FAST'26

Towards Condensed and Efficient Read-Only File System via Sort-Enhanced Compression

Hao Huang[1], Yifeng Zhang[1], Yanqi Pan[1], Wen Xia[1], Xiangyu Zou[1], Darong Yang[1], Jubin Zhong[2], Hua Liao[2]

[1] Harbin Institute of Technology, Shenzhen

[2] Huawei Technologies Co., Ltd

Read-Only Image

Read-only images are **widely used** in IoT Devices, Android Smartphones, and Docker Containers.

Read-Only File System

- **Read-only image gets larger:**
  - System upgrades (Linux kernel, etc.)
  - Increased features (Application functions, etc.)

- **ROFS gets popular: Read-only layout + Compression**
  - SquashFS (Open-source project)
  - EROFS (@ATC'2019)

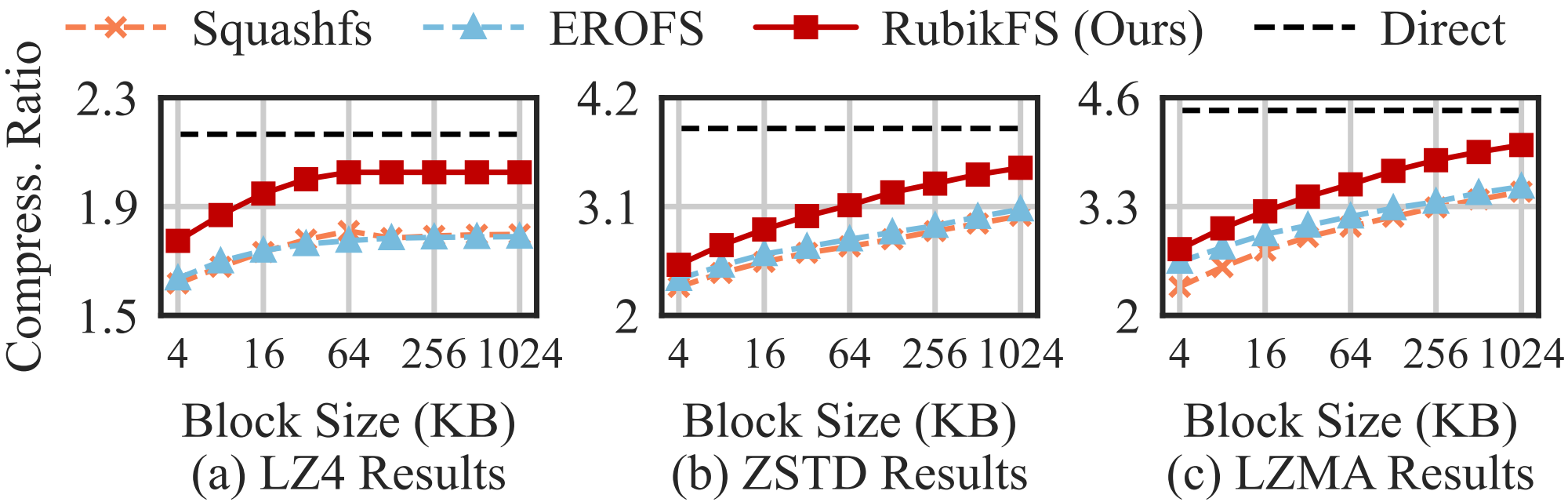
Read-Only File System

- **Read-only image gets larger:**
  - System upgrades (Linux kernel, etc.)
  - Increased features (Application functions, etc.)

- **ROFS gets popular in Read-only layer of Compression**
  - SquashFS (Open-source project)
  - EROFS (@ATC'2019)

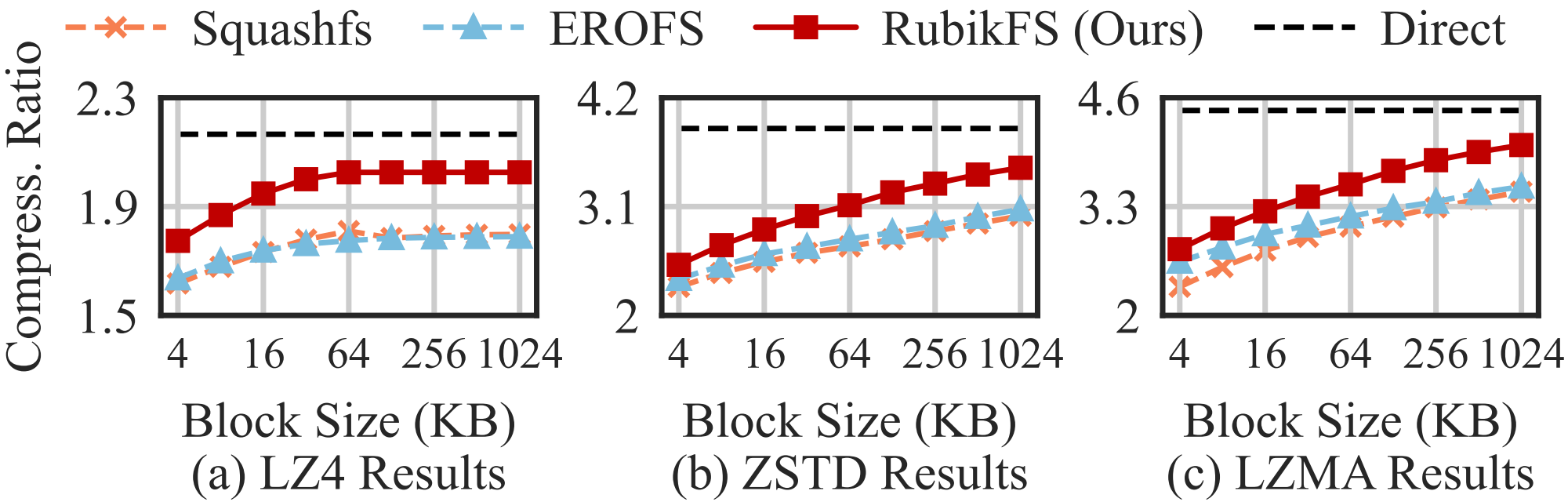
What is the Problem ?

Read-Only Image is Still Large



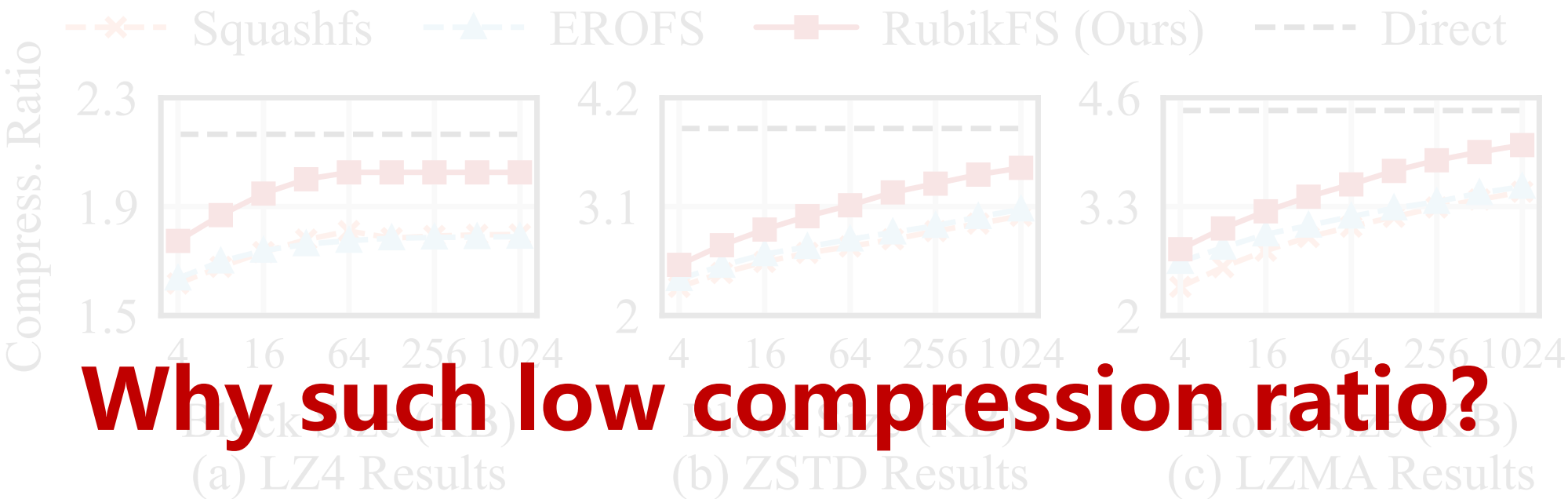
- **Direct:** Near-upper bound of ROFS.
- **Against baseline:** SquashFS and EROFS (@ATC'19)

Read-Only Image is Still Large



- **Direct:** Near-upper bound of ROFS.
- **Against baseline:** SquashFS and EROFS (@ATC'19)
- **Results:** Compression ratio is much lower than Direct

Read-Only Image is Still Large

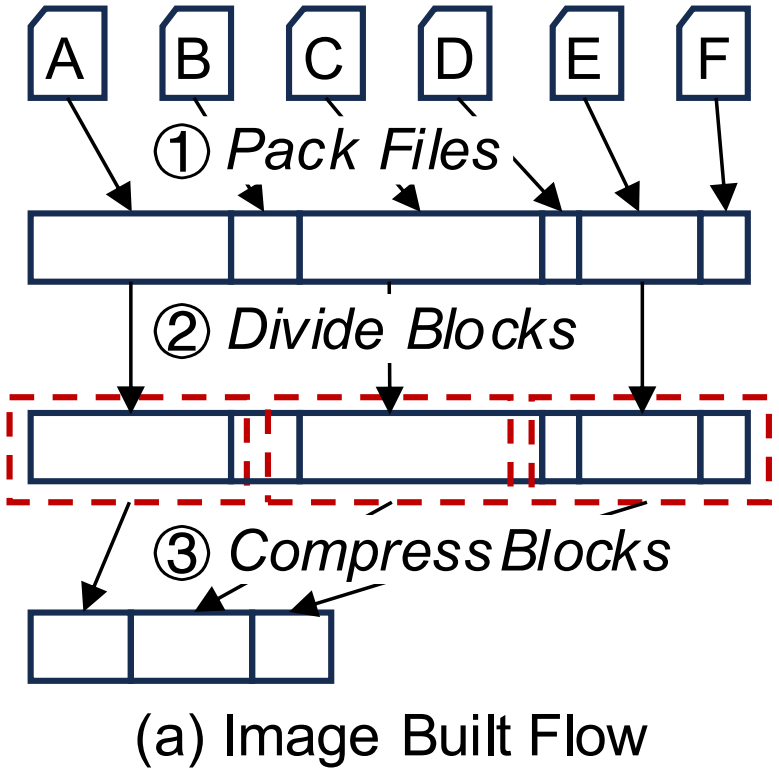


Why such low compression ratio?

- **Direct:** Near-upper bound of ROFS.
- **Against baseline:** SquashFS and EROFS (@ATC'19)
- **Results:** Compression ratio is much lower than Direct

Data Mixture is the Key

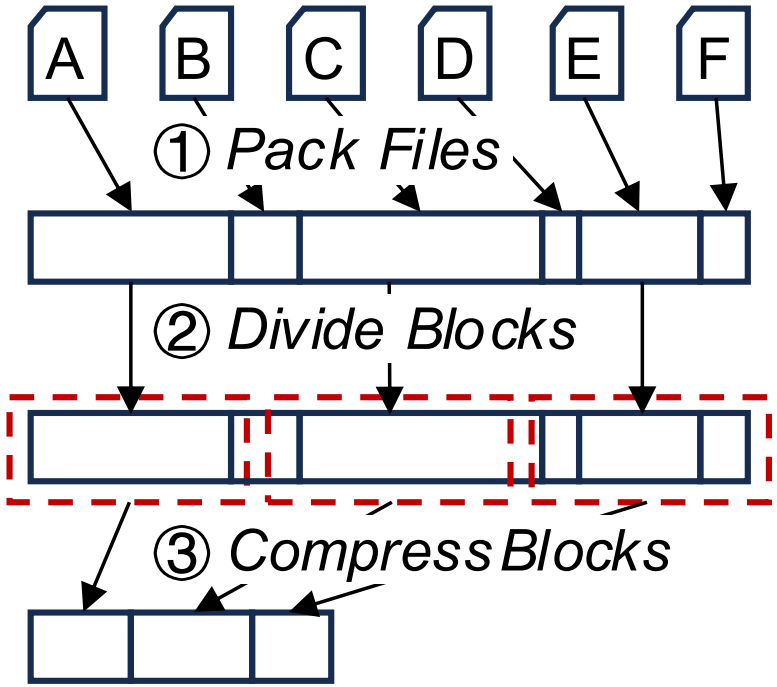
- **Image built flow:** Block division + block compression



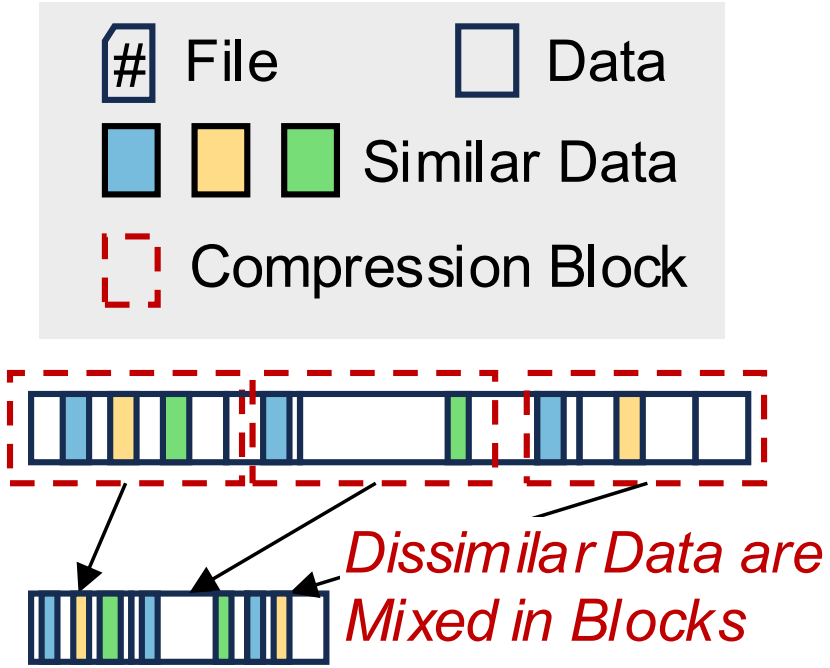
(a) Image Built Flow

Data Mixture is the Key

- **Image built flow:** Block division + block compression
- **Data mixture problem:** Dissimilar & hot/cold data are mixed



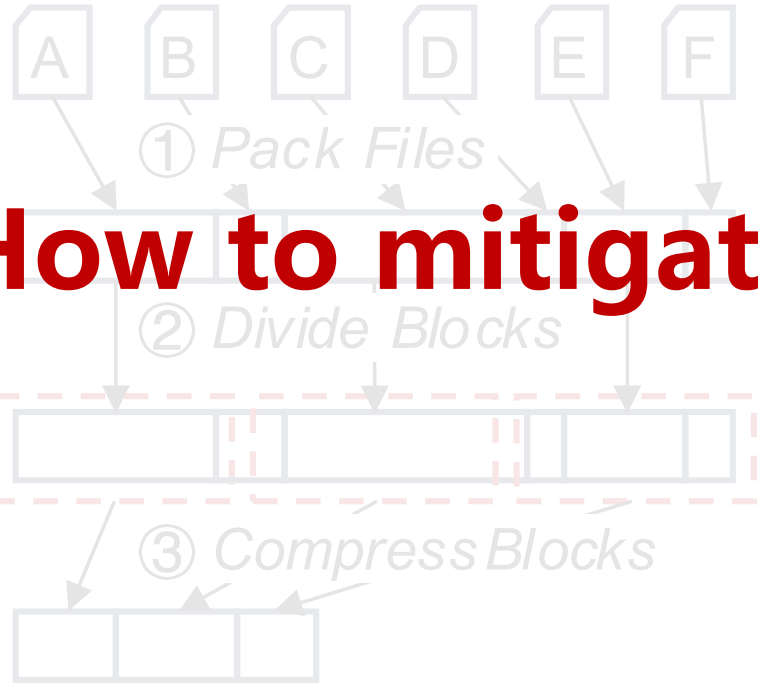
(a) Image Built Flow



(b) Data Mixture Problem

Data Mixture is the Key

- **Image built flow:** Block division + block compression
- **Data mixture problem:** Dissimilar & access-associated data are mixed



(a) Image Built Flow

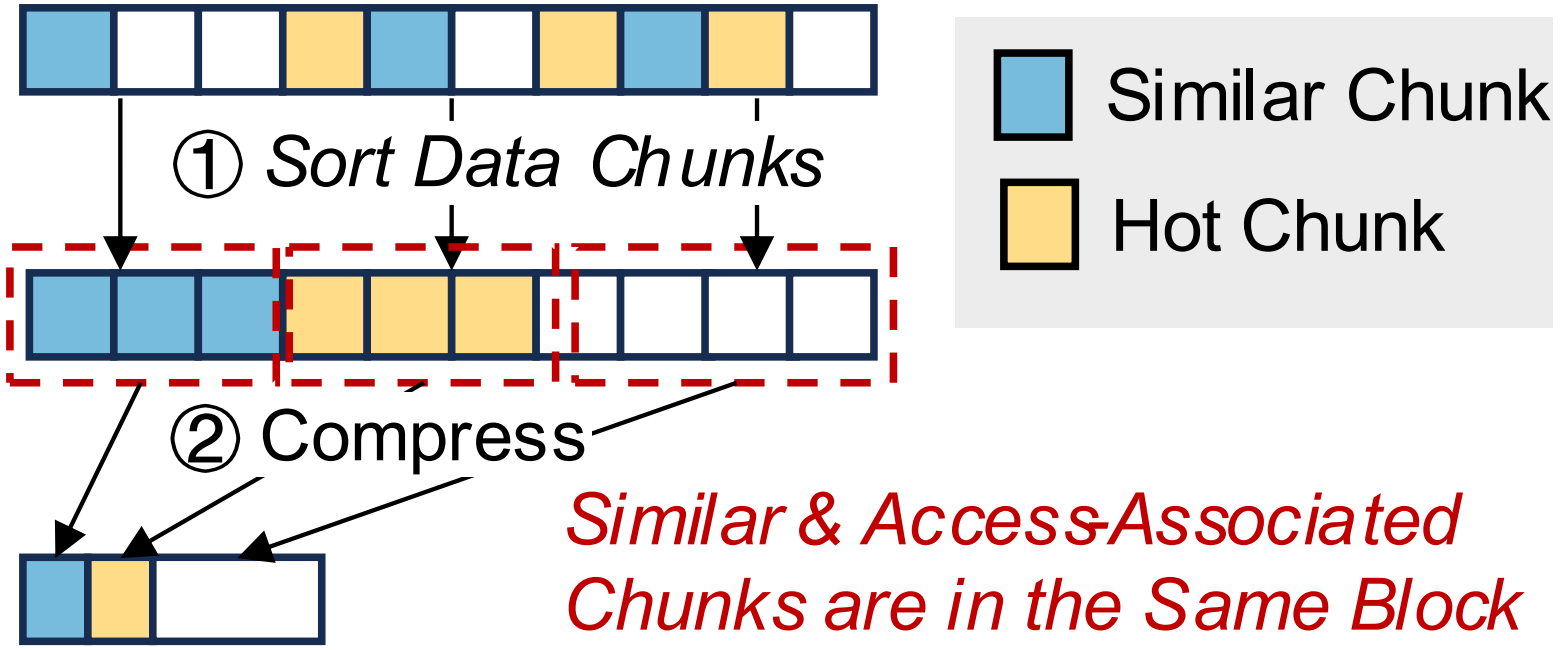
(b) Data Mixture Problem

How to mitigate the data mixture problem?

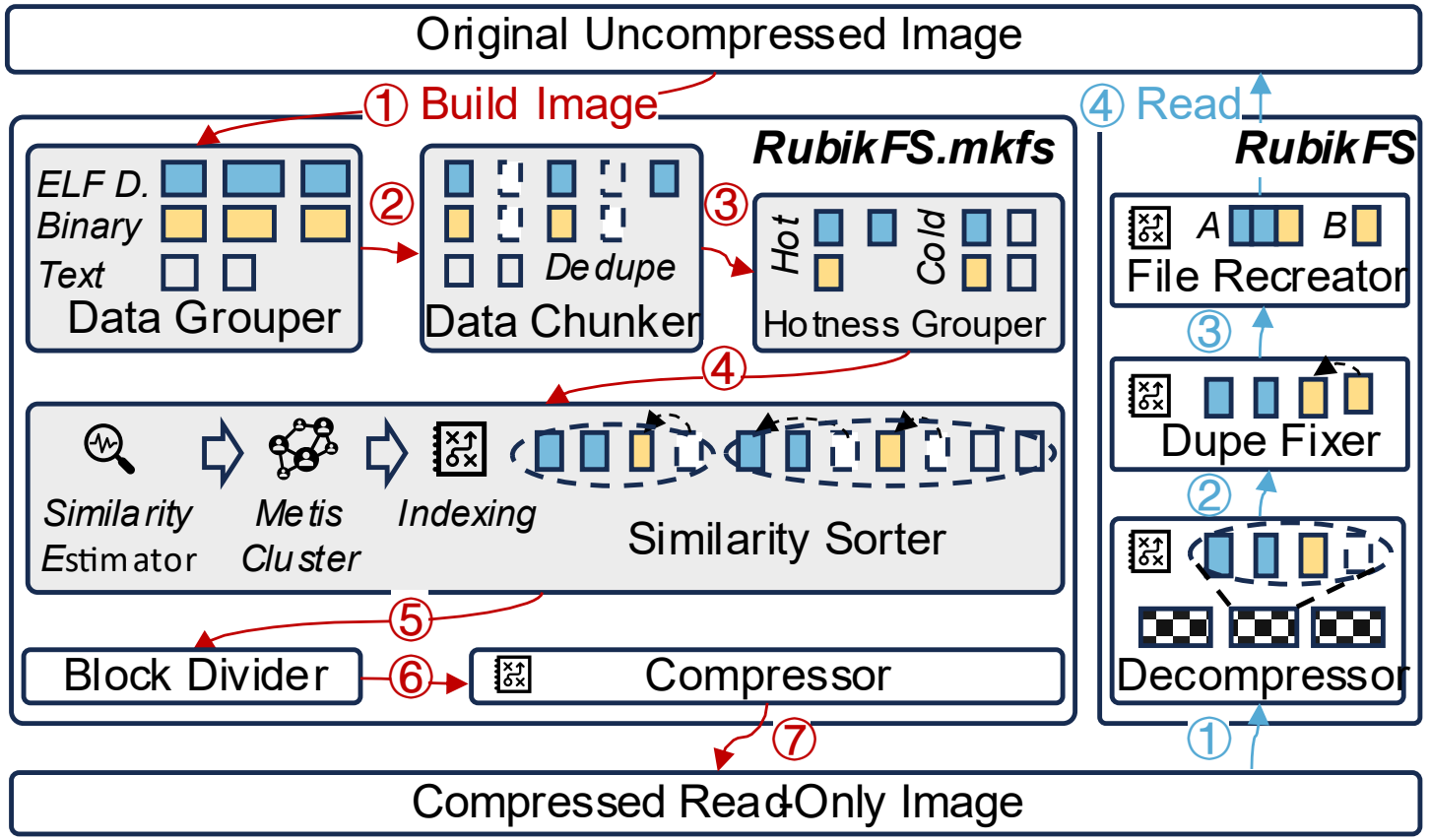
Insight: Sort-Enhanced Compression

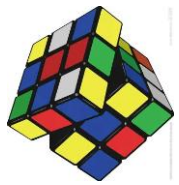
Leveraging sort-enhanced compression to improve:

- **Compression ratio:** Grouping similar chunks into the same block
- **Runtime performance:** Grouping access-associated hot chunks



RubikFS Overview



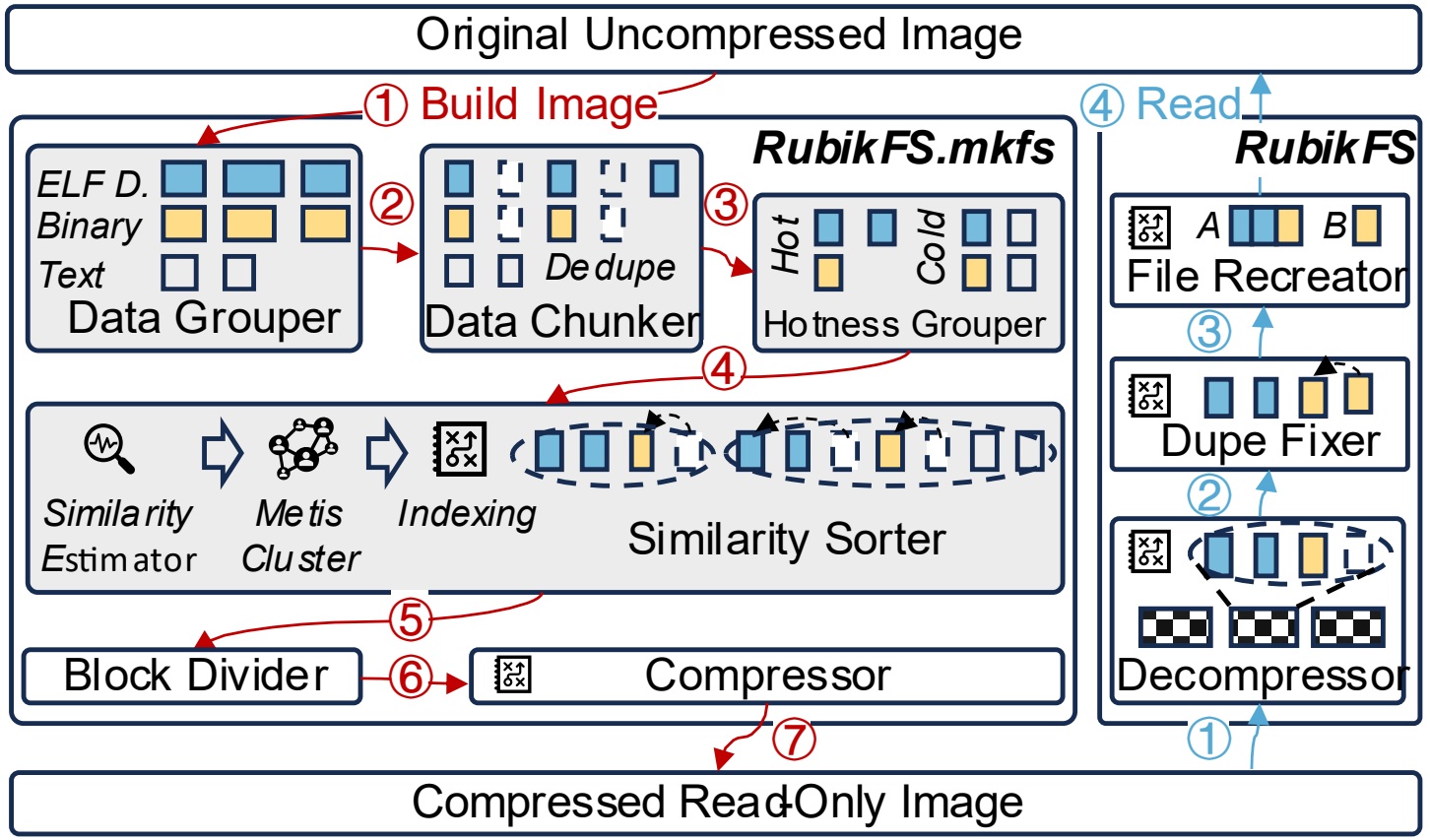


Rubik's cube restores mixed colors into uniform faces, analogous to RubikFS, which clusters similar chunks into the same block.

RubikFS Overview

- **Image builder**
  - Data grouper
  - Data chunker
  - Hotness grouper
  - Similarity sorter
- File system
  - Index
  - ……

See our paper (§4.6)

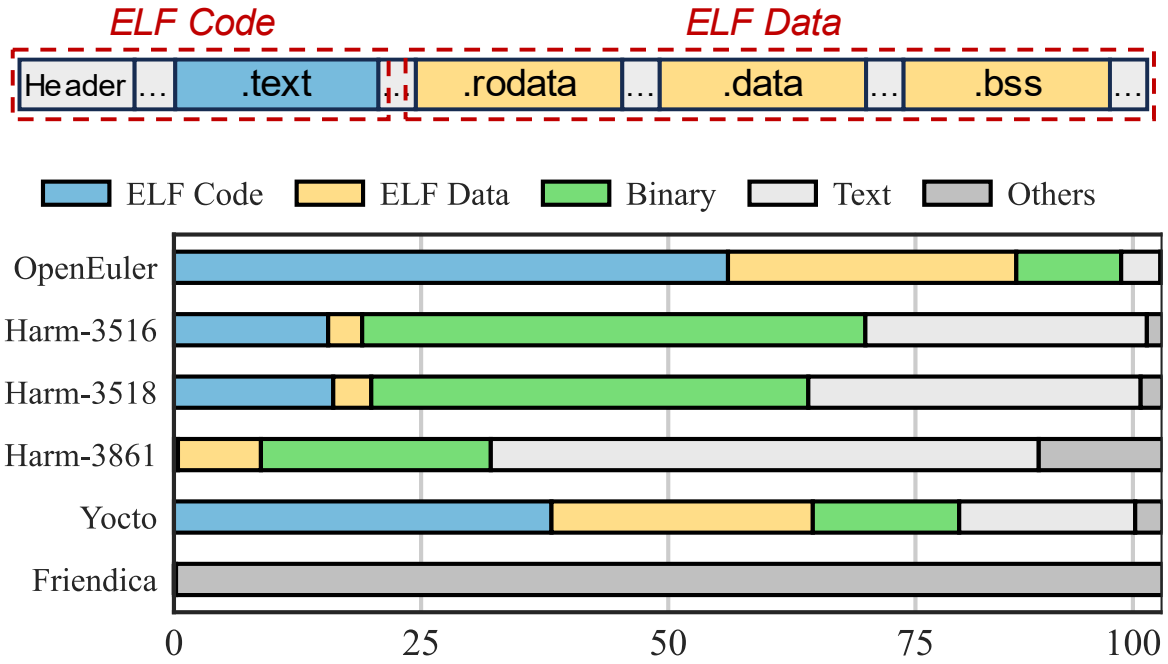


Rubik's cube restores mixed colors into uniform faces, analogous to RubikFS, which clusters similar chunks into the same block.

RubikFS: Data Grouper

Pre-group data through type to mitigate sorting complexity

- **ELF code:** Program code of ELF files
- **ELF data:** Program data of ELF files
- **Binary:** Other executable binaries
- **Text:** Scripts and configuration files
- **Others:** Pictures, videos, audios, etc.



RubikFS: Data Chunker

- **Chunk:** The basic sort unit, which are grouped into blocks to compress
- **Chunk deduplication:** Removing duplicate chunks

RubikFS: Data Chunker

- **Chunk:** The basic sort unit, which are grouped into blocks to compress
- **Chunk deduplication:** Removing duplicate chunks
- **Chunk size configuration:**
  - A trade-off between sort/deduplication granularity and data sequentiality

RubikFS: Data Chunker

- **Chunk:** The basic sort unit, which are grouped into blocks to compress
- **Chunk deduplication:** Removing duplicate chunks
- **Chunk size configuration:**
  - A trade-off between sort/deduplication granularity and data sequentiality
  - $Chunk\ Size = min\left(4KB, \frac{Block\ Size}{16}\right)$

RubikFS: Hotness Grouper

- **Hotness definition:**
  - Data accessed during system startup, since runtime accesses rarely incur additional I/Os for embedded systems.

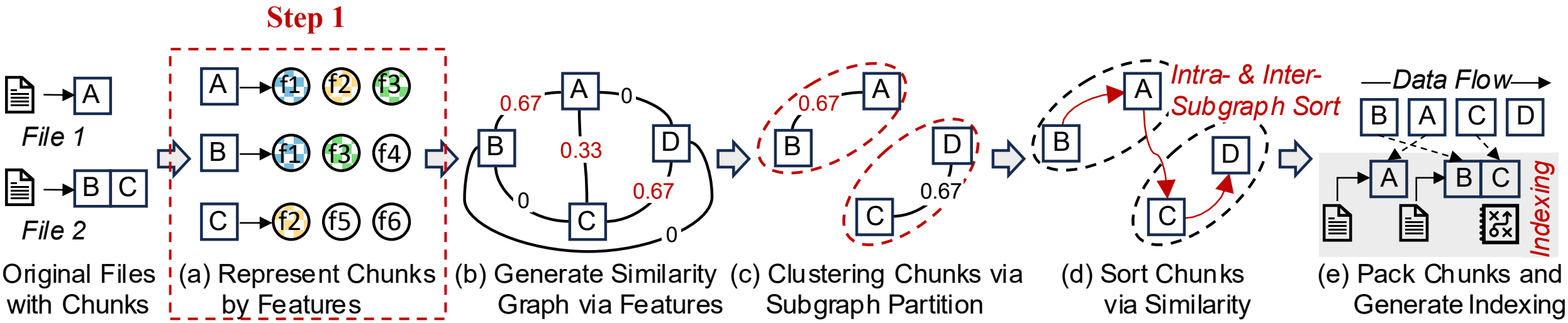
RubikFS: Hotness Grouper

- **Hotness definition:**
  - Data accessed during system startup, since runtime accesses rarely incur additional I/Os for embedded systems.
- **Tracing hot chunks:**
  1. kprobe readpage to capture hot pages
  2. Deploy and run the image until the system reaches steady state
  3. Parse the collected logs into the trace file

RubikFS: Hotness Grouper

- **Hotness definition:**
  - Data accessed during system startup, since runtime accesses rarely incur additional I/Os for embedded systems.
- **Tracing hot chunks:**
  1. kprobe readpage to capture hot pages
  2. Deploy and run the image until the system reaches steady state
  3. Parse the collected logs into the trace file
- **Grouping hot chunks**
  - Divide chunks into hot and cold subgroups

RubikFS: Similarity Sorter



RubikFS: Similarity Sorter

1. Represent chunks by features

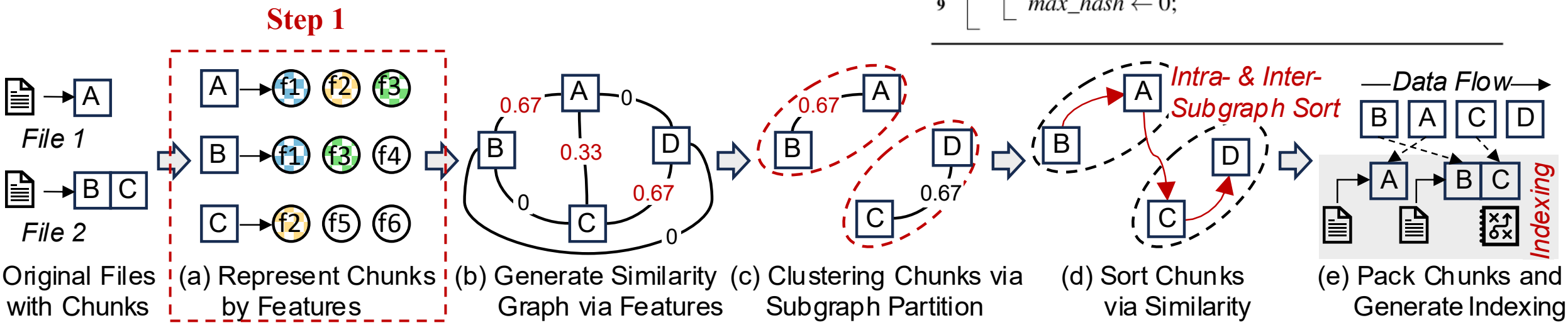
- New feature extraction algorithm
- Quantify chunk similarity (0~1), rather than identifying similarity (0 or 1)

Algorithm 1: Feature Extraction in RubikFS

Input: Chunk ***In***, Chunk Size ***N***, Sampling Rate ***P***, Gear Matrix ***Matrix***

Output: Feature Array ***Features***

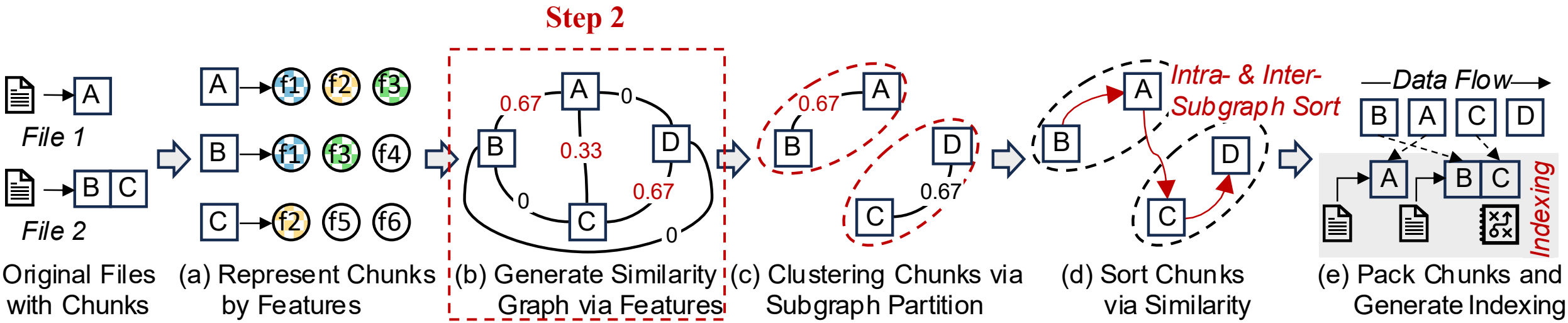
```
1 max_hash ← 0;
2 gear_hash ← 0;
3 for i = 0 to N − 1 do
4     gear_hash ← (gear_hash ≪ 1) + Matrix[In[i]];
5     if gear_hash > max_hash then
6         max_hash ← gear_hash;
7     if (i + 1) mod 1/P = 0 then
8         Features.append(max_hash);
9         max_hash ← 0;
```



RubikFS: Similarity Sorter

2. Generate similarity graph via features

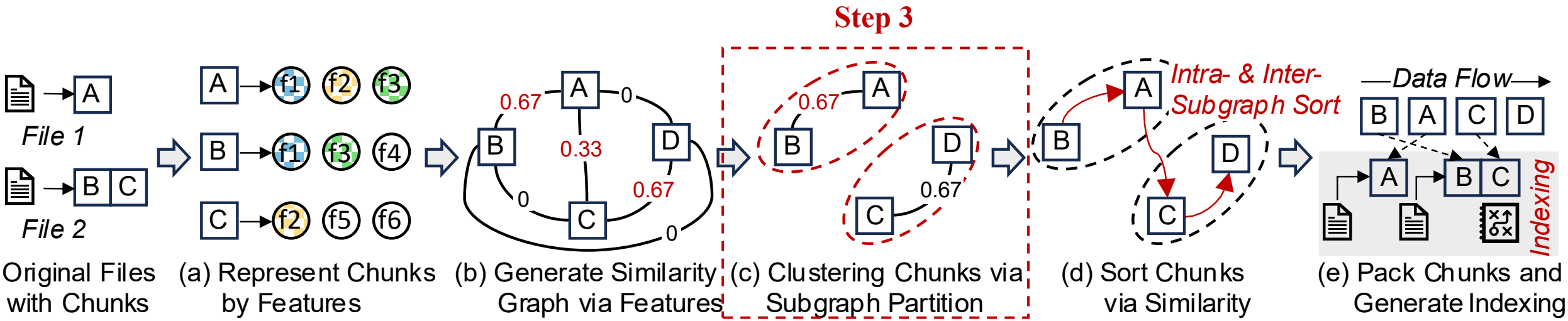
- Chunks are nodes, while chunk similarities are edges
- $Chunk\ Similarity = \frac{Identical\ Features}{Total\ Features}$
- Similarity between *Chunk A* and *Chunk B* is $\frac{2 \times 2}{6} = 0.67$



RubikFS: Similarity Sorter

3. Cluster chunks via subgraph partition

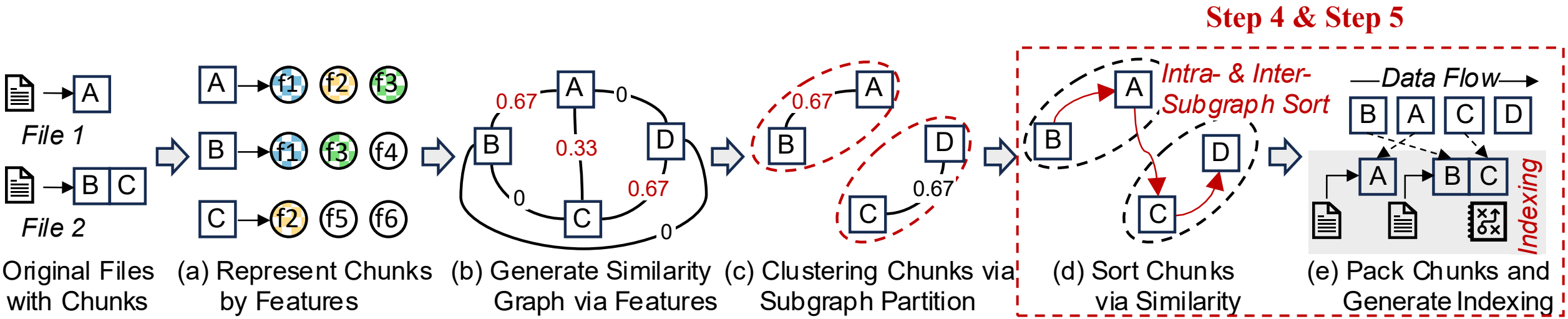
- **Partition goals:** Edges in the same subgraph are of low value, while edges across two subgraphs are of low value
- **Partition algorithm:** METIS algorithm



RubikFS: Similarity Sorter

4. **Sort chunks via similarity**
   - Chunks with the most identical features are placed at the front

5. **Pack chunks and generate indexing**
   - Indexing is used to recover data during access



RubikFS Evaluation

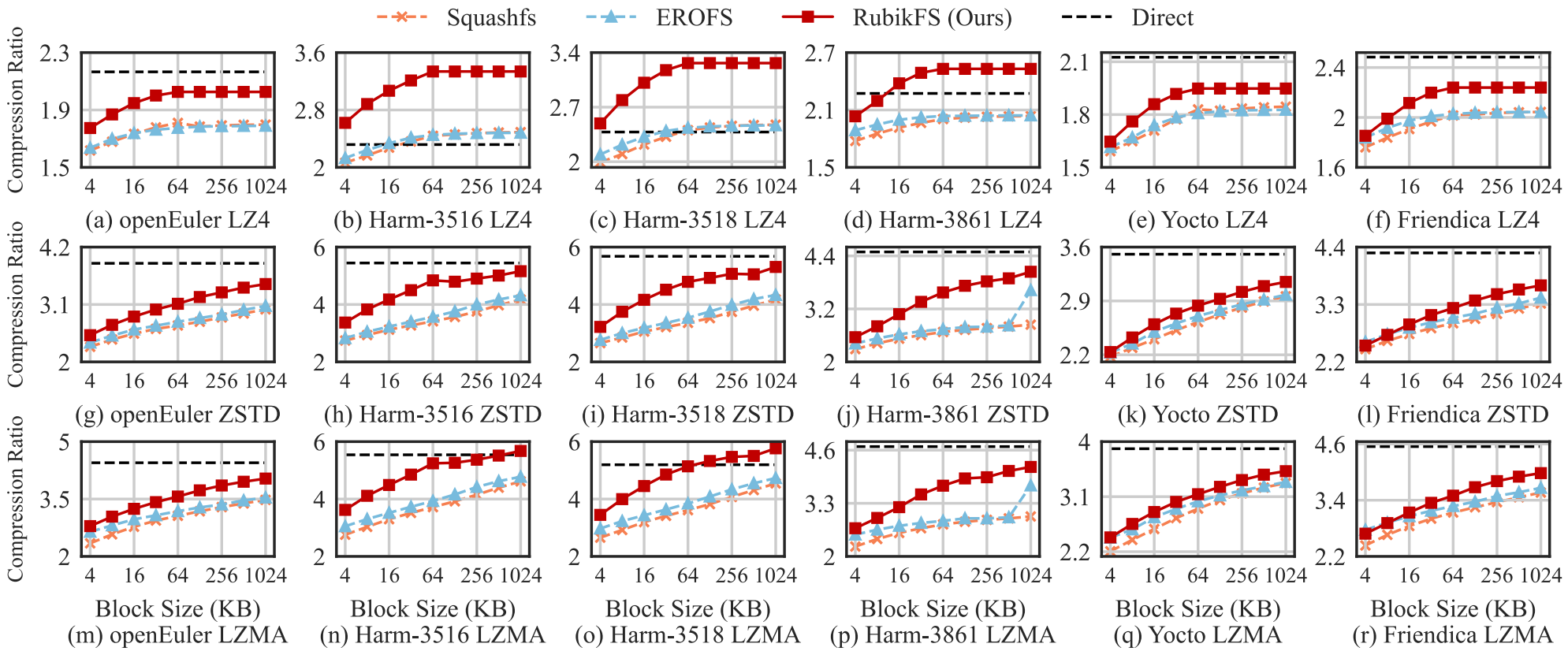
- **Experimental setup**
  - FEMU with 75 µs page read latency
  - Linux kernel 6.16, 2 CPU cores, 1 GiB DRAM
  - Compression algorithms: LZ4, ZSTD, LZMA

RubikFS Evaluation

- **Experimental setup**
  - FEMU with 75 µs read latency
  - Linux kernel 6.16, 2 CPU cores, 1 GiB DRAM
  - Compression algorithms: LZ4, ZSTD, LZMA

- **Competitors**
  - Direct (near-upper bound)
  - SquashFS, EROFS
  - **RubikFS (base on EROFS)**

Compression Ratio Evaluation



- **Setup:** 6 open-source images with 3 compression algorithms.
- **Result 1:** RubikFS increases the compression ratio by up to **42.60%**.

Read Amplification Evaluation

Config.		Comp.	File System			
			RubikFS	w/o	EROFS	Squashfs
12% Hotness	Time (s)	LZ4	1.21	2.88	2.89	3.46
		ZSTD	1.63	3.03	2.99	4.00
		LZMA	3.72	9.40	9.15	9.09
	Read Size (MB)	LZ4	16.41	55.72	56.00	45.13
		ZSTD	13.37	36.60	34.89	25.62
		LZMA	10.13	31.02	29.87	21.58
40% Hotness	Time (s)	LZ4	3.02	4.37	4.85	5.57
		ZSTD	3.03	3.88	4.30	6.39
		LZMA	6.34	11.01	11.75	13.84
	Read Size (MB)	LZ4	32.15	72.71	75.46	66.86
		ZSTD	24.39	41.60	44.88	40.14
		LZMA	20.82	35.00	39.47	34.83

- **Setup:** The openEuler image with 12% and 40% hot data.
- **Result 2:** RubikFS reduces unnecessary reads by up to **70.70%**.

Read Amplification Evaluation

Config.		Comp.	File System			
			RubikFS	w/o	EROFS	Squashfs
12% Hotness	Time (s)	LZ4	1.21	2.88	2.89	3.46
		ZSTD	1.63	3.03	2.99	4.00
		LZMA	3.72	9.40	9.15	9.09
	Read Size (MB)	LZ4	16.41	55.72	56.00	45.13
		ZSTD	13.37	36.60	34.89	25.62
		LZMA	10.13	31.02	29.87	21.58
40% Hotness	Time (s)	LZ4	3.02	4.37	4.85	5.57
		ZSTD	3.03	3.88	4.30	6.39
		LZMA	6.34	11.01	11.75	13.84
	Read Size (MB)	LZ4	32.15	72.71	75.46	66.86
		ZSTD	24.39	41.60	44.88	40.14
		LZMA	20.82	35.00	39.47	34.83

Other results:

- *Breakdown Analysis (§6.3)*
- *Image Build Time (§6.5)*
- *Sensitive Analysis (§6.6)*

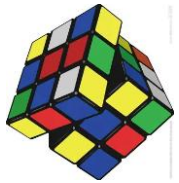
Please refer to our paper

- **Setup:** The openEuler image with 12% and 40% hot data.
- **Result 2:** RubikFS reduces unnecessary reads by up to **70.70%**.

Conclusion

1. **Problem: Data mixture in read-only file systems**
   - Low compression ratio since dissimilar data are mixed in blocks
   - High read amplification due to large block size (e.g., 1 MB)

2. **Motivation: Sort-enhanced compression**
   - Similar and hot data are grouped into the same blocks.

3. **RubikFS, a ROFS based on sorting** 
   - Sorting is fully compatible with ROFS (e.g., EROFS, SquashFS)
   - Results. Outperforms competitors in both compression ratio and read amplification

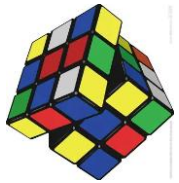
Conclusion

1. **Problem: Data mixture in read-only file systems**
   - Low compression ratio since dissimilar data are mixed in blocks
   - High read amplification due to large block size (e.g., 1 MB)

2. **Motivation: Sort-enhanced compression**
   - Similar and hot data are grouped into the same blocks.

Thanks & QA

3. **RubikFS, a ROFS based on sorting**



   - Sorting is fully compatible with ROFS (e.g., EROFS, SquashFS)
   - Results. Outperforms competitors in both compression ratio and read amplification