Handling data skew adaptively in Spark using Dynamic Repartitioning

Zoltán Zvara
zoltan.zvara@sztaki.hu

MTA SZTAKI

ERICSSON

Introduction

- Hungarian Academy of Sciences, Institute for Computer Science and Control (MTA SZTAKI)

- Research institute with strong industry ties

- Big Data projects using Spark, Flink, Cassandra, Hadoop etc.

- Multiple telco use cases lately, with challenging data volume and distribution

MTA SZTAKI

Agenda

- Our data-skew story

- Problem definitions & aims

- Dynamic Repartitioning
  - Architecture
  - Component breakdown
  - Repartitioning mechanism
  - Benchmark results

- Visualization

- Conclusion

Motivation

- We have developed an application aggregating telco data that tested well on toy data

- When deploying it against the real dataset the application seemed healthy

- However it could become surprisingly slow or even crash

- What did go wrong?
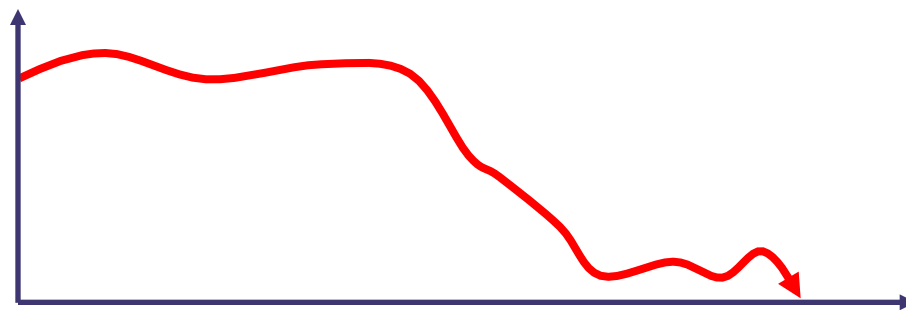
Our data-skew story

- We have use-cases when map-side combine is not an option: **`groupBy, join`**

- 80% of the traffic generated by 20% of the communication towers

- Most of our data is 80–20 rule

The problem

- Using default hashing is not going to distribute the data uniformly
- Some *unknown* partition(s) to contain a lot of records on the reducer side
- Slow tasks will appear
- Data distribution is not known in advance
- „Concept drifts" are common
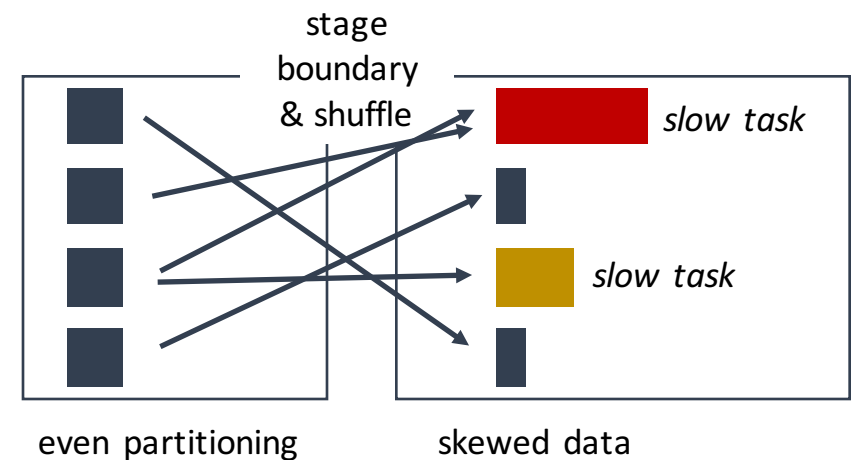
Aim

Generally, make Spark a **data**-aware distributed **data**-processing framework

- Collect information about the data-characteristics on-the-fly
- Partition the data as uniformly as possible on-the-fly
- Handle arbitrary data distributions
- Mitigate the problem of slow tasks
- Be lightweight and efficient
- Should not require user-guidance

```
spark.repartitioning = true
```

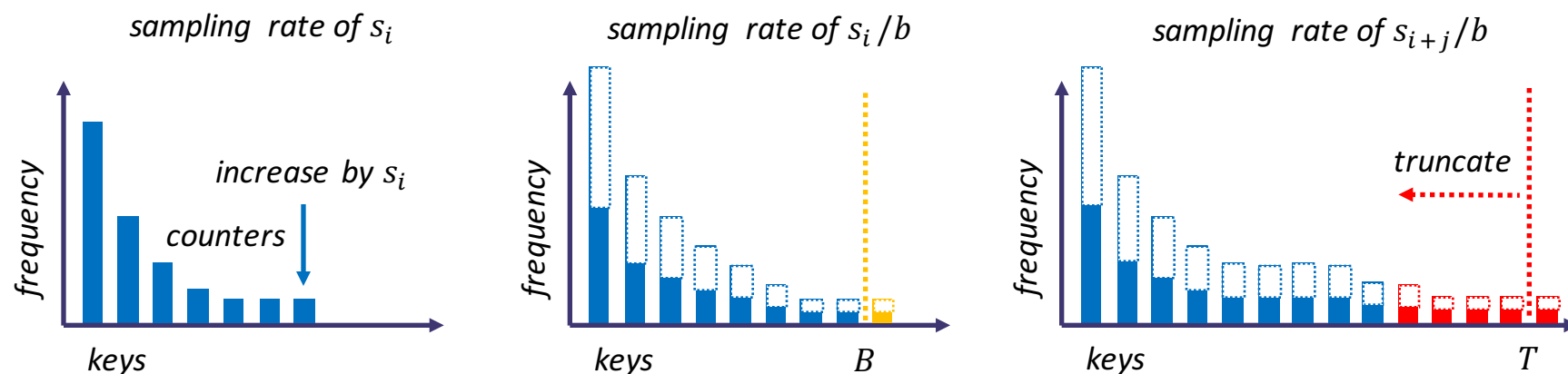# Architecture
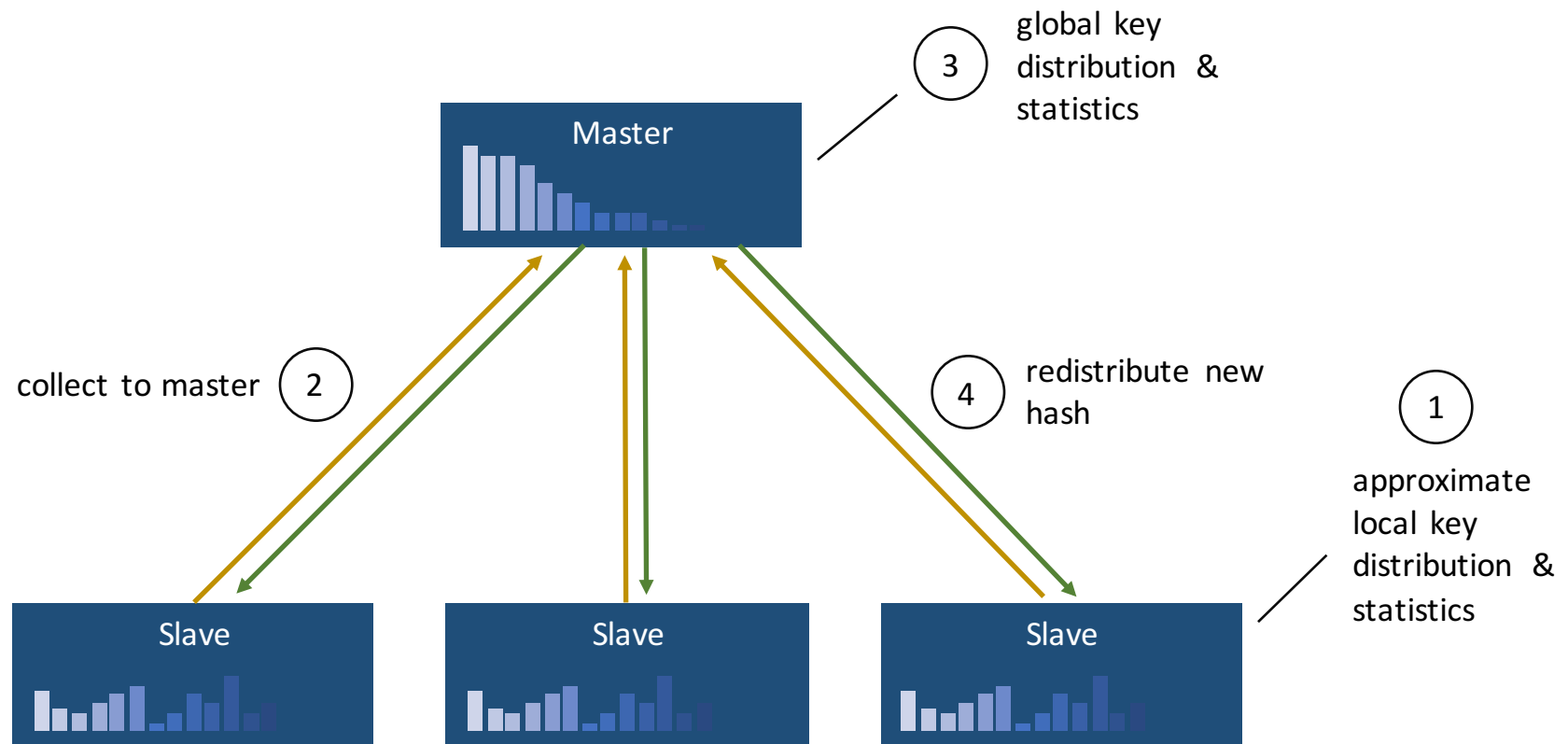
Driver perspective

- `RepartitioningTrackerMaster` part of `SparkEnv`
- Listens to job & stage submissions
- Holds a variety of repartitioning strategies for each job & stage
- Decides when & how to (re)partition

Executor perspective

- `RepartitioningTrackerWorker` part of `SparkEnv`
- Duties:
  - Stores `ScannerStrategies` (`Scanner` included) received from the RTM
  - Instantiates and binds `Scanners` to `TaskMetrics` (where data-characteristics is collected)
  - Defines an interface for `Scanners` to send `DataCharacteristics` back to the RTM

Scalable sampling

- Key-distributions are approximated with a strategy, that is
  - not sensitive to early or late concept drifts,
  - lightweight and efficient,
  - scalable by using a backoff strategy

sampling rate of s_i

frequency

increase by s_i

counters

keys

sampling rate of s_i/b

frequency

keys

B

sampling rate of s_{i+j}/b

frequency

truncate

keys

T

Complexity-sensitive sampling

- Reducer run-time can highly correlate with the *computational complexity* of the values for a given key
- Calculating object size is costly in JVM and in some cases shows little correlation to computational complexity (function on the next stage)
- Solution:
  - If the user object is `Weightable`, use complexity-sensitive sampling
  - When increasing the counter for a specific value, consider its *complexity*

Scalable sampling in numbers

- In Spark, it has been implemented with Accumulators
  - Not so efficient, but we wanted to implement it with minimal impact on the existing code
- Optimized with micro-benchmarking
- Main factors:
  - Sampling strategy - *aggressiveness* (*initial sampling ratio, back-off factor, etc*...)
  - Complexity of the current stage (mapper)
- Current stage's runtime:
  - When used throughout the execution of the whole stage it adds 5-15% to runtime
  - After repartitioning, we cut out the sampler's code-path; in practice, it adds **0.2-1.5%** to runtime

Main new task-level metrics

- `TaskMetrics`
  - `RepartitioningInfo` – new hash function, repartition strategy
  - `ShuffleReadMetrics`
    - `(Weightable)DataCharacteristics` – used for testing the correctness of the repartitioning & for execution visualization tools
  - `ShuffleWriteMetrics`
    - `(Weightable)DataCharacteristics` – scanned periodically by `Scanners` based on `ScannerStrategy`
    - `insertionTime`
    - `repartitioningTime`
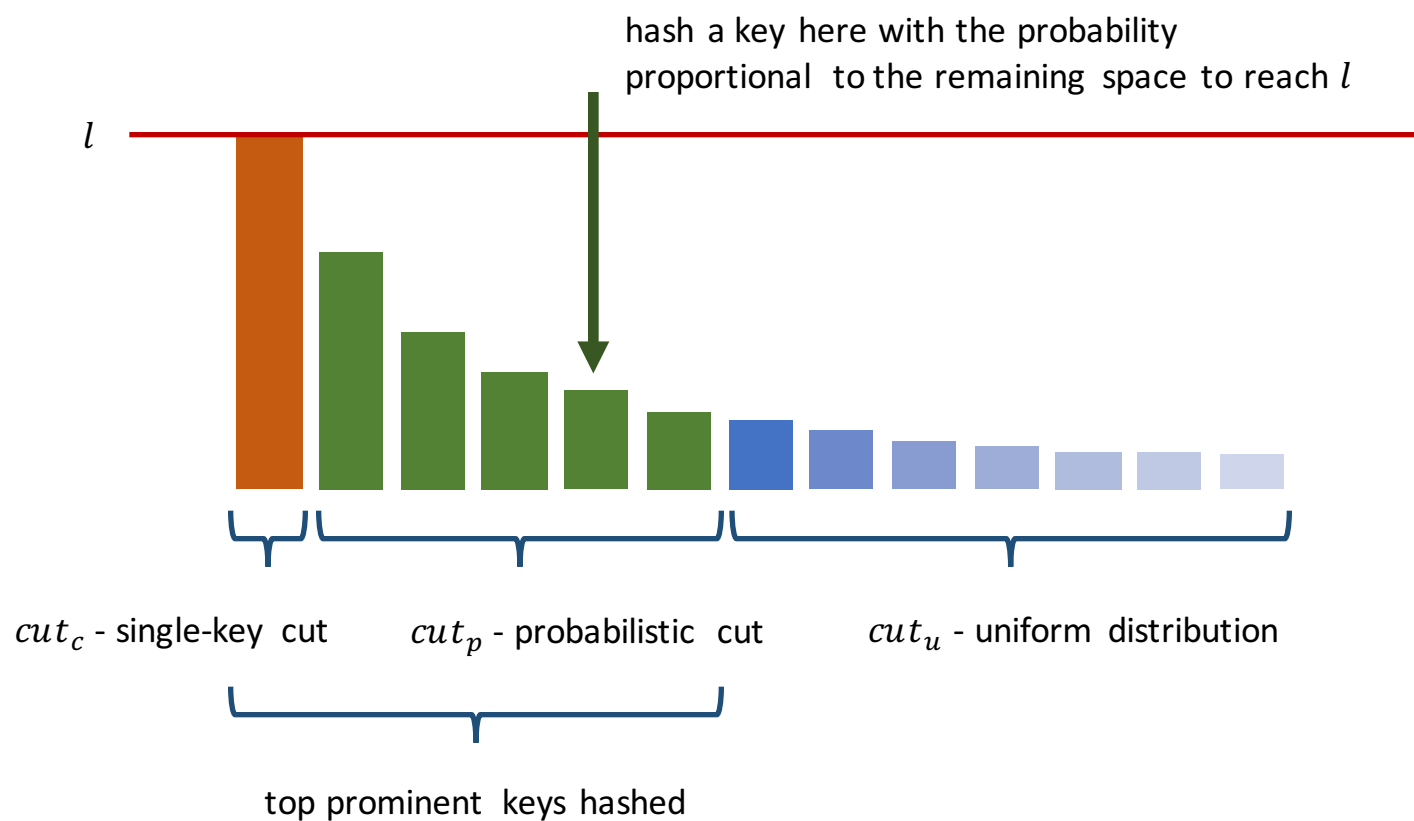   - `inMemory`
      - `ofSpills`

Scanner

- Instantiated for each task before the executor starts them
- Different implementations: `Throughput, Timed, Histogram`
- `ScannerStrategy` defines:
  - when to send to the RTM,
  - histogram-compaction level.

Decision to repartition

- `RepartitioningTrackerMaster` can use different decision strategies:
  - number of local histograms needed,
  - global histogram's distance from uniform distribution,
  - preferences in the construction of the new hash function.

Construction of the new hash function

hash a key here with the probability proportional to the remaining space to reach l

l

cut_c - single-key cut cut_p - probabilistic cut cut_u - uniform distribution

top prominent keys hashed

New hash function in numbers

- More complex than a `hashCode`
- We need to evaluate it for every record
- Micro-benchmark (for example `String`):
  - Number of partitions: 512
  - `HashPartitioner`: AVG time to hash a record is 90.033 ns
  - `KeyIsolatorPartitioner`: AVG time to hash a record is 121.933 ns
- In practice it adds negligible overhead, under 1%

Repartitioning

- Reorganize previous naive hashing
- Usually happens in-memory
- In practice, adds additional 1-8% overhead (usually the lower end), based on:
  - complexity of the mapper,
 - length of the scanning process.

More numbers (groupBy)

MusicTimeseries – groupBy on tags from a listenings-stream
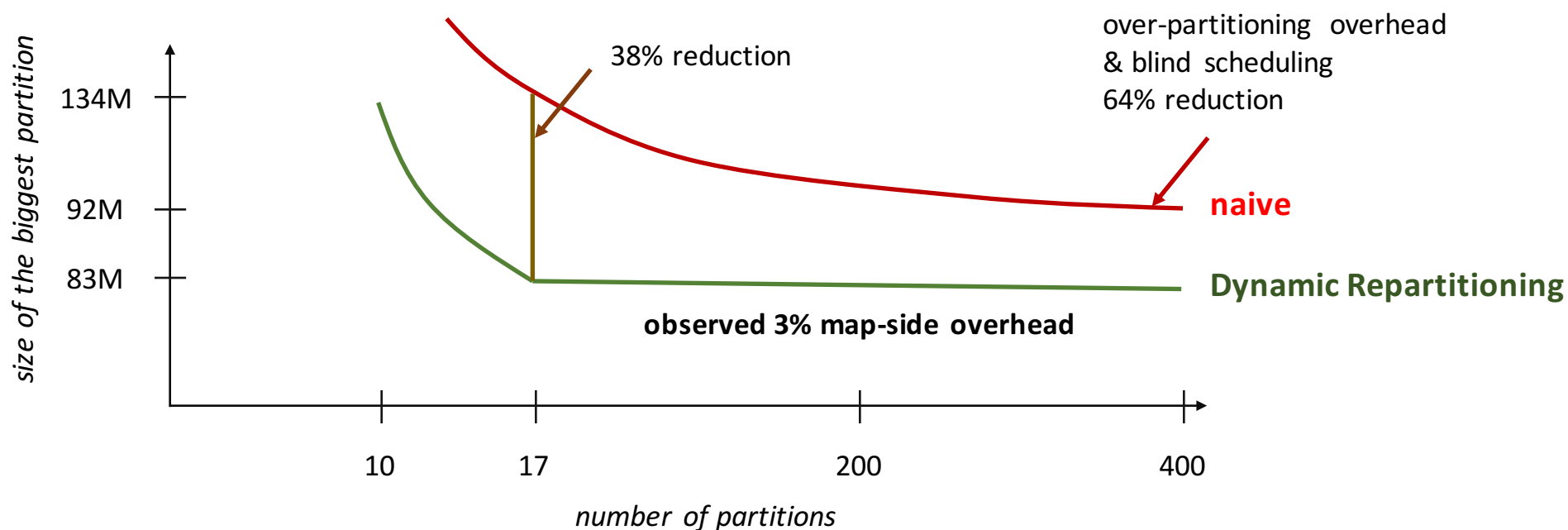
More numbers (join)

- Joining tracks with tags
- Tracks dataset is skewed
- Number of partitions is set to 33
- Naive:
  - size of the biggest partition = **4.14M**
  - reducer's stage runtime = **251 seconds**
- Dynamic Repartitioning
  - size of the biggest partition = **2.01M**
  - reducer's stage runtime = **124 seconds**
  - heavy map, only 0.9% overhead
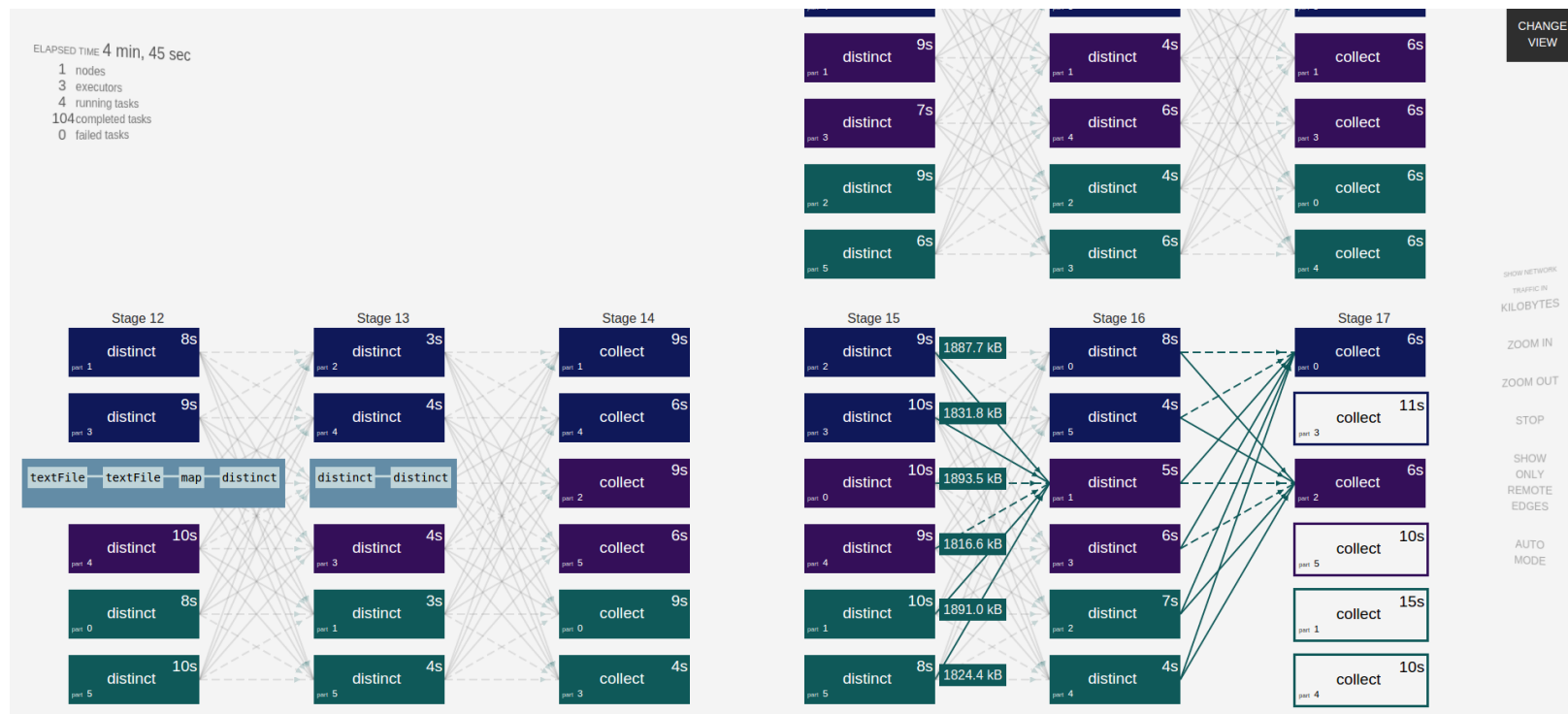

Partition sizes
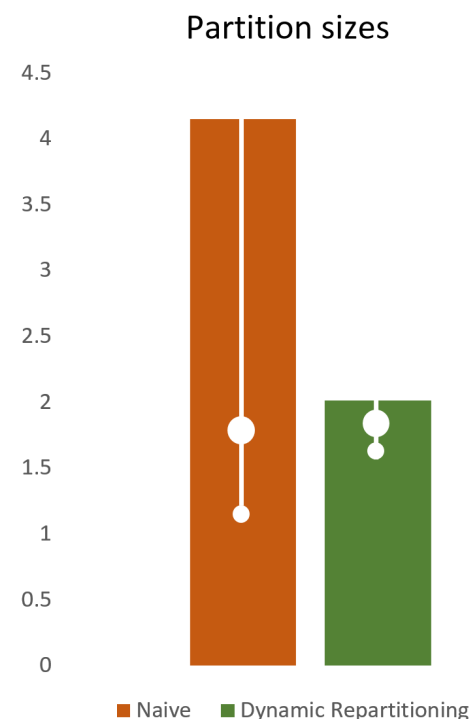
Spark REST API

- New metrics are available through the REST API

- Added new queries to the REST API, for example: „what happened in the last 3 second?"

- BlockFetches are collected to ShuffleReadMetrics

```
recordsRead: 8399,
dataCharacteristics: {
    3424: 19.75,
    115752: 32.25,
    204710: 19.75,
    254186: 17.25
}
```

```
remoteBlocksFetched: 0,
remoteBlockFetchInfos: [ ],
localBlocksFetched: 10,
localBlockFetchInfos: [
    - {
        - blockId: {
            shuffleId: 6,
            mapId: 0,
            reduceId: 7,
            shuffle: true,
            rdd: false,
            broadcast: false
        },
        bytes: 871162
    },
    - {
        - blockId: {
            shuffleId: 6,
            mapId: 1,
            reduceId: 7,
            shuffle: true,
            rdd: false,
            broadcast: false
        },
        bytes: 872696
    },
```
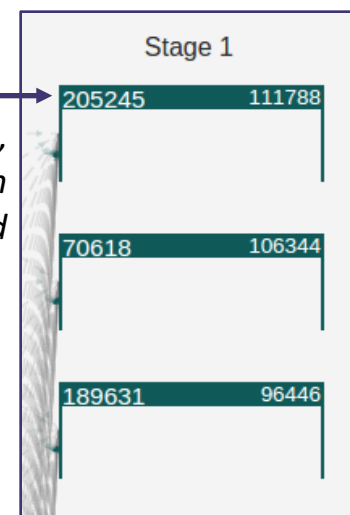
Execution visualization of Spark jobs

Repartitioning in the visualization
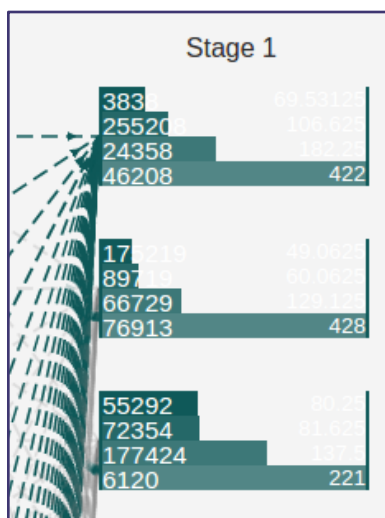
heavy keys create
heavy partitions (slow tasks)

heavy is alone,
size of the biggest partition
is minimized

Conclusion

- Our Dynamic Repartitioning can handle data skew dynamically, on-the-fly on any workload and arbitrary key-distributions
- With very little overhead, data skew can be handled in a natural & general way
- Visualizations can aid developers to better understand issues and bottlenecks of certain workloads
- Making **Spark** *data-aware* pays off

Thank you for your attention

Zoltán Zvara
zoltan.zvara@sztaki.hu