Introduction to BigData Processing

(Distributed Operations on Spark Datasets)

cours 2024

arnaud.nauwynck@gmail.com

This document:
https://github.com/Arnaud-Nauwynck/Presentations/bigdata/
1-Introduction-Spark-Distributed-Dataset-Operations

Outline

- from List<T> to distributed **Dataset<T>**
- Immutability, Functional API
- processing workflow:
  Input -> Transformations -> Output
- **narrow** operations (=per partitions)
- **wide** operations (=shuffled)

Distributed Processing Goal :
Handle Tera << Peta Bytes << ... of Data
but
on Commodity hardwares cluster (Giga of RAM)

RAM Compute Resources
to process
in Giga Bytes

Storage Files to process
in Giga/Tera Bytes

RAM
= **Giga** Bytes

Disk
= **Giga(SSD) / Tera** Bytes

Data Lake
= **Peta** Bytes

What are the Top #4 Challenges ?

Challenge #1 (most difficult) =

Manage Failures (be Safe/Resilient)
in a Fragile Distributed Sub-Systems

Challenge #2 (most obvious) =

Scale to Huge Data limits

even with restricted Resources

Challenge #3 (most differenciating) =

Be **Fast / Efficient** at using & sharing resources

Scale to CPUs at clusters level

Make compromises CPU/RAM/Network/Disk

Challenge #4 (for success) =

Keep Things Simple

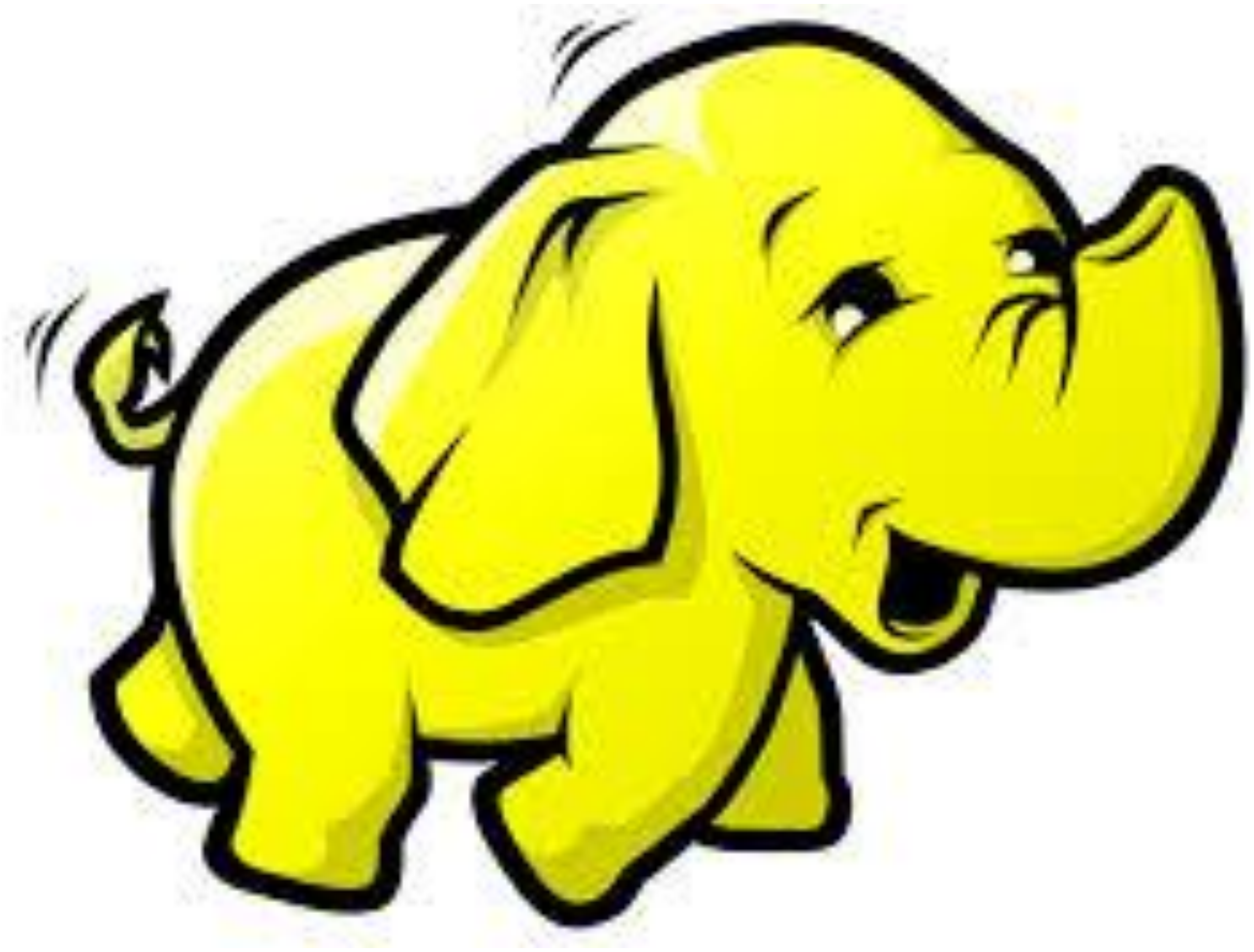
Architecture for Open / Powerfull / Wide / Simple

become a Standard

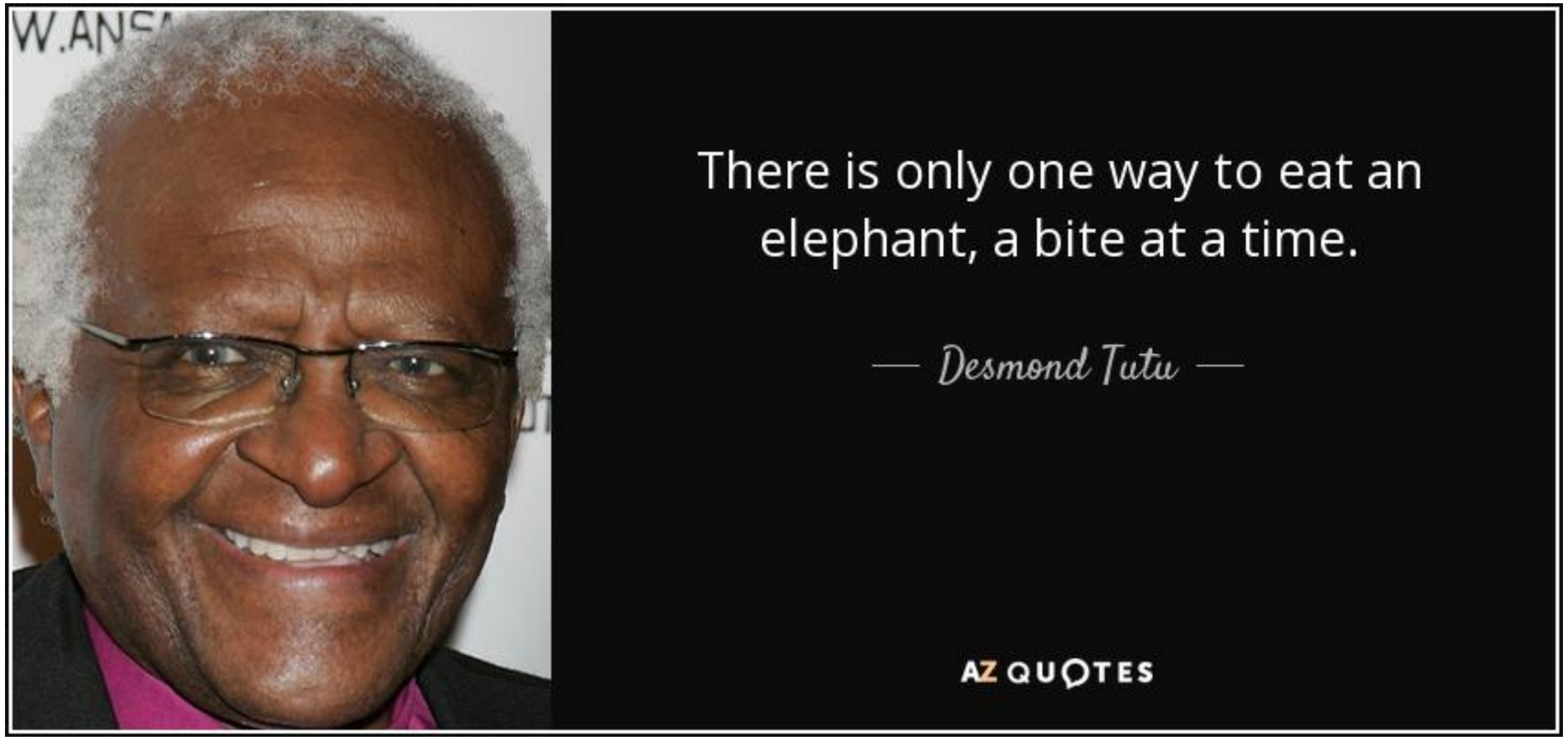
Traditional Databases vs BigData

Traditional OLTP Databases	**BigData Processing**
Interactive	Batches
ACID Transactions	NO "per-row" Transactions (NO Update/Delete)
Use SAN disks	Use HDFS (distributed storage),
mostly Scale vertically	Scale Horyzontally
expensive single hardware	cluster of N x commodity hardwares
Tables = optimized structures by DB	**Tables = basic directory + files**
B-Tree	**basic Lists (i.e. Datasets)**
avoid full scans, use cache	**no cache but parallelize reads**
proprietary binary storage format	parquet "columnar" file format
Single Server, Closed - SPOF	Distributed & Open

How can you eat an Elephant ?



https://www.azquotes.com/quote/529521 (African Proverb)


There is only one way to eat an elephant, a bite at a time.

— Desmond Tutu —

AZ QUOTES

split into pieces (partitions),

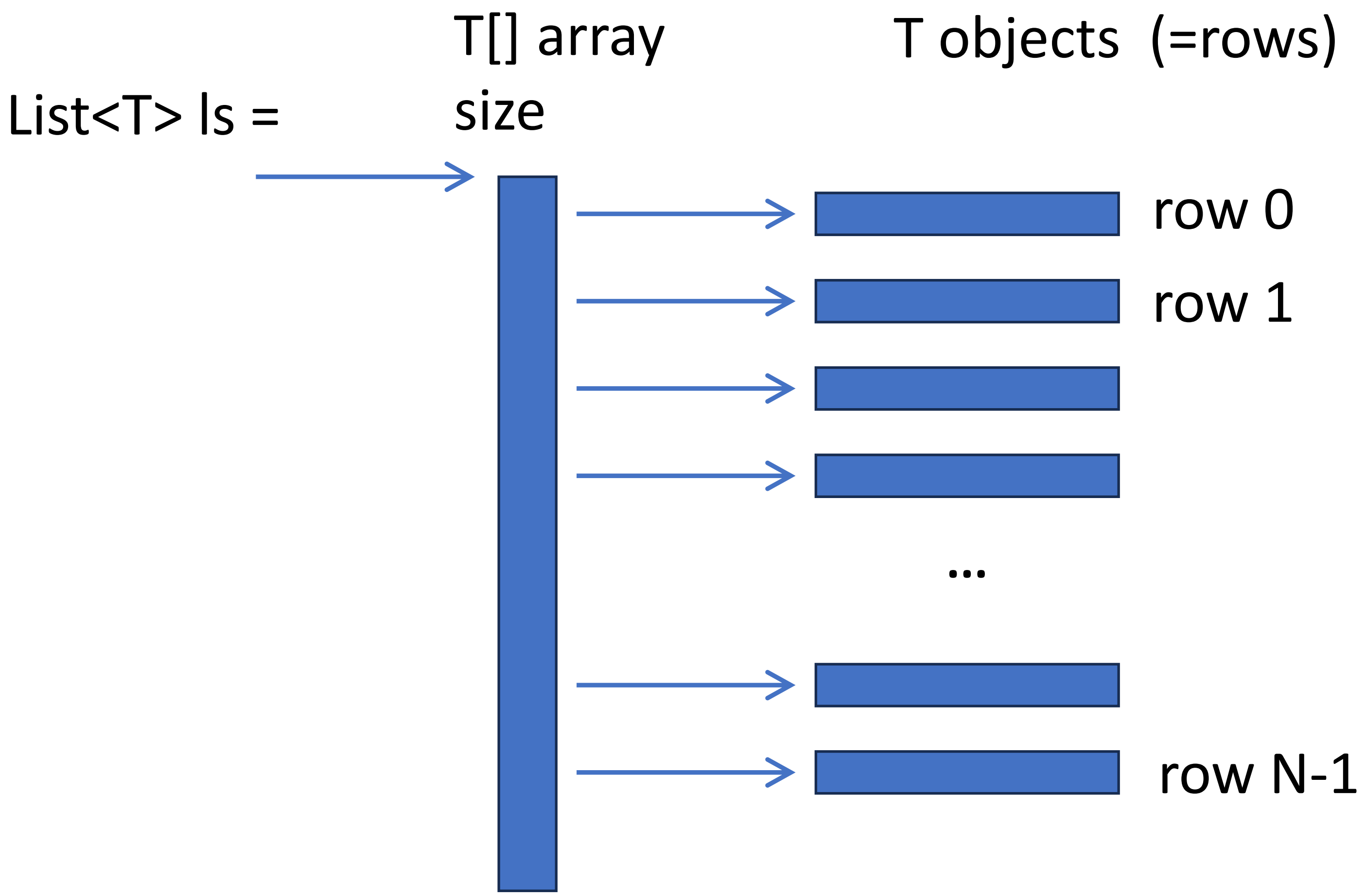
and then

iterate one partition at a time
(or parallelize + iterate if possible)

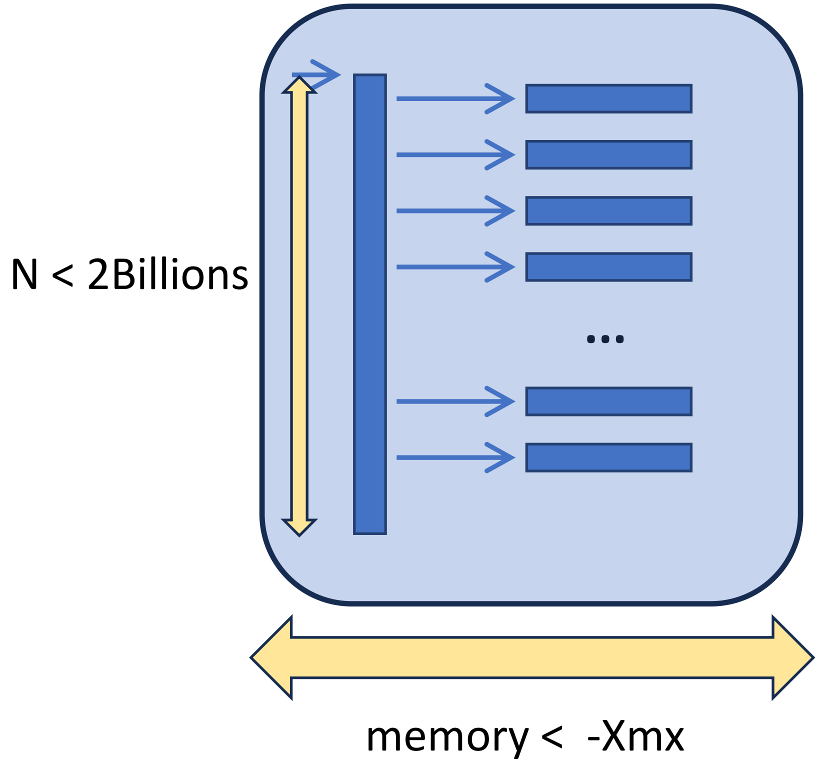
Outline

- from List<T> to distributed **Dataset<T>**
- Immutability, Functional API
- processing workflow:
  Input -> Transformations -> Output
- **narrow** operations (=per partitions)
- **wide** operations (=shuffled)

java.util.ArrayList<T>



List<T> Java VM Restrictions



Restriction 1/ array are indexed by "int" (32 bits)

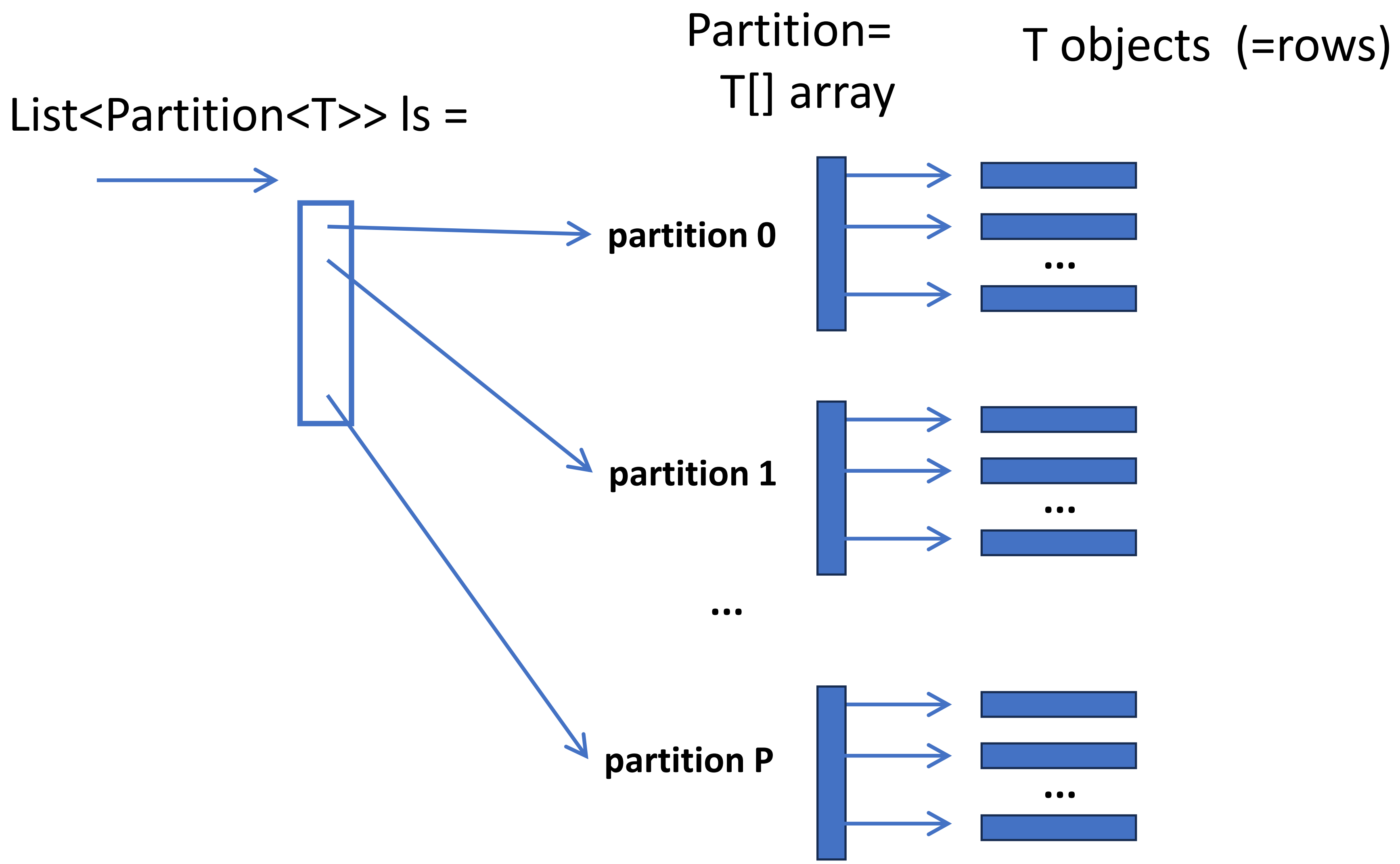
number of elements : N < 2 ^32 - 1 **~ 2 Billions**

Restriction 2/ objects are in heap memory (-Xmx)

total memory size (in bytes) < -Xmx

(ex: -Xmx128g)

Splitting List<T> in sub-list List<Partition<T>>



List< Partition<T> > restrictions

NO MORE Restriction on number of elements

rows are indexed by   [partitionIndex][indexWithinPartition]
can be > 2^32

STILL Restriction 2/ objects are in heap memory (-Xmx)

total memory size (in bytes) < -Xmx
(ex: -Xmx128g)

Practical API for Iterating on List<List<T>> ?

Old-School Imperative Code Style

```
int partitionCount = ds.partitionCount();
for( int i = 0; i < partitionCount; i++) {
   List<T> currPartition = ds.partition(i);
   int currPartitionLen = currPartition.size();

   for (int j = 0; j < currPartitionLen; j++) {
        T row = currPartition.rowAt(j);

        someUserFunction( row );
   }
}
```

BAD ... UGLY, INNEFICIENT
Does not scale on distributed code
NOT even on Multi-Threads

Object-Oriented Style
using Iterator pattern

```
Iterator<T> iter = ds.iterator();
while(iter.hasNext() {
     T row = iter.next();

     someUserFunction( row )
}
```

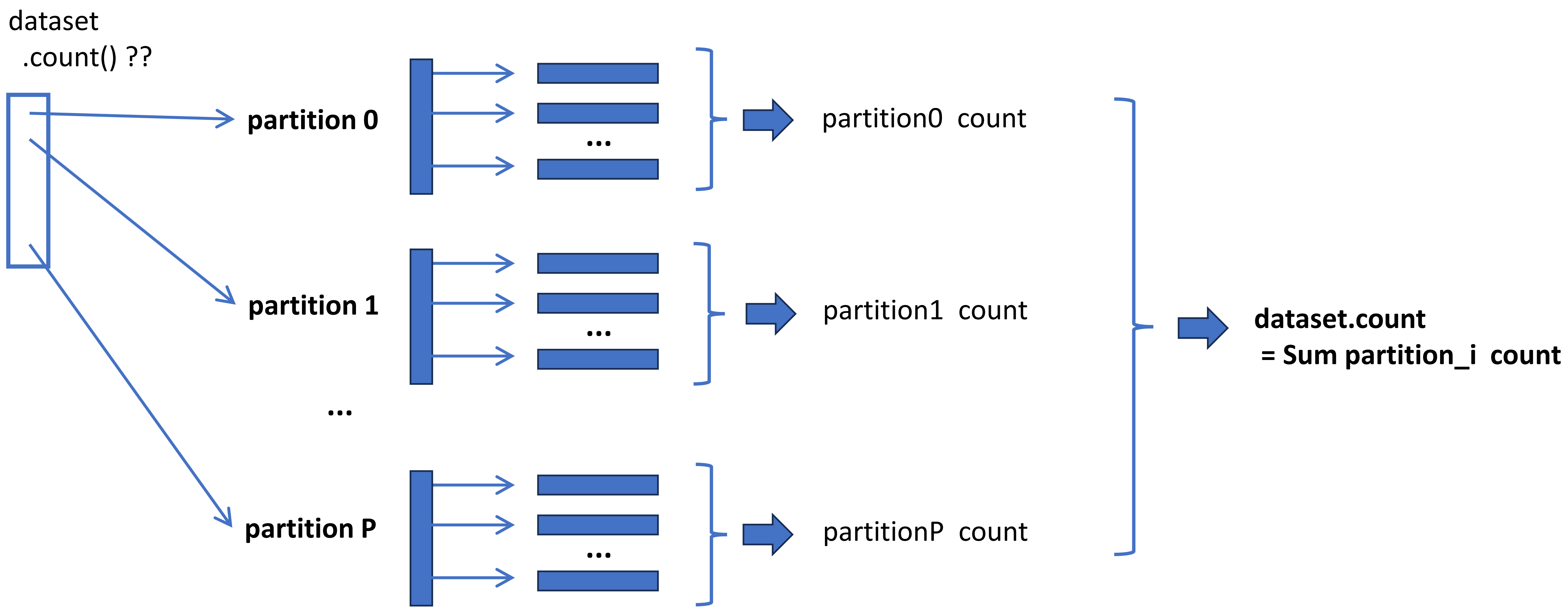
BAD
Does not scale on distributed code
NOT even on Multi-Threads

Functional Style
using Lambda, callbacks

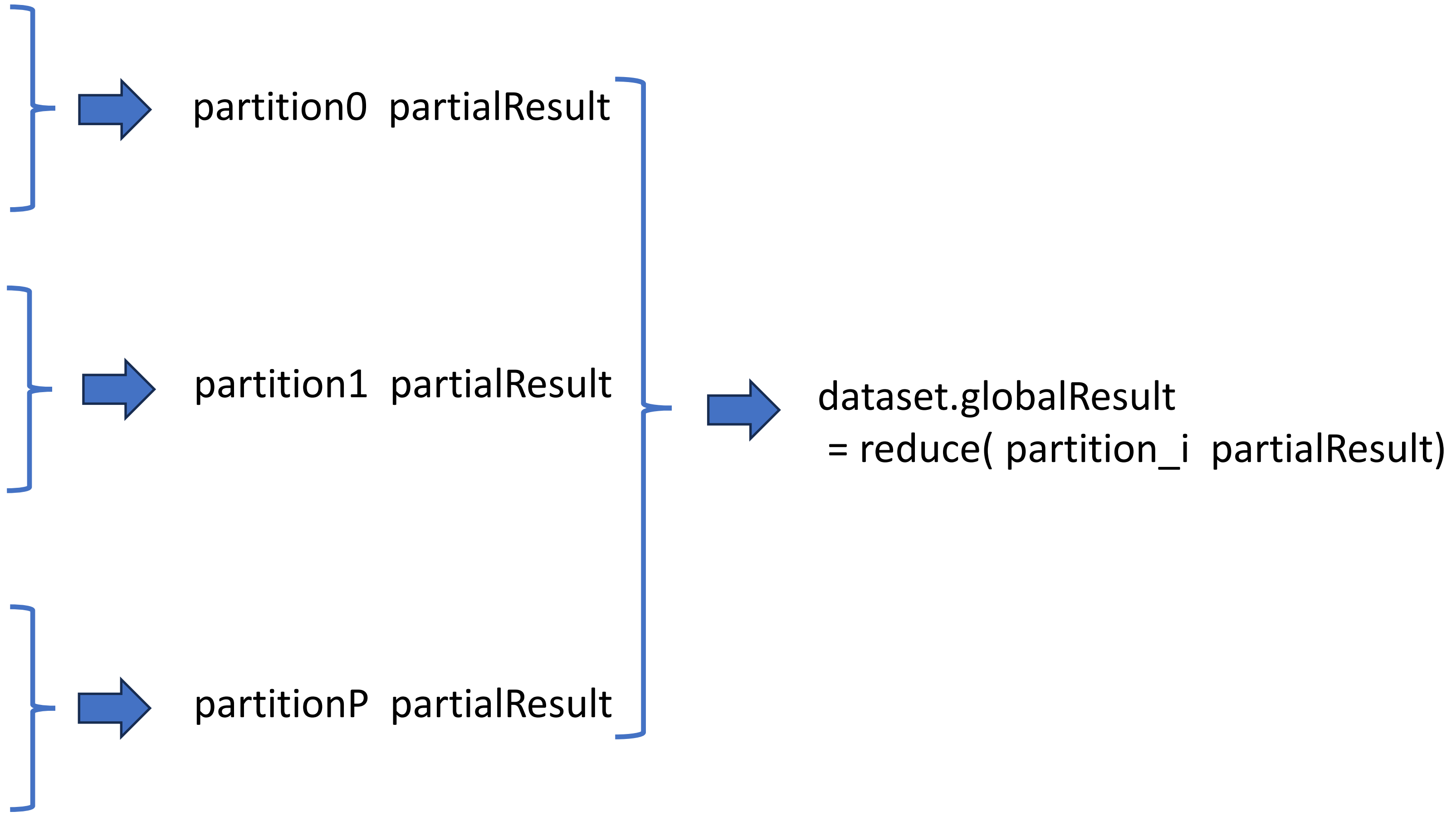
```
ds.forEach( someUserFunction );

// lambda equivalent
ds.forEach(x => someUserFunction(x));
```

OK
(callbacks must be serializable)

Sample Easy Parallelizable Operations: dataset.count()



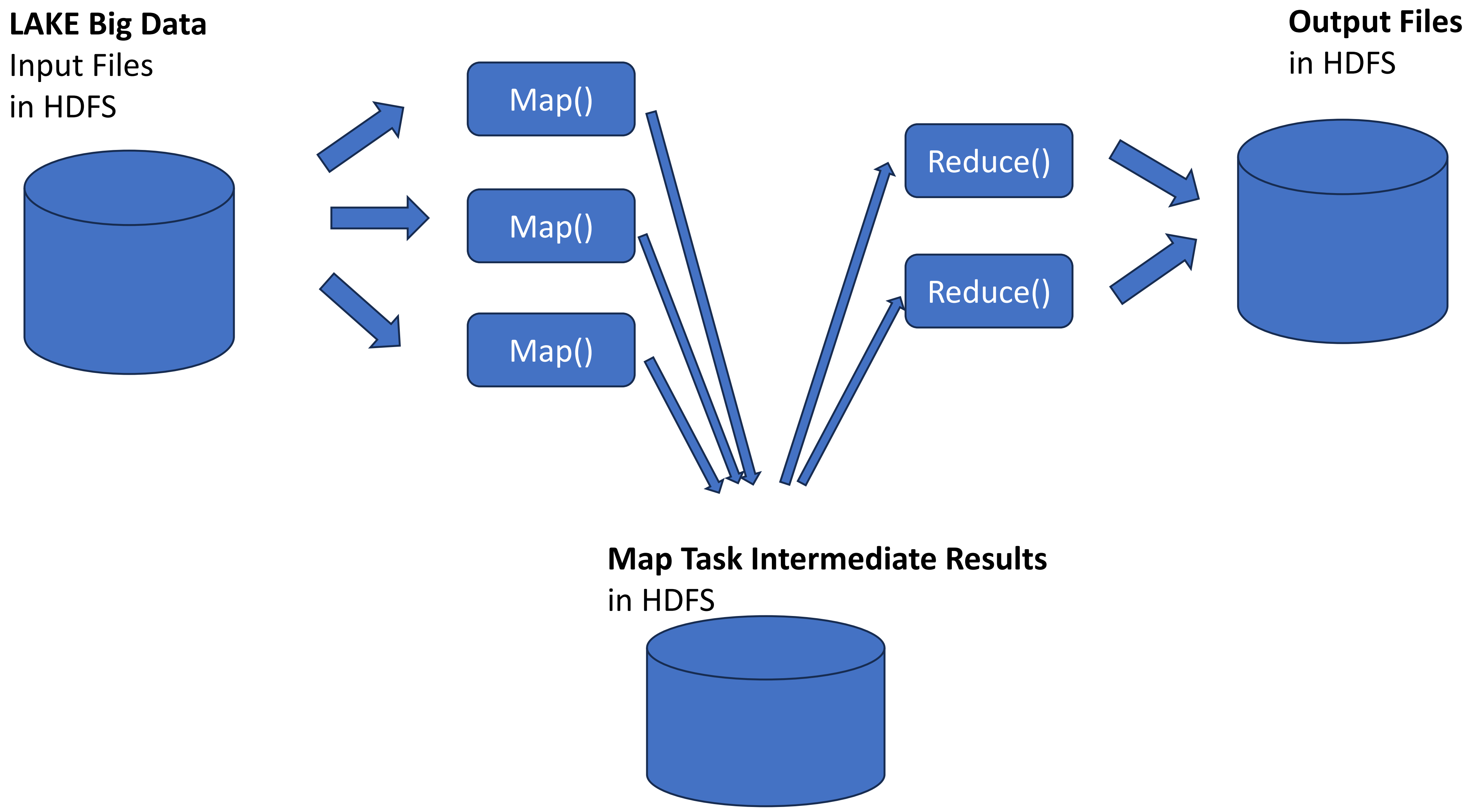
Sample Easy Parallelizable Operations: dataset .count() .find() .first() .min() .max() .sample()



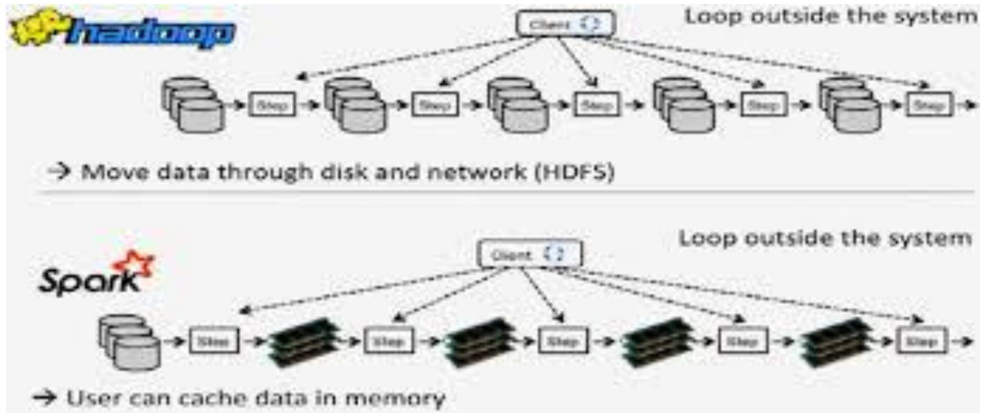
step 1/2 : "**mapper**" side

step 2/2 : shuffle "**reducer**" side

(Hadoop) Map-Reduce … legacy



Spark
Faster save Intermediate Tasks Results



WRONG schema !!

Urban Legend : Spark would be faster because it caches shuffle data in-memory ???

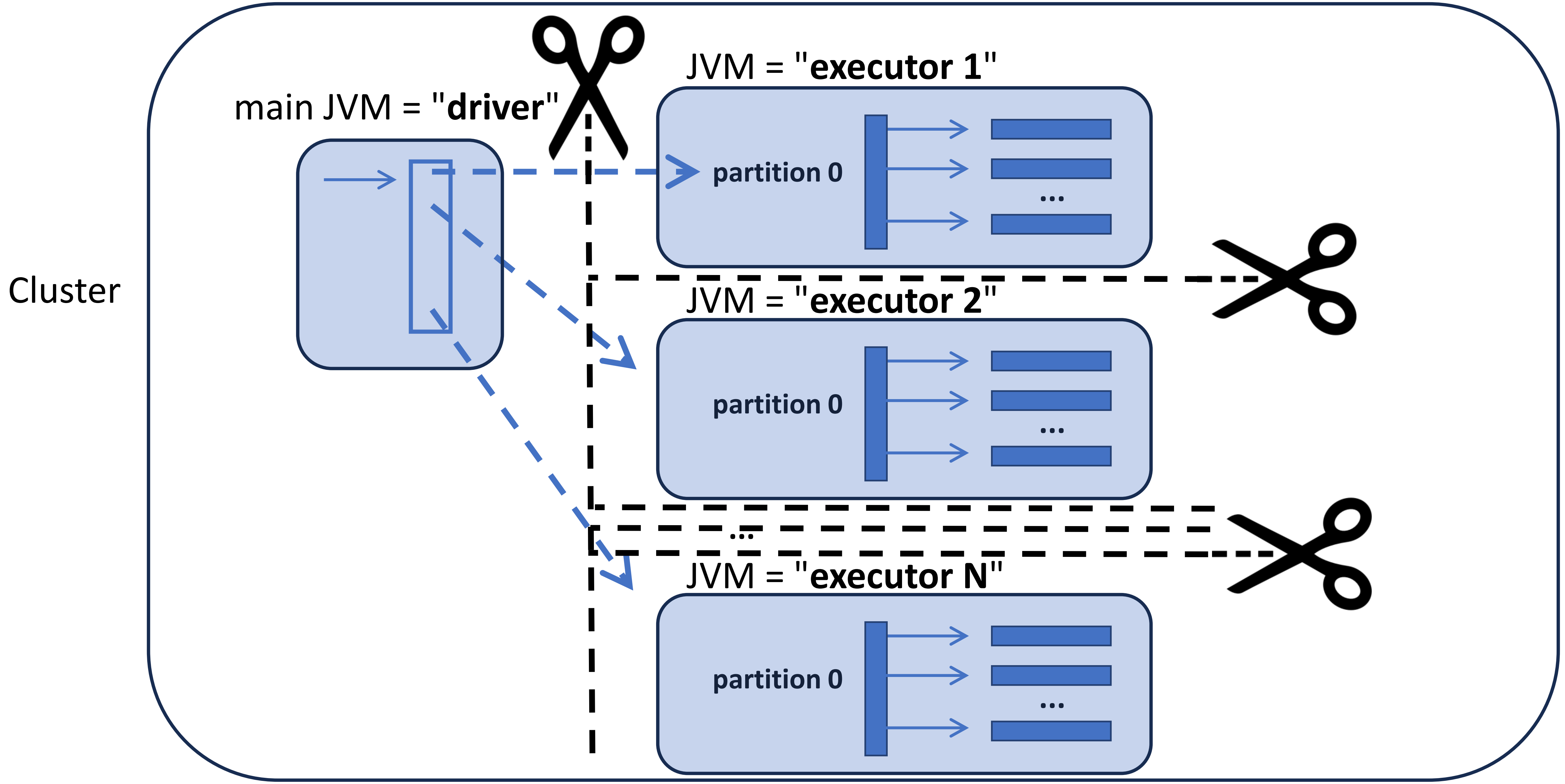
WRONG !!!!

Spark save all shuffle files to Local Disk

.. even maybe several File writes per shuffle ("Spill To Disk")

=> faster than Hdfs, but Less tolerant to failures

Split: Distribute on Multi JVMs Cluster



Distributed in-memory Dataset<T> restrictions

Dataset NO more limited by

- number of elements
- single host node RAM

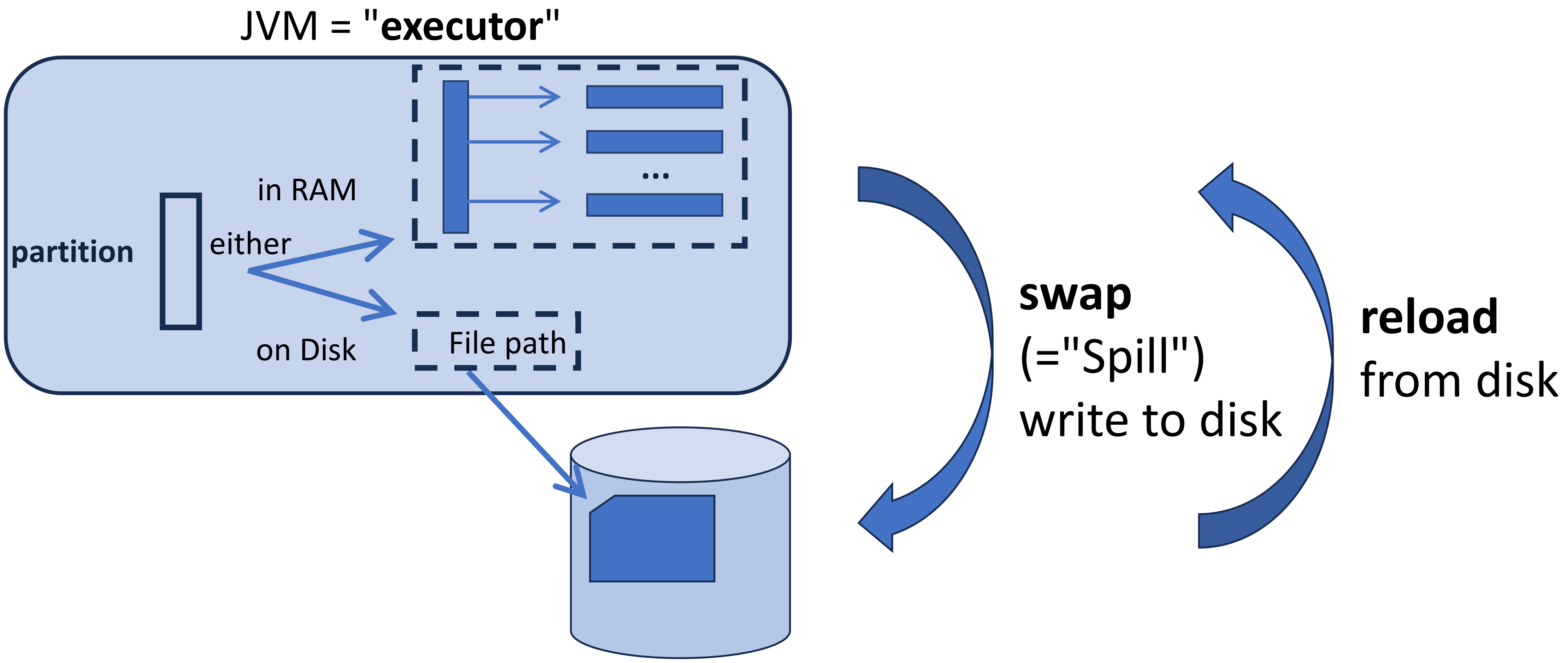
STILL Restrictions :

partition(s) memory size < RAM of a executor holding partition(s)

total memory size < total RAM of cluster

Dataset must be"**equally split and distributed**" (not skewed)

Swap Partition RAM to Local Disk



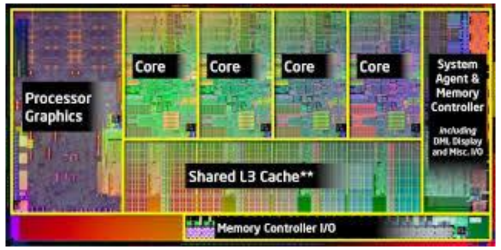
Parallelize on cluster : use N(>=100) CPUs ?

How to process with **multi-threads** ? (use CPU cores)

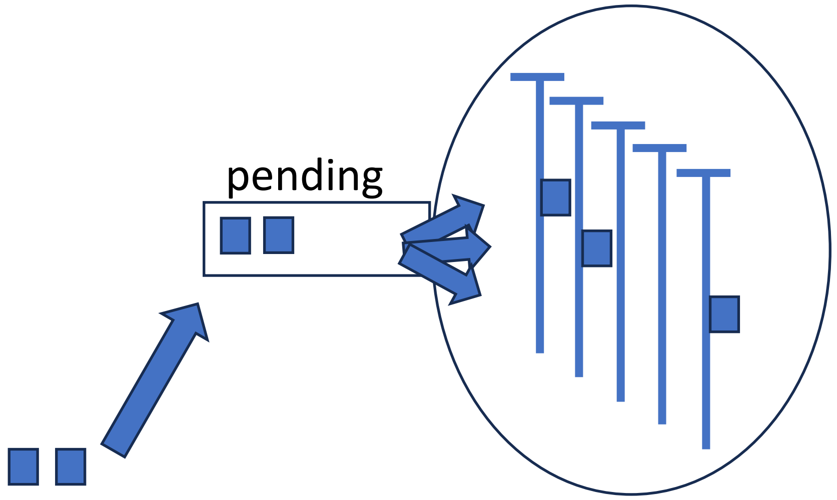
How many partitions can be processed **concurrently** ?

1 Partition on 1 Thread = 1 Task

Physical CPU (Core)



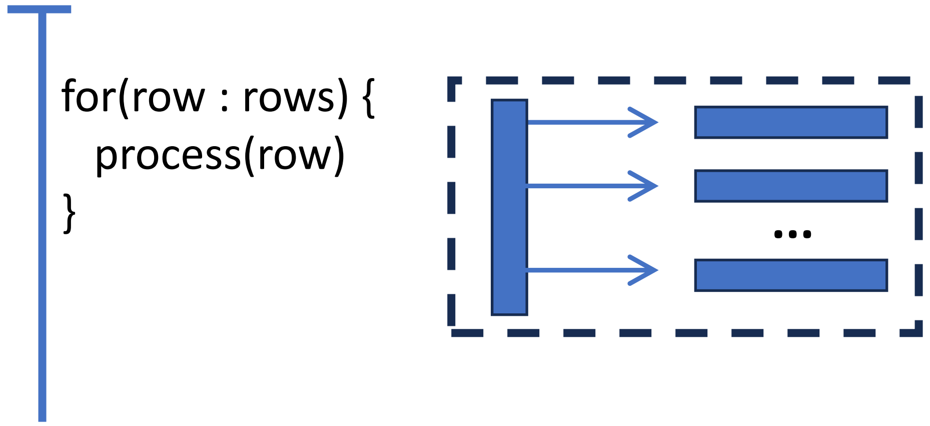
Applicative **TreadPool**
+ FIFO **Task Queue**
= Task executions service



enqueue
Partition to Process
(=**Task**)

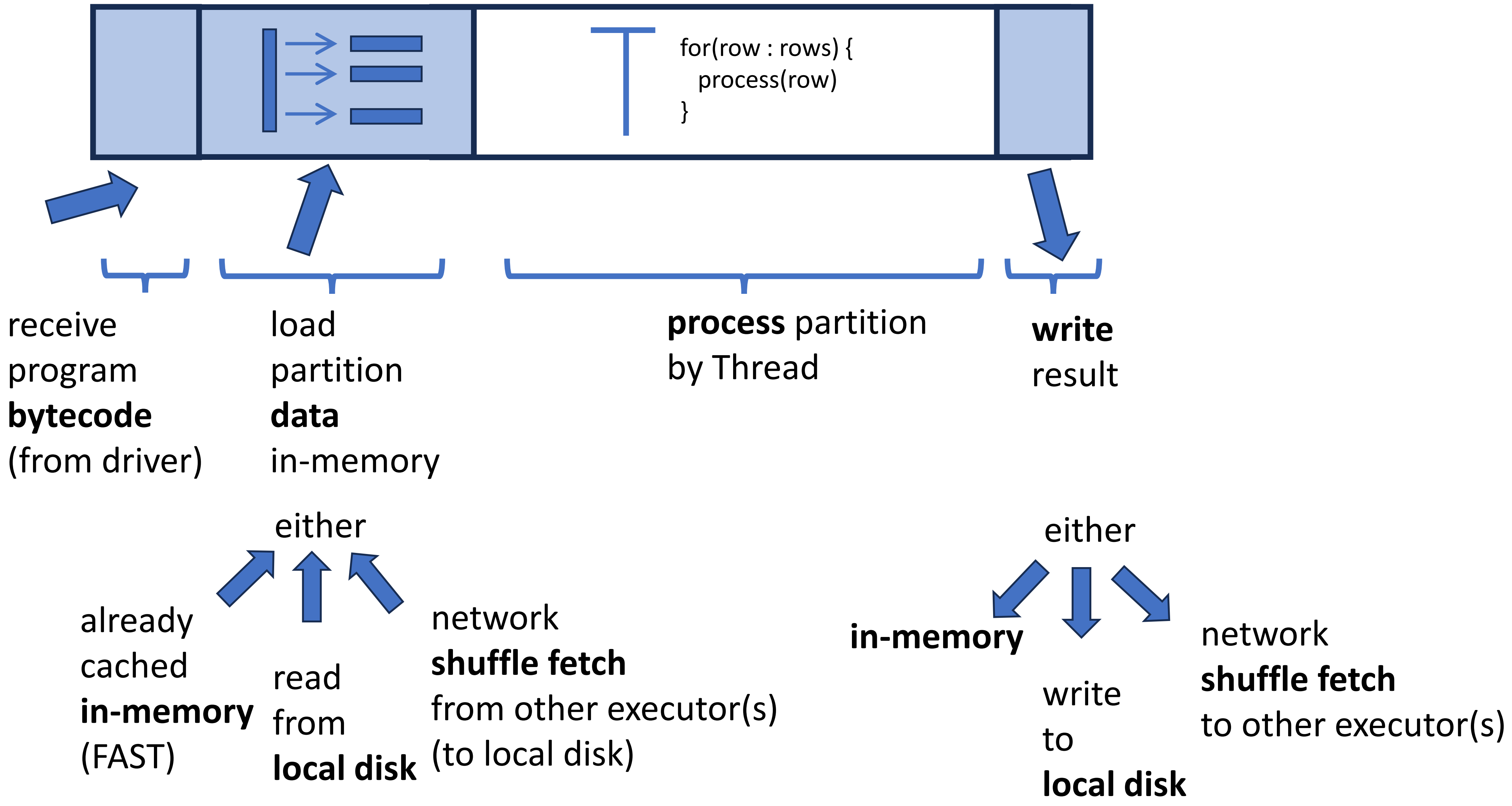
dispatch
1 Partition Task
on
1 free Thread

1 Task = 1 Partition processed by 1 Thread



```
for(row : rows) {
   process(row)
}
```

Life of a Task



```
for(row : rows) {
  process(row)
}
```

receive program **bytecode** (from driver)
load partition **data** in-memory
process partition by Thread
write result
either
already cached **in-memory** (FAST)
read from **local disk**
network **shuffle fetch** from other executor(s) (to local disk)
either
in-memory
write to **local disk**
network **shuffle fetch** to other executor(s)

How Many Concurrent Threads ?

Compromise:

- use as many as possible? thread need cpu

    threads ~= vcore

    avoid unused iddle CPU while waiting for In-Out

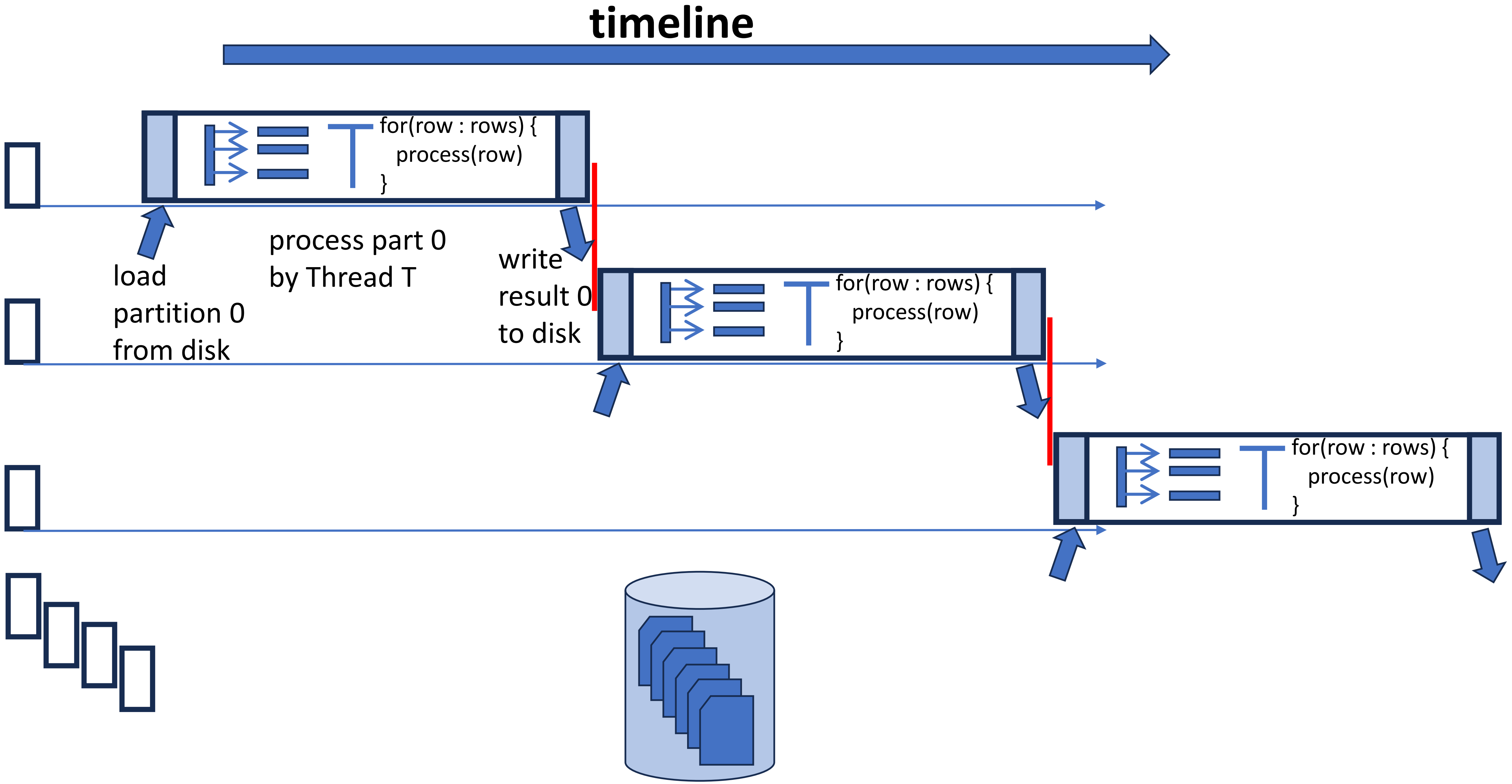
    in general   **threads = vcore = 2 x core - 1**

- can't use too much concurrently:

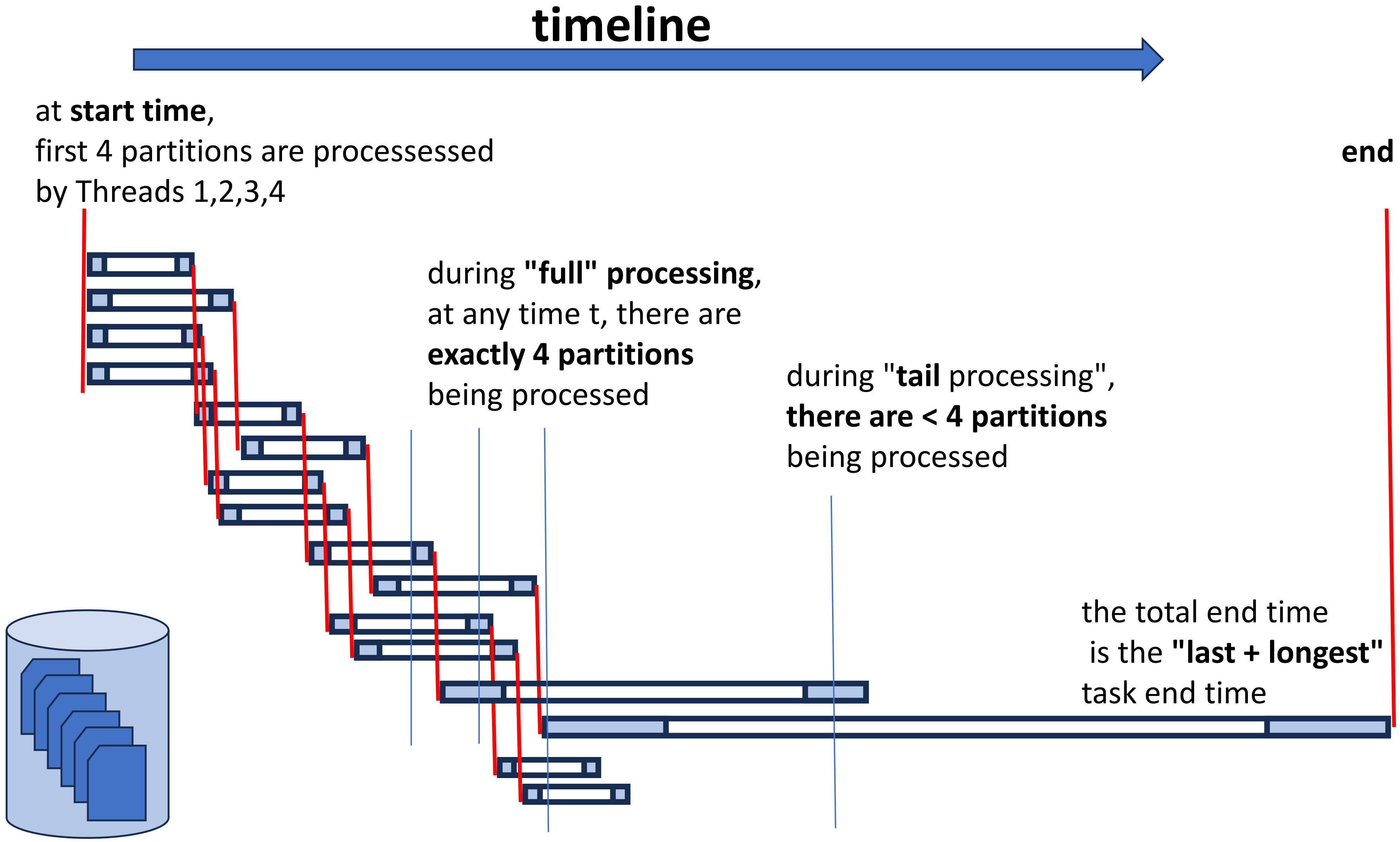
  each task need RAM memory, so

    threads * taskMemory < total RAM

Successive Tasks Timeline for 1 Thread



Timeline for N(100) Partitions - P(4) Threads



at **start time**,
first 4 partitions are processessed
by Threads 1,2,3,4

during **"full" processing**,
at any time t, there are
exactly 4 partitions
being processed

during "**tail** processing",
there are < 4 partitions
being processed

the total end time
is the **"last + longest"**
task end time

Distributed in-memory Dataset<T> restrictions

Dataset NO more limited by
- number of elements
- single host node RAM
- total RAM of cluster

STILL Restrictions :

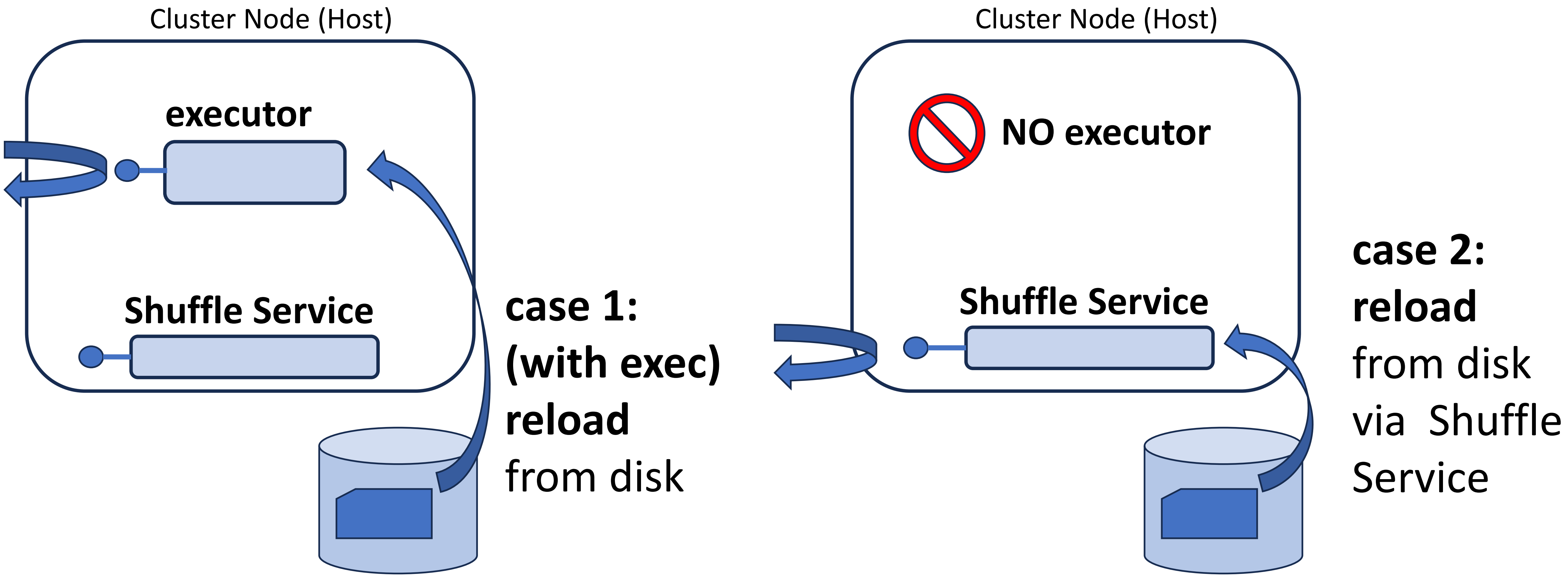
- Dataset must be "equally split and distributed" (not skewed)
- **dependent of local disk(s) of each nodes**
- **no ephemeral executor / nodes (compute linked to storage)**

persistent Disk attached to Host / always served by Executor ...

How to handle **Failures** (Disk/Executor/Host) ?

How to **dynamically scale** Up & Down ??

ShuffleService: to re-Read from Disk when Executor is gone / lost



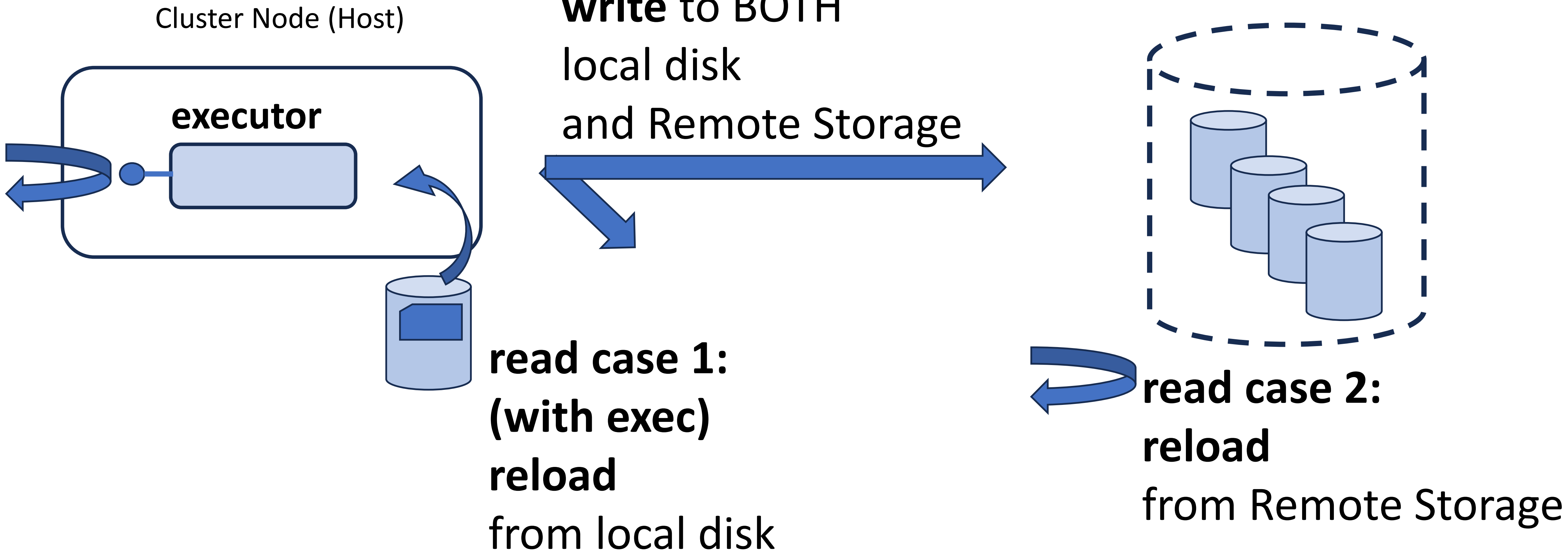
Distributed Dataset<T> restrictions

with **dynamicAllocation** : executor can scale up&down
=> allows to share resources in cluster

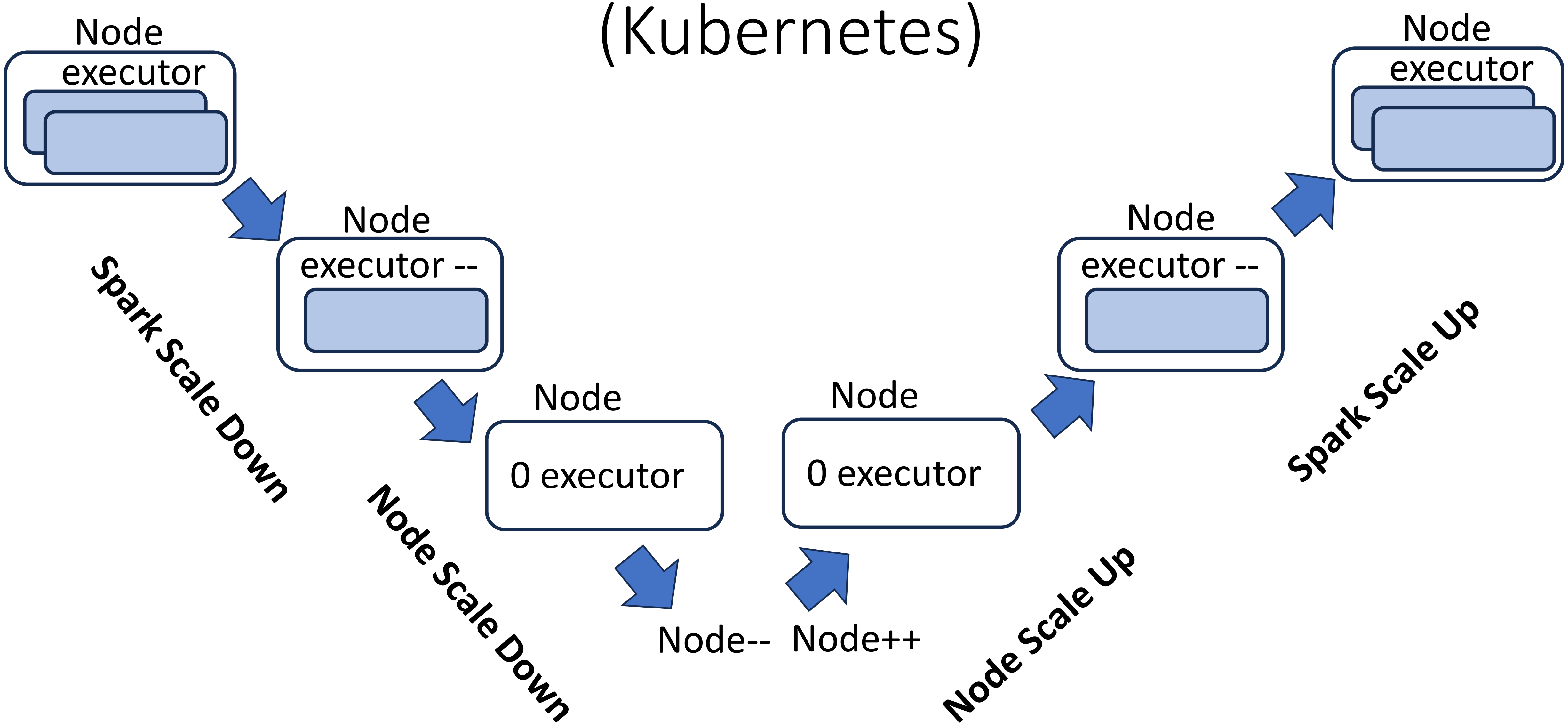
STILL Restrictions :

hosts and disks can still NOT scale
no ephemeral nodes (compute linked to storage)

Current Developments in Spark... using Local Disk + Object Storage: "Remote Shuffle Service"



DynamicAllocation & Using Ephemeral Compute Resources (Kubernetes)



Dataset<T> Restrictions Summaries

Dataset NOT limited by

- number of elements
- single host node RAM
- total RAM of cluster
- local disk(s) of each nodes
- dynamicAllocation: executor can be scale up/down
- compute resource: node can be scale up/down

STILL Restrictions :

Dataset must be "equally split and distributed" (not skewed)
NOT all partitions can be processed simultaneously
Cpu <-> Memory <-> Disk IO <-> Network are trade-off

Outline

- from List<T> to distributed **Dataset<T>**
- **Immutability, Functional API**
- processing workflow:
  Input -> Transformations -> Output
- **narrow** operations (=per partitions)
- **wide** operations (=shuffled)

Immutability : getter(s), new, NO setter

instead of **modifying** existing objects
just **create** new ones !

Functional API

NO **Iterative Code**

```
ds2 = new ..; for(int i =0; i <N; i++) { ds2.add(ds1.get(i));}
```

use **Functions (Lambda)**

```
ds2 = dataset1.map(row -> new Row(...))
.filter(), .flatMap(),  .mapPartition(), .reduceByKey(), etc.
```

Example of CRUD API Replacements
"C" = Create

No **ds=new Dataset()**
for(..) { ds.**addRow**(row); }

use **ds=spark.createDataset(list)**
or **ds=spark.read.format(...).load("path/files/")**
or **ds=spark.sql("SELECT * FROM .. WHERE ..")**

Example of CRUD API Replacements "R" = Read

No

```
    for(int i=0; i<ds.partitionCount(); i++) {
        for(int j=0;  j< ds.part(i).count(); j++) {
            ds.getRow( i, j );  } }
```

use

```
ds.map(row -> {.. })
    or ds.mapPartitions(rowIter -> {
                while(rowIter.hasNext() { .. }
        })
```

Example of CRUD API Replacements
NO "U" = Update !!

No

    ds.**setRow**(row);   row.**set**(col, value)
    Sql "**UPDATE** TABLE .. SET .. WHERE .."

use

  ds2 = ds1.map(row -> new Row(copy with ..));

or install **Iceberg** or DeltaLake Extension

  "UPDATE ..", "UPSERT ..", "SELECT asof version"

Example of CRUD API Replacements
NO "D" = Delete !!

No

ds.**removeRow**()

Sql "**DELETE FROM** TABLE .. WHERE .."

just use

ds2 = ds1.filter(row -> { true/false });

or install **Iceberg** or DeltaLake Extension

"DELETE.."

Outline

- from List<T> to distributed **Dataset<T>**
- **Immutability, Functional API**
- processing workflow:
  Input -> Transformations -> Output
- **narrow** operations (=per partitions)
- **wide** operations (=shuffled)

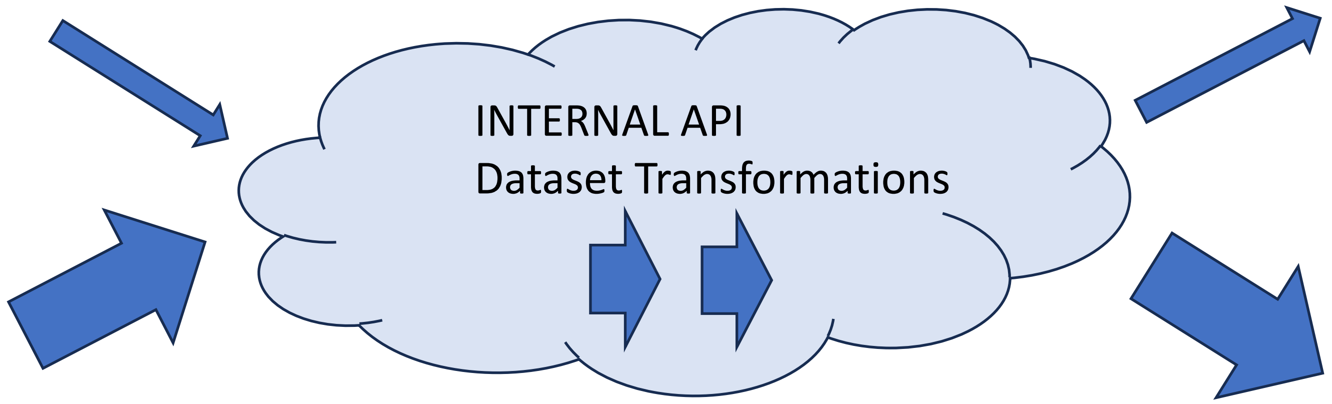
External Input - Internal Dataset API - External Output

dataset = spark.**createDataset**(inMemoryList)

SMALL data only !

inMemoryList = dataset.**collectAsList**()

SMALL data only !



dataset = spark.**read**() ... .**load**(filePath)
dataset = spark.**sql**("SELECT * FROM ..")

ds2 = ds1.**<<transform>>**(..)

dataset.**write**() ... .**save**(filePath)
spark.**sql**("INSERT.. SELECT..")

Hdfs File **Inputs**
or other IO reads

Hdfs File **Outputs**
or other IO writes

Challenge for Distributed Programming

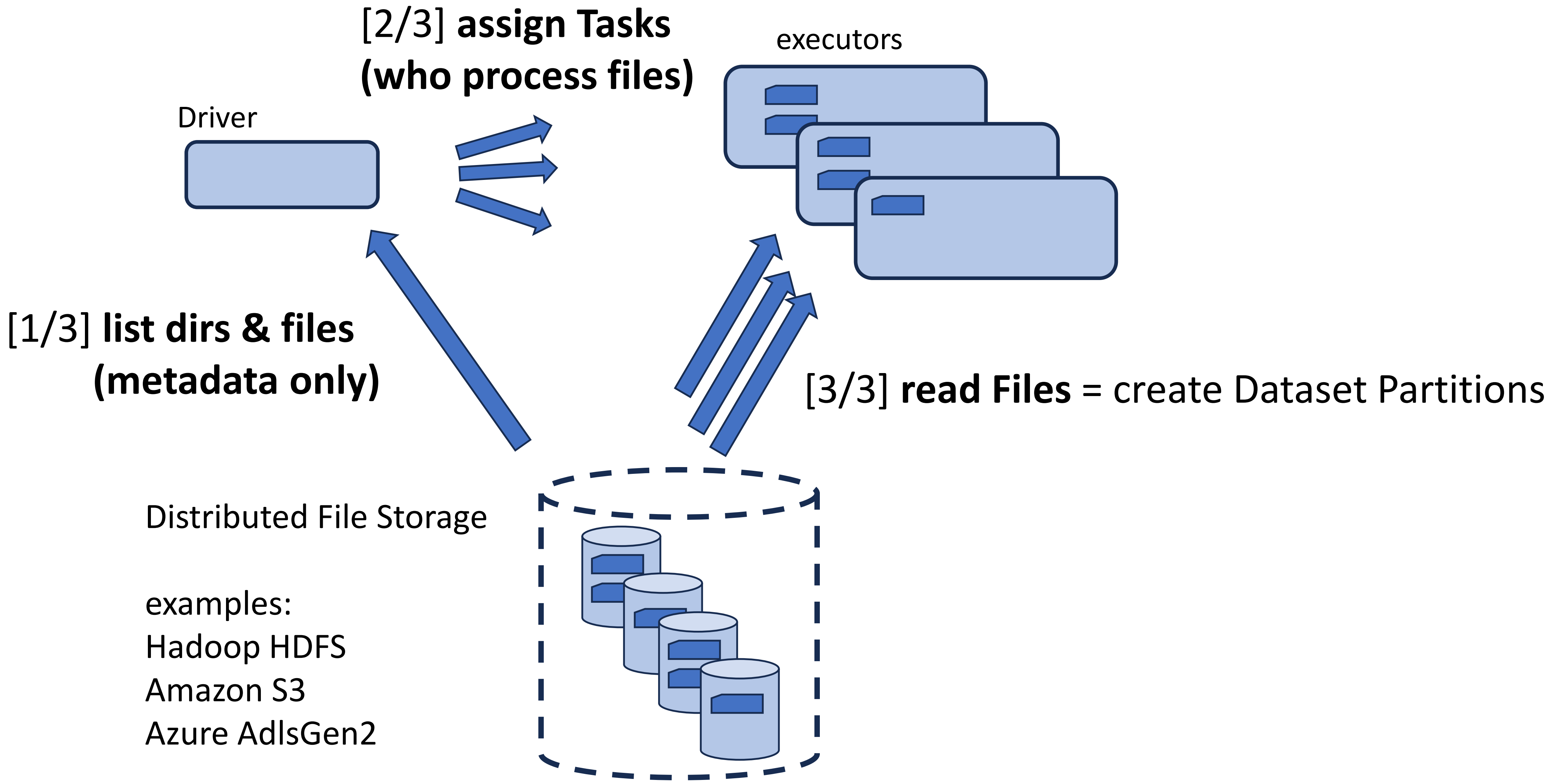
How to ?
- Distributed read Input files ?
- Distributed write result files ?
- transform ?

Concurrency between threads & nodes   (Dataset Immutable => OK)

result on 1 node memory is not "available" on other nodes => need network copy + sync

all "iterative style" programming is impossible

Inputs : Distributed Read Files


[2/3] **assign Tasks (who process files)**
executors
Driver
[1/3] **list dirs & files (metadata only)**
[3/3] **read Files** = create Dataset Partitions
Distributed File Storage
examples:
Hadoop HDFS
Amazon S3
Azure AdlsGen2

Sample Spark ".read()" Code

```
Dataset<Row> ds  = sparkSession
    .read()
    .format("parquet")
    .option("compression", "snappy")
    .load("hdfs://path/dir")


Dataset<Row> ds  = sparkSession
    .sql("SELECT * FROM .. WHERE ..");
```

Read dir, not files ?

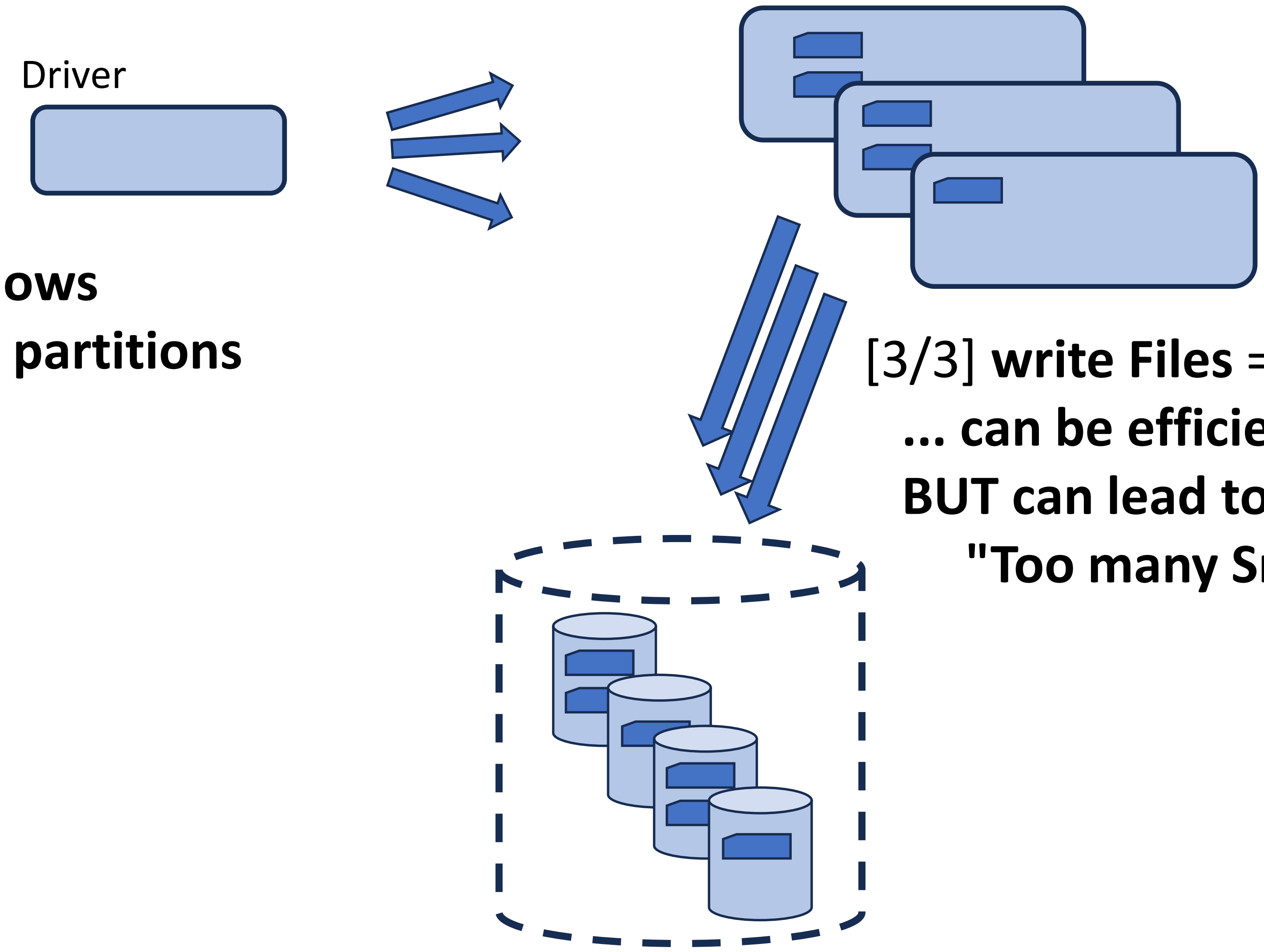
```
sparkSession.read() ..
  .load("hdfs://path/dir")    //  implicitly  "**/*.parquet"  files
```

Spark will discover all sub-dirs
filter out all "_*" and ".*" considered as Hidden files

example: "_SUCCESS", ".part-*.crc" are excluded

Dirs should contain only homogeneous files type

Outputs: Distributed Write Files



[1/3] **knows Dataset partitions**

[2/3] **assign write Tasks**

[3/3] **write Files** = 1 per dataset partition
... can be efficiently parallelized
BUT can lead to problem
"Too many Small files"

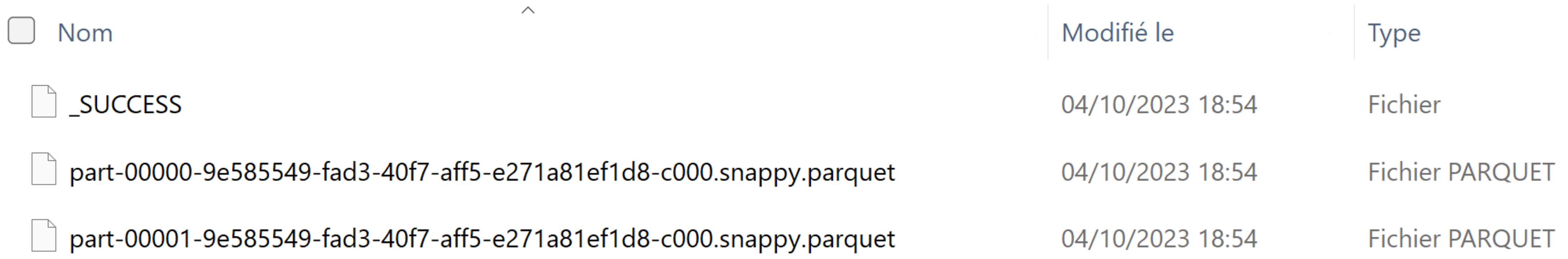
Sample Spark ".write()" Code

```
Dataset<Row> ds  = ...
ds.write()
   .format("parquet")
   .option("compression", "snappy")
   .save("hdfs://path/dir")



sparkSession
   .sql("INSERT ... SELECT .. ");
```

Write ... UUID Generated filenames
1 File per Partition

to write in a Distributed way, Spark generates random UUID filename,
and "part-000xx" index for each partition
"_SUCCESS" is an empty marker file

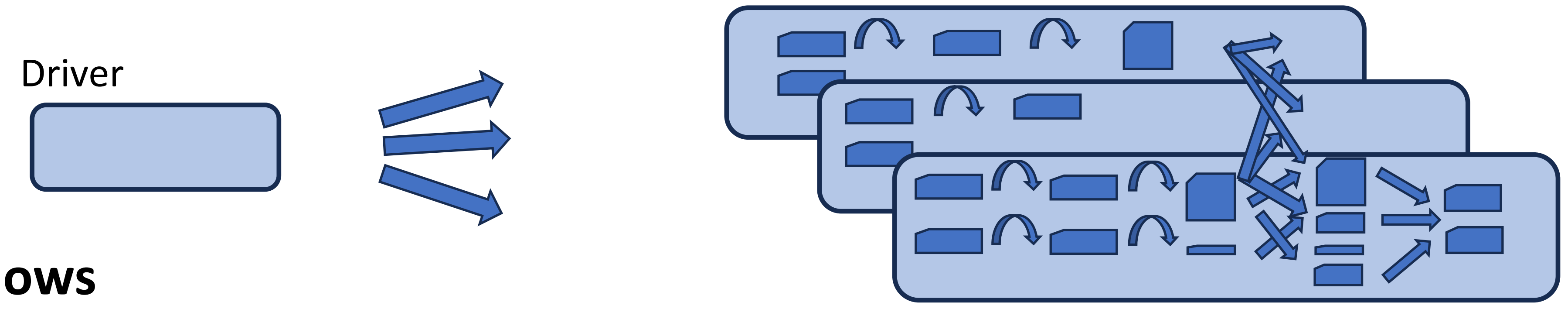
example:



Nom	Modifié le	Type
_SUCCESS	04/10/2023 18:54	Fichier
part-00000-9e585549-fad3-40f7-aff5-e271a81ef1d8-c000.snappy.parquet	04/10/2023 18:54	Fichier PARQUET
part-00001-9e585549-fad3-40f7-aff5-e271a81ef1d8-c000.snappy.parquet	04/10/2023 18:54	Fichier PARQUET

(Input ->) Transformations (-> Outputs)

[2/3] **assign Tasks**



[1/3] **knows**
Dataset partitions

[3/3] **transformation Tasks**

can change partitions topology (change count/size)

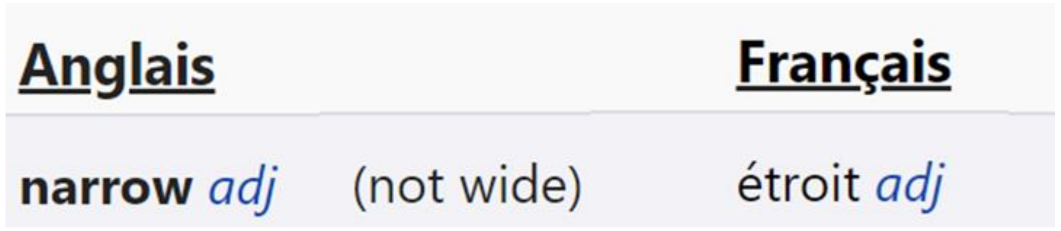
can redistribute data on cluster

Outline

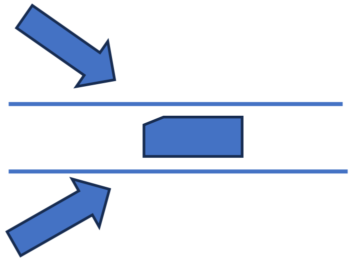
- from List<T> to distributed **Dataset<T>**
- Immutability, Functional API
- processing workflow:
  Input -> Transformations -> Output
- **narrow** operations (=per partitions)
- **wide** operations (=shuffled)

"Narrow" ?

https://www.wordreference.com/enfr/narrow

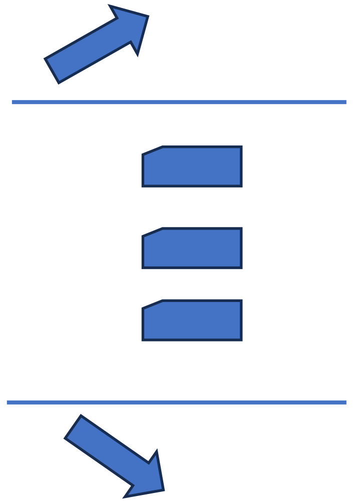


Anglais		Français
narrow *adj*	(not wide)	étroit *adj*



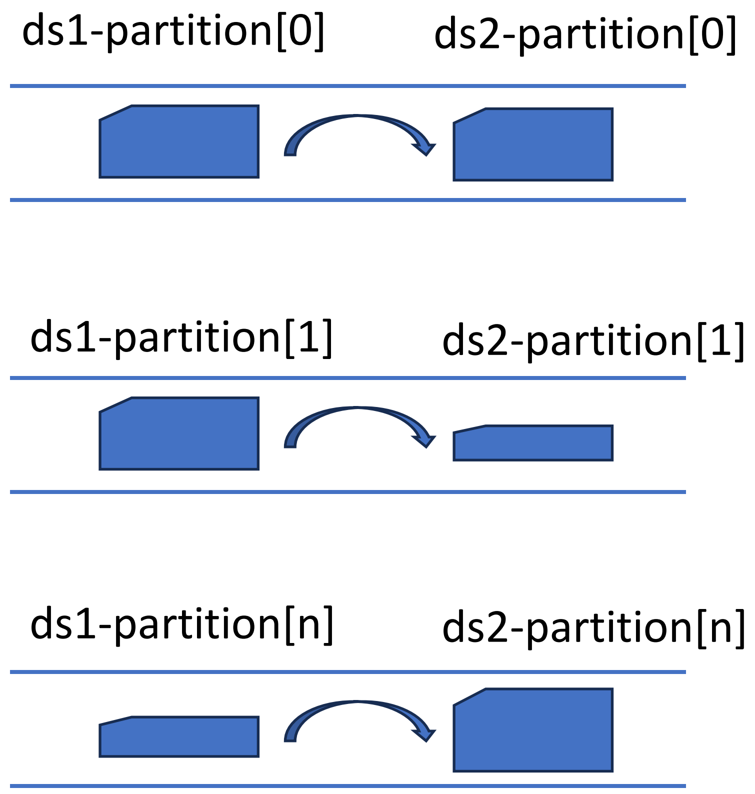
narrow

$\neq$ **wide**



Narrow Transformations

ds2 = ds1.narrowTransform(..)



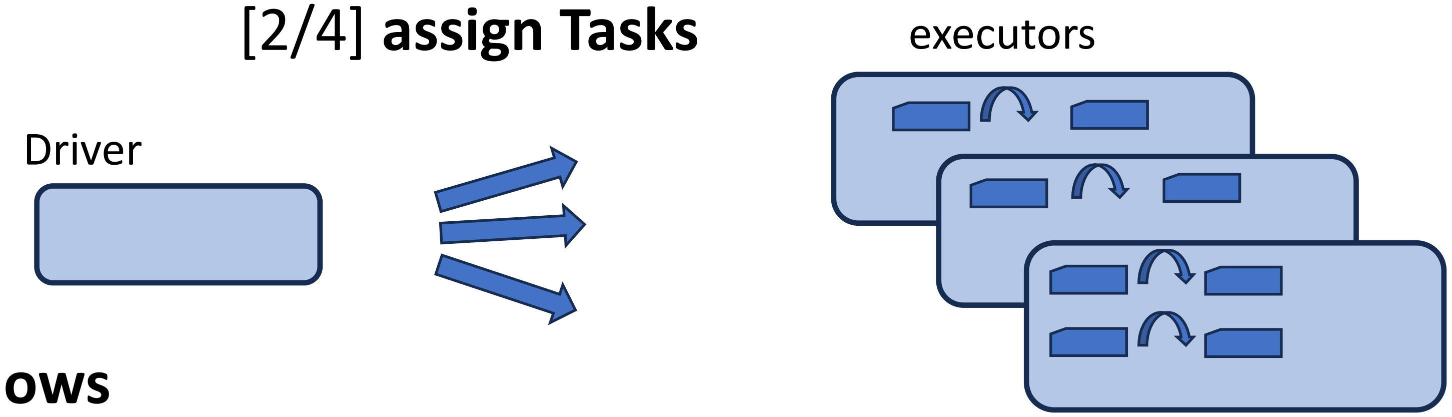
locally independent (parallelisable) transformations **for each partition**

not necessarily "row by row", but partition by partition

NO network data movement between partition

Narrow Transformation

[2/4] **assign Tasks**



[1/4] **knows**
Dataset partitions

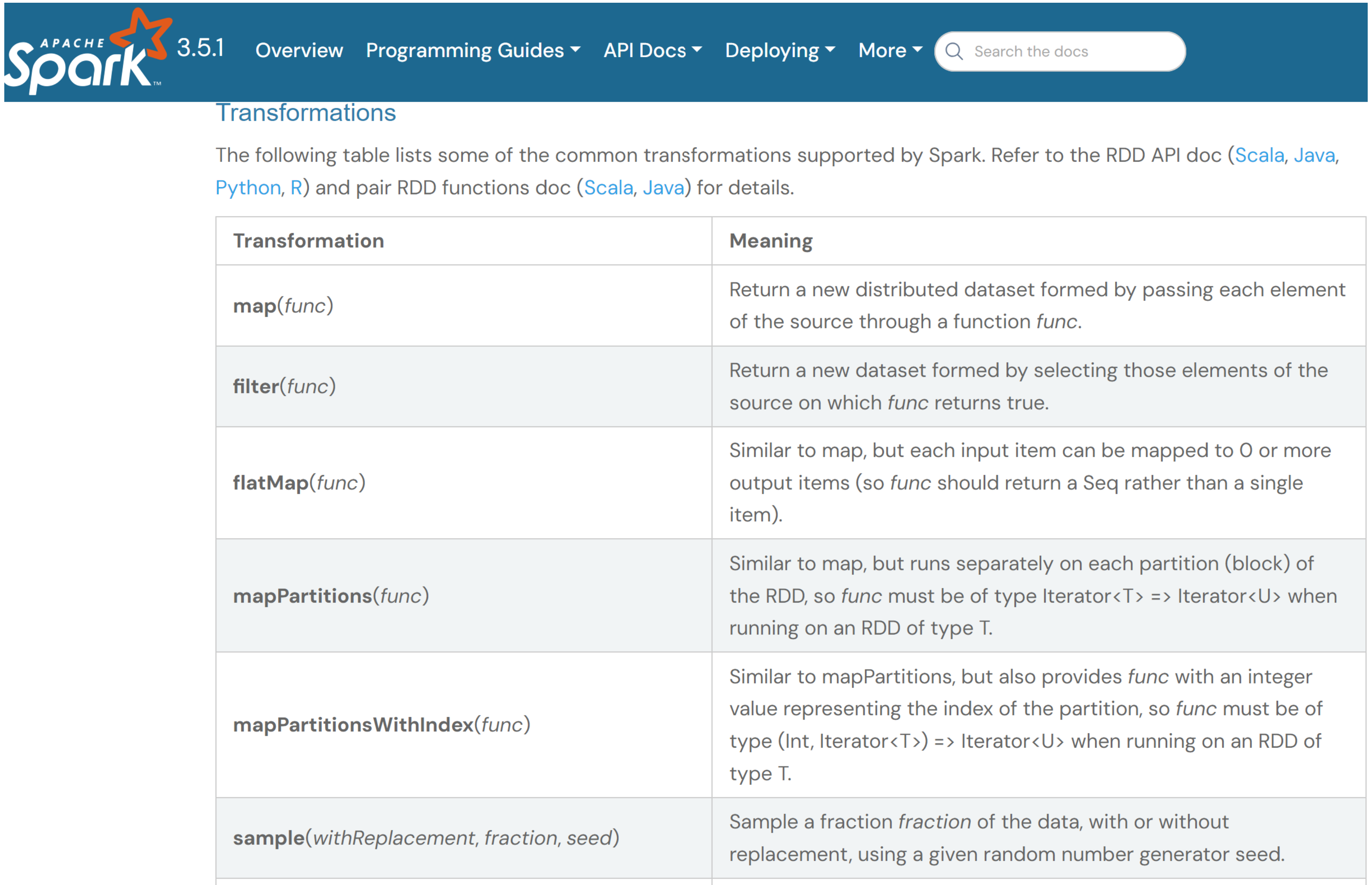
[3/4] **transformation Tasks**

[4/4] **wait result status**
.. register new dataset

on each executor,
if there were N src partition(s),
=> there will be N result partition(s)

Narrow Transformations

https://spark.apache.org/docs/latest/rdd-programming-guide.html#rdd-operations



Transformations

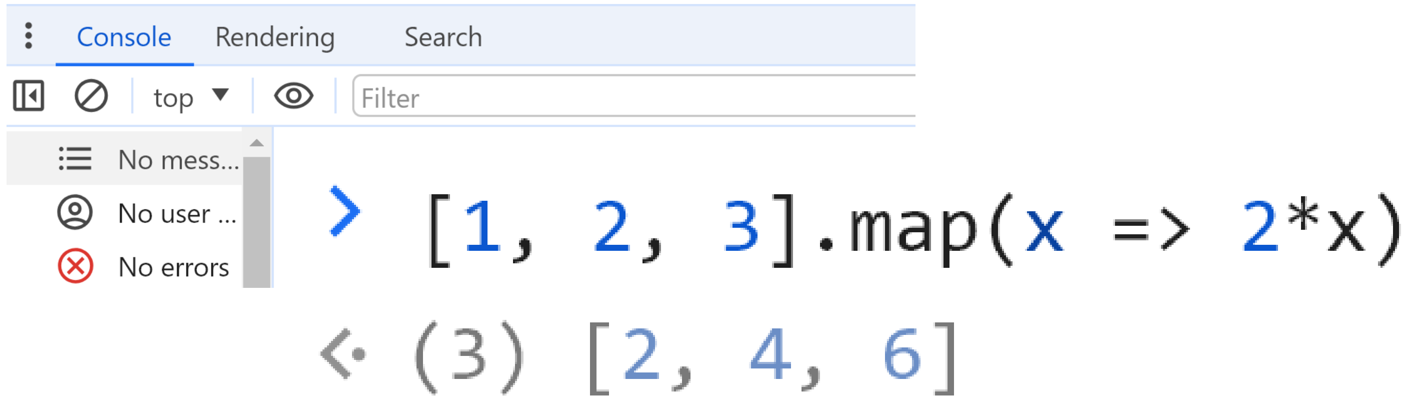
The following table lists some of the common transformations supported by Spark. Refer to the RDD API doc (Scala, Java, Python, R) and pair RDD functions doc (Scala, Java) for details.

Transformation	Meaning
map(*func*)	Return a new distributed dataset formed by passing each element of the source through a function *func*.
filter(*func*)	Return a new dataset formed by selecting those elements of the source on which *func* returns true.
flatMap(*func*)	Similar to map, but each input item can be mapped to 0 or more output items (so *func* should return a Seq rather than a single item).
mapPartitions(*func*)	Similar to map, but runs separately on each partition (block) of the RDD, so *func* must be of type Iterator<T> => Iterator<U> when running on an RDD of type T.
mapPartitionsWithIndex(*func*)	Similar to mapPartitions, but also provides *func* with an integer value representing the index of the partition, so *func* must be of type (Int, Iterator<T>) => Iterator<U> when running on an RDD of type T.
sample(*withReplacement*, *fraction*, *seed*)	Sample a fraction *fraction* of the data, with or without replacement, using a given random number generator seed.

.map(x => f(x))

as in any functional langage

example in JavaScript (use Chrome DevTools: F12)

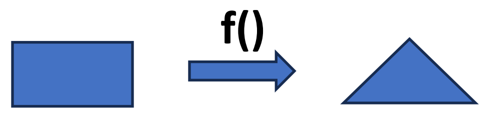


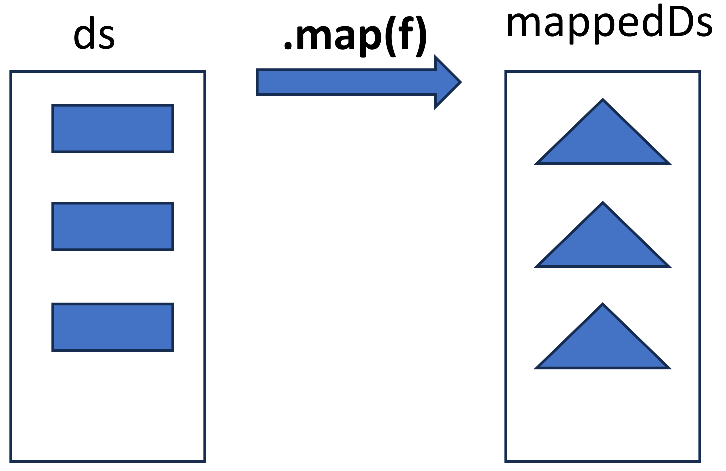
```
> [1, 2, 3].map(x => 2*x)
<· (3) [2, 4, 6]
```

dataset.map(rowFunction : T => U)

```
Dataset<T>  ds = ...

def f (row: T): U = { ... }

Dataset<U>  mappedDs = ds.map( row =>  f(row) )
                    // idem  = ds.map(f)
```

f : Function T -> U





.map() Row columns <=> sql: "SELECT <...>" columns managment

Dataset<T> ds = ...
Dataset<U> mappedDs = **ds.map(f)**

when T=U=Row .. columns

```
 ds.select("col1", col2", "colX")
    .withColumn("computed", ...expr)
    .withColumnRename("computed", "col3"
    .drop("colX")
```

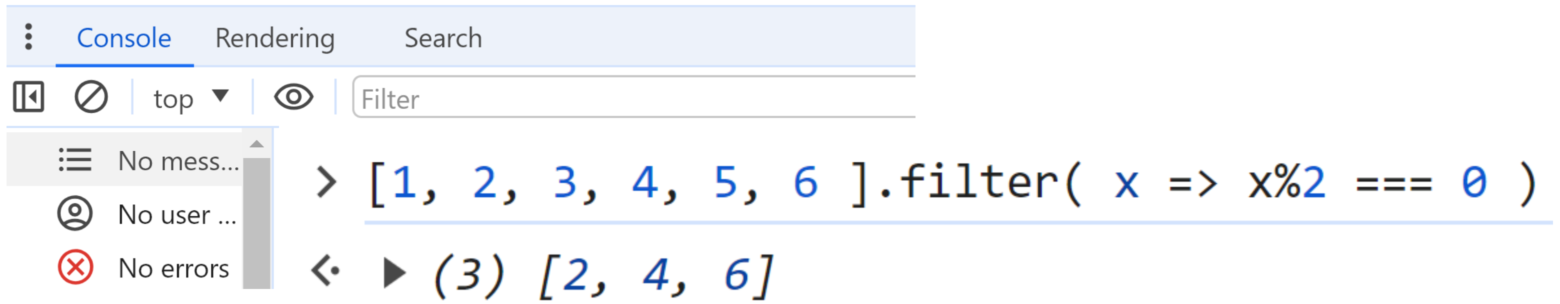
⟺

```
SELECT col1, col2, colX,
          ...expr as computed,
          computed as col3,
FROM ...
```

.filter(row => booleanFunc(row))

as in any functional langage

example in JavaScript (use Chrome DevTools: F12)

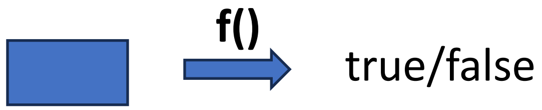


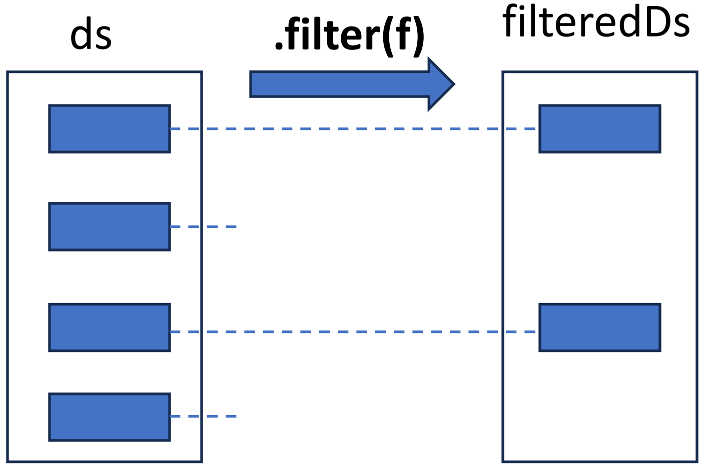
```
> [1, 2, 3, 4, 5, 6 ].filter( x => x%2 === 0 )
<· ▶ (3) [2, 4, 6]
```

dataset.filter(rowPredicate : T => boolean)

```
Dataset<T>  ds = ...

def f (row: T): boolean = { ... }

Dataset<T>  filteredDs = ds.filter( row =>  f(row) )
                        // idem  = ds.filter(f)
```

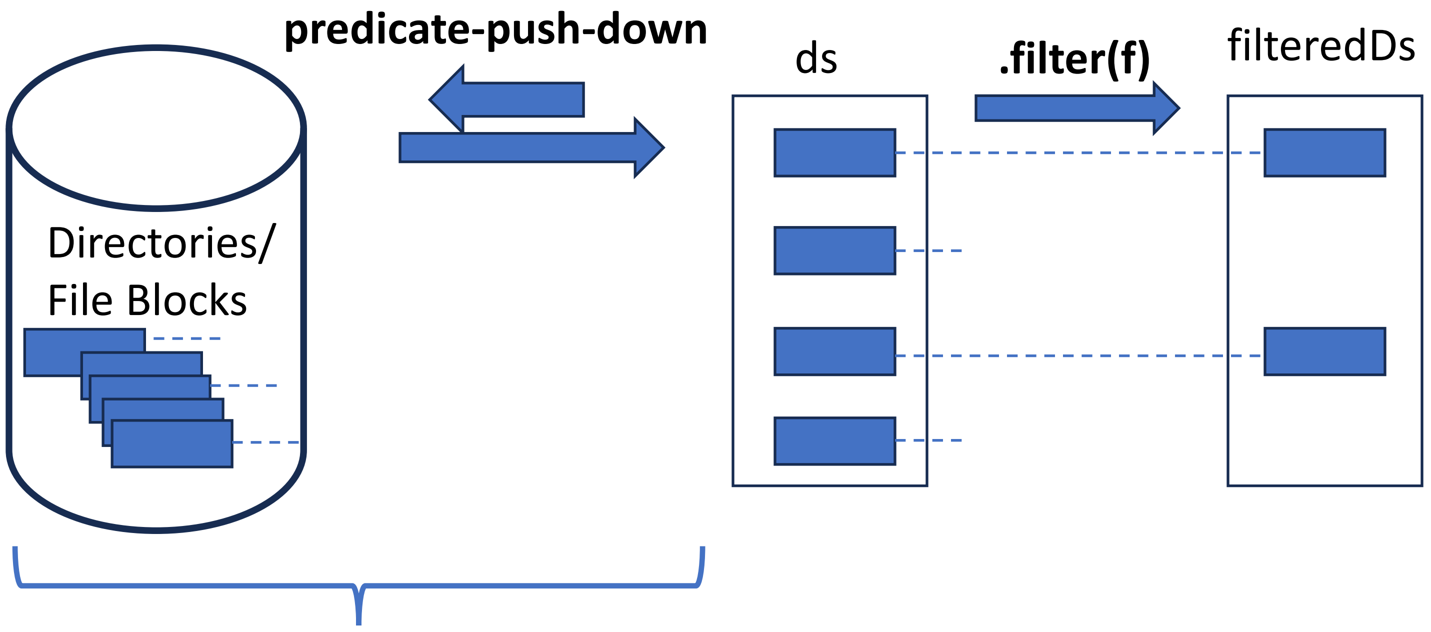
f : Function T -> boolean





dataset.filter(sqlWhere : String)
dataset.filter(columnExpr : Column)

```
Dataset<T>  ds = ...
Dataset<T>  filteredDs = ds.filter( "col = 123" )   // in SQL
                     // ~ ds.filter( ds.col("col").eq(lit(123) )  // Column api
```



Optim: Avoid reading useless Dir / Files/ blocks

.filter() <=> sql: "WHERE <...>"

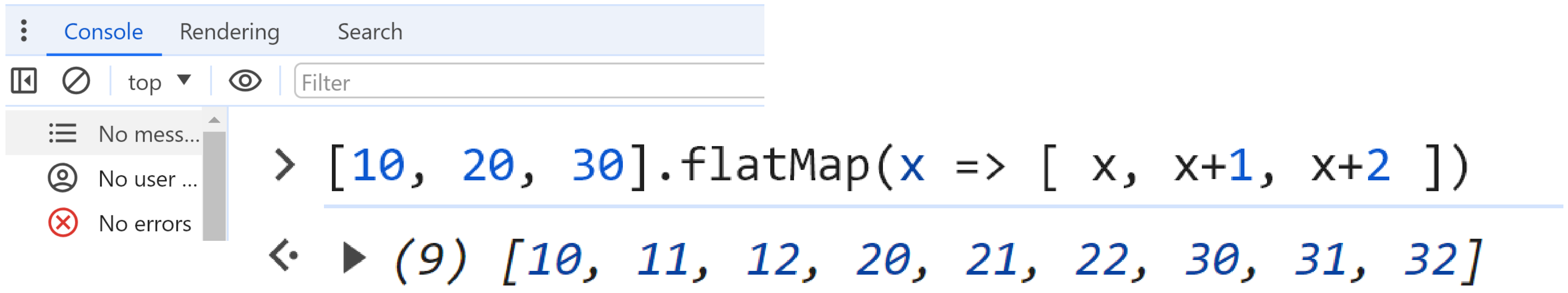
```
ds
    .filter("col1 == 1")
    .filter( col("col2").eq (lit(2))
    .filter(x => pred(x))
```

```
SELECT *
FROM ...
WHERE
          col1 = 1
  and col1 = 2
  and ... ??
```

.flatMap(row => listFunc(row))

as in any functional langage

example in JavaScript (use Chrome DevTools: F12)

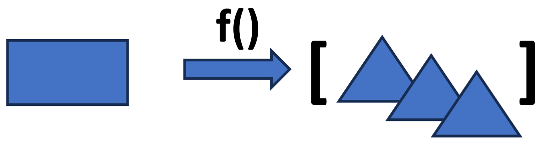


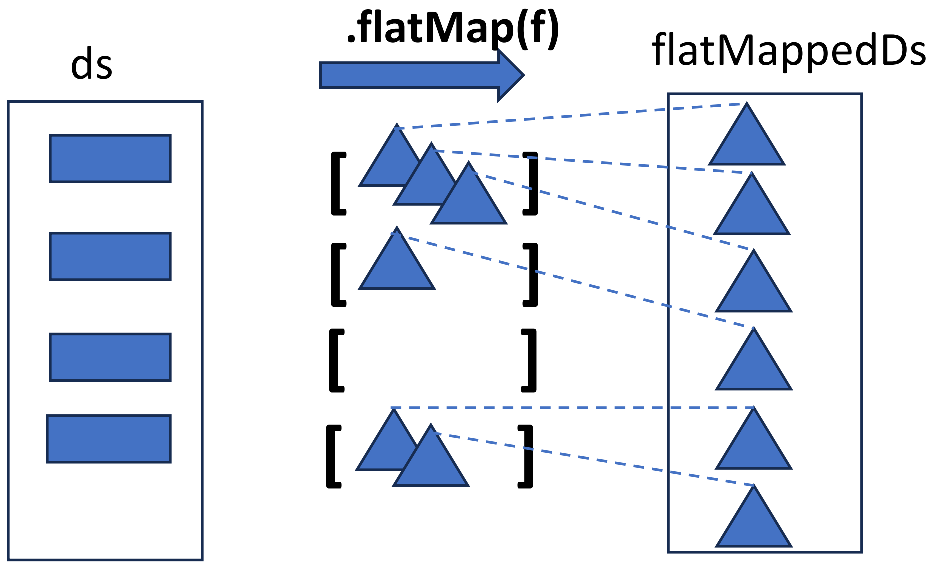
```
> [10, 20, 30].flatMap(x => [ x, x+1, x+2 ])
<· ▶ (9) [10, 11, 12, 20, 21, 22, 30, 31, 32]
```

dataset.flatMap(rowFunction : T => Iterator<U>)

```
Dataset<T>  ds = ...

def f (row: T): Iterator<U> = { ... }

Dataset<U>  flatMappedDs = ds.flatMap( row =>  f(row) )
                               // idem  = ds.flatMap(f)
```

f : Function T -> U

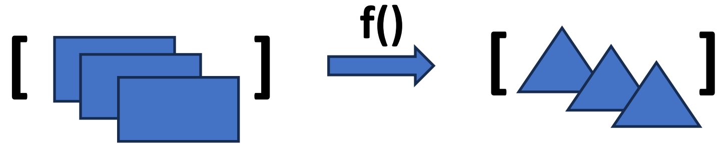


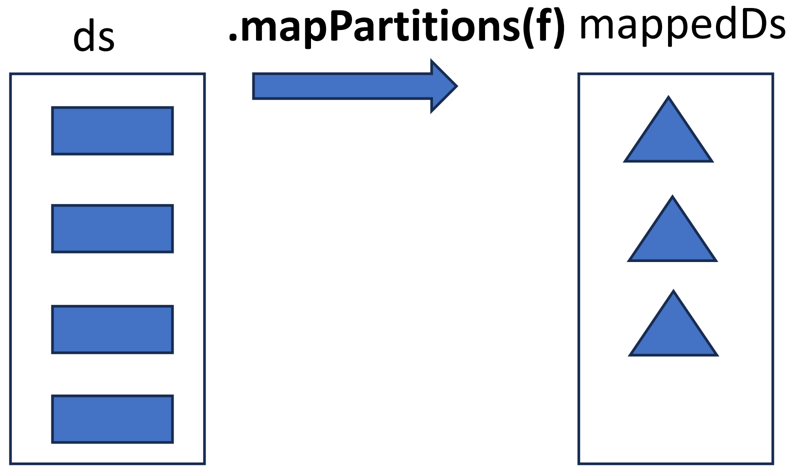


dataset.mapPartitions(rowIter => f(rowIter))

```
Dataset<T>  ds = ...

def f (iter: Iterator<T>): Iterator<U> = { ... }

Dataset<T>  mappedDs = **ds.mapPartitions(f)**
```

f : Iterator<T> -> Iterator<U>





Remarks on mapPartitions() vs .flatMap(), .map(), .filter()

Both .map() and .filter() can be implemented using .flatMap

.**map**(f)   <==> .flatMap(row => [f(row)])

.filter(pred) <==>   .flatMap(row => pred(row)? [row] : [ ])

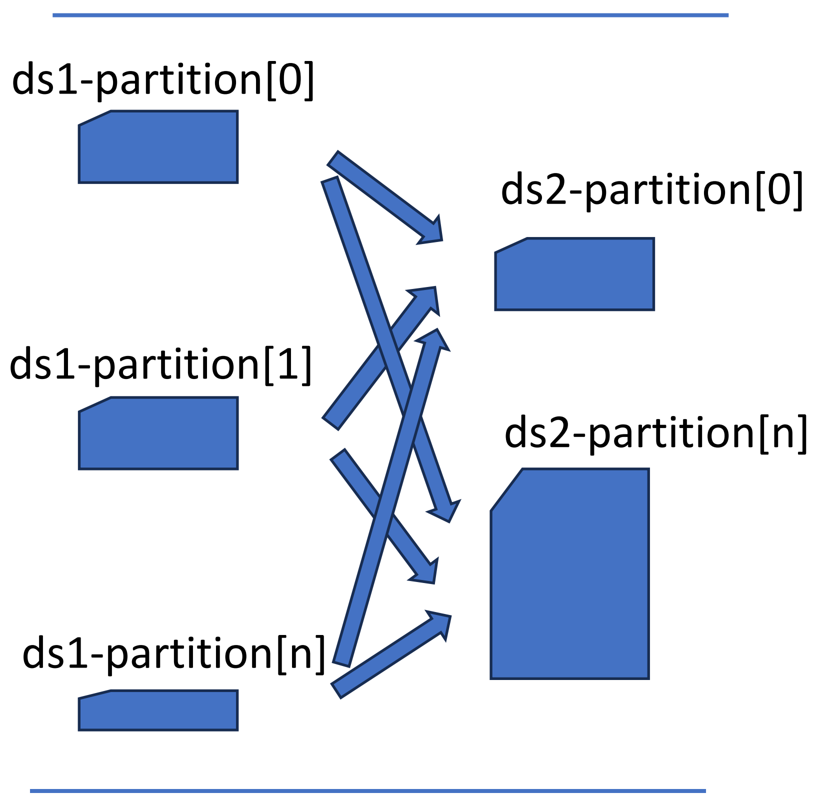
Even .flatMap()   can be implemented using .mapPartitions()

Outline

- from List<T> to distributed **Dataset<T>**
- Immutability, Functional API
- processing workflow:
  Input -> Transformations -> Output
- **narrow** operations (=per partitions)
- **wide** operations (=shuffled)

Wide Transformations

ds2 = ds1.wideTransform(..)



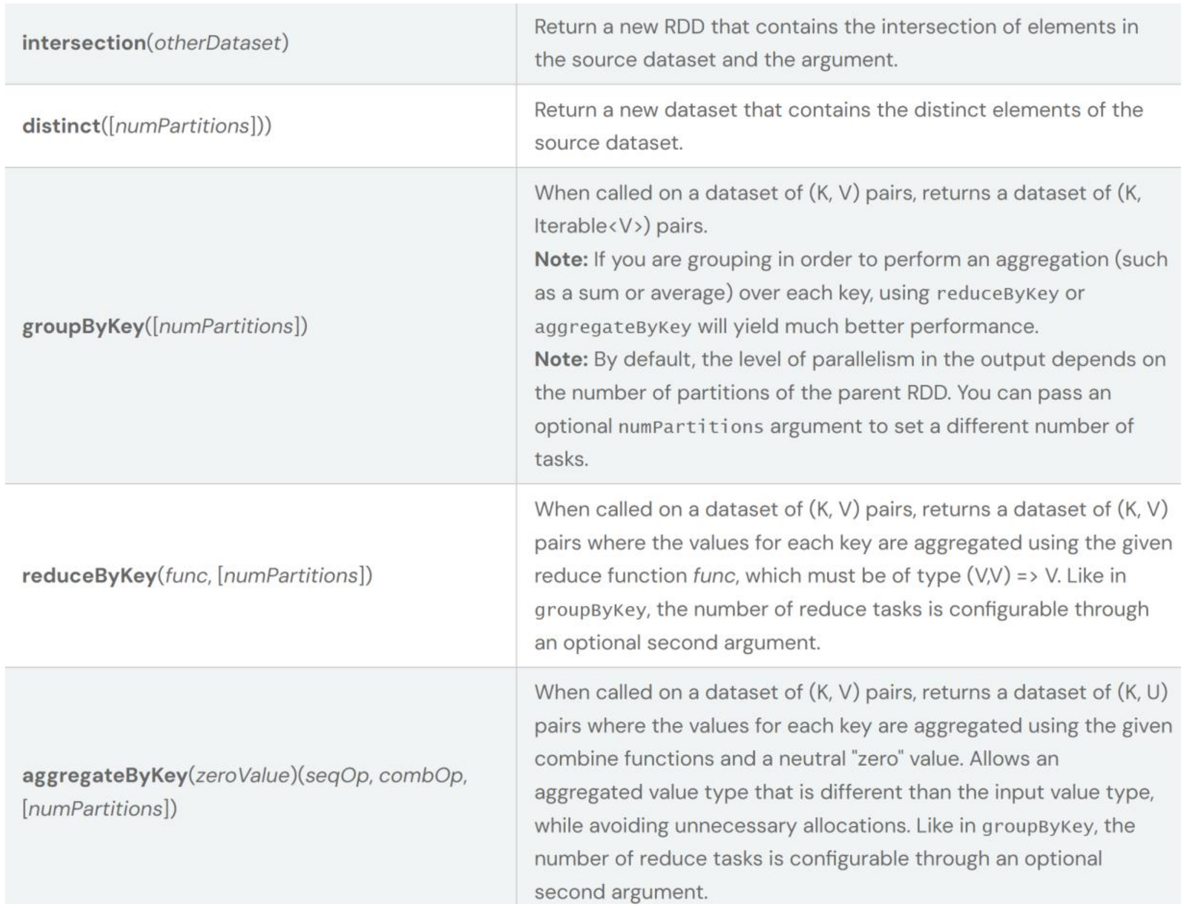
Shuffle are all inter-dependent

not necessarily preserving
partition topology (count/sizes)

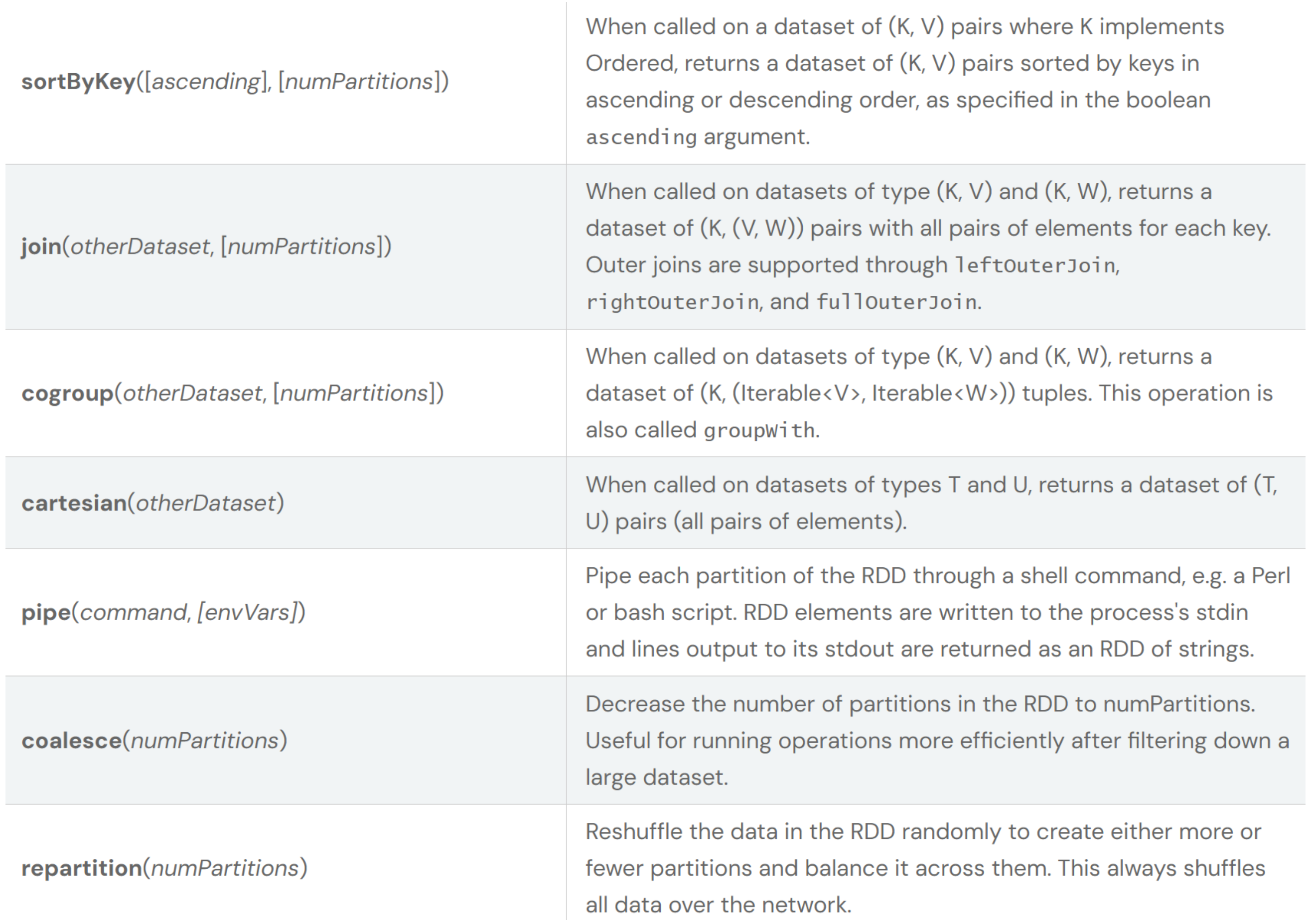
network data movements
between mapped/reduced partitions

Wide Transformations

https://spark.apache.org/docs/latest/rdd-programming-guide.html#rdd-operations



intersection(*otherDataset*)	Return a new RDD that contains the intersection of elements in the source dataset and the argument.
distinct([*numPartitions*]))	Return a new dataset that contains the distinct elements of the source dataset.
groupByKey([*numPartitions*])	When called on a dataset of (K, V) pairs, returns a dataset of (K, Iterable<V>) pairs. **Note:** If you are grouping in order to perform an aggregation (such as a sum or average) over each key, using `reduceByKey` or `aggregateByKey` will yield much better performance. **Note:** By default, the level of parallelism in the output depends on the number of partitions of the parent RDD. You can pass an optional `numPartitions` argument to set a different number of tasks.
reduceByKey(*func*, [*numPartitions*])	When called on a dataset of (K, V) pairs, returns a dataset of (K, V) pairs where the values for each key are aggregated using the given reduce function *func*, which must be of type (V,V) => V. Like in `groupByKey`, the number of reduce tasks is configurable through an optional second argument.
aggregateByKey(*zeroValue*)(*seqOp*, *combOp*, [*numPartitions*])	When called on a dataset of (K, V) pairs, returns a dataset of (K, U) pairs where the values for each key are aggregated using the given combine functions and a neutral "zero" value. Allows an aggregated value type that is different than the input value type, while avoiding unnecessary allocations. Like in `groupByKey`, the number of reduce tasks is configurable through an optional second argument.



sortByKey([*ascending*], [*numPartitions*])	When called on a dataset of (K, V) pairs where K implements Ordered, returns a dataset of (K, V) pairs sorted by keys in ascending or descending order, as specified in the boolean `ascending` argument.
join(*otherDataset*, [*numPartitions*])	When called on datasets of type (K, V) and (K, W), returns a dataset of (K, (V, W)) pairs with all pairs of elements for each key. Outer joins are supported through `leftOuterJoin`, `rightOuterJoin`, and `fullOuterJoin`.
cogroup(*otherDataset*, [*numPartitions*])	When called on datasets of type (K, V) and (K, W), returns a dataset of (K, (Iterable<V>, Iterable<W>)) tuples. This operation is also called `groupWith`.
cartesian(*otherDataset*)	When called on datasets of types T and U, returns a dataset of (T, U) pairs (all pairs of elements).
pipe(*command*, *[envVars]*)	Pipe each partition of the RDD through a shell command, e.g. a Perl or bash script. RDD elements are written to the process's stdin and lines output to its stdout are returned as an RDD of strings.
coalesce(*numPartitions*)	Decrease the number of partitions in the RDD to numPartitions. Useful for running operations more efficiently after filtering down a large dataset.
repartition(*numPartitions*)	Reshuffle the data in the RDD randomly to create either more or fewer partitions and balance it across them. This always shuffles all data over the network.

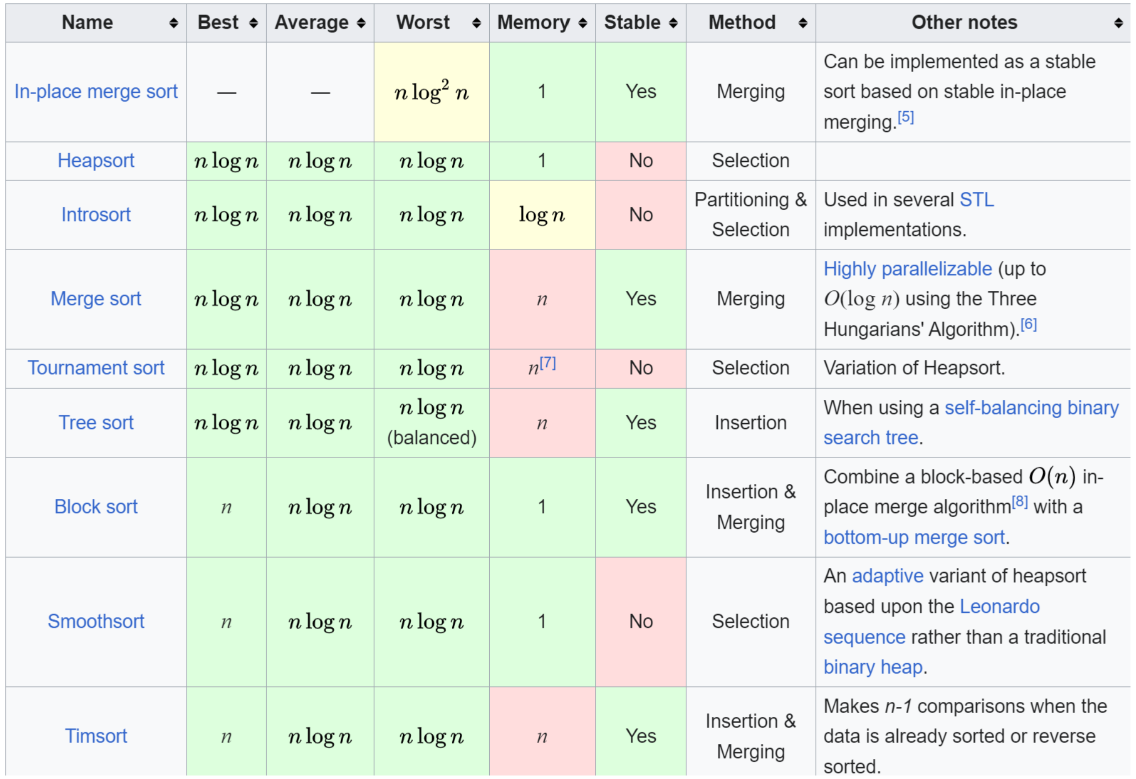
Wide Transformations… focus on

```
.sortByKey( [col1, col2.. ] )

.repartition( N )
.repartition( [col1,col2..],  N)

.join( otherDataset, joinedCols)
```

Local Sorting

non-distributed sorting algorithms are classical
best "local" complexity = **N x log(N) ops / N memory size**
ex: QuickSort, TimSort, MergeSort, CountingSort, RadixSort, ...

Problem : **how to distribute ?**



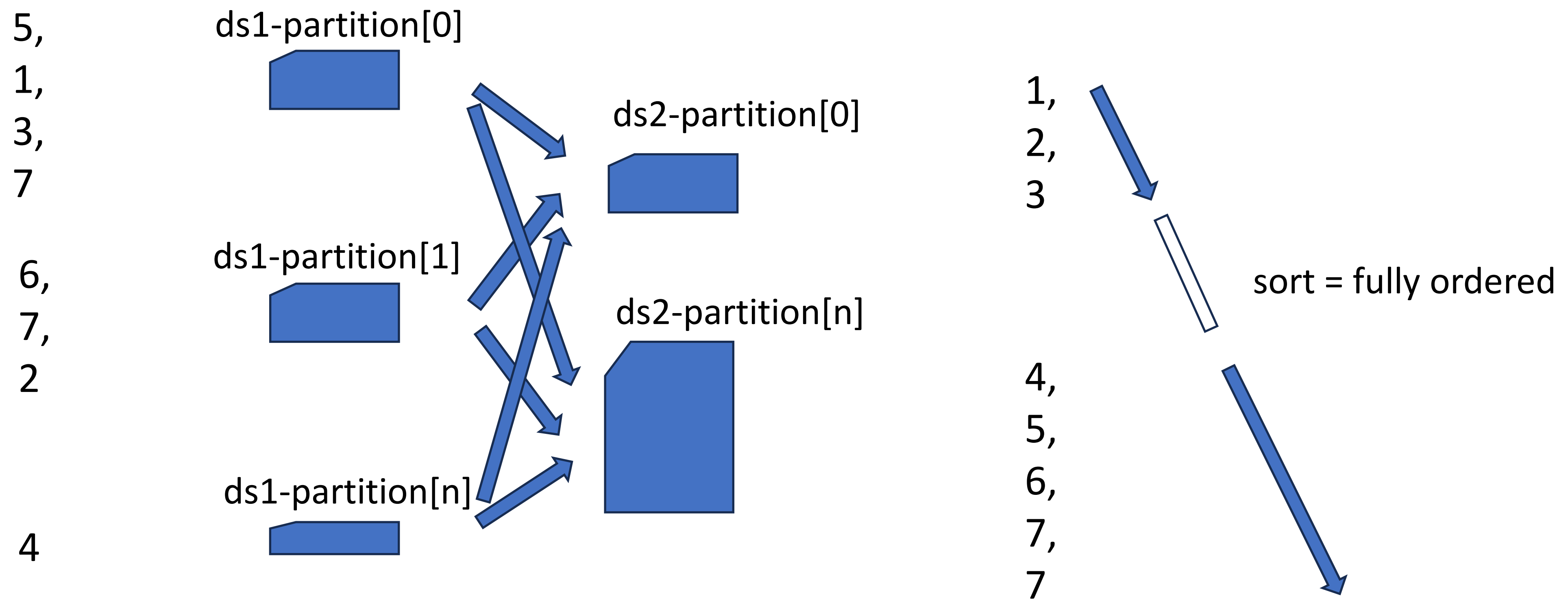
Name	Best	Average	Worst	Memory	Stable	Method	Other notes
In-place merge sort	—	—	$n\log^2 n$	1	Yes	Merging	Can be implemented as a stable sort based on stable in-place merging.[5]
Heapsort	$n\log n$	$n\log n$	$n\log n$	1	No	Selection	
Introsort	$n\log n$	$n\log n$	$n\log n$	$\log n$	No	Partitioning & Selection	Used in several STL implementations.
Merge sort	$n\log n$	$n\log n$	$n\log n$	n	Yes	Merging	Highly parallelizable (up to $O(\log n)$ using the Three Hungarians' Algorithm).[6]
Tournament sort	$n\log n$	$n\log n$	$n\log n$	n[7]	No	Selection	Variation of Heapsort.
Tree sort	$n\log n$	$n\log n$	$n\log n$ (balanced)	n	Yes	Insertion	When using a self-balancing binary search tree.
Block sort	n	$n\log n$	$n\log n$	1	Yes	Insertion & Merging	Combine a block-based $O(n)$ in-place merge algorithm[8] with a bottom-up merge sort.
Smoothsort	n	$n\log n$	$n\log n$	1	No	Selection	An adaptive variant of heapsort based upon the Leonardo sequence rather than a traditional binary heap.
Timsort	n	$n\log n$	$n\log n$	n	Yes	Insertion & Merging	Makes *n-1* comparisons when the data is already sorted or reverse sorted.

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

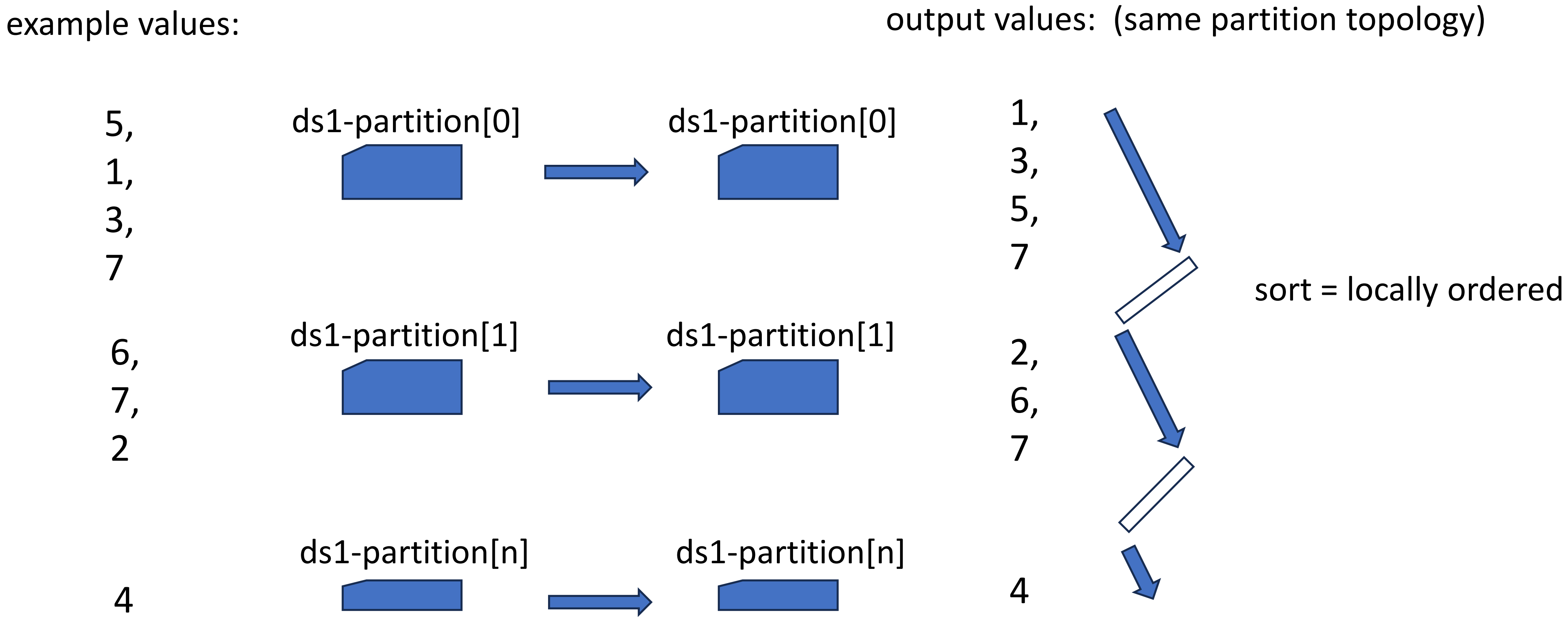
Dataset Sort

example values:

example split output values:
(2 partitions, split at .. < 4 <= ..)



!= Narrow (Local) .sortWithinPartitions()



API: .sort(col1, ... colN)   synonym: .orderBy()

Dataset<T> ds1 = ...

Dataset<T> ds2 = ds1**.sort**("col1", "col2");

equivalent: **.orderBy**("col1", "col2")

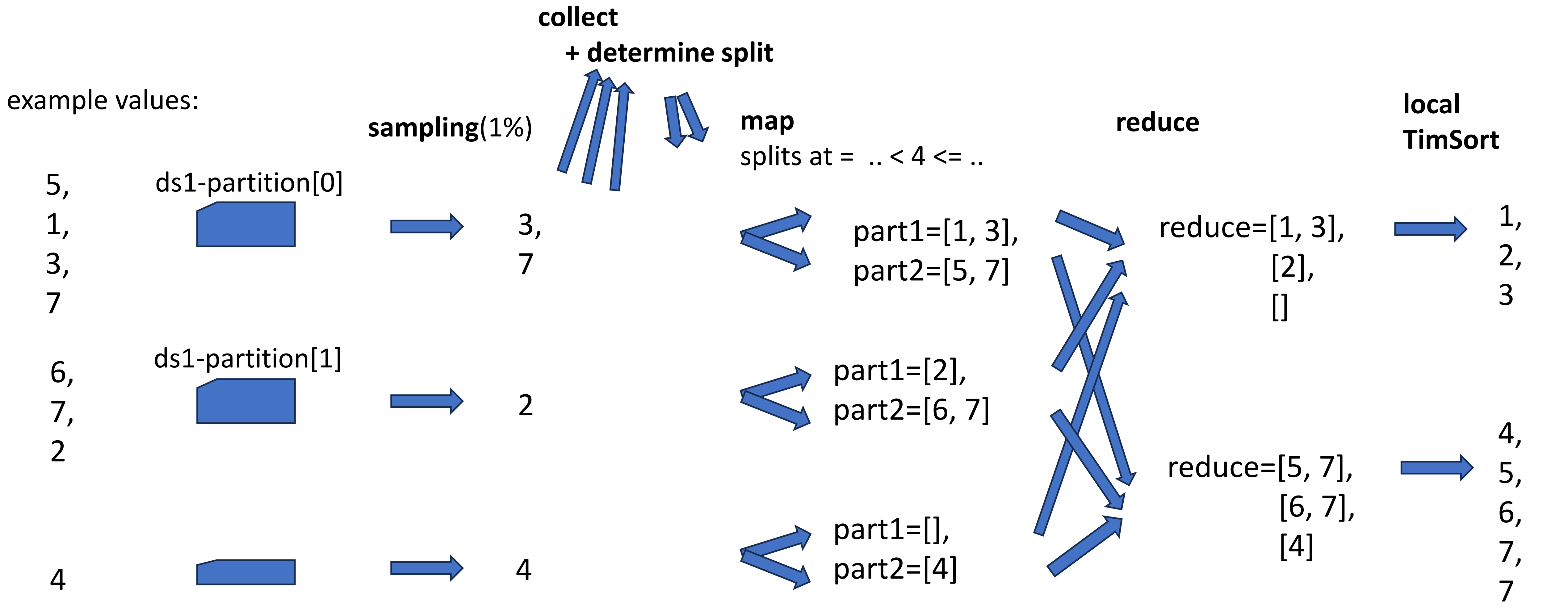
.sort(col("col1"), **col**("col2") **)**

see also

.orderBy(col("col1")**.ascending**, col("col2")**.descending**)
.orderBy(col("col1").ascending**.null_first**, col("col2").descending**.null_last**)

NOTICE: no lambda, nor "comparator" objects

Internal Sort Algorithm = **Sampling** values + Determine **Split** limits + **Repartition** by Range + **TimSort**



.sort() - SQL: "ORDER BY" (!= "SORT BY")

```
ds
   .sort(col("col1"),
           col("col2").descending )
```

⟺

```
SELECT *
FROM ...
ORDER BY col1, col2 DESC
```

(Standard SQL)

```
ds
   .sortWithinPartitions("col3")
```

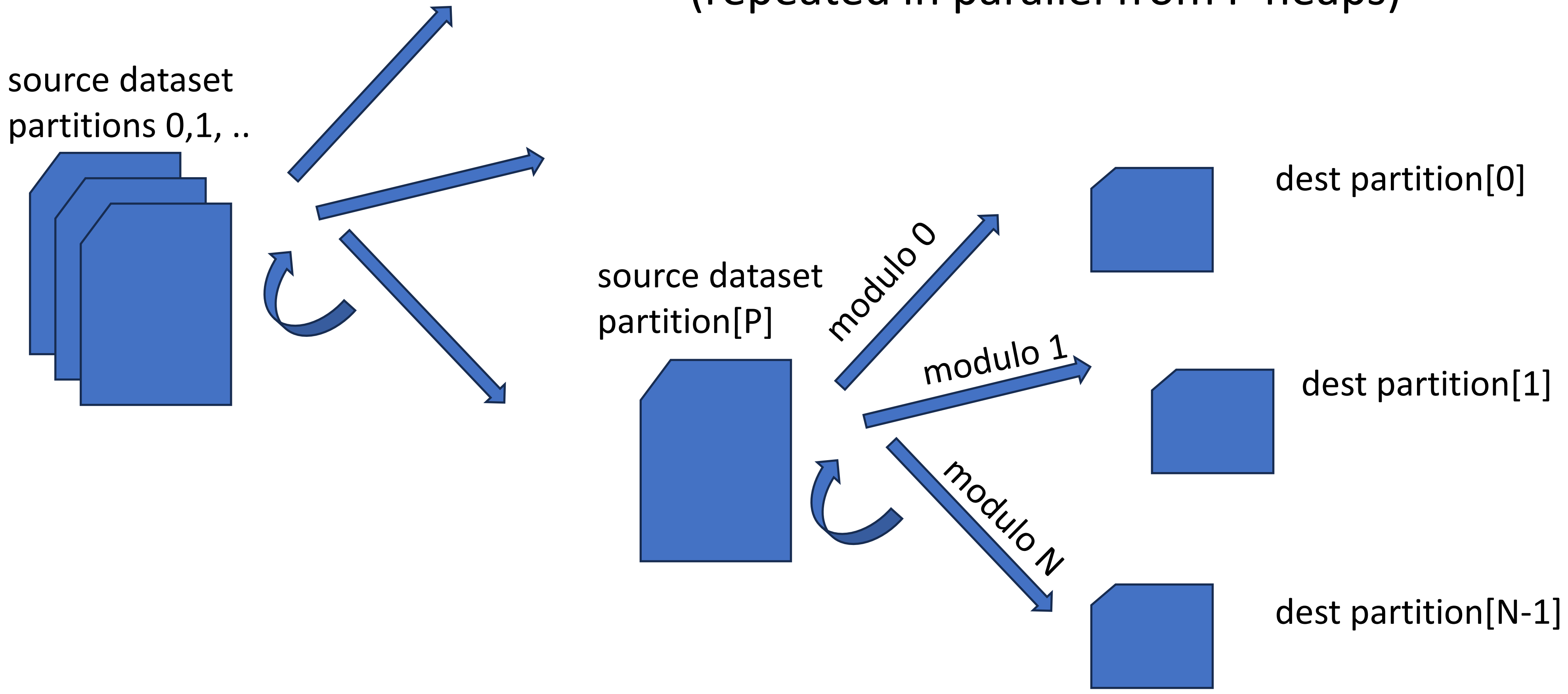
⟺

```
SELECT *
FROM ...
SORT BY col3
```

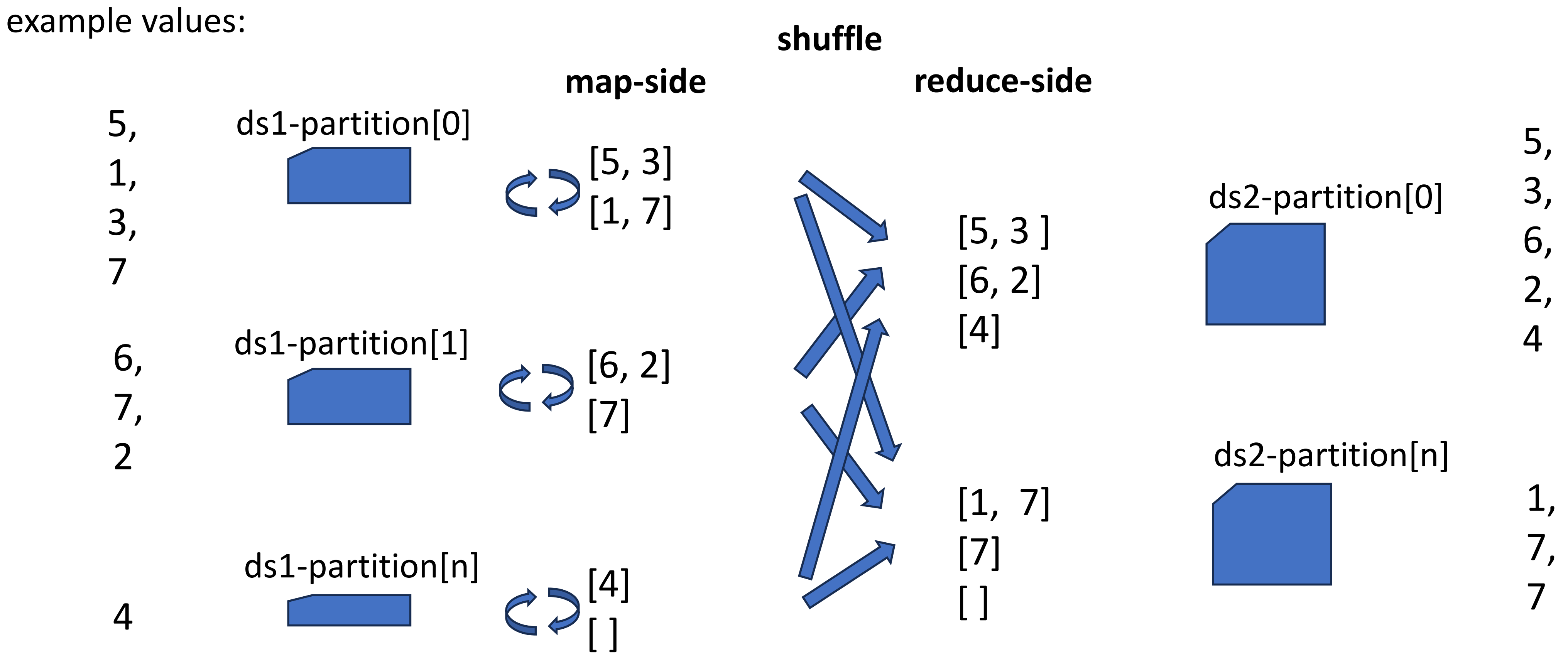
NON-standard
!! SQL Extension !!

.repartition(N)
Round-Robin Repartition

same as **Dealing Cards** to N players
(repeated in parallel from P heaps)



.repartition(N)
= round-robin Map + Reduce



.repartition(columns)   groupBy

.repartition (cardFamily)   => 4

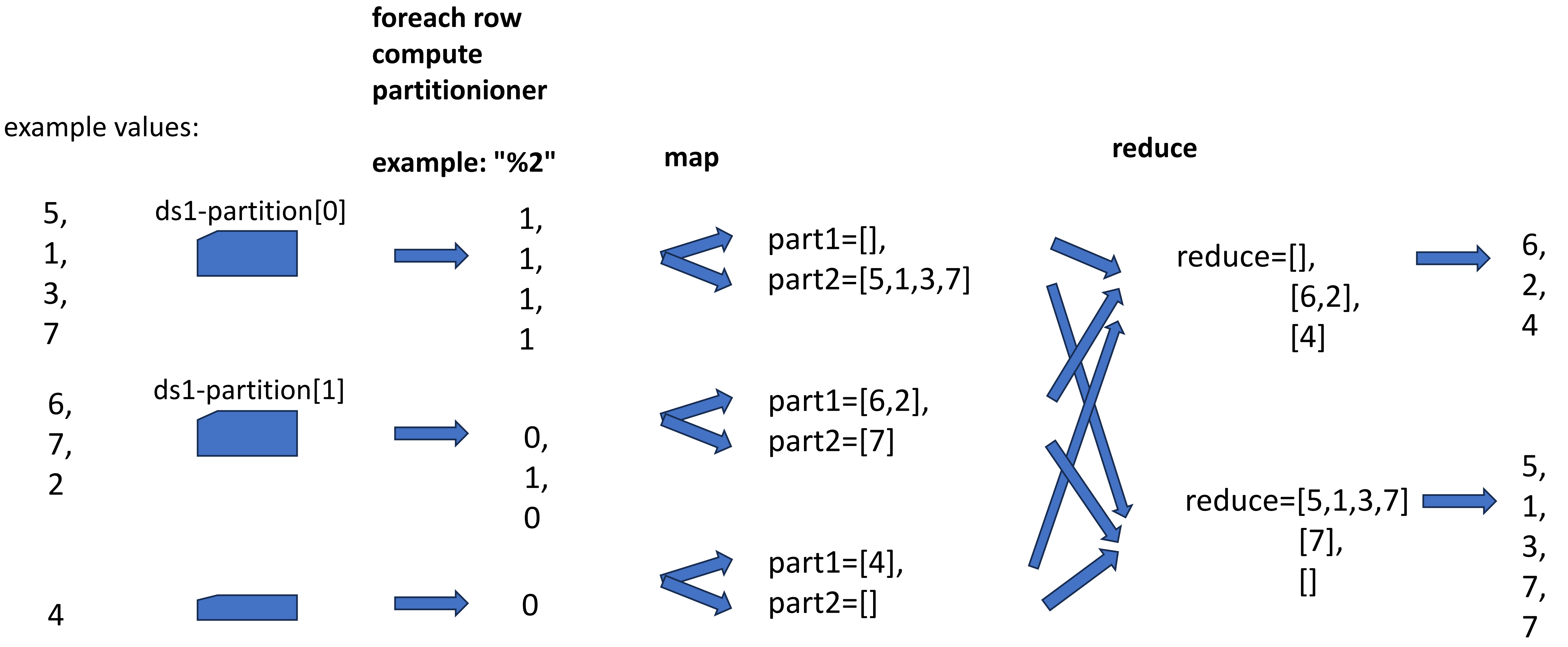
.repartition (cardColor [red/black])   => 2

.repartition (cardValue[1,2,3..])   => 13

.repartition (cardFamily, cardValue)   => 52 = 4*13

.repartition ([cardFamily, cardValue], 20)   => 20

.repartition(column) = Mapper groupBy - Reduce



.repartition(col) <=> sql: "GROUP BY <...>"

```
ds
    .repartition("col1", "col2")
    .mapPartitions(..)
```

⇔

```
SELECT  col1, col2,
            sum(*),count(*),min(*), distinct
            <<analytical>>(..)
FROM ...
GROUP BY col1, col2
```

.repartition(col1..colP, **hashModuloN**)

when col1,col2,..colP   give too many repartition groups (millions of groups ?)

=> spark will compute hashCode, then modulo default=200

You can change globally "**spark.shuffle.partitions**" (200)
or specifically by call

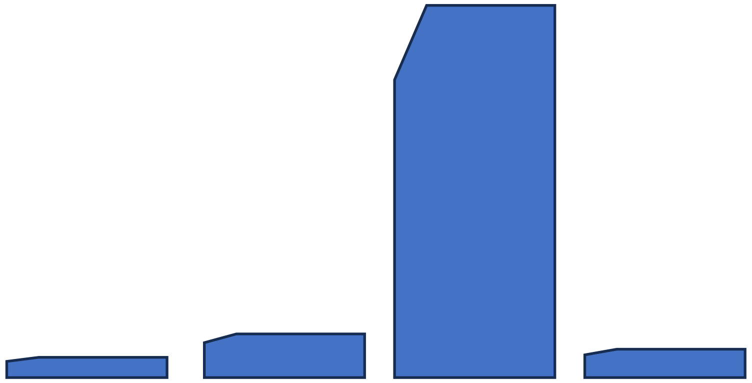
.repartition(col1, col2)

➡ equivalent: .repartition(col1, col2, N=200)

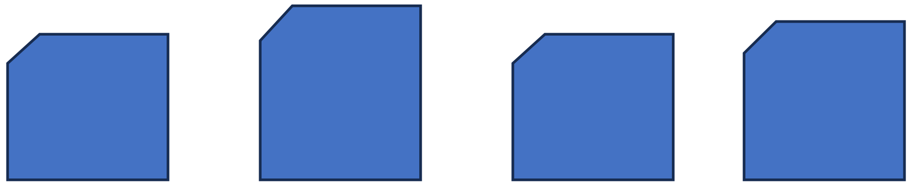
➡ partitioner function:

func(row) { abs(hash(row.col1 ^ row.col2)) % N }

Transformations to Skewed or Well-Balanced Partitions ?



Skewed



Well Balanced / Equi-Distributed

Transformations to Skewed or Well-Balanced Partitions ?

.sort()   =>   **sampling randomness** might produce unbalanced data
but generally ok

.repartition(N)   =>   exactly N x **equi-distributed** +/-1 x P rows

.repartition(col1, col2)   => most probably **SKEWED DATA** !!
(must choose [col1,col2] carefully)

.repartition(highCardinalityCol, N)   => most probably N x equi-distributed

.repartition(highCardinalityCol)   => most probably 200 x equi-distributed

Joins

```
Dataset<T>   ds1 =  ...

Dataset<U>   ds2 =  ...

Dataset<Pair<T,U>>   joinedDs =  ds1.join( ds2,  joinExpr,  joinType)
```

Join - SQL "FROM .. JOIN .. ON .."

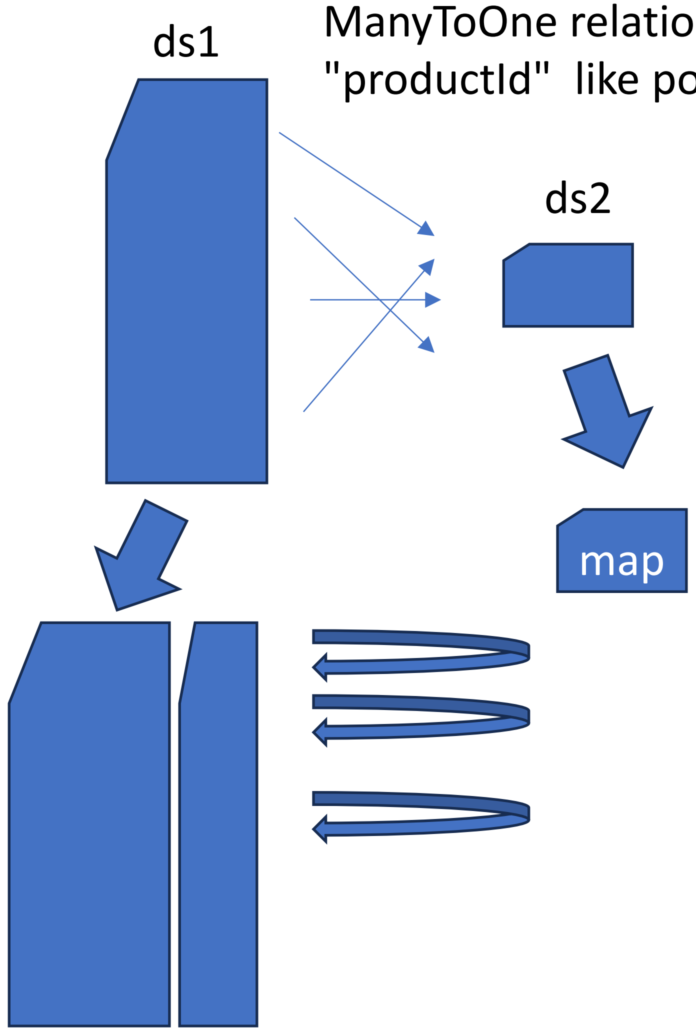
Typical Star(*) Schema: 1 Big **Fact** table, N small **Dimensions** Tables

```
Dataset<Row>  sellDs = ..  // FACT table (big)  has foreign key to "productId"

Dataset<Row>  productDs = .. // Dimension table (small) .. has primaryKey "id"


Dataset<Row> sellEnrichedDs =  sellDs.join(productDs,
                                       sellDs.col("productId") == productDs.col("id"),
                                       "left-outer");
```

SQL:

SELECT s.*, p.*
  FROM Sell s
  LEFT OUTER JOIN Product p **ON** s.productId = p.id

Local Join Algorithm using right HashMap



ManyToOne relationship
"productId" like pointer to PK "id"

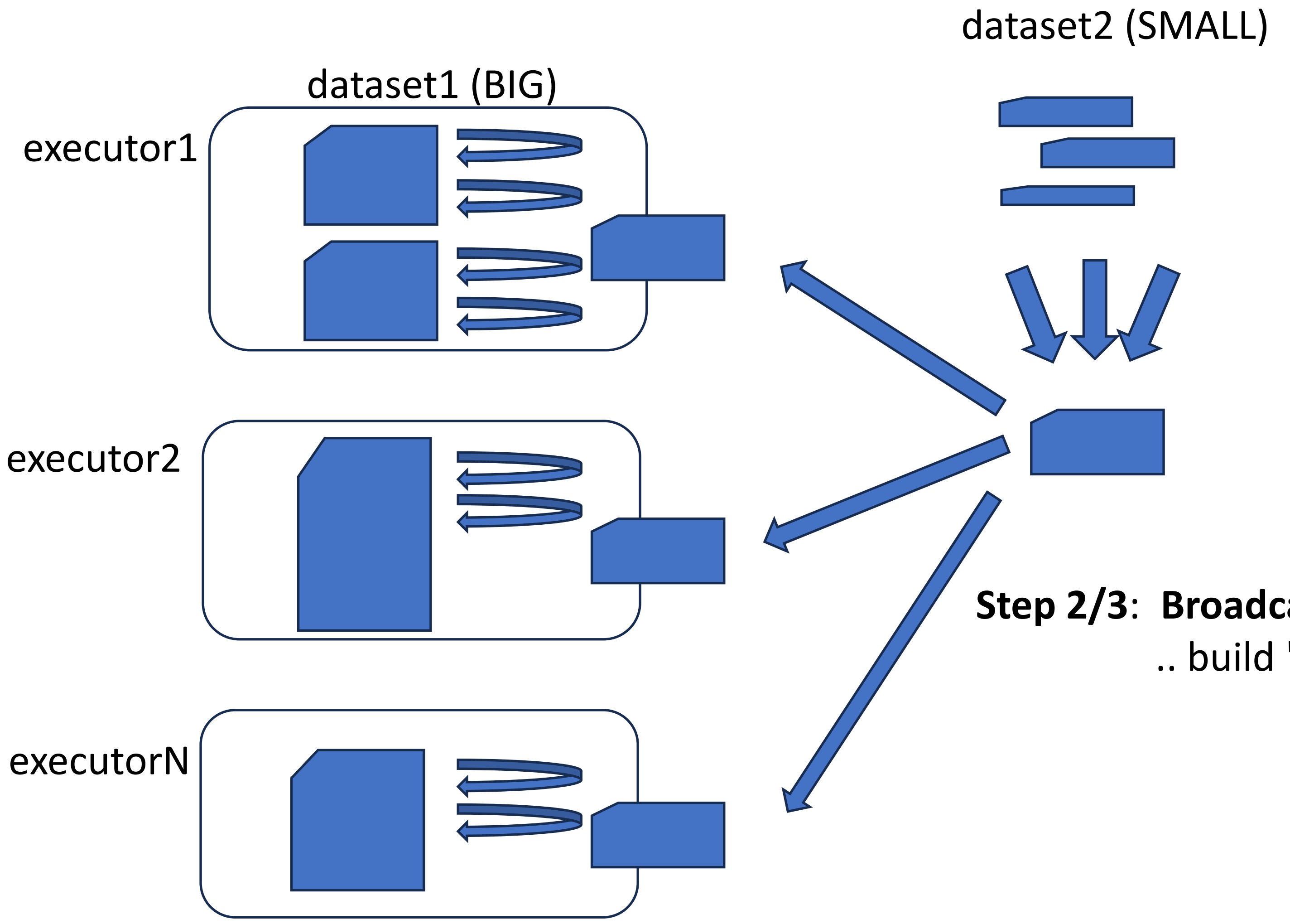
Step 1/2: prepare index all ds2 rows by id in HashMap

```
ds2HashMap = new HashMap<int,T2>();
for(e : ds2) {
  ds2HashMap.put(e.id, e);
}
```

Step 2/2: foreach item in ds1, lookup corresponding ds2 row by joinId

```
res = new ArrayList<Pair<T,U>>();
for(e : ds1) {
  res.add(new Pair(e,  ds2HashMap.get(e.joinId) ) );
}
```

Spark .. BroadcastHashJoin

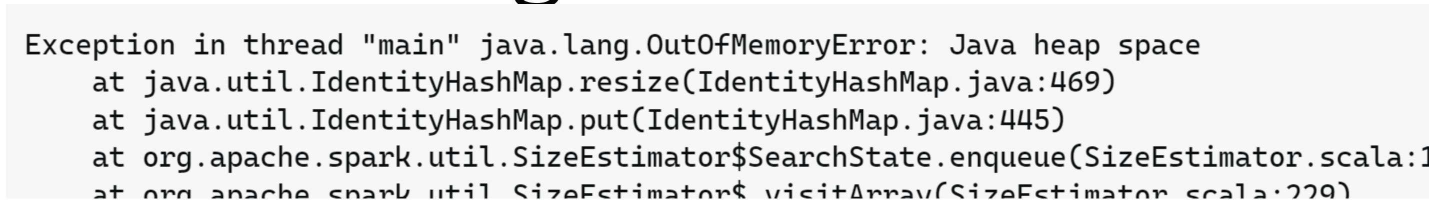


Step 1/3: **collect** ALL ds2 values on driver

Step 2/3: **Broadcast** (copy) of ds2 to all executors
.. build "HashMap" full local copies

Step 3/3: **foreach** item in ds1, **lookup** corresponding ds2 in HashMap by joinId

Problem ... How to Join 2 Big Datasets ?

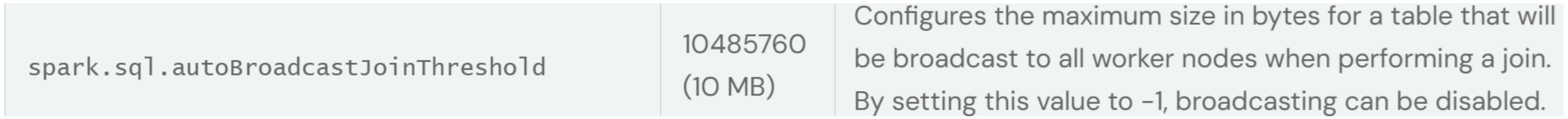


```
Exception in thread "main" java.lang.OutOfMemoryError: Java heap space
    at java.util.IdentityHashMap.resize(IdentityHashMap.java:469)
    at java.util.IdentityHashMap.put(IdentityHashMap.java:445)
    at org.apache.spark.util.SizeEstimator$SearchState.enqueue(SizeEstimator.scala:1
    at org.apache.spark.util.SizeEstimator$.visitArray(SizeEstimator.scala:229)
```

can NOT **collect** data to a single driver
so can not **broadcast**

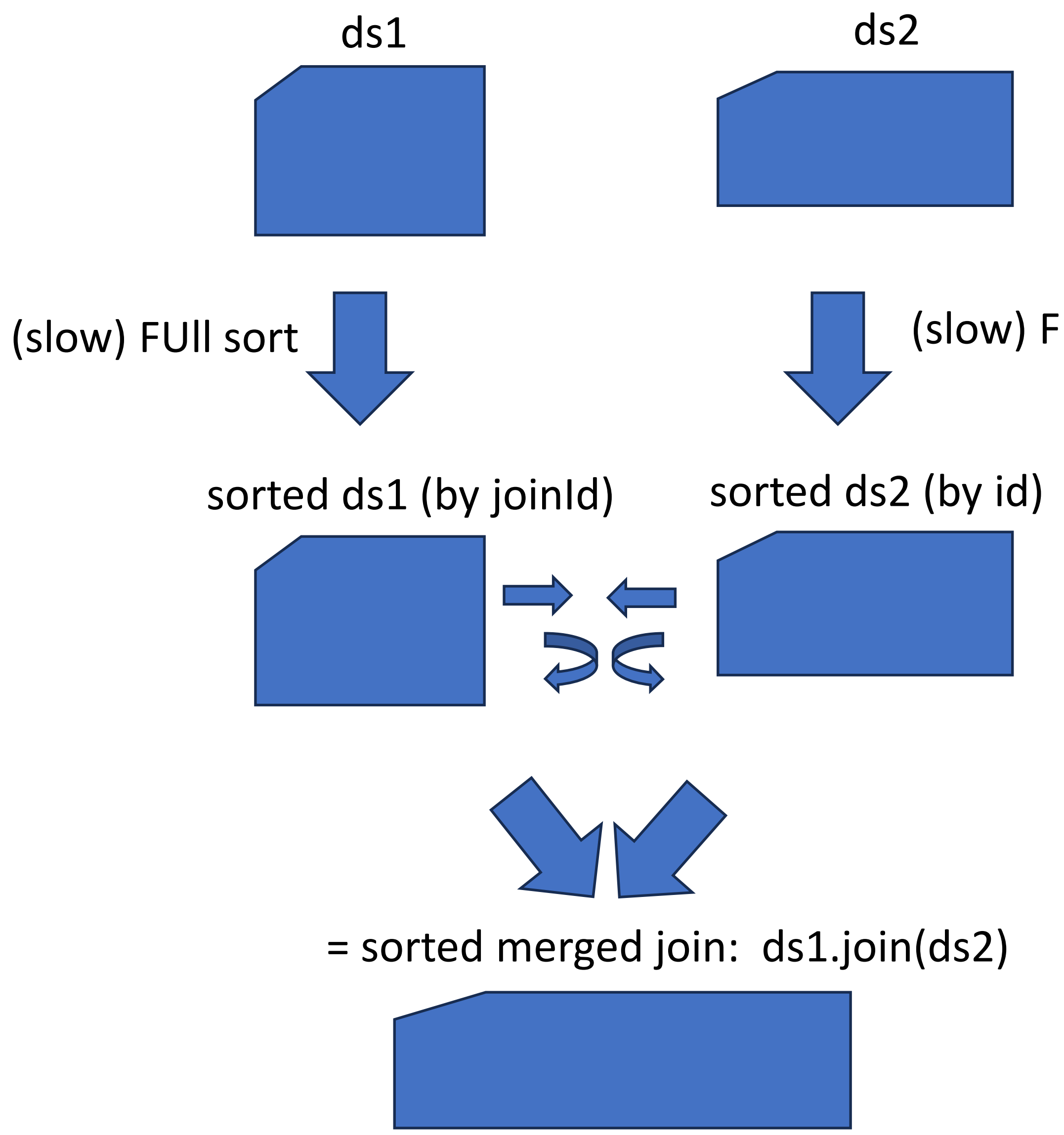
& can not have **copy** of data on each N x executors

spark conf:



`spark.sql.autoBroadcastJoinThreshold`	10485760 (10 MB)	Configures the maximum size in bytes for a table that will be broadcast to all worker nodes when performing a join. By setting this value to -1, broadcasting can be disabled.

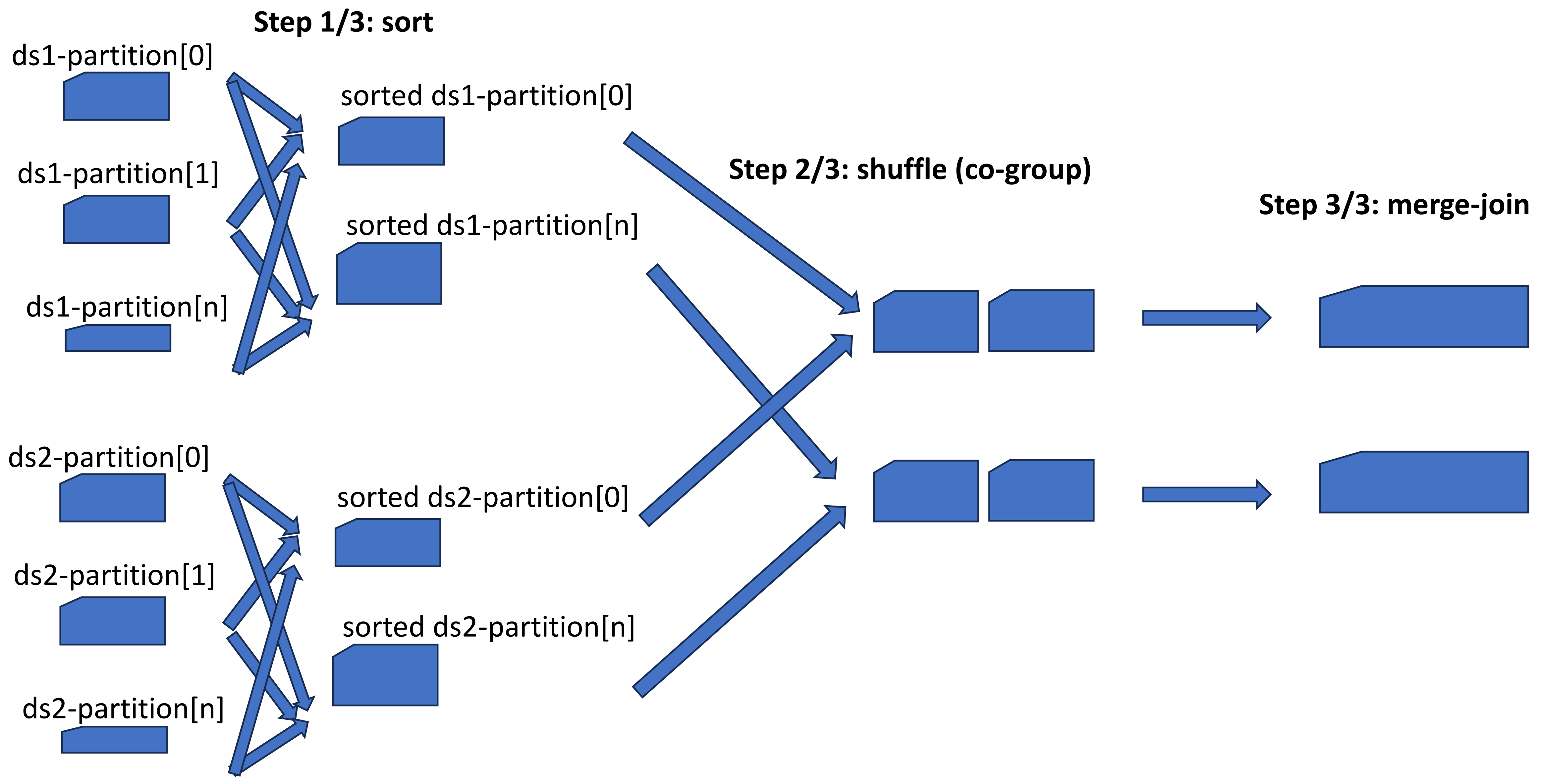
Sort Merge Join Algorithm



iterator in both ds1 AND ds2
simultaneously:

```
if (iter1 > iter2)   iter1.next()
else if (iter1 < iter2)   iter2.next()
else {  join..
      iter1.next(); iter2.next()
}
```

(Distributed) Sort Merge Join



Outline

- from List<T> to distributed **Dataset<T>**
- Immutability, Functional API
- processing workflow:
  Input -> Transformations -> Output
- **narrow** operations (=per partitions)
- **wide** operations (=shuffled)

➡ **Conclusion, Next Steps**

Conclusion

Only a "Short" Introduction to "Big Data" Distributed operations challenges

Dataset = **distributed List on a cluster**,
the sky is the limit
Immutable and using **Functional API**
implements basic operators (narrow and wide)

The core fondamentals of Spark for processings

Next Steps

cf Lessons 2, 3

- Spark Architecture

- Parquet file Format (Dir & Files, partitions, columnar, Optims..)

- ...