Zero-Copy Merge with RNTuples

EUGENIO MARINELLI

Supervisors:
Jakob Blomer
Javier Lopez Gomez

Introduction

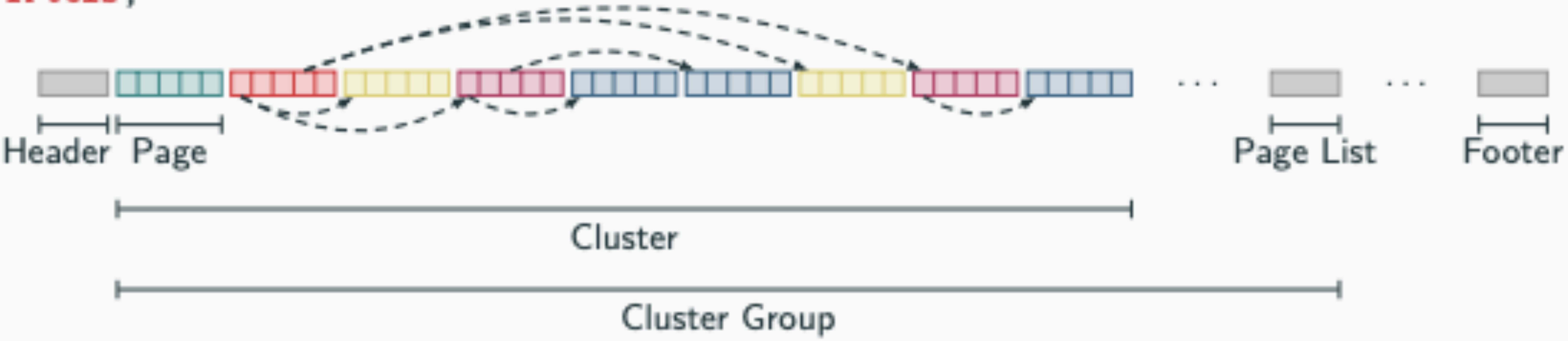
- Data from HEP events is rapidly increasing
- Need of a modern software stack for modern storage hardware and systems
- ROOT RNTuple
  - Modern redesign of ROOT TTree I/O subsystem
  - Available in ROOT7
  - HEP events stored as records of properties
    - Properties stored columnar-wise on disk
- RNTuple fits the HEP events workload
  - Write-once-read-many columnar access
  - Often **merging** of many RNTuples content is need
    - Significant resource consumption

Contribution

- In this work:
  - extensive analysis of the challenges related to merging algorithm of RNTuples
  - first implementation of zero-copy merge
  - comparison with alternative implementations
  - overview of those aspects related to this algorithm that still need to be investigated.

ROOT7 RNTuples Hierarchy (bottom-up)

```
struct Event {
    int fId;
    vector<Particle> fPtcls;
};
struct Particle {
    float fE;
    vector<int> fIds;
};
```



- Pages
  - partition columns
- Clusters
  - Group of columns for a certain HEP events range
- Cluster Groups
  - Set of clusters

ROOT7 RNTuple Metadata

- Header
  - RNTuple schema
- Footer
  - Information about clusters, clusters groups, etc
- Page Lists
  - Offsets of pages in the clusters
  - Stored after each cluster group

RNTuple Merging Operation

- Given m RNTuples
- We want one resulting RNTuple with:
  - New header
  - Same content of all the source RNTuples
  - Same number of clusters and cluster groups, of the all the source RNTuples
  - New page list with updated offsets
  - New footer
- Note: we want to **share** the content without copying it

Reflink and Block Sharing

Mechanism of duplicating files by sharing blocks at file system level

Pros:

- Note: different from hardlink mechanism
- Finer granularity
  - file system blocks rather than whole files
- Different metadata, i.e. different inode
- Based on copy-on-write mechanism
  - Blocks are duplicated when one of the copy is modified

Reflink and Block Sharing: Challenges

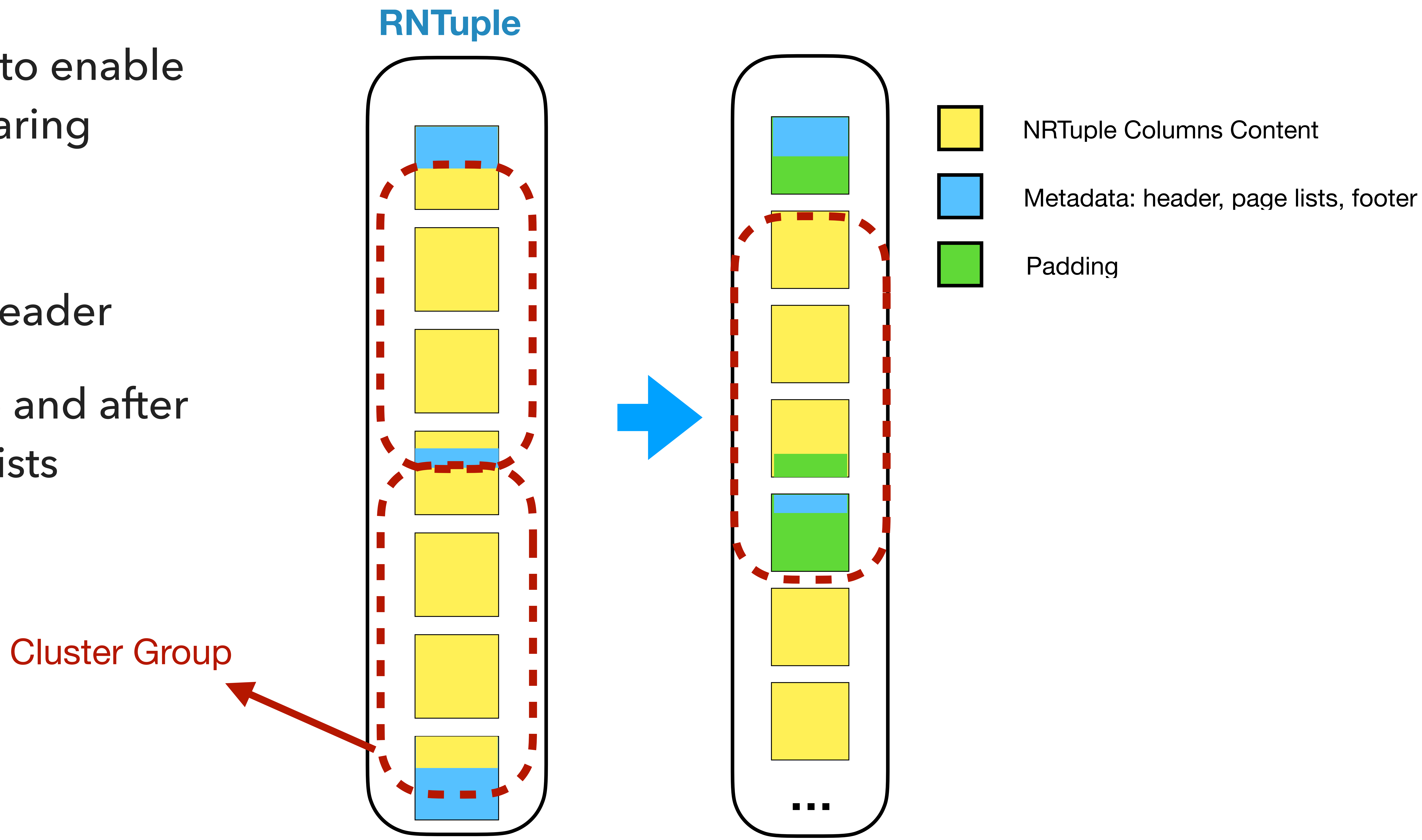
- Available ONLY on certain file systems:
  - brtfs
  - XFS
- Accessible through system calls (on Linux only):
  - ioctl
  - copy_file_range
- Addresses range need to be aligned to the file system block size
  - typically 4KB (default) or higher

Reflinks from userspace

- ioctl
  - Fails if:
    - used out of XFS or btrfs
    - addresses are not aligned
  - If conditions met, blocks are duplicated in nearly constant time
- copy_file_range
  - more general version of ioctl
  - if used on XFS and btrfs and addresses are aligned
    - blocks are shared
  - if used on other file systems (EXT3, EXT4, etc):
    - efficiently copy the blocks at kernel level - w/o passing through the use-espace
  - if used on NFS enables server to server copy

RNTuple Padding

- Needed to enable block sharing
- Applied:
  - after header
  - before and after page lists



Zero-Copy Merge

- Works with one cluster groups at time
- For each cluster group:
  - clone the file blocks
  - update metadata
    - local offsets become global offsets

Evaluation

- Are blocks really shared?

```
[root@phsft-cvm01 test7]# xfs_bmap -vp ntpl1.root
ntpl1.root:
 EXT: FILE-OFFSET        BLOCK-RANGE          AG AG-OFFSET          TOTAL FLAGS
   0: [0..7]:            105009056..105009063  2 (151456..151463)       8 000000
   1: [8..300007]:       105009064..105309063  2 (151464..451463)  300000 100000
   2: [300008..300095]:  105309064..105309151  2 (451464..451551)      88 000000
[root@phsft-cvm01 test7]# xfs_bmap -vp ntpl2.root
ntpl2.root:
 EXT: FILE-OFFSET        BLOCK-RANGE          AG AG-OFFSET          TOTAL FLAGS
   0: [0..7]:            105309152..105309159  2 (451552..451559)       8 000000
   1: [8..480007]:       105309160..105789159  2 (451560..931559)  480000 100000
   2: [480008..480135]:  105789160..105789287  2 (931560..931687)     128 000000
[root@phsft-cvm01 test7]# xfs_bmap -vp ntplmerged.root
ntplmerged.root:
 EXT: FILE-OFFSET        BLOCK-RANGE          AG AG-OFFSET              TOTAL FLAGS
   0: [0..7]:            157286488..157286495  3 (88..95)                   8 000000
   1: [8..300007]:       105009064..105309063  2 (151464..451463)      300000 100000
   2: [300008..300087]:  171841608..171841687  3 (14555208..14555287)      80 000000
   3: [300088..780087]:  105309160..105789159  2 (451560..931559)      480000 100000
   4: [780088..780215]:  171841688..171841815  3 (14555288..14555415)     128 000000
```

RNTuple 1

RNTuple 2

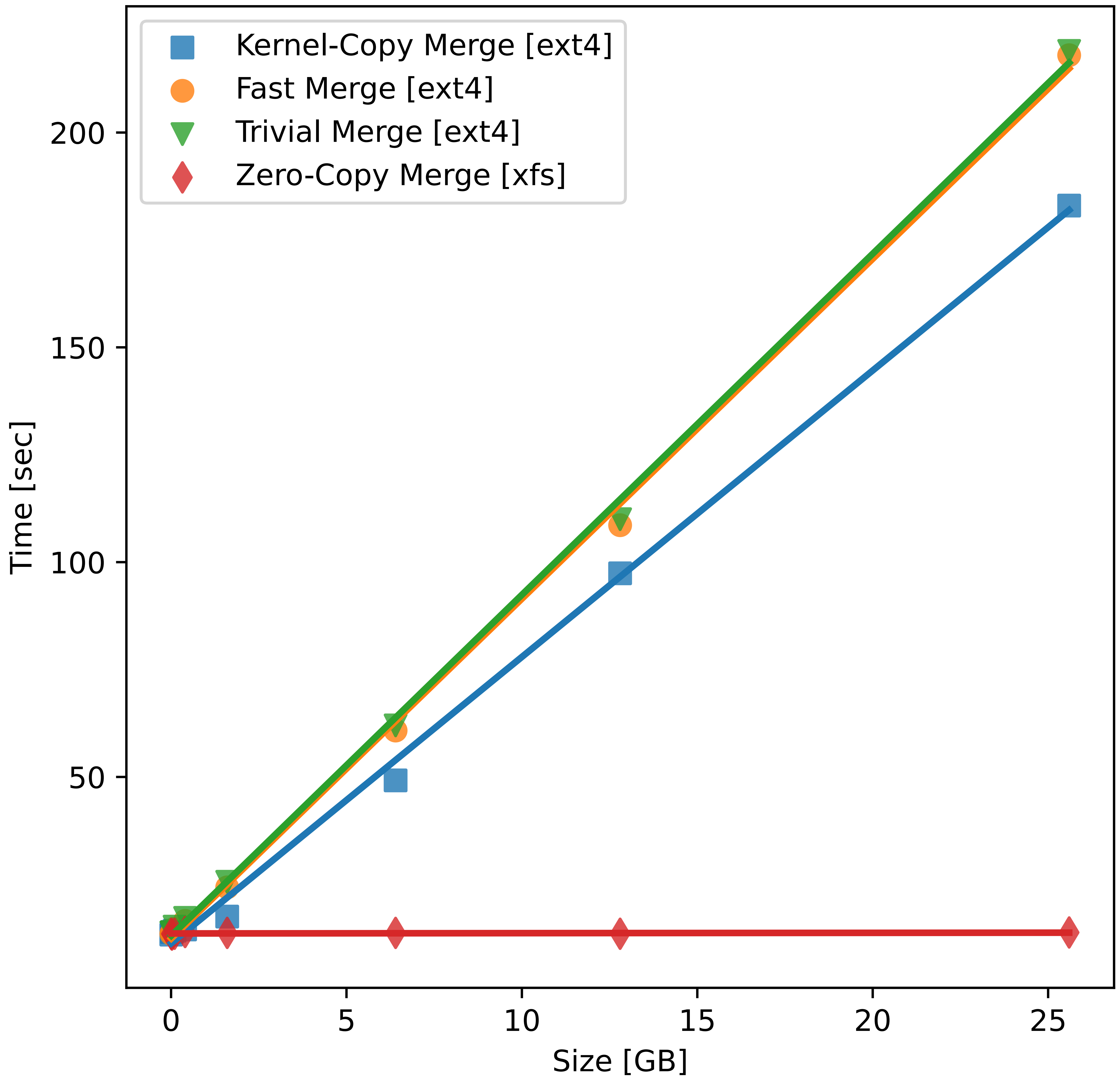
Merged RNTuple

Evaluation

- Zero-Copy merge compared with:
  - Trivial merge
    - read and write data using standard RNTuple API
  - Fast merge
    - read from source files and write on destination file
  - Kernel merge
    - file copy in kernel space

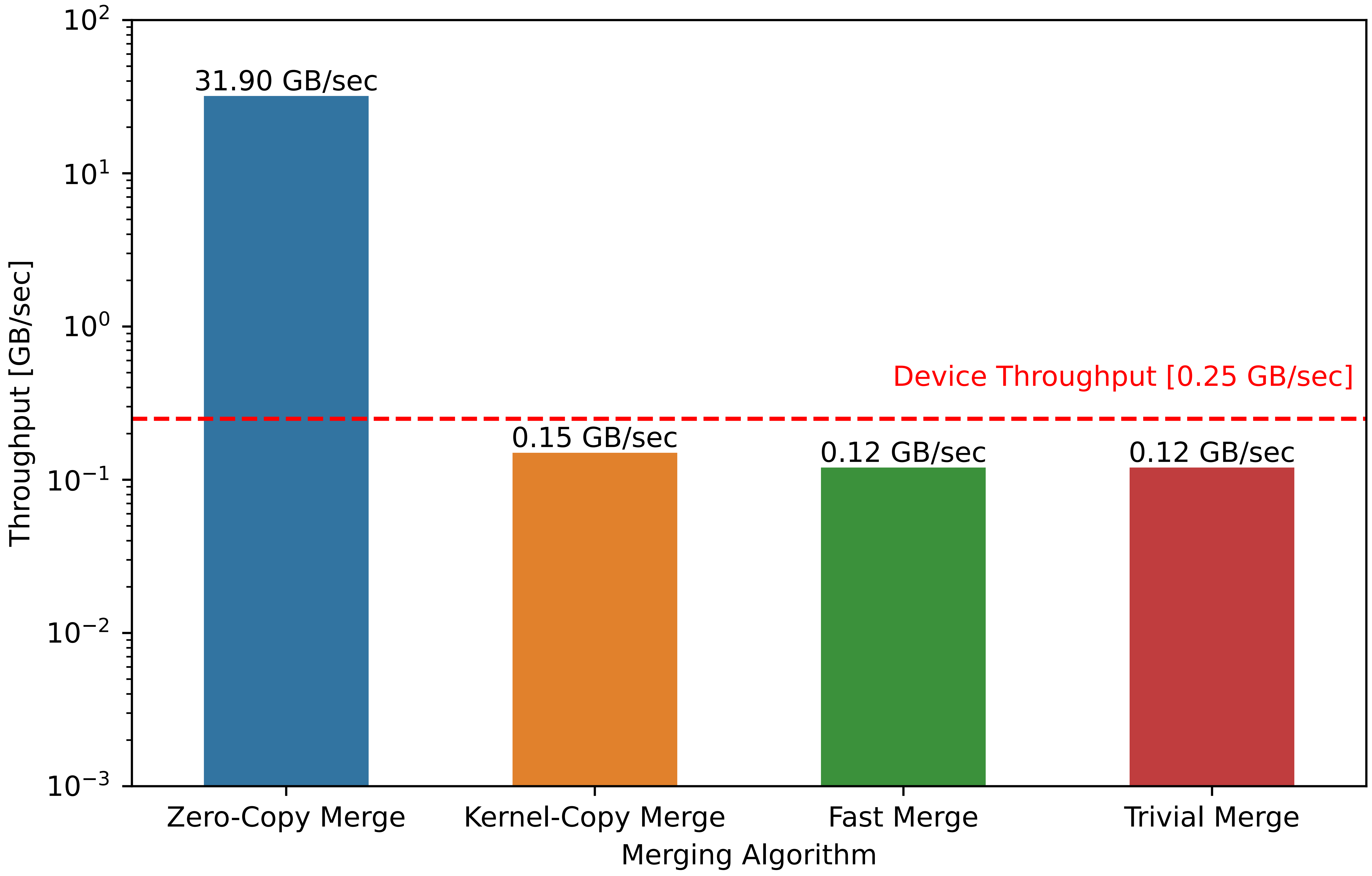
Evaluation: Execution Time

- Zero copy merge nearly constant
  - metadata updates is the only contribution
- Other merge grows linearly
- Test data is uncompressed



Evaluation: Throughput

- We achieve higher throughput than the device limit



Conclusion

- This first Zero-Copy merge implementation seems to perform well in many basic cases
- It outperforms alternative implementations
- Enables the RNTuple merge in almost constant time
- Ideally suit object store
  - By nature it enables zero-copy merge

Future Works

- Many questions still to be investigated
- Can we avoid padding?
- How to deal with OS different from Linux?
- What if we use file systems with different blocks sizes?
- What happens if on disk location of pages is interleaved with other objects?
  - Writing two RNTuples concurrently
- How to deal with different storage backends (such as object store Intel DAOS)

THANK YOU!