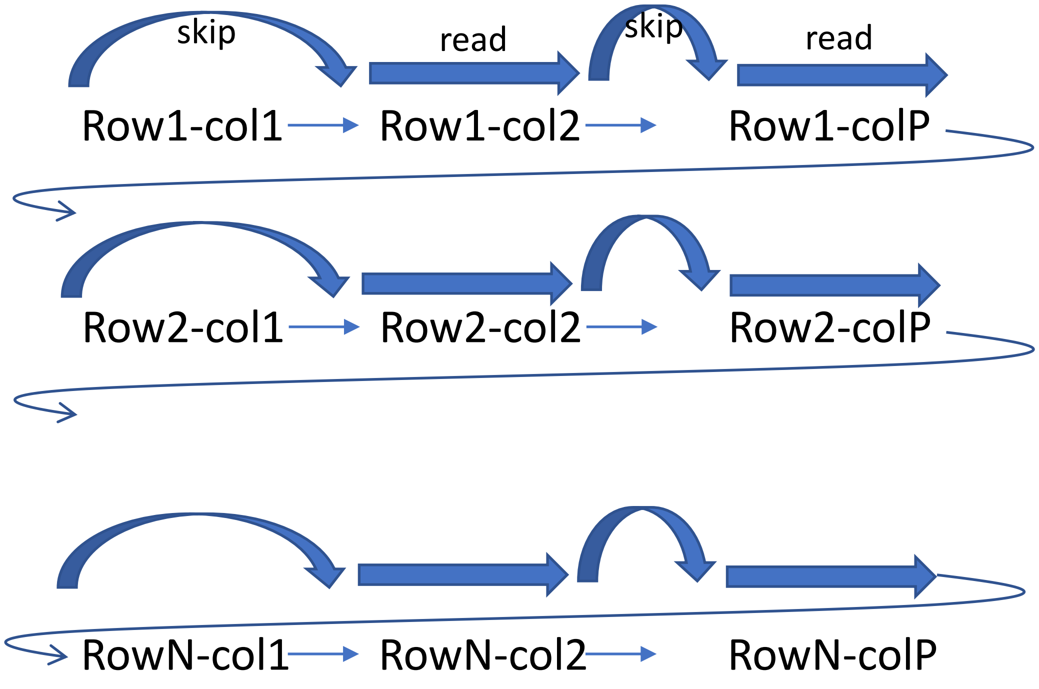


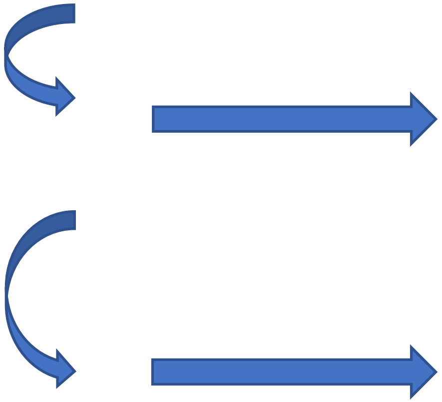
Comparison .. Full Read & Garbage

2*N skips
+ 2*N small unitary reads

vs

2 skips
+ 2 array reads





Much faster
Fewer data IO / fewer ops

Optim: « Column Pruning »

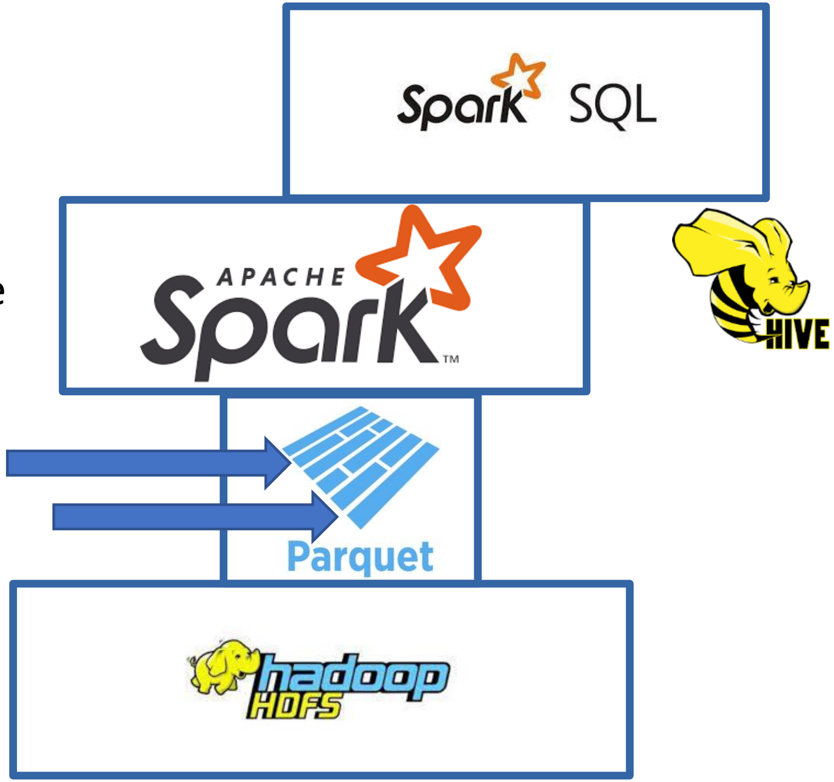
From SQL to Parquet IO .. Hadoop IO

Select col2, colP from table…
(prune all other columns)

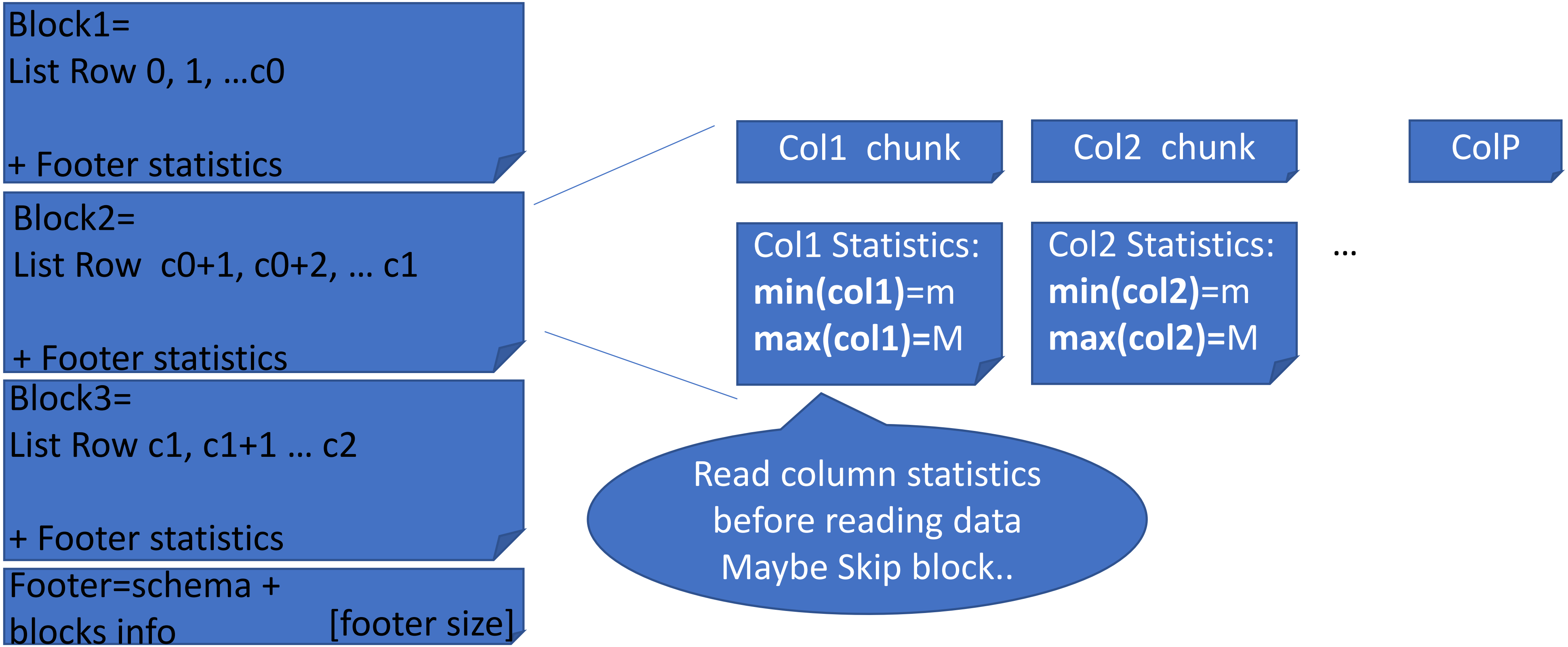
DataSet / RDD on Parquet file

Parquet API
to read columns chunks

Fewer IO



Last but not Least Optim
Using page-column statistics



Predicate… skip with statistics (maybe False Positive)

Example:

SELECT col2, colP FROM … WHERE **col3 = value3**

Col3 chunk

Col3 Statistics:
min(col3)=m
max(col3)=M

value3 ?

m <= ? < M

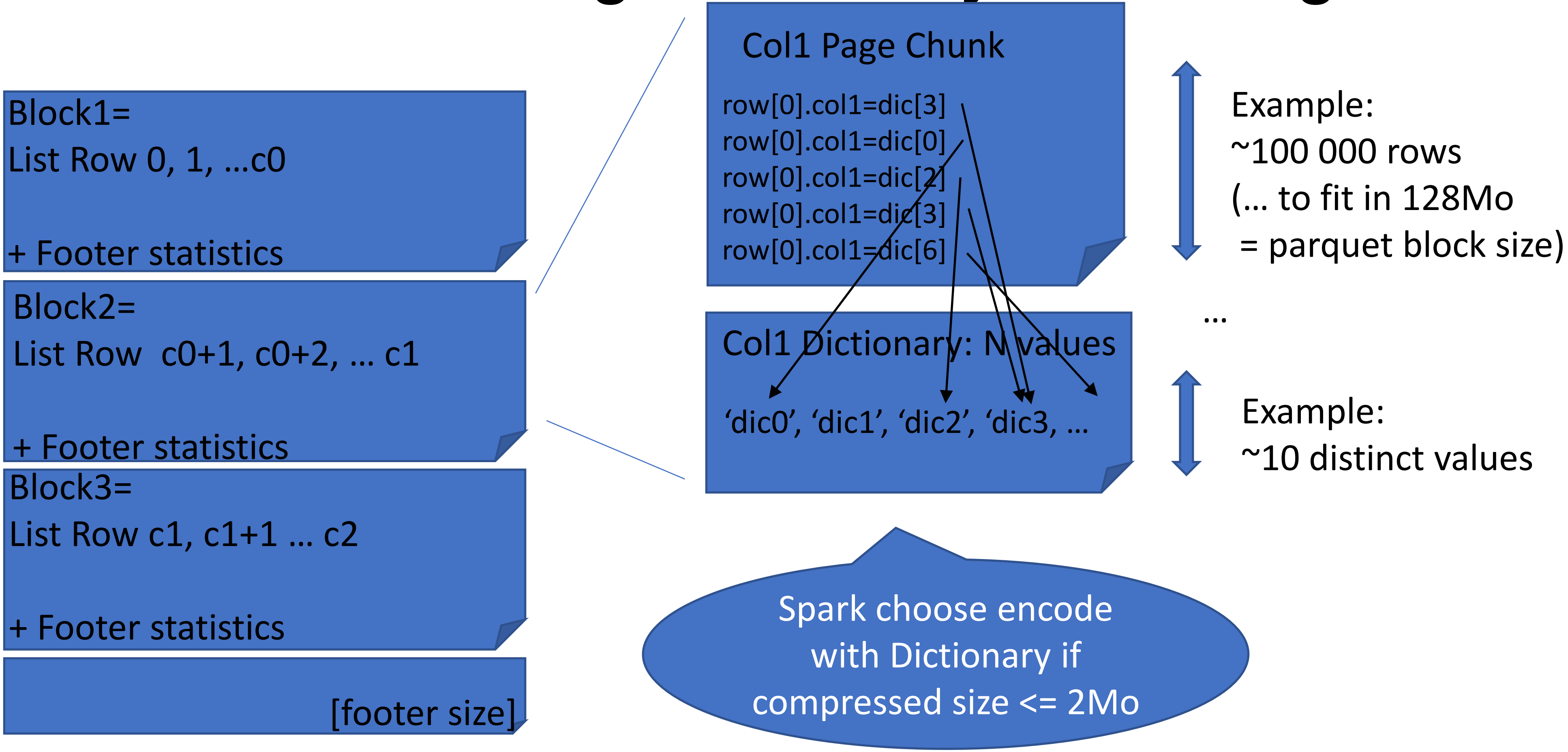
If (**(value3 < m) OR (value3 > M)**)

… AND check for null to please SQL semantic ?!

⇒Impossible to find row in this block

⇒Skip block!

Column with small number of distinct values … Stored using Dictionary encoding



Predicate Push-Down for « col=‘value’ » or « col in [‘value1’, .. ‘valueN’] »

Example:

SELECT col2, colP FROM …

WHERE **col3 = ‘value3’** and **col4 in [‘value1’, ‘value2’, value3’]**

For each page chunk of col3
If encoded as Dictionary
=> read dictionary
then if ‘value3’ not in dictionary
=> SKIP Row Group !!!

Bloom Filter: mask=Union(hash(..))

New in Parquet … (older in ORC)
statistics can also contain Bloom Filter masks

value

Hash{1..k}(value)

0000100001 in?
0100001010

Col chunk

Col Statistics:
min(col)=m
max(col)=M

Col Bloom masks{1..k}
0100110101101010
1000111010

Bitmask h = hash(value)
If (**(h & bloom) == h**)
… AND check for null to please SQL semantic ?!
⇒Impossible to find row in this block
⇒Skip block!

k hashes, m bits, n elements
=> False positive rate ~ $(1-e^{-kn/m})^k$

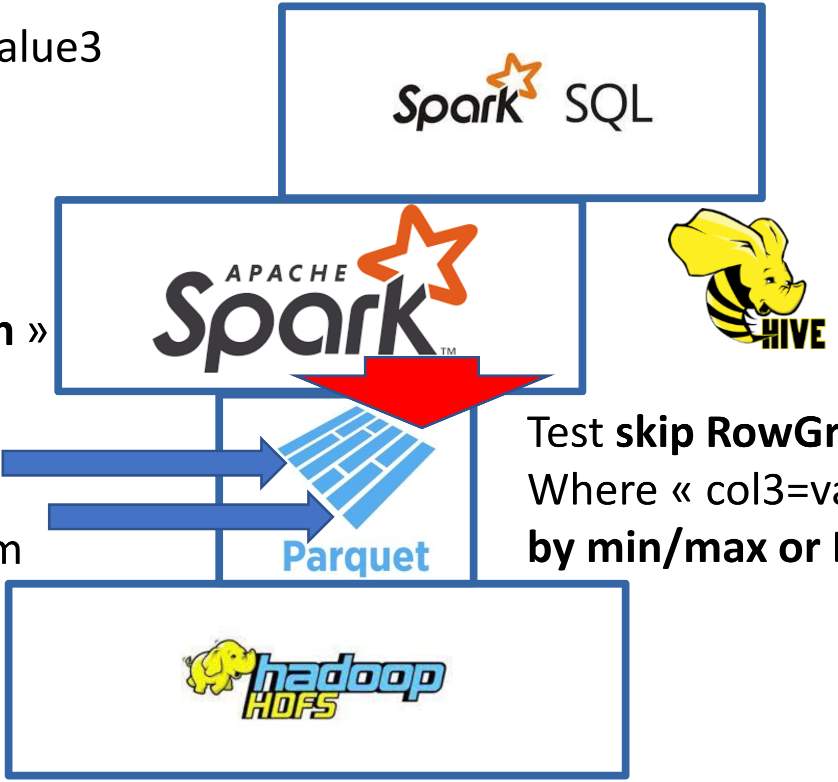
« PPD » : Predicate-Push-Down

Select .. from … where col3=value3

DataSet / RDD on Parquet file
WITH PREDICATE « **pushed-down** »

Parquet API
to read columns statistics+bloom

Fewer IO



Test **skip RowGroups**(=blocks)
Where « col3=value3 »
by min/max or Bloom

Sort + parquet.block.size
for better Predicate-Push-Down

When writting PARQUET files
… think to optimize reads later (PPD)

Example: id in range 1..1000 predicate id=542

Unsorted, Big block 128M

min=2 max=997	min=5 max=998	min=4 max=992
read	read	read

… value within min/Max of all blocks
=> NO skipped block … only False positives

Sorted + Small blocks 16M

min=2 max=201	min=202 max=350	min=352 max=501	min=502 max=593	min=595 max=992
skip	skip	skip	read	skip

How to « Write » parquet files : Adapt for best « Reads » later

Dataset<Row> ds = spark.sql(« … »);
// ds contains probably **200 partitions** (default value after a SHUFFLE)

ds = **ds.repartition(1);**   // equivalent to « .coalesce(1) »
    // or   ds.repartition(2) // or 3 … if RDD does not fit in spark.executer.memory !!

ds = **ds.sortWithinPartition(« colA », « colB »,   …   « colID »)**
    // sort by general columns first « colA » (example portfolio, region, productType…
    // last by « id » column

ds.write()**.format(« hive »).**mode(SaveMode.overwrite).**insertInto**(« db.table_name »);

Recap 5 Optimizations

1/ typed schema, binary encoding, dictionary + compression

2/ **splittable** file (blocks) = distributed

3/ Hive Metastore **Partition Pruning** = skip/scan dirs

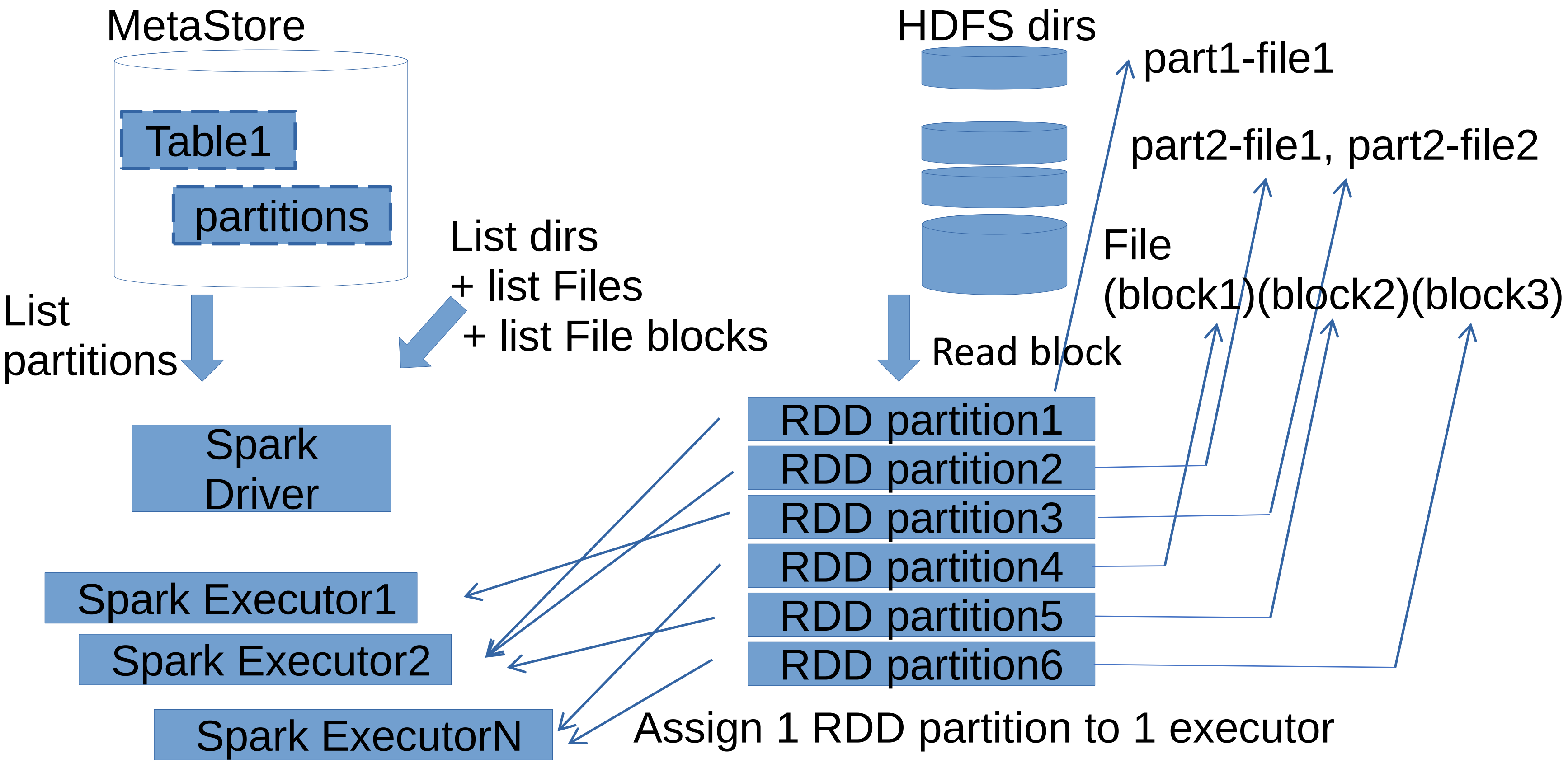
4/ **Column Pruning** (Columnar storage format) = seek + array read

5/ **Predicate-Push-Down** = skip using statistics, bloom filter

Recap Optimizations 1/5
Schema, Binary Encoding, Dictionary

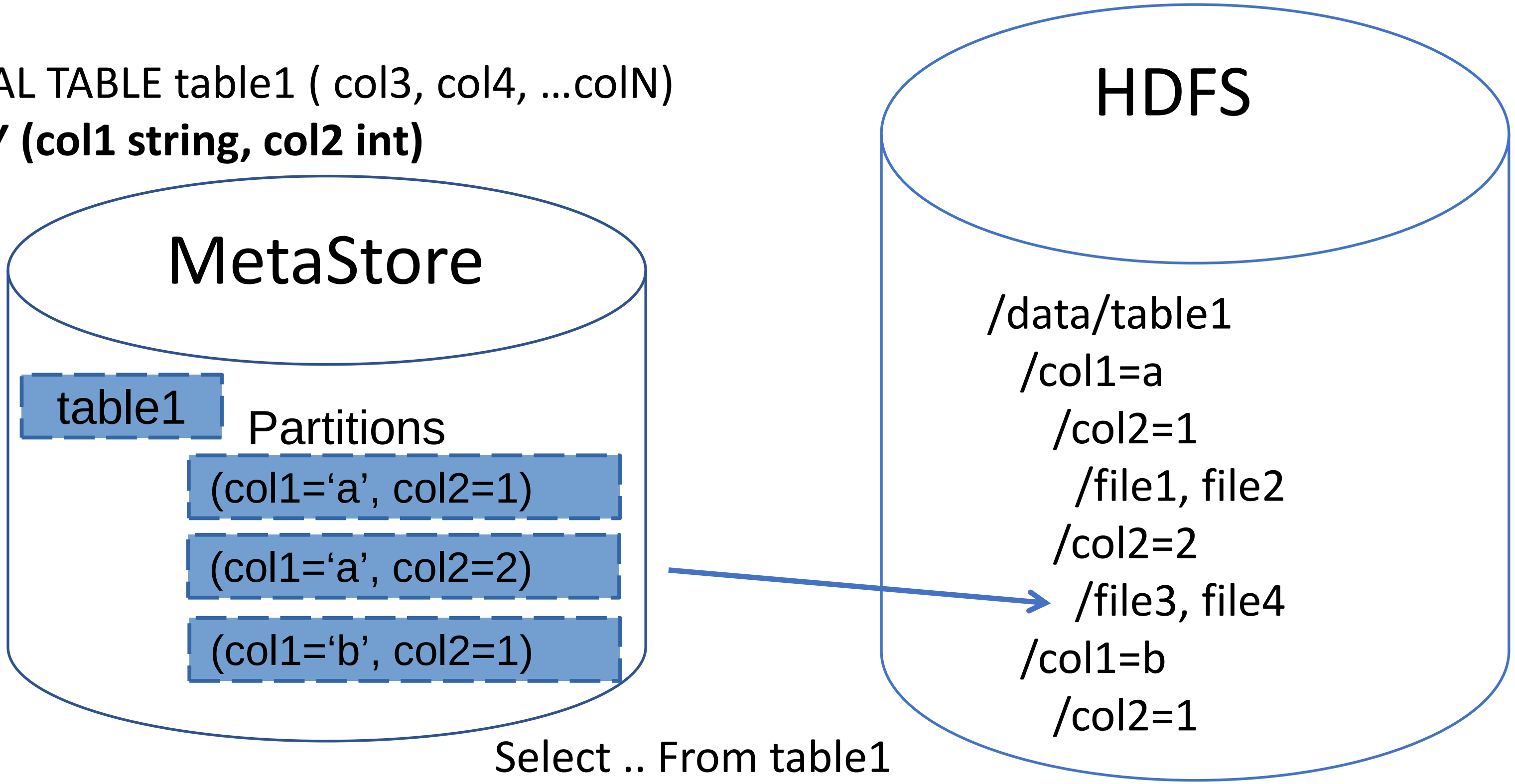
CSV, Xml, ND-JSON	Schema-less file formats ! … innefficient text encoding Redundant <xml> value</xml> or « json »: « value»
PARQUET, ORC	Strongly typed Schema embedded in file … efficient binary encoding Efficient incremental encoding, or Dictionary

Recap Optimizations 2/5
Distributed RDD: Splittable File Blocks



Recap Optimizations 3/5
Hive Metastore Partitions Pruning

CREATE EXTERNAL TABLE table1 (col3, col4, …colN)
PARTITIONED BY (col1 string, col2 int)



Select .. From table1
Where col1=a and col2=2
-- partitioned columns

Recap Optimizations 4/5

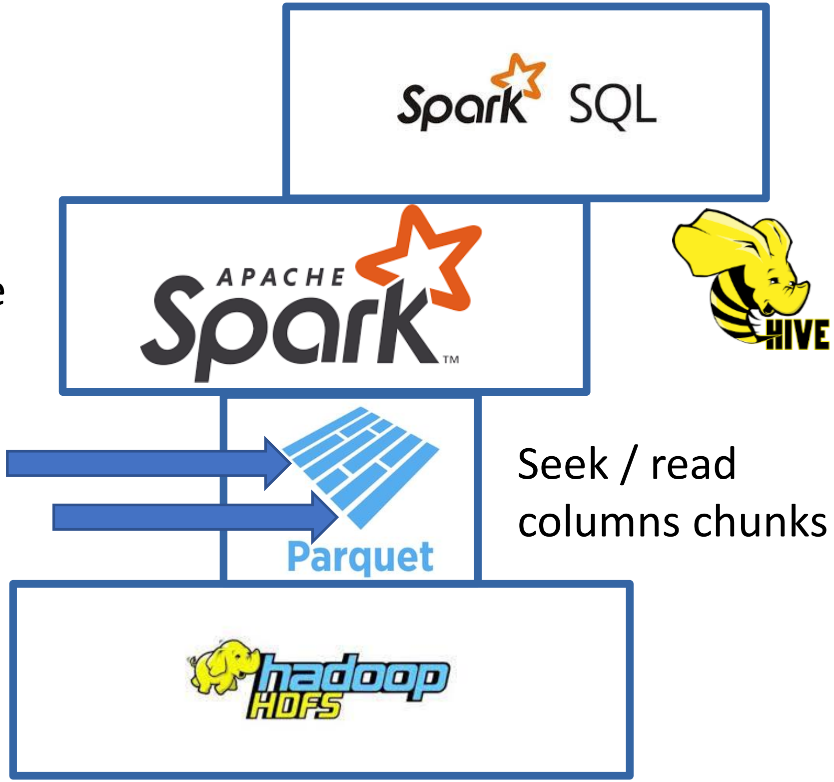
Columns Pruning
(seek in Columnar Format)

Select col2, colP from table…
(prune all other columns)

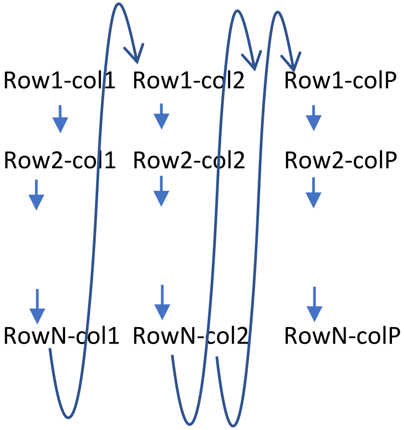
DataSet / RDD on Parquet file

Parquet API
to read columns chunks

Fewer IO



Seek / read
columns chunks



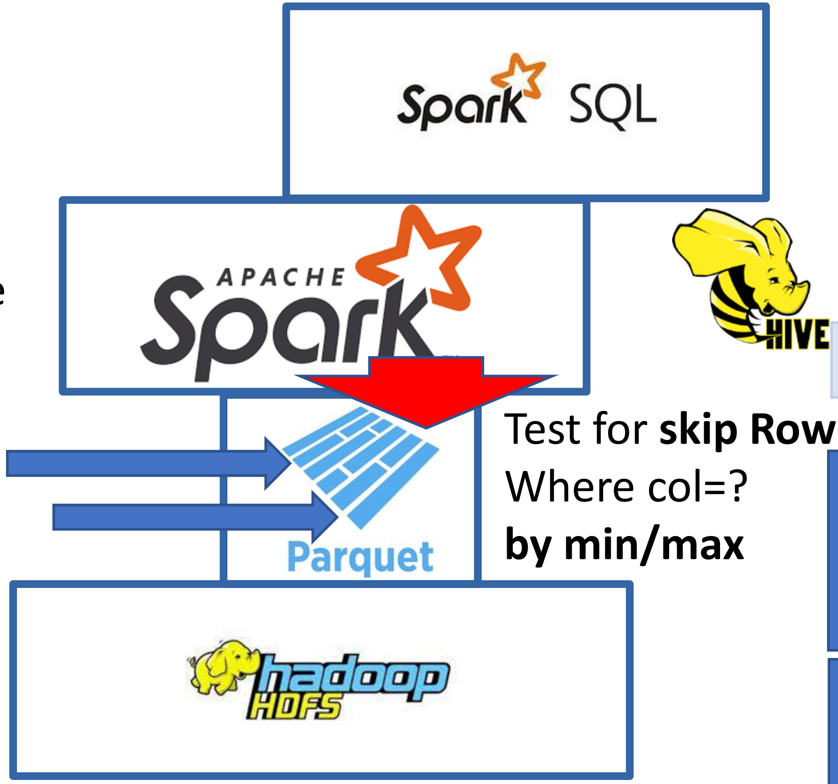
Recap Optimizations 5/5
PredicatePushDown (min-max statistics/Bloom)

Select col2, colP from table…
(prune all other columns)

DataSet / RDD on Parquet file

Parquet API
to read columns chunks

Fewer IO



Col chunk

Test for **skip RowGroups**
Where col=?
by min/max

Col Statistics:
min(col)=m
max(col)=M

Col Bloom masks{1..k}
0100110101101010
1000111010

Next… part 5

Spark