RDataFrame Enhancements for Distributed Processing

SWAN-TOTEM Helix Nebula Project

51st Parallelism, Performance and Programming model meeting

Javier Cervantes, Enric Tejedor

Helix Nebula Project

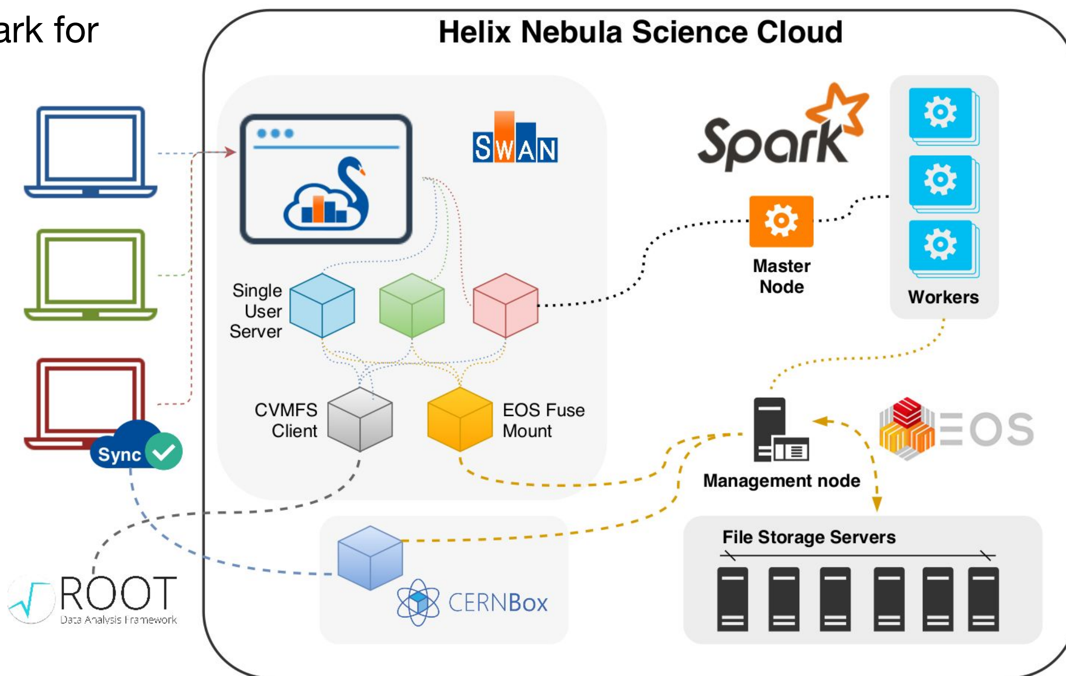
- European Partnership between IT providers and three research centres (CERN, EMBL and ESA)
  - 40 public and private partners
- Science cloud platform for the European research community (Horizon 2020)
  - Cloud services serving scientific users from a wide range of domains.
- Collaboration at CERN:
  - Interactive Data Analysis for End Users on Helix Nebula Science Cloud (HNSciCloud)
  - Contributors:
    - TOTEM Experiment
    - EP-SFT (ROOT)
    - IT-Databases
    - IT-Storage

Helix Nebula Project

- **First test:** HN Deployment (2017)
  - Provide existing CERN *Products* as a service

Helix Nebula Project

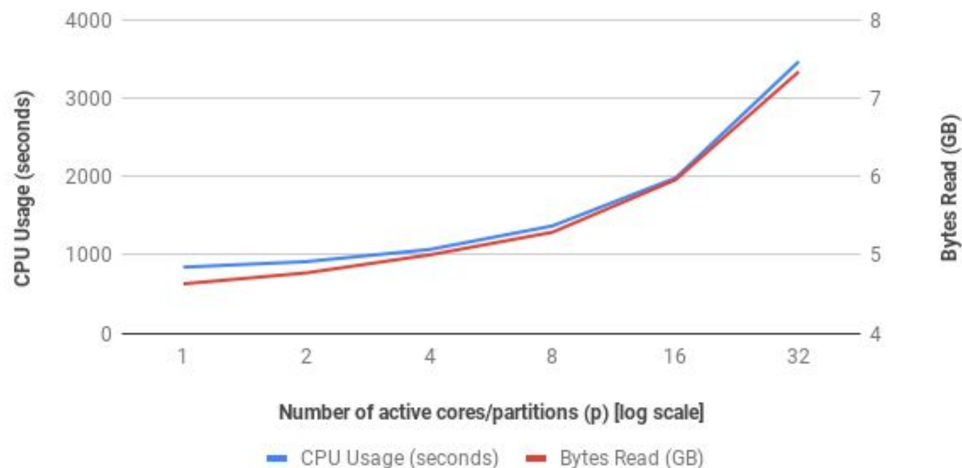
- **Current test:** HN TOTEM Test (2018)
    - Interactive analysis using Spark for distributed processing
    - TOTEM proton-proton elastic scattering analysis



Motivation - The Problem

ROOT v6.14

- When running the TOTEM analysis with distributed RDataFrame, the number of bytes read and CPU usage increased with the number of partitions of the input dataset



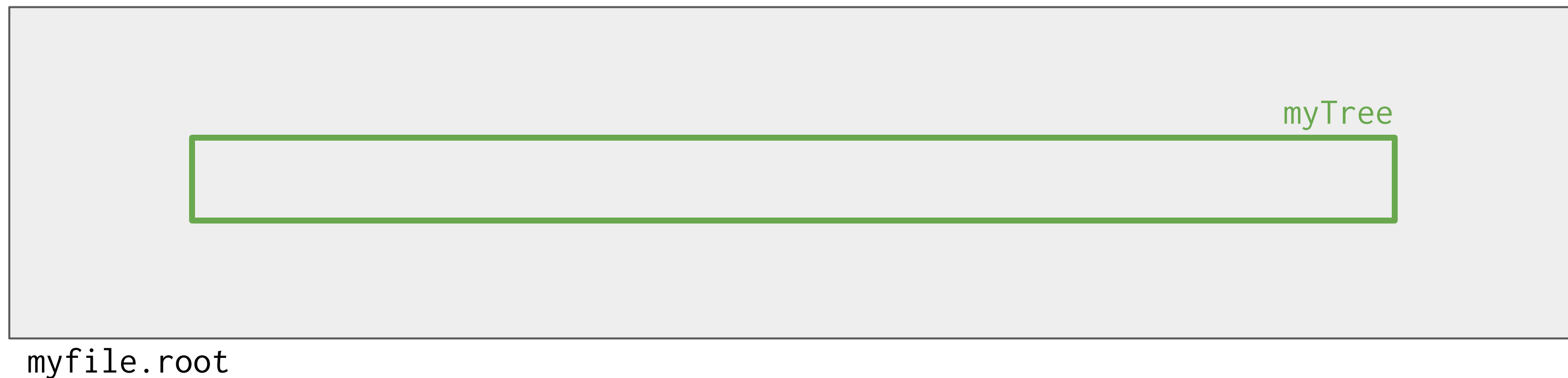
Credit: Prasanth Kothuri

ROOT File Format concepts learned

Reading from a ROOT File

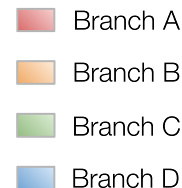
myfile.root

Reading from a ROOT File



- ROOT File may contain one or more `TTrees`

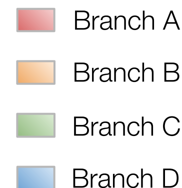
Reading from a ROOT File



- ROOT File may contain one or more `TTrees`
- `TTrees` contain `branches` where final values are stored in `leaves`
    - Columnar representation in memory

Reading from a ROOT File

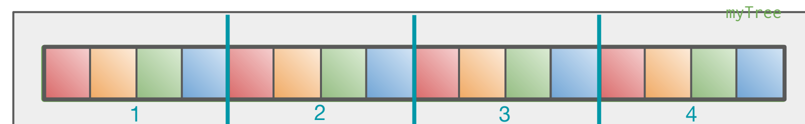
myfile.root



- ROOT File may contain one or more `TTrees`
- `TTrees` contain `branches` where final values are stored in `leaves`
  - Columnar representation in memory
- `Branches` are grouped in `clusters`

Reading from a ROOT File

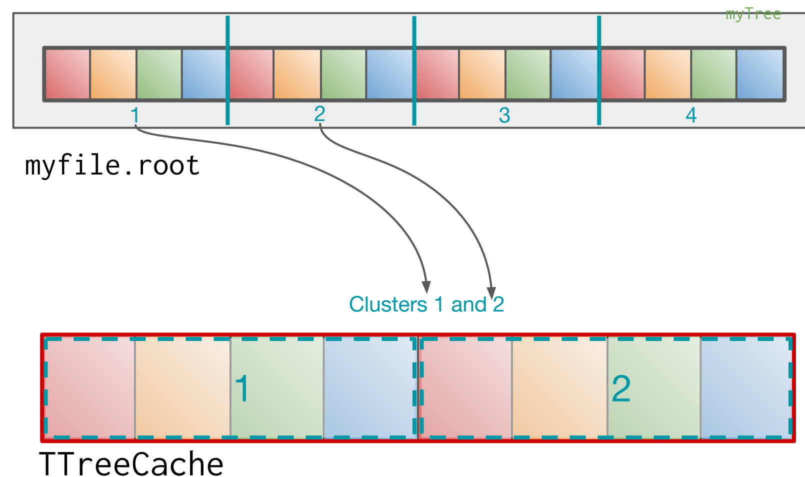
- ROOT is optimized for **sequential reading** of an entire file (all branches / all clusters)
  - Minimize access to disk
  - Cache data for future accesses (`TTreeCache`)



`myfile.root`

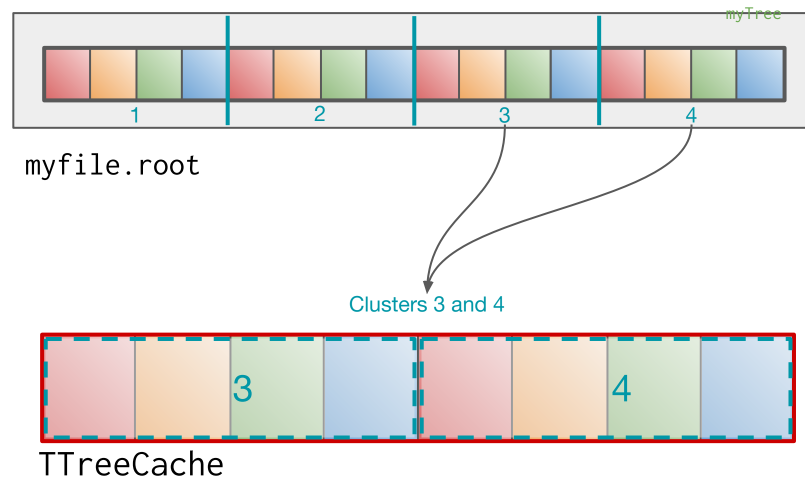
Reading from a ROOT File

- ROOT is optimized for **sequential reading** of an entire file (all branches / all clusters)
  - Minimize access to disk
  - Cache data for future accesses (`TTreeCache`)
- By default `TTreeCache` reads in consecutive clusters until:
  - the end of the file
  - or the cache is full



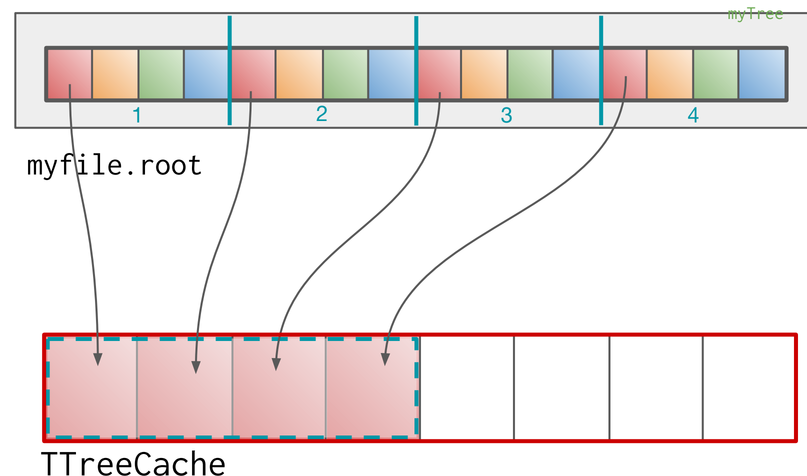
Reading from a ROOT File

- ROOT is optimized for **sequential reading** of an entire file (all branches / all clusters)
  - Minimize access to disk
  - Cache data for future accesses (`TTreeCache`)
- By default `TTreeCache` reads in consecutive clusters until:
  - the end of the file
  - or the cache is full

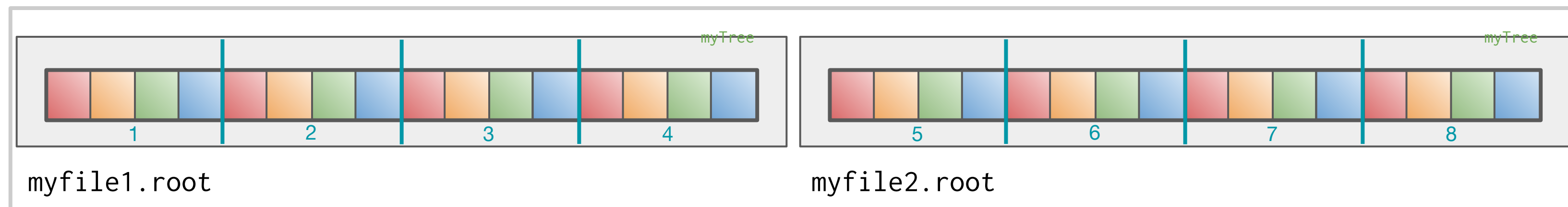


Reading from a ROOT File

- ROOT is optimized for **sequential reading** of an entire file (all branches / all clusters)
  - Minimize access to disk
  - Cache data for future accesses (`TTreeCache`)
- By default `TTreeCache` reads in consecutive clusters until:
  - the end of the file
  - or the cache is full
- Users can force the `TTreeCache` to read in only those branches being processed



Reading from a ROOT File



- For big datasets split in different files we work with a `TChain`
  - `TChain =` tree stored in multiple files
  - Tree clusters never exceed the boundaries of a file

Problems identified

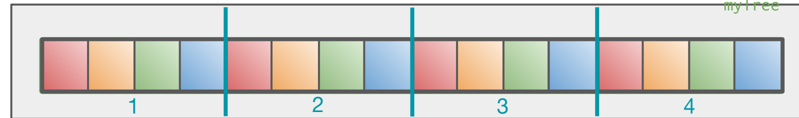
1. Cluster prefetching - [ROOT-9773]
   - When a task was processing a range of entries, clusters outside the range were prefetched in the Spark executor processes
2. Branch prefetching - [ROOT-9802]
   - Branches that were not processed in the `RDataFrame` computation were also read
3. Full sweep of `TChain` - [DistROOT changes]
   - Problematic if the range was located towards the end of the chain
4. `RDataFrame` reset the chain current entry to 0 after the event loop - PR#3001
   - Caused an unnecessary extra cluster prefetching

1 Cluster prefetching

[ROOT-9773]

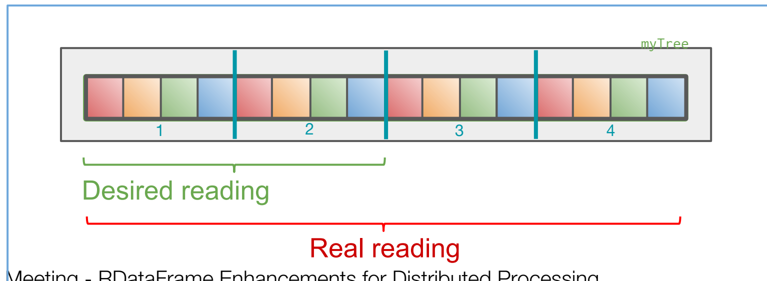
When a task was processing a range of entries, clusters outside the range were prefetched in the Spark executor processes

- Processing **one** file `myfile.root` with **two** Spark workers

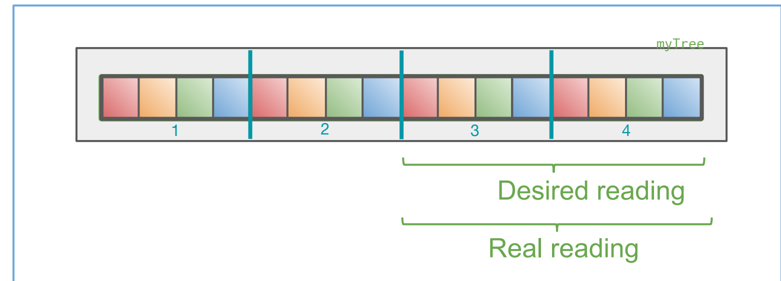


myfile.root

Worker 1



Worker 2

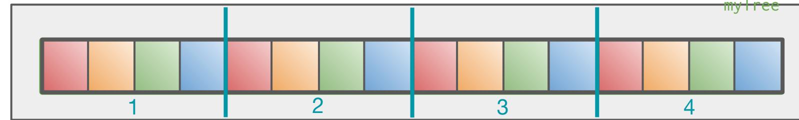


1 Cluster prefetching

[ROOT-9773]

When a task was processing a range of entries, clusters outside the range were prefetched in the Spark executor processes

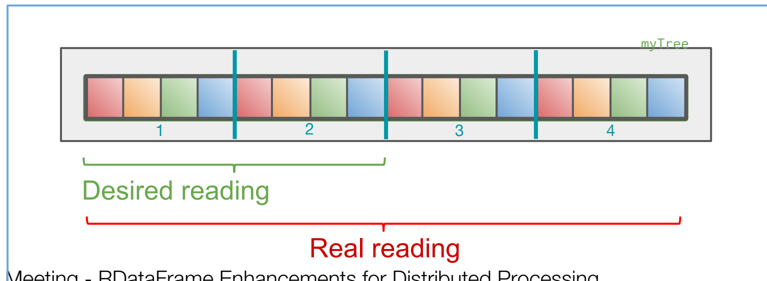
- Processing **one** file `myfile.root` with **two** Spark workers



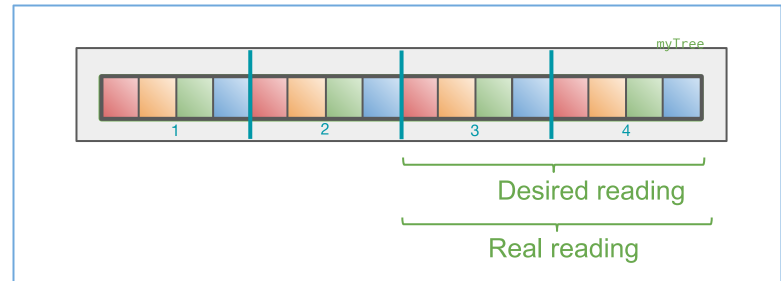
1.5x Reading

`myfile.root`

Worker 1



Worker 2

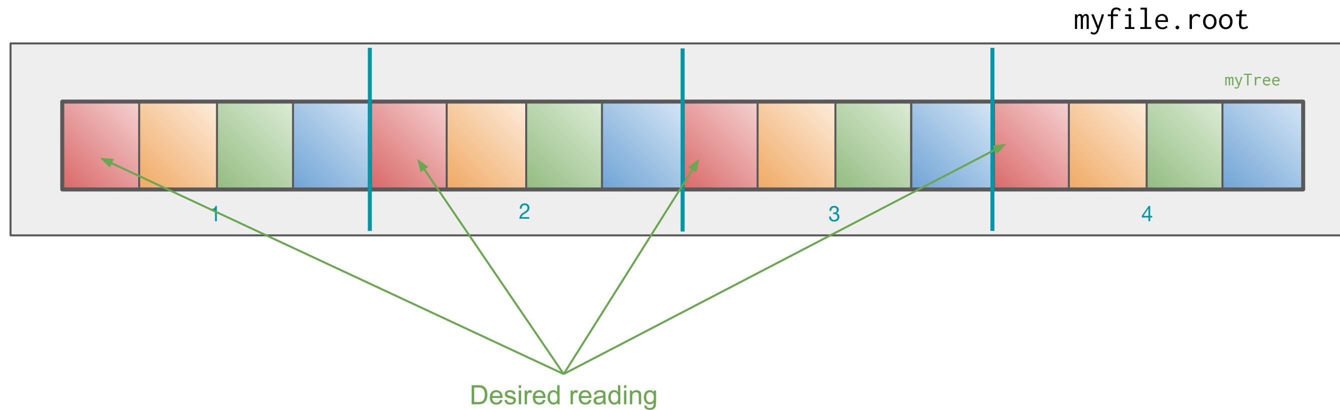


[ROOT-9802]

2 Branch prefetching

Branches that were not processed in the `RDataFrame` computation were also read

- Processing **one branch** from file `myfile.root`

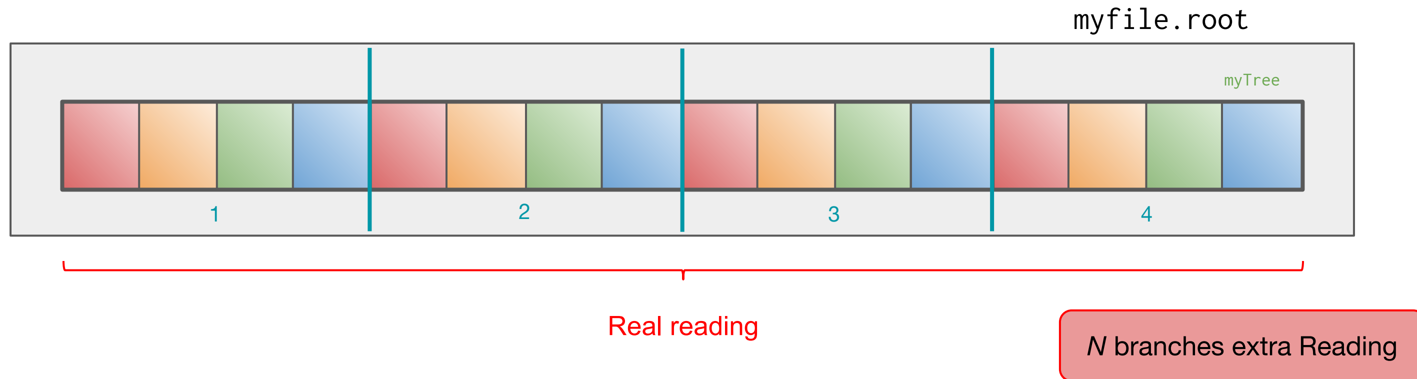


[ROOT-9802]

2 Branch prefetching

Branches that were not processed in the `RDataFrame` computation were also read

- Processing **one branch** from file `myfile.root`

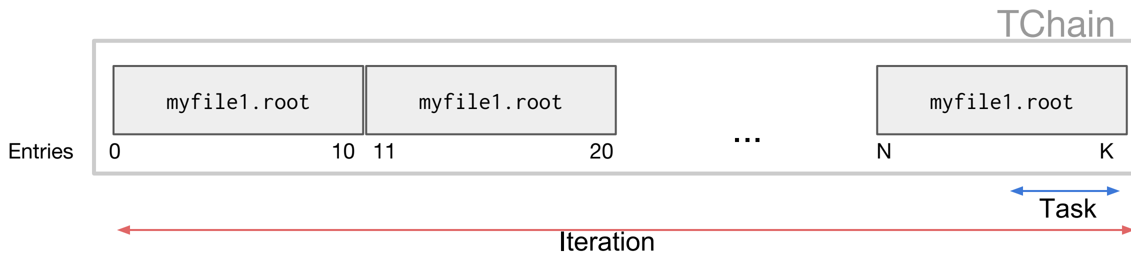


* Specific instructions to force caching of a single branch were not considered

[DistROOT changes]

3 Full sweep of `TChain`

- Problematic if the range was located towards the end of the chain



- Task processing last part of the TChain still iterates from the beginning of the chain (although it does not read)
  - Time consuming

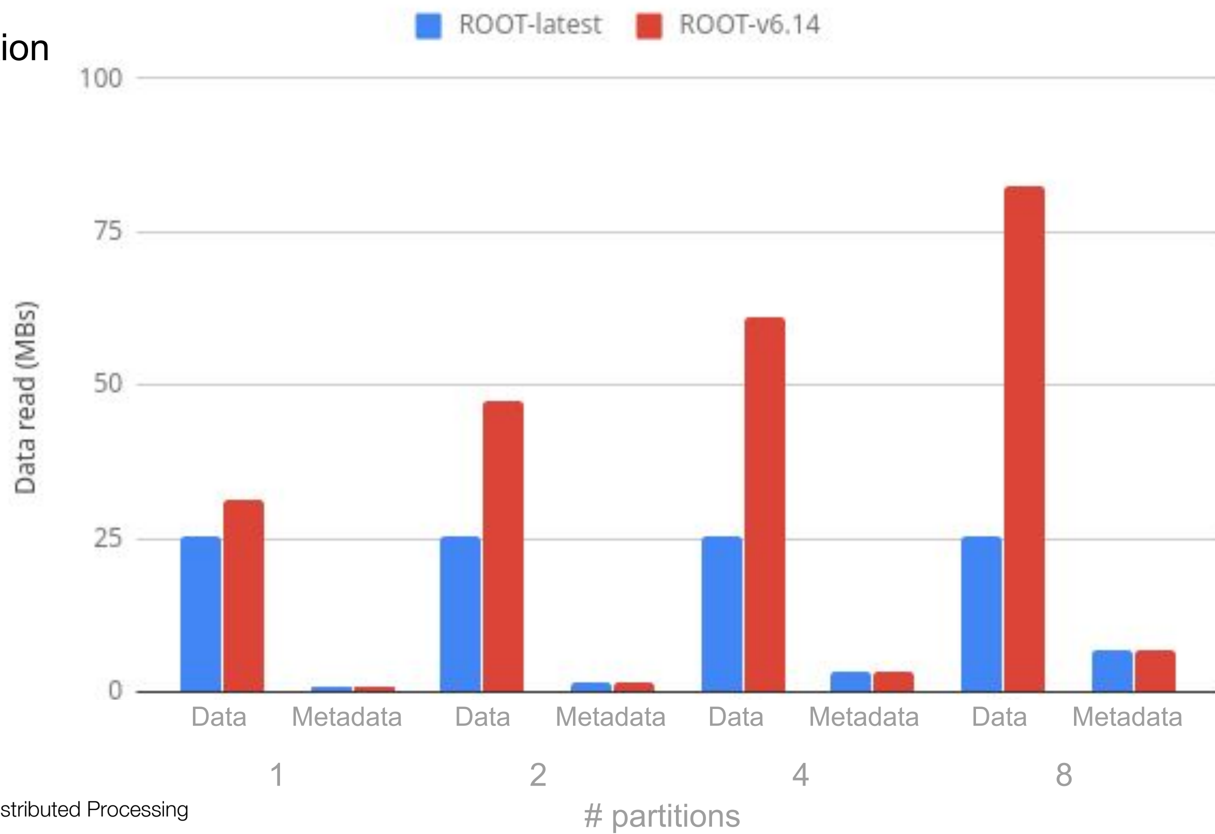
Solutions applied

1. Cluster prefetching
   - ✓ Respect cluster boundaries when creating entry ranges
   - ✓ Prefetch only the clusters of current range into the cache
2. Branch prefetching
   - ✓ Only cache branches processed during the `RDataFrame` computation
3. Partial sweep of `TChain`
   - ✓ Spark executors create a `TChain` with only the files of current range
4. `RDataFrame` reset the chain current entry to 0 after the event loop
   - ✓ Reset without reading

Reading Comparison

1 2 4

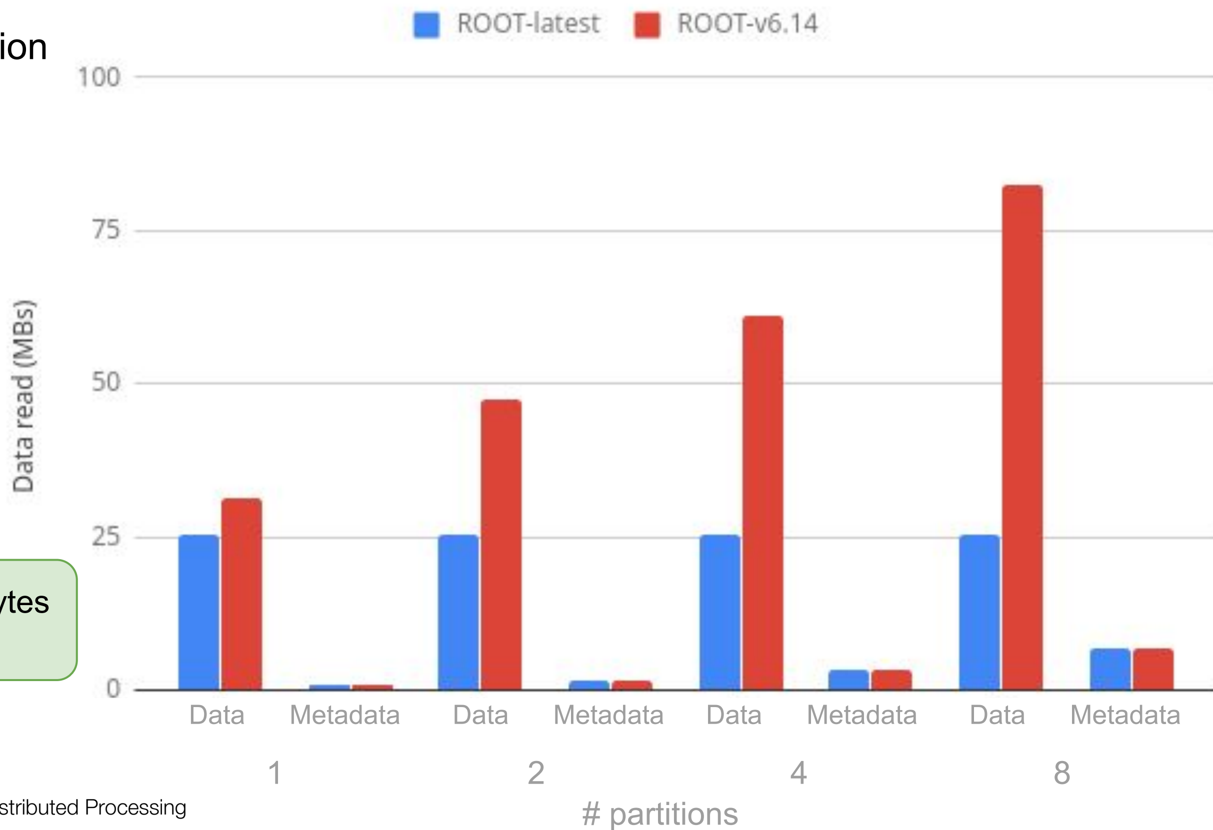
- No Spark, local execution
- Input: 1 File from DS1
- Distill + Distributions



Reading Comparison

1 2 4

- No Spark, local execution
- Input: 1 File from DS1
- Distill + Distributions

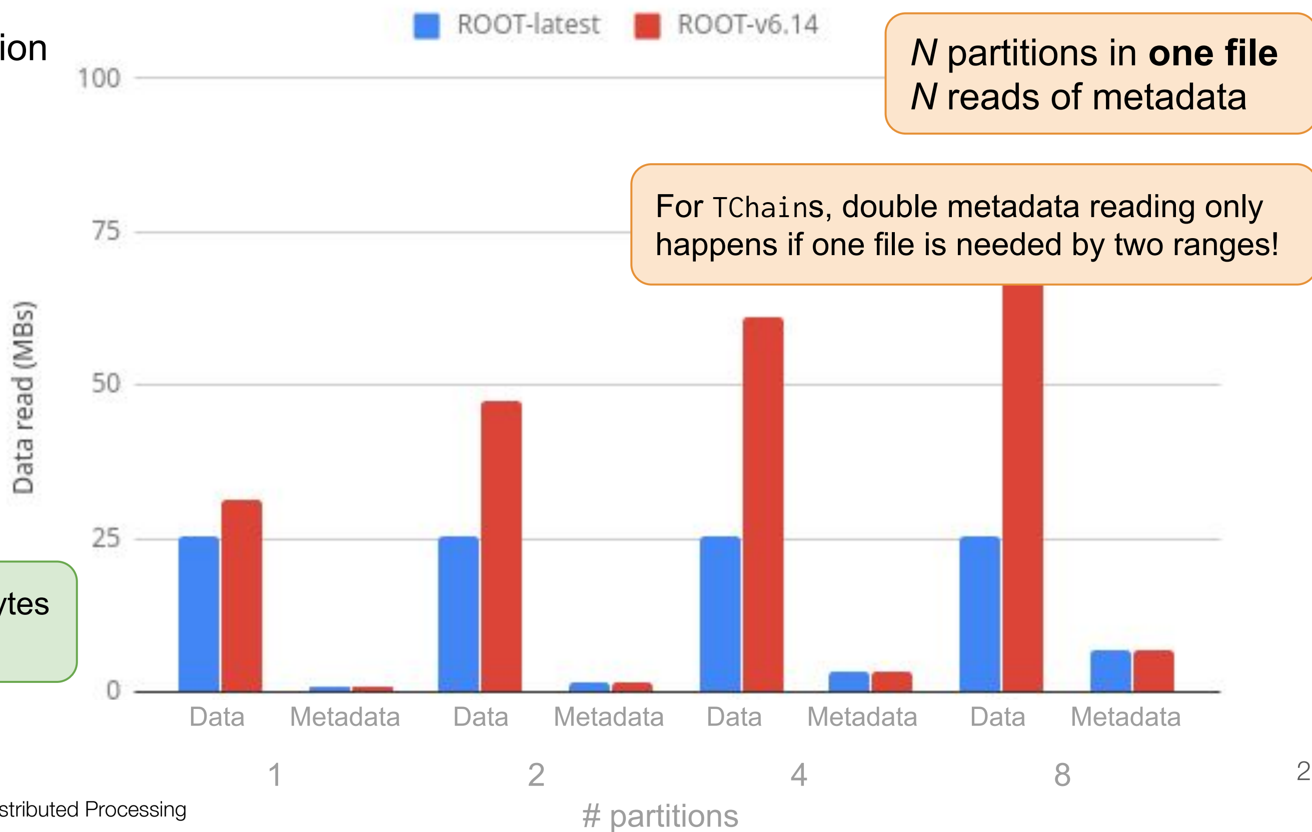


Stable on (non meta) bytes read wrt # partitions

Reading Comparison

1 2 4

- No Spark, local execution
- Input: 1 File from DS1
- Distill + Distributions



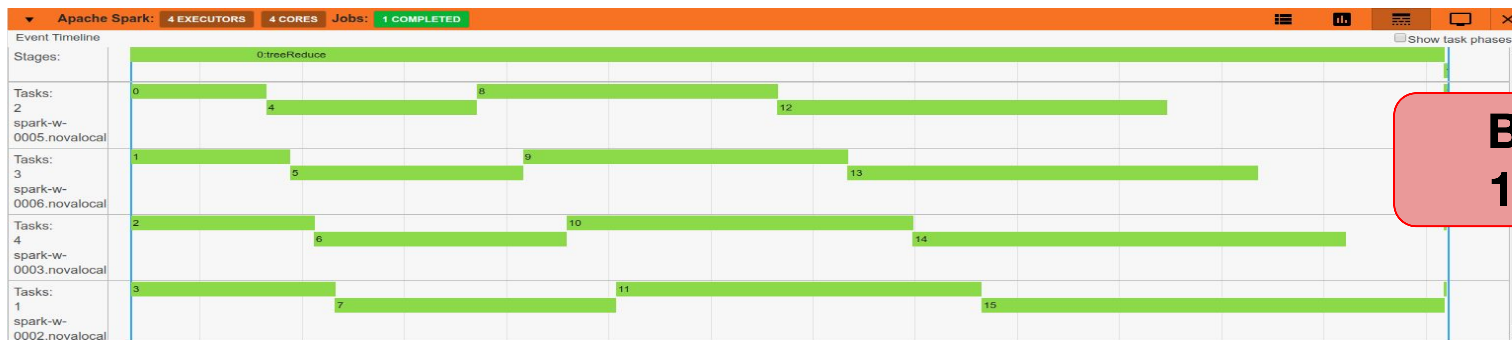
N partitions in **one file**
N reads of metadata

For `TChains`, double metadata reading only happens if one file is needed by two ranges!

Stable on (non meta) bytes read wrt # partitions

Performance Comparison (3)

- Spark execution, HN cloud, 4 executors
- Input: DS1 (91GB, 41 files), 16 partitions (by entries)
- Distill + Distributions

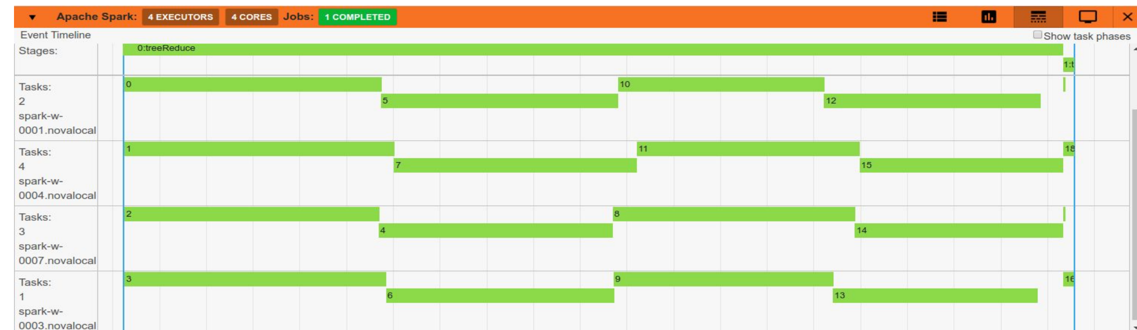


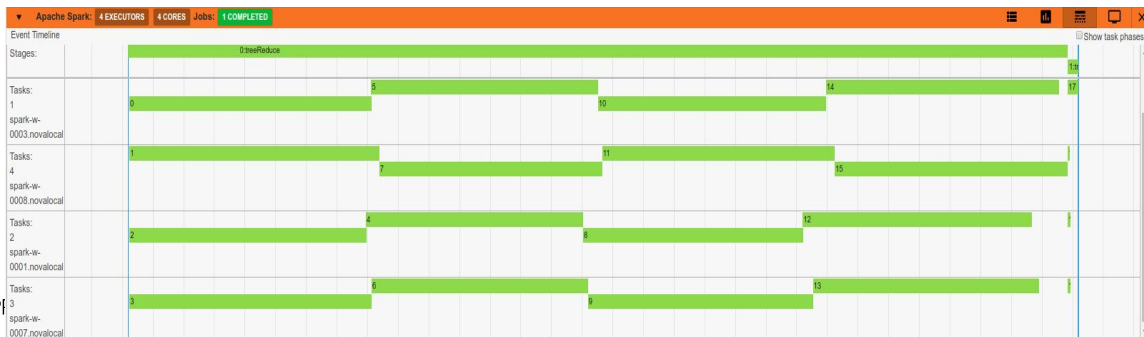
Before
12 min

After
5 min

Performance Comparison - Entries vs Files

- Spark execution, HN cloud, 4 executors
- Input: DS1 (91GB, 41 files), 16 partitions (by entries or files)
- Distill + Distributions





Partitioning

File-based
5 min

Entry-based
5 min

Spark RDataFrame vs RDataFrame

- All tests running on the same VM
  - RDF via ssh-ing into the machine
  - Spark RDF via SWAN + Spark
- Little overhead
  - Generation of ranges (accesses TChain metadata)
  - Spark itself
- Multithreaded execution slower in plain RDF (To be investigated)

Spark RDataFrame vs RDataFrame