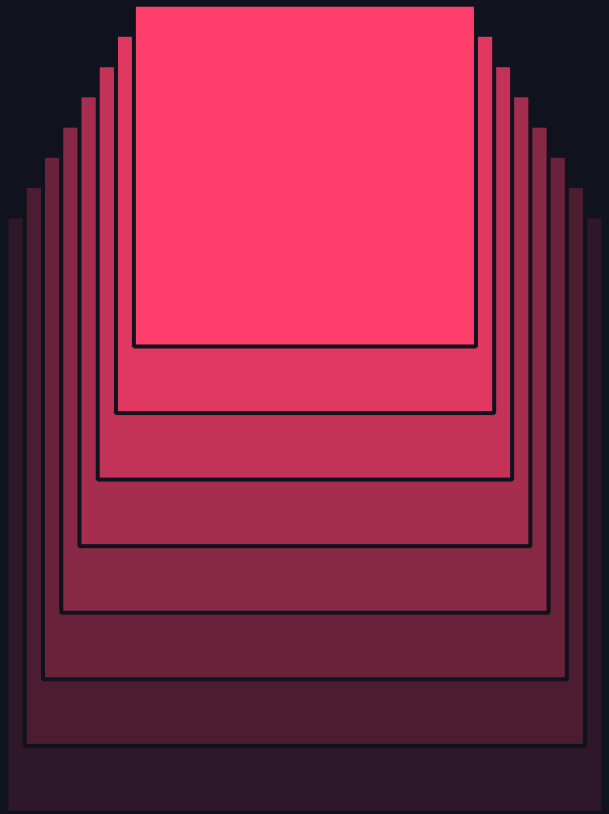
DATA+AI SUMMIT BY databricks

DELTA TENSOR

Efficient Tensor Storage in Delta Lake



Liaoliao Liu, Liam Bao, Zhiyu Wu

Our team

Liaoliao Liu

liu.liao@northeastern.edu

- Incoming SDE @ AWS Athena
- CS Graduate @ Northeastern University

Liam Bao

bao.zhiw@northeastern.edu

- Incoming SWE @ Snowflake
- CS Graduate @ Northeastern University

Zhiyu Wu

wu.zhiyu@northeastern.edu

- CS Graduate @ Northeastern University

Contents

01.

INTRODUCTION

Background

- Exponential growth of AI/ML applications
- Inefficient methods for tensor storage
  - Redundant storage space
  - Slow processing speed

Related works

- Focus on vectors (1d arrays) rather than tensors (nd arrays)

- Miss cloud-native environment: cloud storage offers more than just disks

Objective

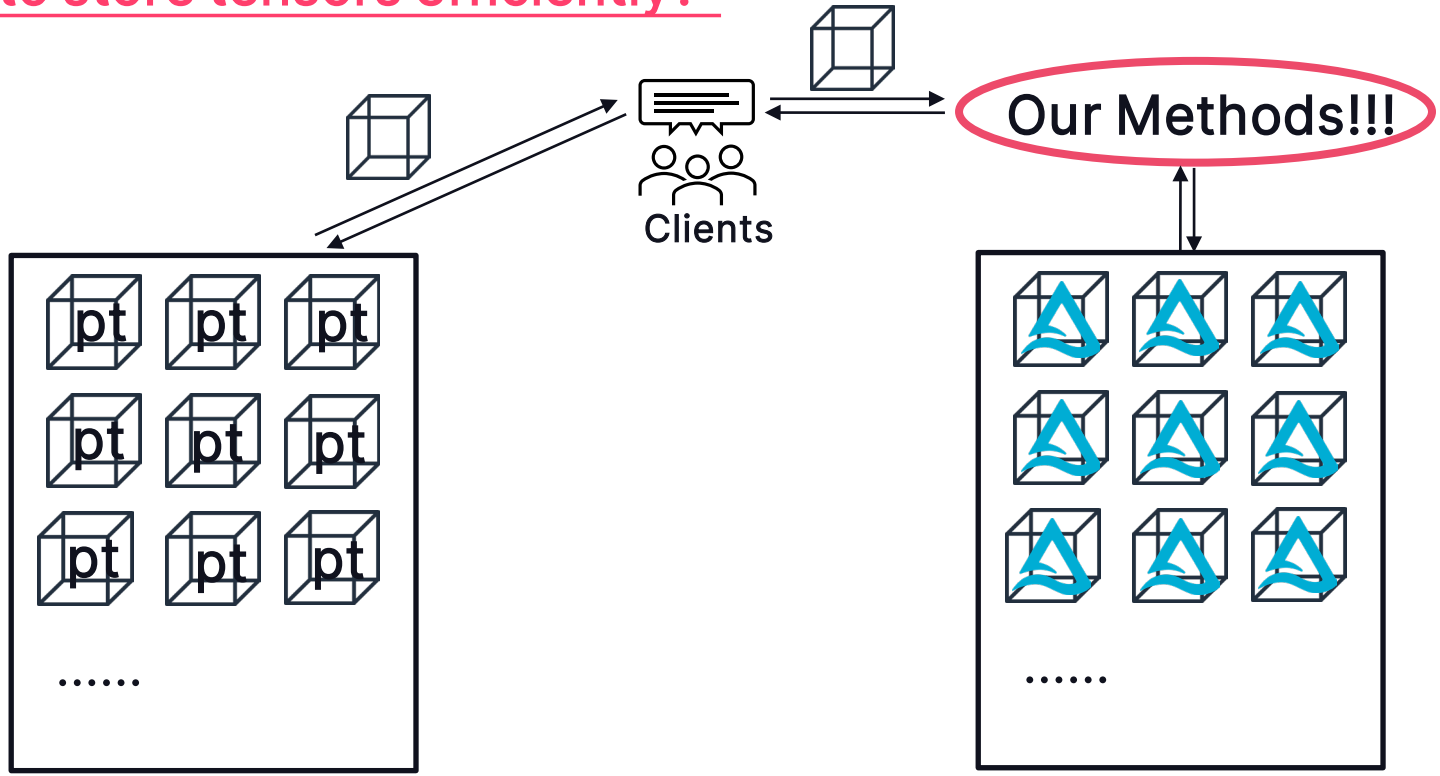
- Efficient tensor storage in cloud object storage

02.

PROBLEM STATEMENT

Problem Statement

How to store tensors efficiently?



Problem Statement

Terminologies

Definitions

Vector:

$$a = [a_1, a_2, \dots, a_n]$$

Tensor:

$$X = \{x_{i^1,i^2,\dots,i^N} \mid i^j \in [1, d_j] \text{ for } j = 1,2,\dots,N\}$$

Tensor encoding/decoding:

$$X_{encode} = F(X)$$

$$X = F^{-1}(X_{encode})$$

Evaluation Metrics

Compression Ratio:

$$\boldsymbol{C_r = \frac{S_{encode}}{S_{binary}}}$$

Writing/Reading Time:

$$t_{en(X)} = Elapsed(F(X))$$

$$t_{de(X_{encode})} = Elapsed(F^{-1}(X_{encode}))$$

$$\boldsymbol{t_{write} = t_{ser} + t_{en}\ (X)}$$

$$\boldsymbol{t_{read\_tensor} = t_{des} + t_{de}\ (X_{encode})}$$

$$\boldsymbol{t_{read\_slice} = t_{des} + t_{de}\ (X_{encode[\dots]})}$$

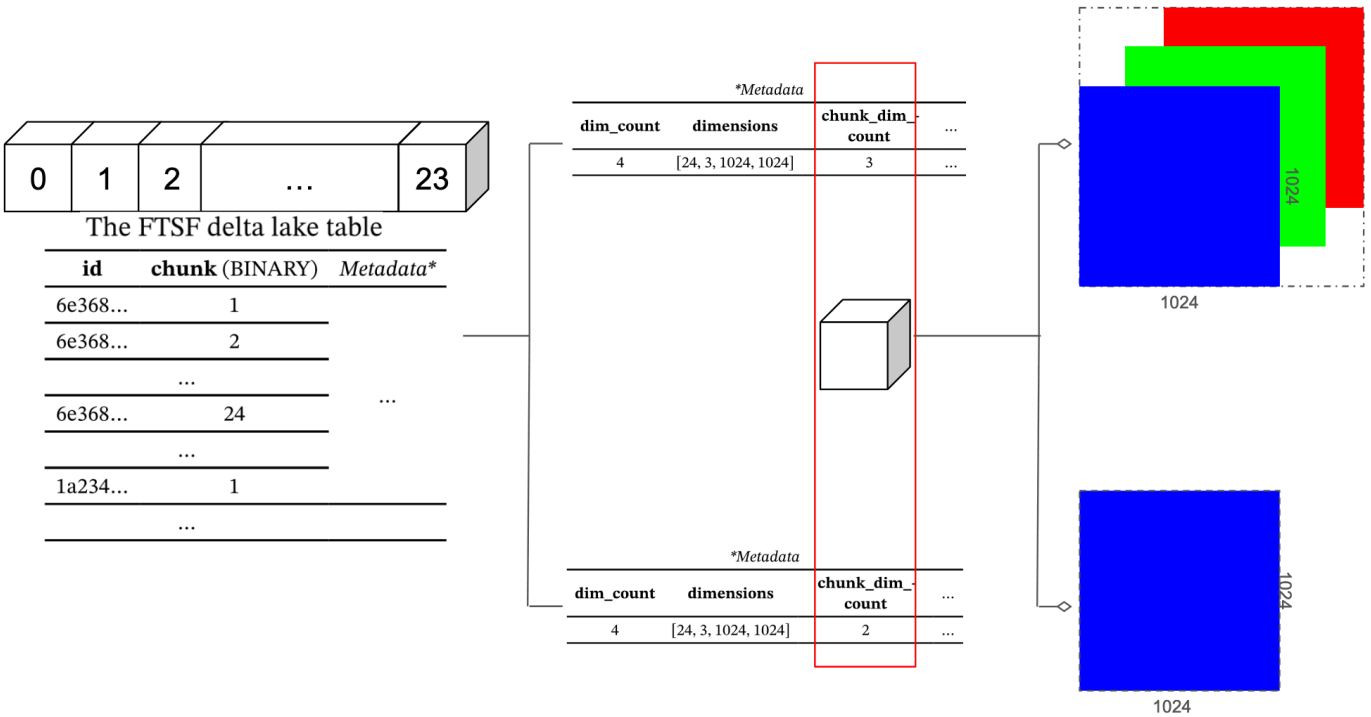
03.

METHODS

Method - General Tensors

Flattened Tensor Storage Format (FTSF)

- Tenet: only fetch the elements needed (OFTEN).

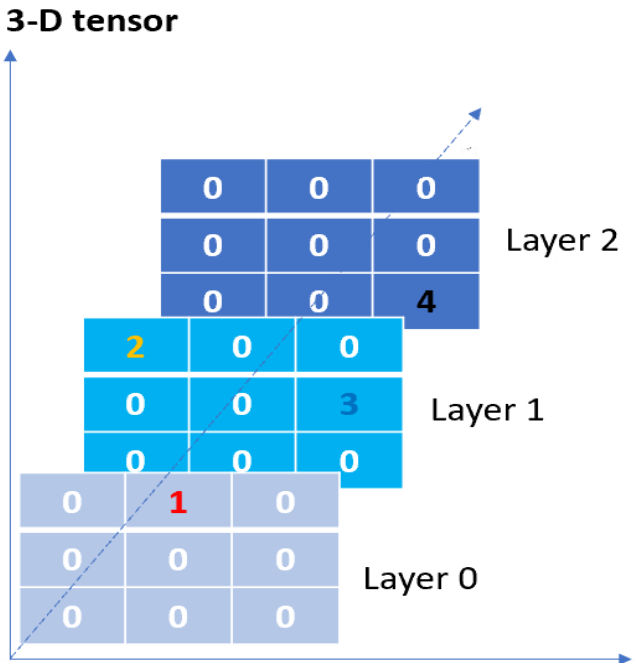


Why sparse matters

Curse of dimensionality:
As dimensions increase, data sparsity grows rapidly, demanding exponentially more data for reliable results.

Name	Non-zeros	Order	Dimensions	Sparse COO Size	Estimated Dense Size
Amazon Reviews	1,741,809,018	3	4,821,207 x 1,774,269 x 1,805,187	55.7 GB	123,534.3 PB
Uber Pickups	3,309,490	4	183 x 24 x 1,140 x 1,717	51.4 MB	68.8 GB
VAST 2015 Mini-Challenge 1	26,021,945	5	165,427 x 11,374 x 2 x 100 x 89	570.0 MB	267.9 TB

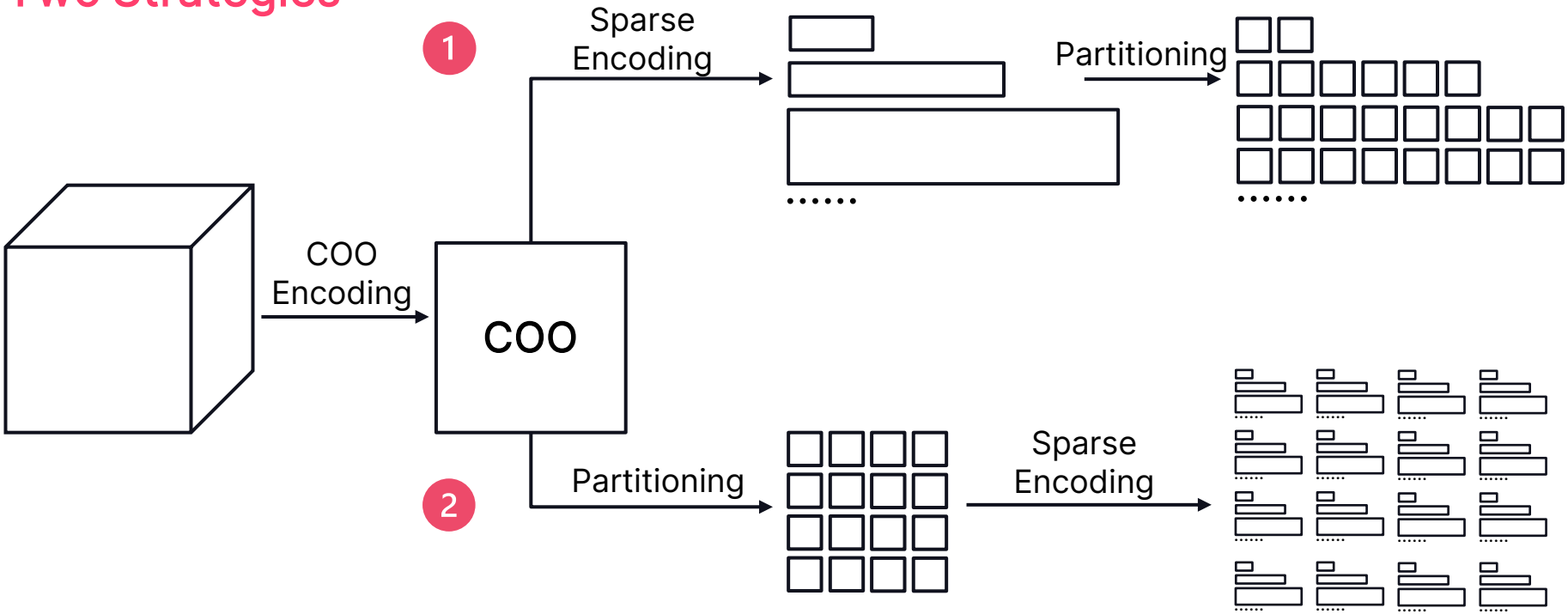
Sparse Encoding - Coordinate Format (COO)



COO:
[
[0, 1, 1, 2], → Layer
[0, 0, 1, 2], → Row
[1, 0, 2, 2], → Column
[1, 2, 3, 4] → Value
]

Methods - Sparse tensors

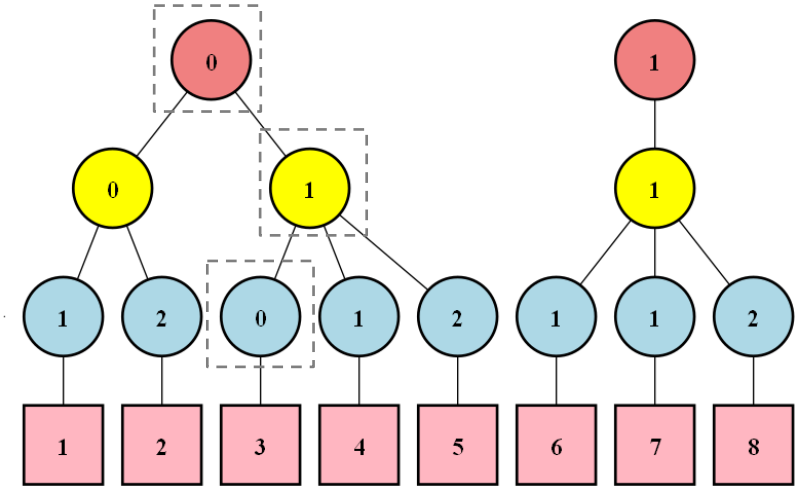
Two Strategies



Method 1 - Sparse Tensors

Compressed Sparse Fiber (CSF)

rows	0	0	0	0	0	1	1	1
columns	0	0	1	1	1	1	1	1
tubes	1	2	0	1	2	0	1	2
values	*1*	*2*	*3*	*4*	*5*	*6*	*7*	*8*



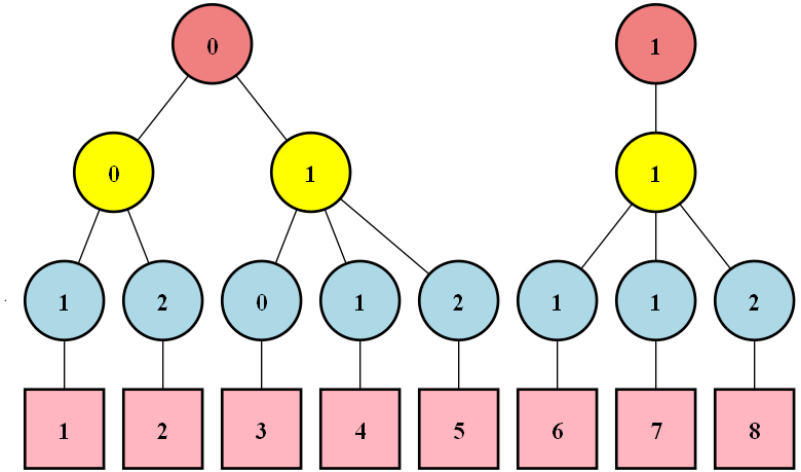
COO format

CSF Trie representation

Example: 2 x 2 x 3 tensor

Method 1 - Sparse Tensors

Compressed Sparse Fiber (CSF)



CSF Trie representation

fptr[0]	0	2	3					
fptr[1]	0	2	5	8				
fids[0]	0	1						
fids[1]	0	1	1					
fids[2]	1	2	0	1	2	1	1	2
values	*1*	*2*	*3*	*4*	*5*	*6*	*7*	*8*

Slicing Pointer

Level Indices

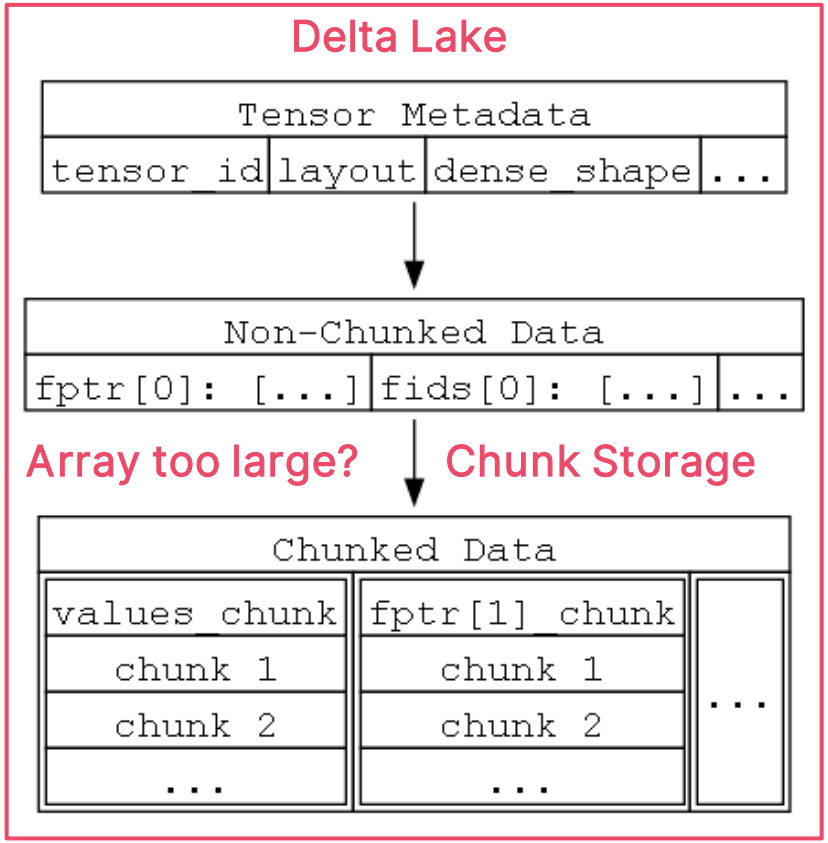
CSF Array representation

Method 1 - Sparse Tensors

Compressed Sparse Fiber (CSF)

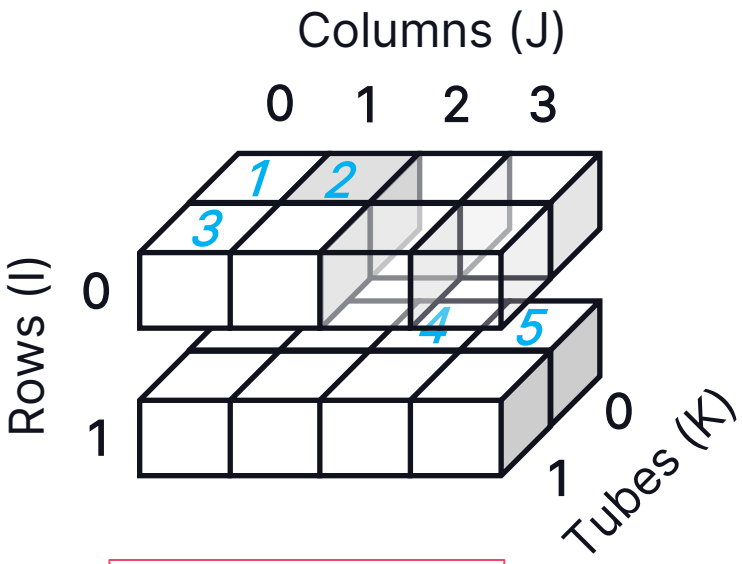
fptr[0]	0	2	3					
fptr[1]	0	2	5	8				
fids[0]	0	1						
fids[1]	0	1	1					
fids[2]	1	2	0	1	2	1	1	2
values	*1*	*2*	*3*	*4*	*5*	*6*	*7*	*8*

CSF Array representation



Method 2 - Sparse Tensors

Block Sparse Generic Storage (BSGS)



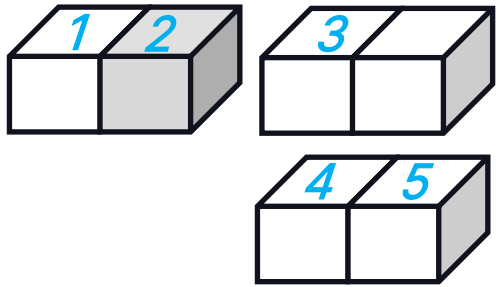
2 x 4 x 2 tensor

COO format

rows	0	0	0	1	1
columns	0	1	0	2	3
tubes	0	0	1	0	0
vals	*1*	*2*	*3*	*4*	*5*

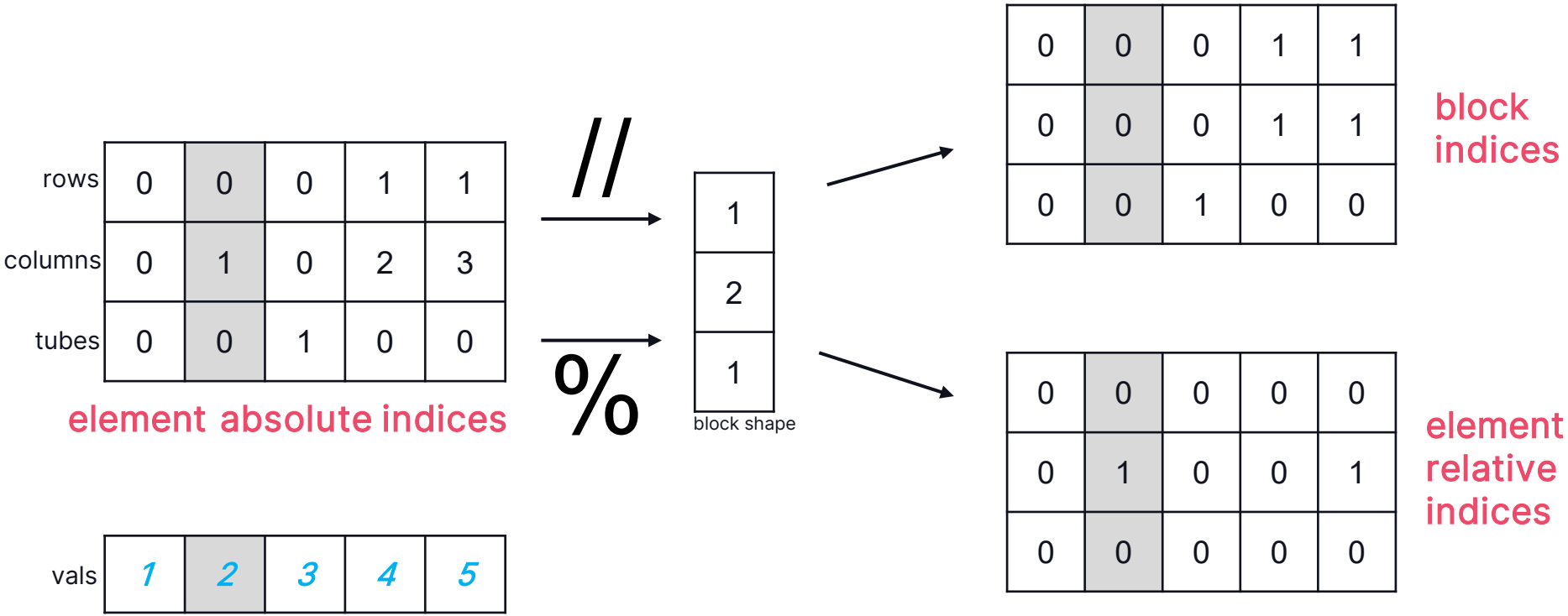
block shape

rows	1
columns	2
tubes	1



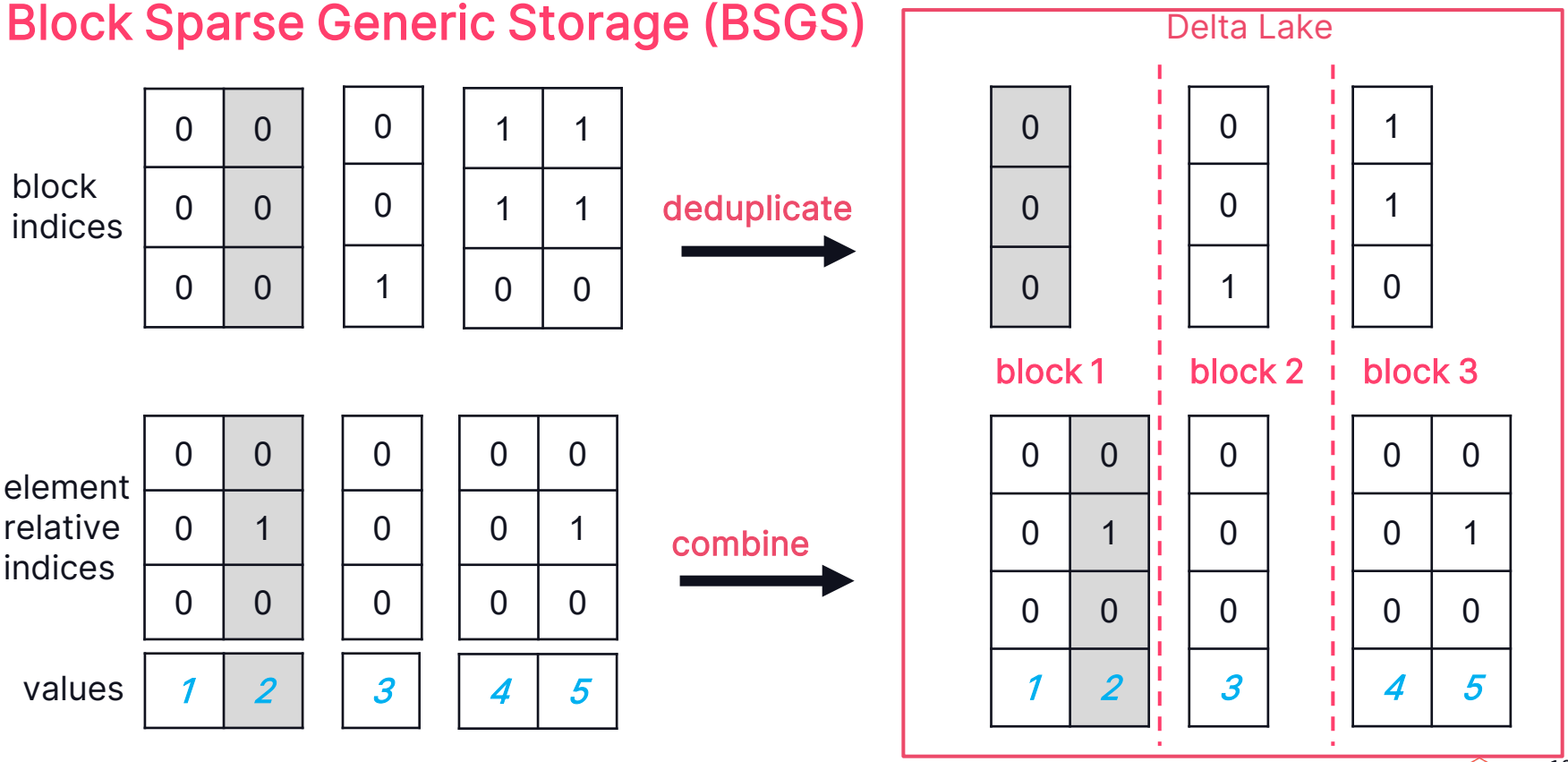
Method 2 - Sparse Tensors

Block Sparse Generic Storage (BSGS)



Method 2 - Sparse Tensors

Block Sparse Generic Storage (BSGS)



Method 2 - Sparse Tensors

Block Sparse Generic Storage (BSGS)

Internal table - column compression:

id	dense_shape	block_shape	block_indices	block_values
1, 3	[2,4,2], 3	[1,2,1], 3	[0,0,0]	[[0,0], [0,1], [0,0], [1,2]]
			[0,0,1]	[[0], [0], [0], [3]]
			[1,1,0]	[[0,0], [0,1], [0,0], [4,5]]

Tensor Slicing Operation - read tensor

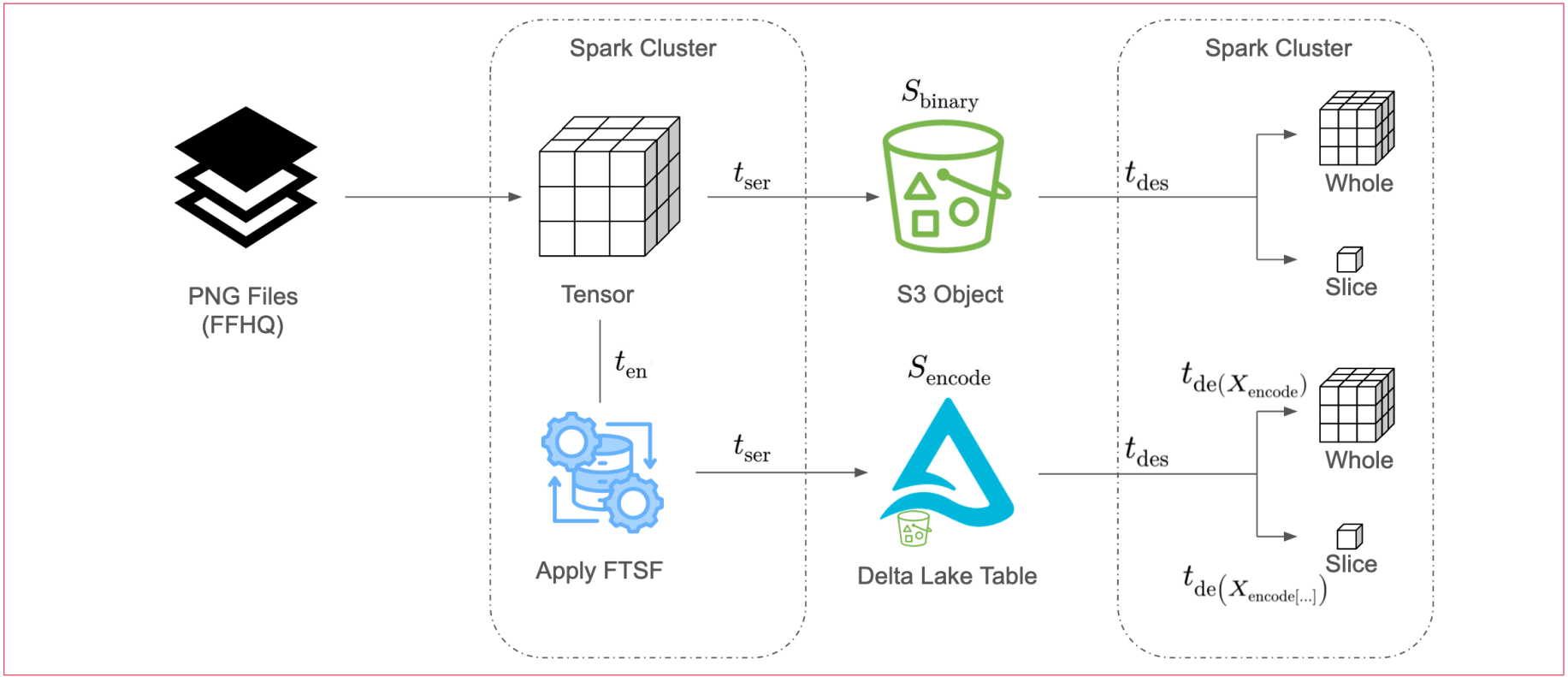
Steps to get tensor[0, :, :]:

1. Filter table using id.
2. Get block_shape and dense_shape, calculate the slice shape 1 x 4 x 2.
3. Scan the block_indices column to get block_values.
4. Construct the slice

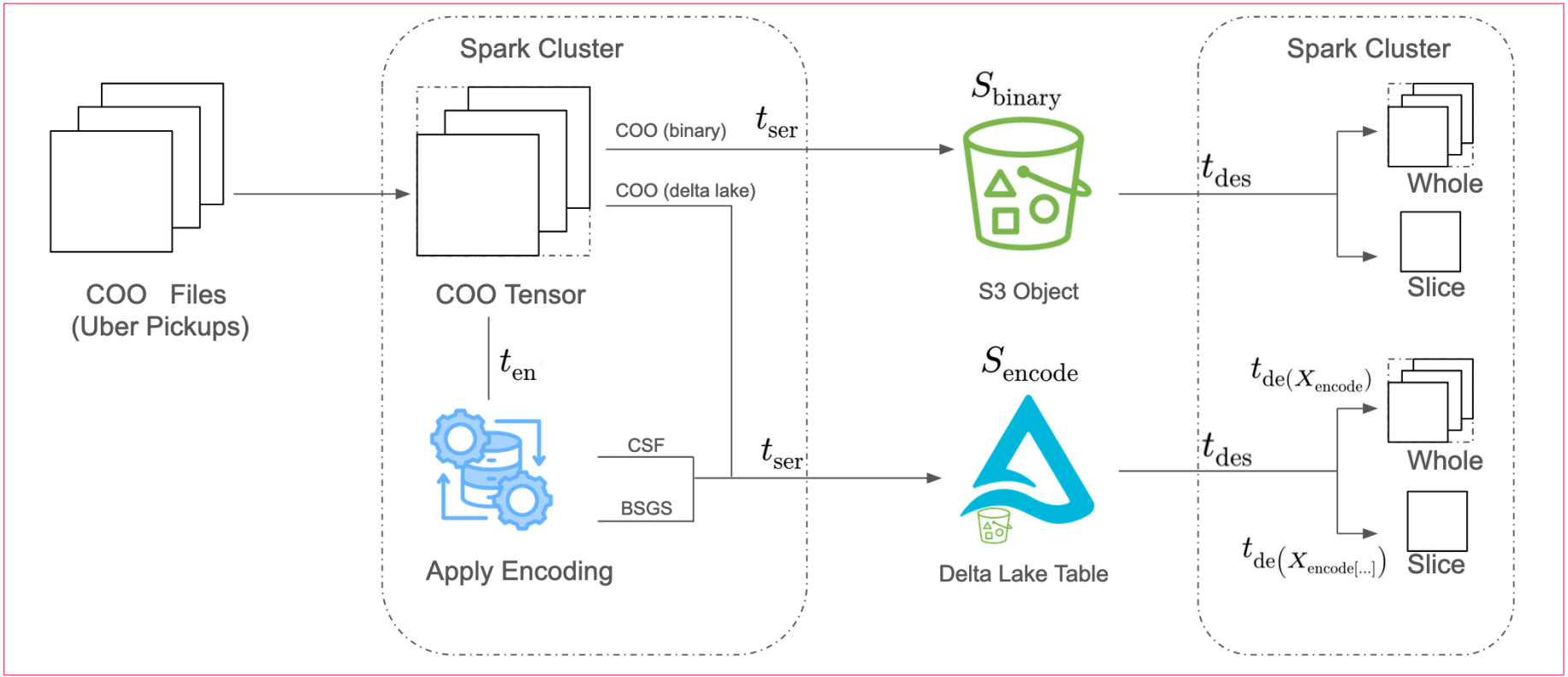
[[0,0,0],
[0,1,0],
[0,0,0],
[1,2,3]]

04.
EXPERIMENTS

Experiments - General Tensors



Experiments - Sparse Tensors



05.

RESULTS

Results - General Tensors

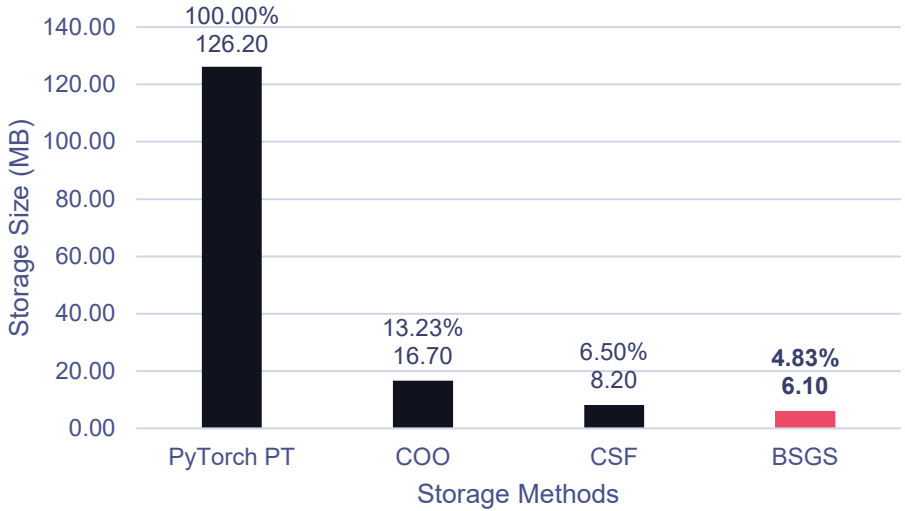
Slice reading is efficient, **reducing time by 90.04%** by fetching only relevant chunks.

	Storage Size (GB)	Write Tensor (s)	Read Tensor (s)	Read Slice (s)
Binary	14.6	135.69	379.51	494.33
FTSF	13.3	251.77	474.51	49.24
Δ	-8.90%	85.52%	25.02%	-90.04%

Results - Sparse Tensors

Evaluation – Storage Size

All the sparse tensor methods save the storage space compared to the PT file format with compression rates less than 13.23%.

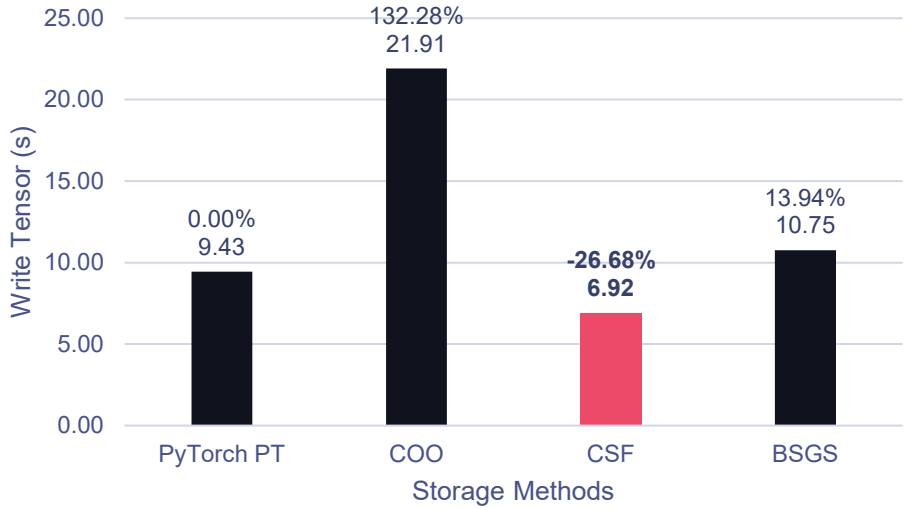


Best Performance

Results - Sparse Tensors

Evaluation – Tensor Writing

CSF and BSGS achieve comparable performance, with CSF being the most efficient, taking 26.68% less time compared to PT.

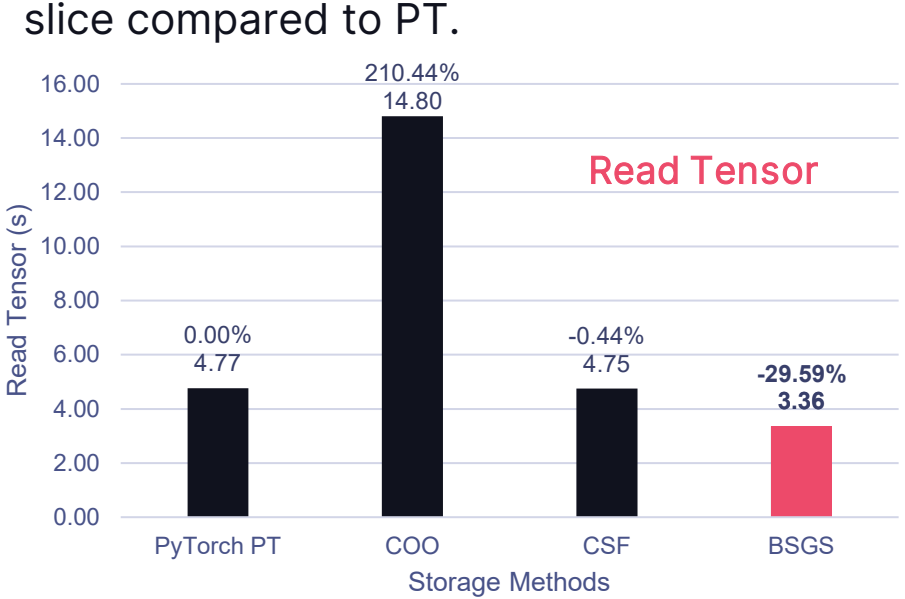


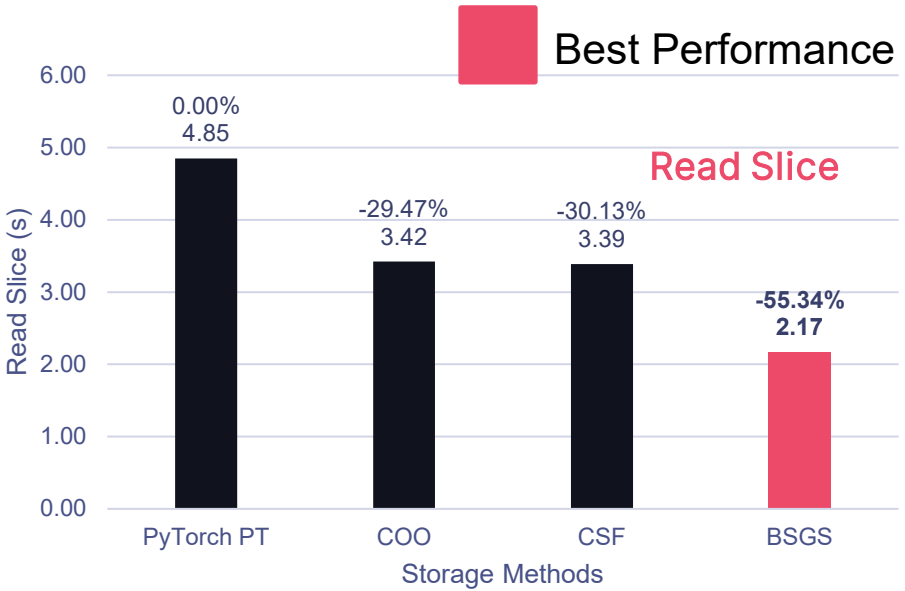
Best Performance

Results - Sparse Tensors

Evaluation – Tensor Reading

CSF and BSGS achieve comparable performance, with BSGS being the most efficient, taking 29.59% less time to read entire tensor and 55.34% less time to read tensor slice compared to PT.





06.

CONCLUSION

Conclusion

- Cloud-native tensor storage solution
- Customized methods for general and sparse tensors
- Remarkable space savings especially for sparse tensors
- Competitive performance for tensor writing, reading, and slicing operations

THANK YOU

Contact us:

liu.liao@northeastern.edu
bao.zhiw@northeastern.edu
wu.zhiyu@northeastern.edu

View our paper: