Data-Aware Spark

Zoltán Zvara
zoltan.zvara@sztaki.hu

MTA SZTAKI

This project has received funding from the European Union's Horizon 2020 research and innovation program under grant agreement No 688191.

ERICSSON

Introduction

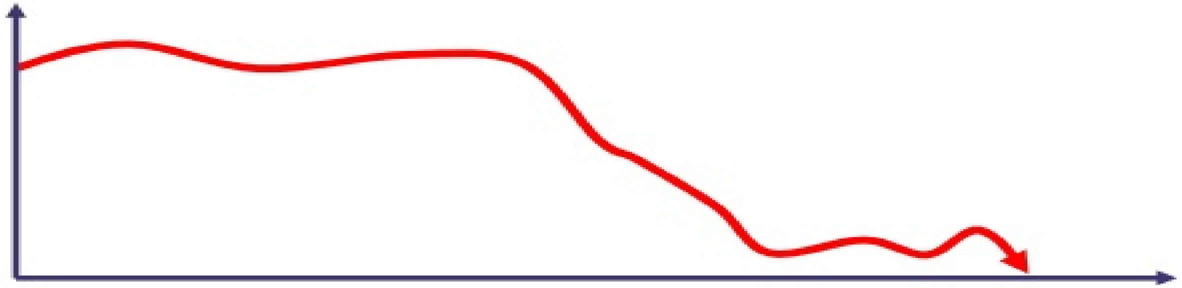
- Hungarian Academy of Sciences, Institute for Computer Science and Control (MTA SZTAKI)
- Research institute with strong industry ties
- Big Data projects using Spark, Flink, Hadoop, Couchbase, etc...
- Multiple IoT and telecommunication use cases, with challenging data volume, velocity and **distribution**

Agenda

- Data-skew story
- Problem definitions & goals
- **Dynamic Repartitioning**
  - Architecture
  - Component breakdown
  - Repartitioning mechanism
  - Benchmark results
- **Tracing**
- Visualization
- Conclusion

Motivation

- Applications aggregating IoT, social network, telecommunication 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 went wrong?

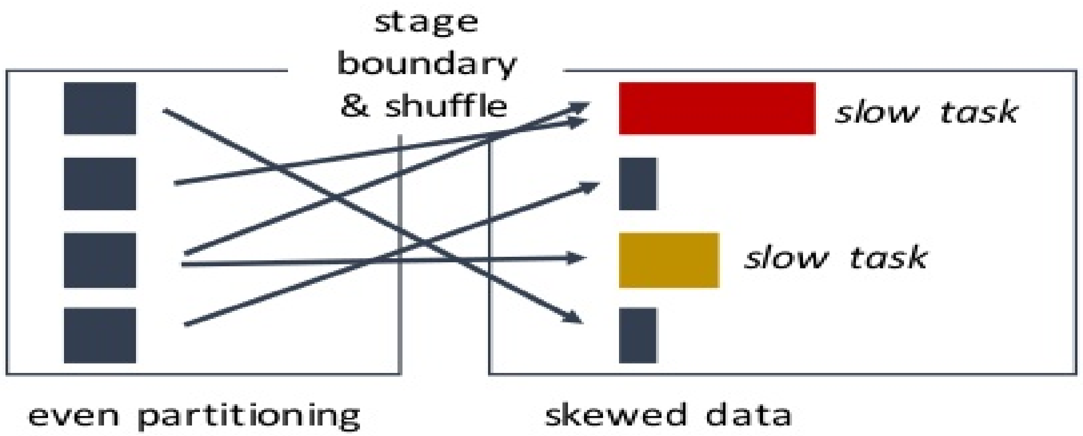


Our data-skew story

- We have use-cases when map-side combine is not an option: **`groupBy, join`**
- Power laws (Pareto, Zipfian)
- „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



Ultimate goal

Spark to be a **data**-aware distributed **data**-processing framework

F1

Generic, engine-level load balancing & data-skew mitigation

Low level, works with any APIs built on top of it

F2

MTA SZTAKI

Completely online

Requires no offline statistics

Does not require simulation on a subset of your data

F3

MTA SZTAKI

Distribution agnostic

No need to identify the underlying distribution

Does not require to handle special corner-cases

F4

For the batch & the streaming guys under one hood

Same architecture and same core logic for every workload

Support for unified workloads

F5

System-aware

Dynamically understand system-specific trade-offs

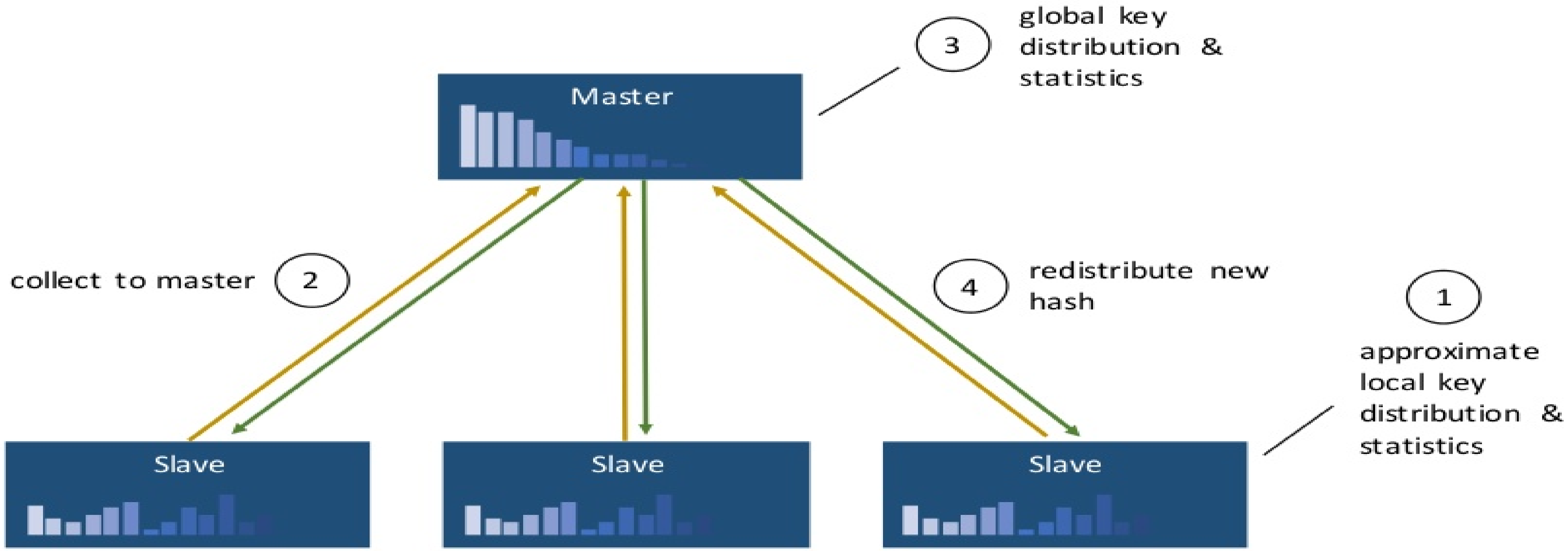
F6

Does not need any user-guidance or configuration

Out-of-the box solution, the user does not need to play with the configuration

`spark.data-aware = true`

Architecture



Driver perspective

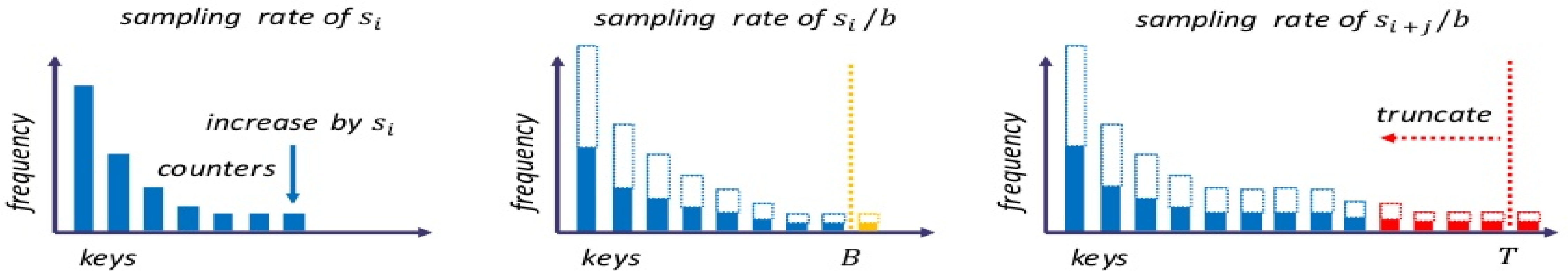
- `RepartitioningTrackerMaster` part of SparkEnv
- Listens to job & stage submissions
- Holds a variety of repartitioning strategies for each job & stage
- Decides when & how to (re)partition
- Collects feedbacks

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



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*

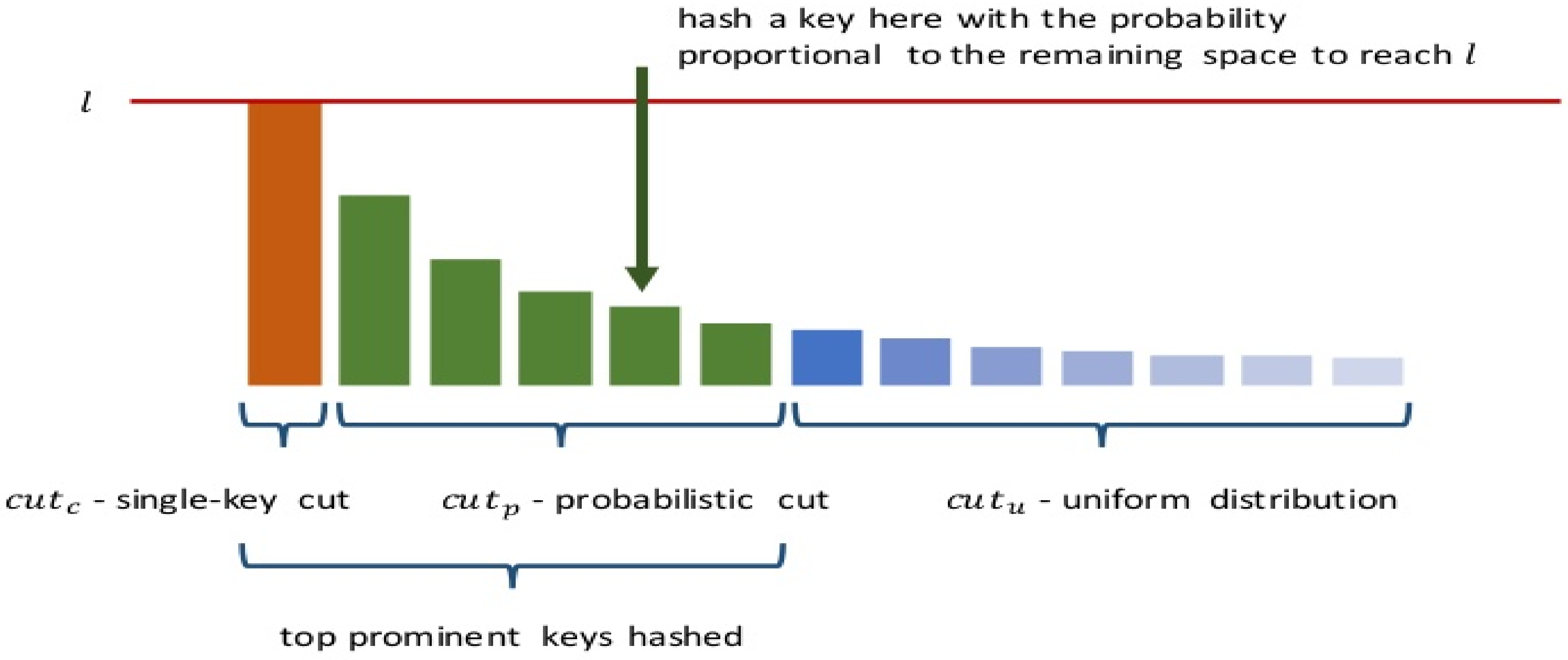
Scalable sampling in numbers

- 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

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



New hash function in numbers

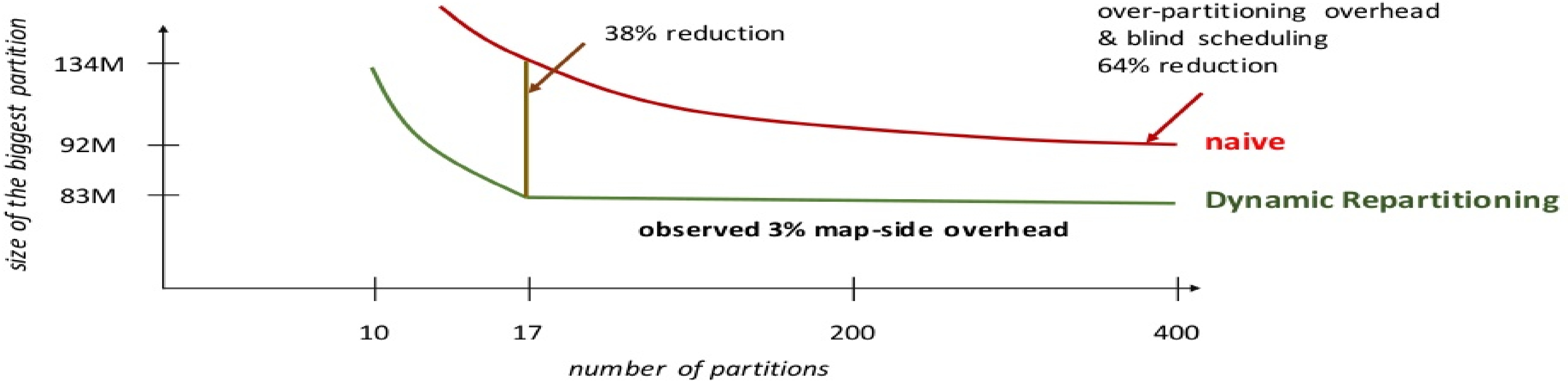
- More complex than a hashCode
- We need to evaluate it for every record
- Micro-benchmark (for example String):
  - Number of partitions: 512
  - HashPartitioner: AVG time to hash a record is 90.033 ns
  - KeyIsolatorPartitioner: AVG time to hash a record is 121.933 ns
- In practice it adds negligible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

Repartitioning

- Reorganize previous naive hashing or buffer the data before 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.
- For streaming: update the hash in DStream graph

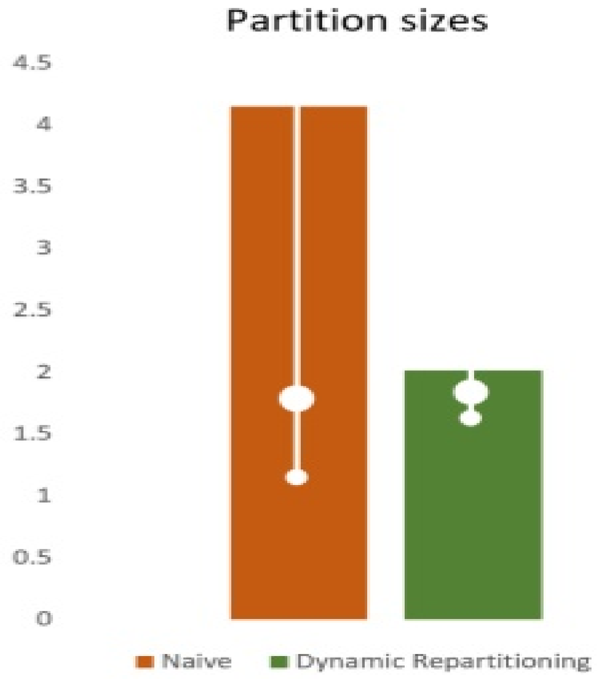
Batch groupBy

MusicTimeseries – groupBy on tags from a listenings-stream



Batch join

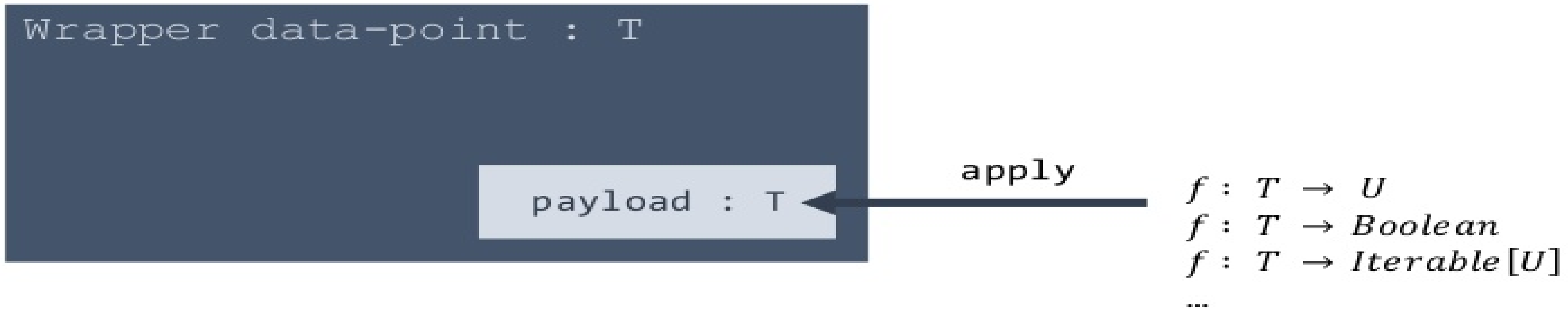
- Joining tracks with tags
- Tracks dataset is skewed
- Number of partitions set to 33
- Naive:
  - size of the biggest partition = **4.14M**
  - reducer’s stage runtime = **251 seconds**
- Dynamic Repartitioning
  - size of the biggest partition = **2.01M**
  - reducer’s stage runtime = **124 seconds**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



Tracing

Goal: capture and record a data-point's lifecycle throughout the whole execution

Rewritten Spark's core to handle wrapped data-points.

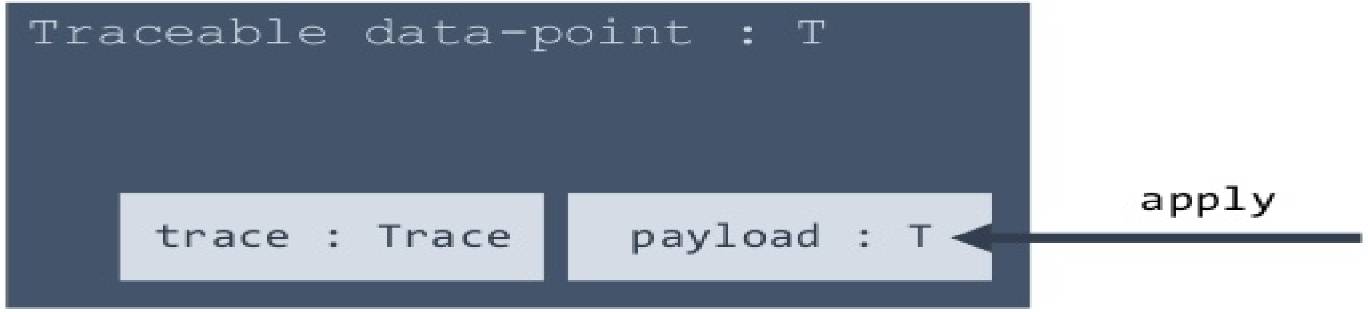


Traceables in Spark

Each record is a Traceable object (a Wrapper implementation).

Apply function on payload and report/build trace.

`spark.wrapper.class = com.ericsson.ark.spark.Traceable`



$f : T \rightarrow U$

$f : T \rightarrow Boolean$

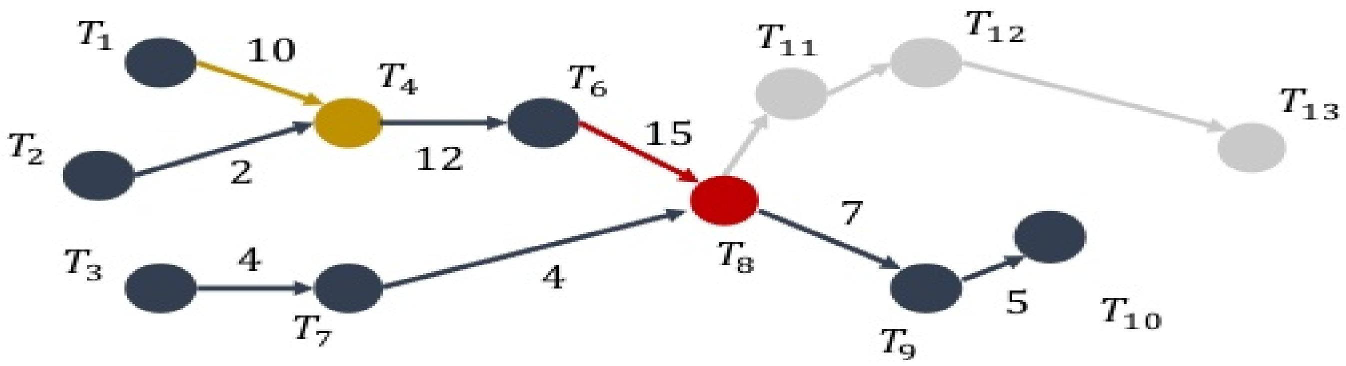
$f : T \rightarrow Iterable[U]$

…

Tracing implementation

Capture trace informations (deep profiling):

- by reporting to an external service;
- piggyback aggregated trace routes on data-points.



- T_i can be any Spark operator
- we attach useful metrics to edges and vertices (current load, time spent at task)

F7

Provide data-characteristics to monitor & debug applications

Users can sneak-peak into the dataflow

Users can debug and monitor applications more easily

F7

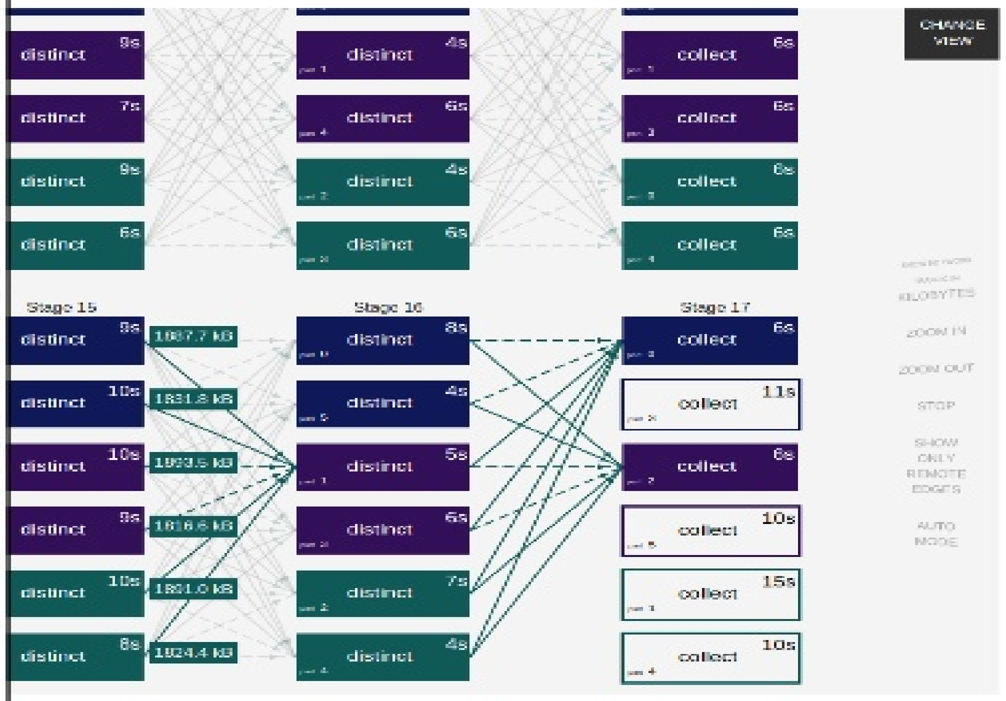
- New metrics are available through the REST API
- Added new queries to the REST API, for example: „what happened in the last 3 second?”
- BlockFetches are collected to ShuffleReadMetrics


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

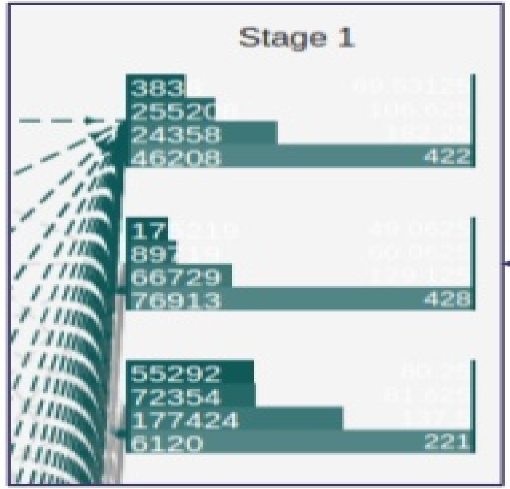


```
remoteBlocksFetched: 0,
remoteBlockFetchInfos: [ ],
localBlocksFetched: 10,
localBlockFetchInfos: [
  - {
      - blockId: {
            shuffleId: 6,
            mapId: 0,
            reduceId: 7,
            shuffle: true,
            rdd: false,
            broadcast: false
        },
       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

Execution visualization of Spark jobs

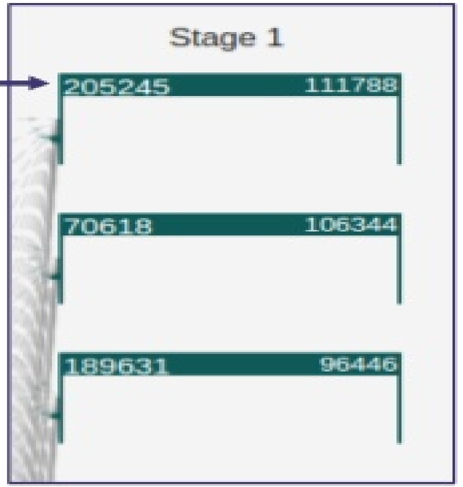


Repartitioning in the visualization



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
- Tracing can help us to improve the co-location of related services
- 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

MTA SZTAKI

ERICSSON