PUSHtap: PIM-based In-Memory HTAP with Unified Data Storage Format

Yilong Zhao, Mingyu Gao, Huanchen Zhang, Fangxin Liu, Gongye Chen, He Xian, Haibing Guan, and Li Jiang

Shanghai Jiao Tong University, Shanghai Qi Zhi Institute, Tsinghua University

ASPLOS' 2025

March 24, 2026

Outline

- **Background: HTAP Database**
  2 workloads: **OLTP** (**row-wise** transactions) and **OLAP** (**column-wise** analytics).
- **Problem：** **Row-store** vs. **column-store** trade-off -- no format fits both
- **Insight:** Exploit access divergence -- **CPU interleaved** & **PIM contiguous** access
  → 1 data format serving 2 workloads: **PIM-enabled unified storage for HTAP**.
- **Challenges → Solutions：**

1 Alignment of Heterogeneous Data Types	Unified Data Format (Compact Aligned Format + Block-circulant Placement)
2 PIM Parallelism	
3 Fragmentation in Version Control	Format-aware MVCC & Defragmentation
4 CPU-PIM Contention	Architecture support: Two-phase Execution

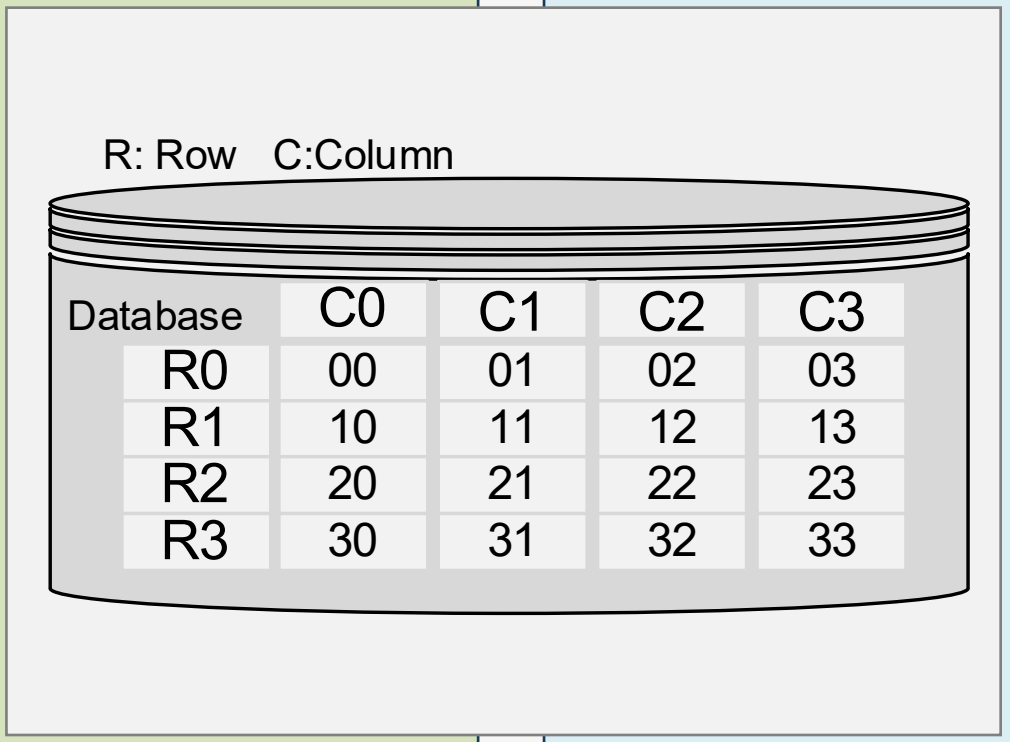
- **Evaluation：** Zsim+Ramulator2 simulation | 8-Channel DDR5 | 20GB CH Benchmark
- **Results：** 3.4× peak OLTP throughput | 4.4× OLAP throughput under peak OLTP (vs. MI baseline)

HTAP: The Convergence of Conflicting Workloads

Hybrid transaction/analytical processing (HTAP) database, 2 processing sets:

Online transaction processing (OLTP):

Online analytical processing (OLAP):

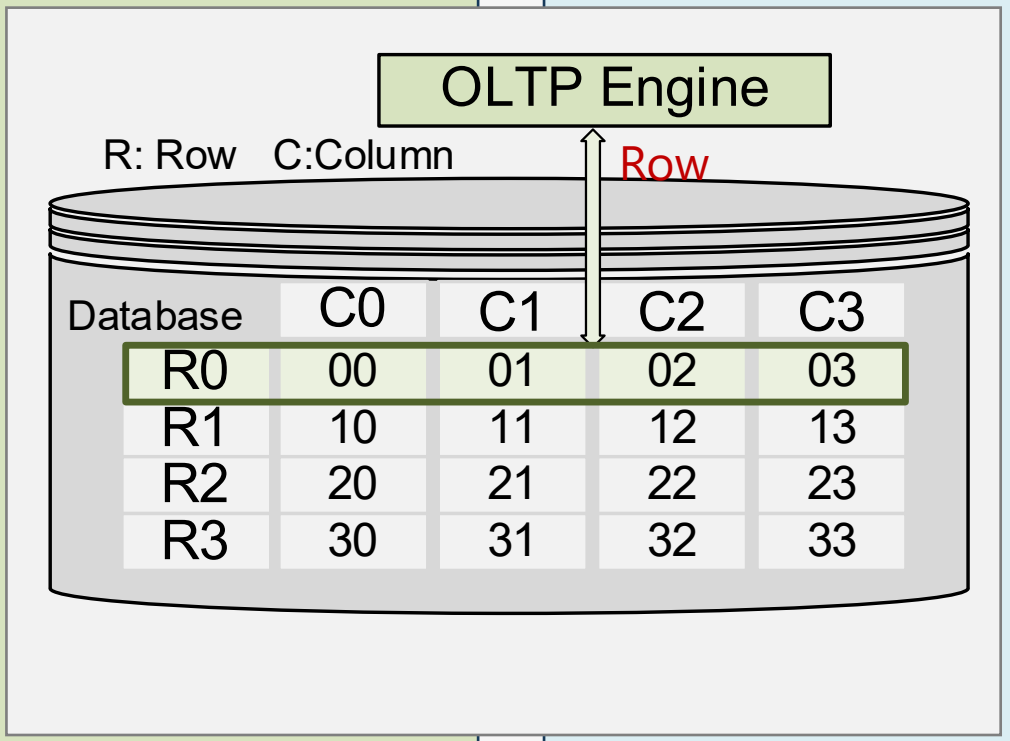


HTAP: The Convergence of Conflicting Workloads

Hybrid transaction/analytical processing (HTAP) database, 2 processing sets:

Online transaction processing (OLTP):

Access Pattern:
Row-wise, random
Bottleneck:
Computation-intensive



Ideal Format:
Row-store

Online analytical processing (OLAP):

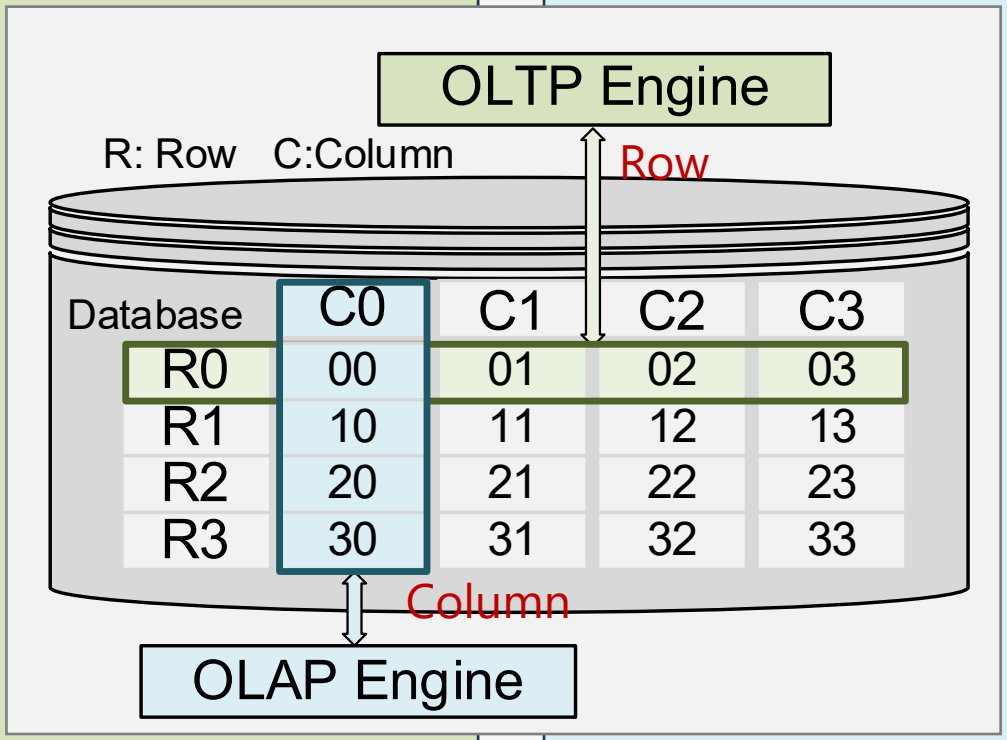
HTAP: The Convergence of Conflicting Workloads

Hybrid transaction/analytical processing (HTAP) database, 2 processing sets:

Online transaction processing (OLTP):

Access Pattern:
Row-wise, random
Bottleneck:
Computation-intensive

Ideal Format:
Row-store



Online analytical processing (OLAP):

Access Pattern:
Column-wise, sequential
Bottleneck:
Memory-intensive

Ideal Format:
Column-store

HTAP: The Convergence of Conflicting Workloads

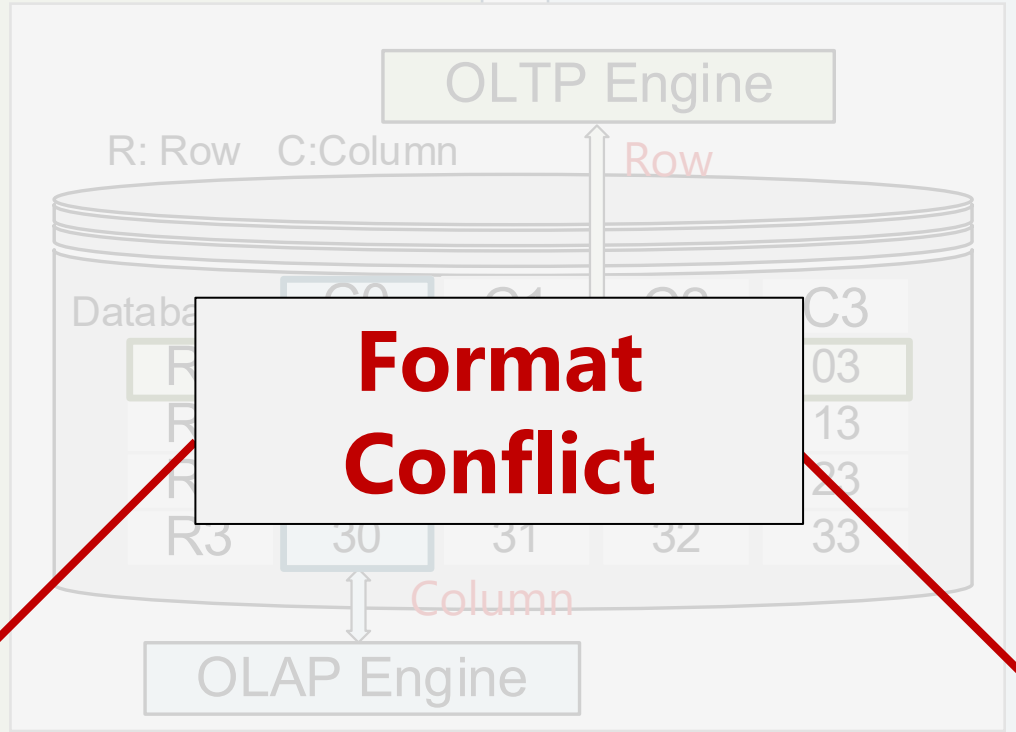
Hybrid transaction/analytical processing (HTAP) database, 2 processing sets:

Online transaction processing (OLTP):

Access Pattern:
Row-wise, random
Bottleneck:
Computation-intensive

Online analytical processing (OLAP):

Access Pattern:
Column-wise, sequential
Bottleneck:
Memory-intensive



Format Conflict

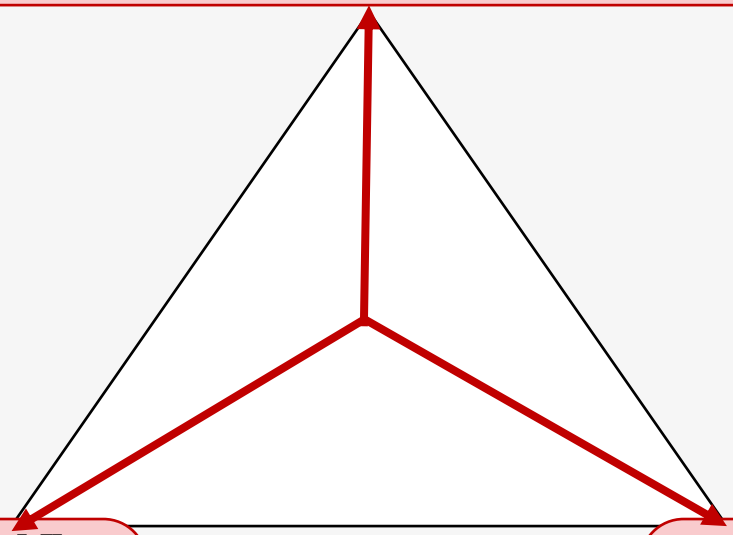
Ideal Format: Row-store

Ideal Format: Column-store

The "Impossible Trinity" of HTAP Design Goals

Three goals of an ideal HTAP:[SIGMOD'17]

Workload-specific optimizations (*W*):
Optimized performance for both workloads



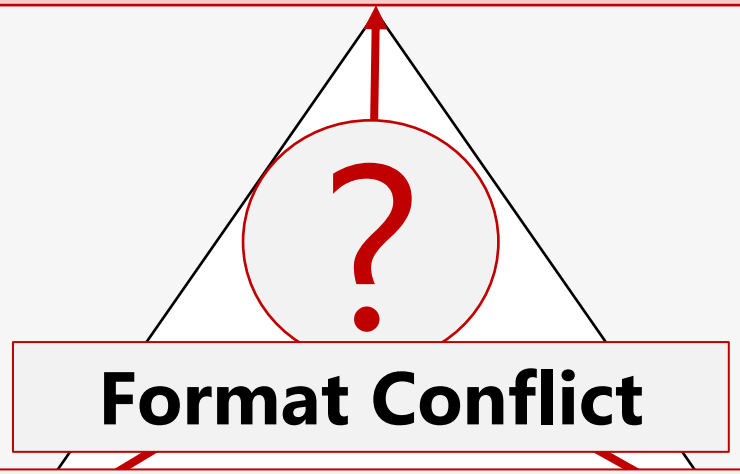
Performance isolation (*I*):
Zero performance interference

Data freshness (*F*):
OLAP need up-to-date data

The "Impossible Trinity" of HTAP Design Goals

Three goals of an ideal HTAP:[SIGMOD'17]

Workload-specific optimizations (*W*):
Optimized performance for both workloads



Current solutions sacrifice one or more goals to achieve the others

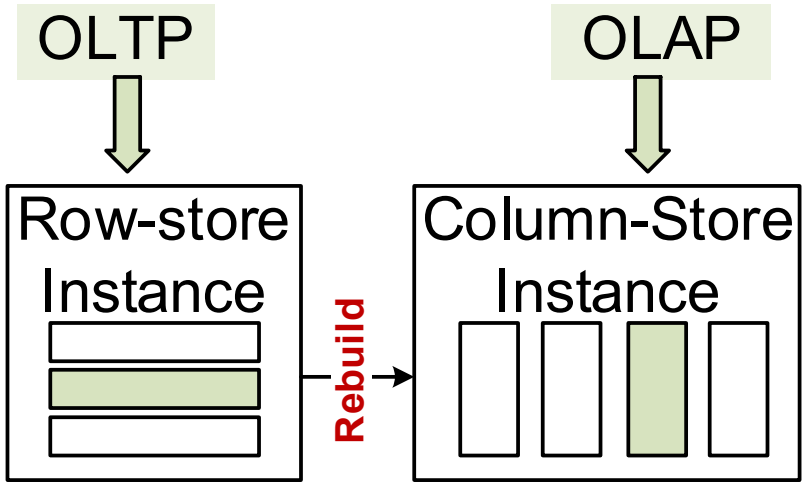
Performance isolation (*I*):
Zero performance interference

Data freshness (*F*):
OLAP need up-to-date data

Current solution 1 - Multi-instance, Multi-format

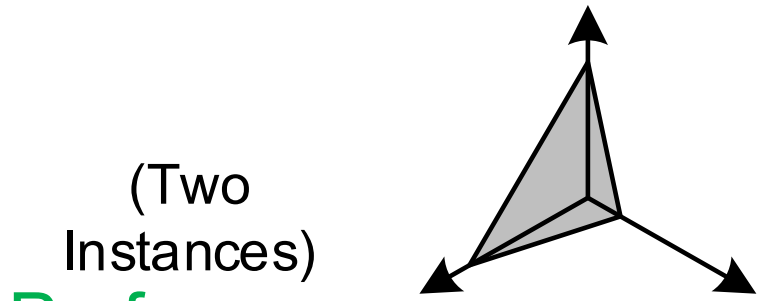
Data Format:

A row-store instance +
a column-store instance



Can only periodically rebuild due to large overhead

Goals:



Workload-specific Optimization (High)

(Efficient Format)

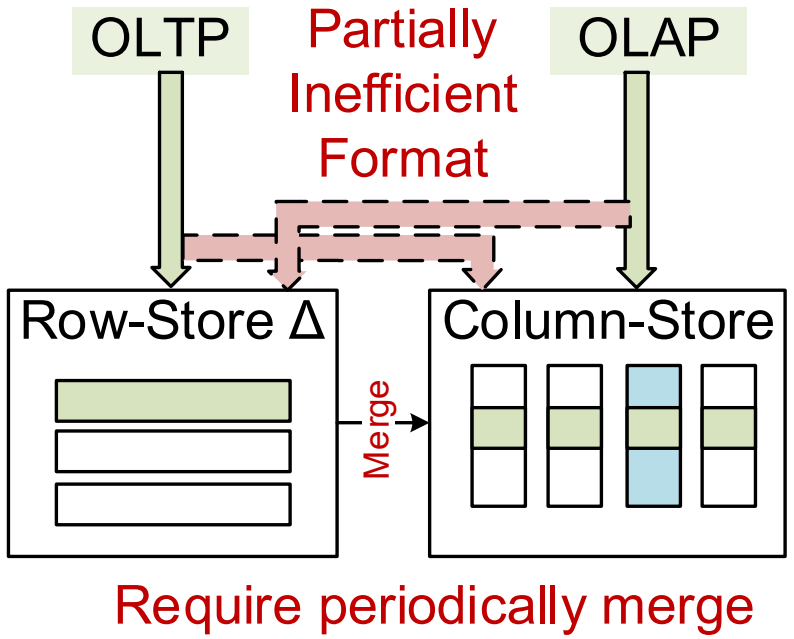
(Two Instances)
Performance Isolation (High)

(Periodically rebuild)
Data Freshness (Low)

Current solution 2 - Single-instance, Mixed-format

Data Format:

Primary column-store + row-store Δ

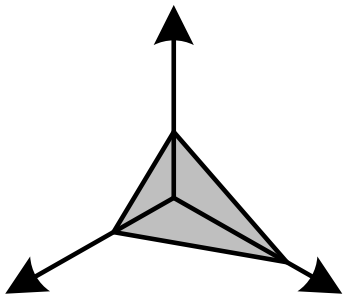


Require periodically merge

Goals:

Workload-specific Optimization (Medium)

(Partially Inefficient Format)



(Periodically Merge)
Performance Isolation (Medium)

(Single Instance)
Data Freshness (High)

Summary: Limitations of Current HTAP Solutions

Format	Performance Isolation	Data Freshness	Workload-Specific Optimization
Multi-instance	H ●	L ○	H ●
Mixed-format	M ◐	H ●	M ◐

Summary: Data format conflicts prevent CPU-only systems from simultaneously achieving all three goals

Summary: Limitations of Current HTAP Solutions

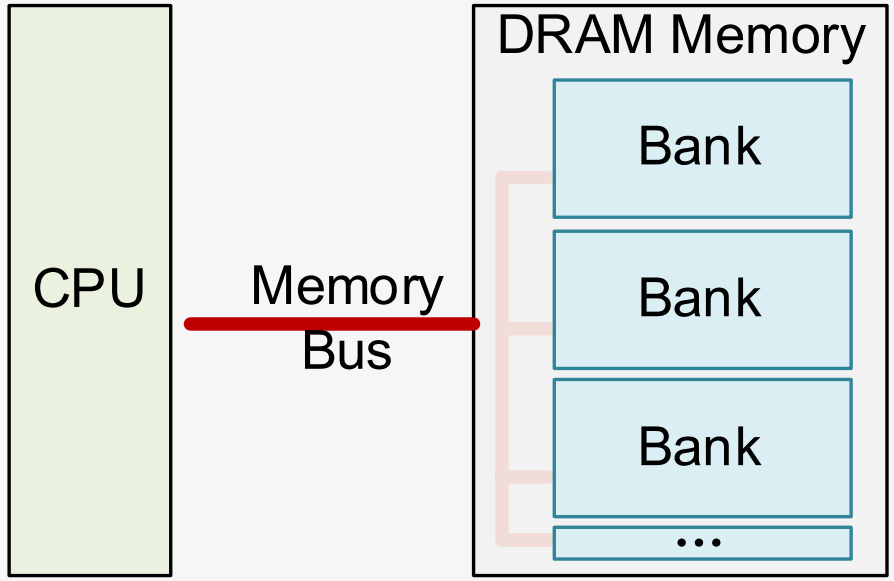
Format	Performance Isolation	Data Freshness	Workload-Specific Optimization
Multi-instance	H	L	H
Mixed-format	M	H	M
Ideal PUSHtap?	**H**	**H**	**H**

Resolve the data format conflict?

Summary: Data format conflicts prevent CPU-only systems from simultaneously achieving all three goals

Background: Processing-in-memory (PIM)

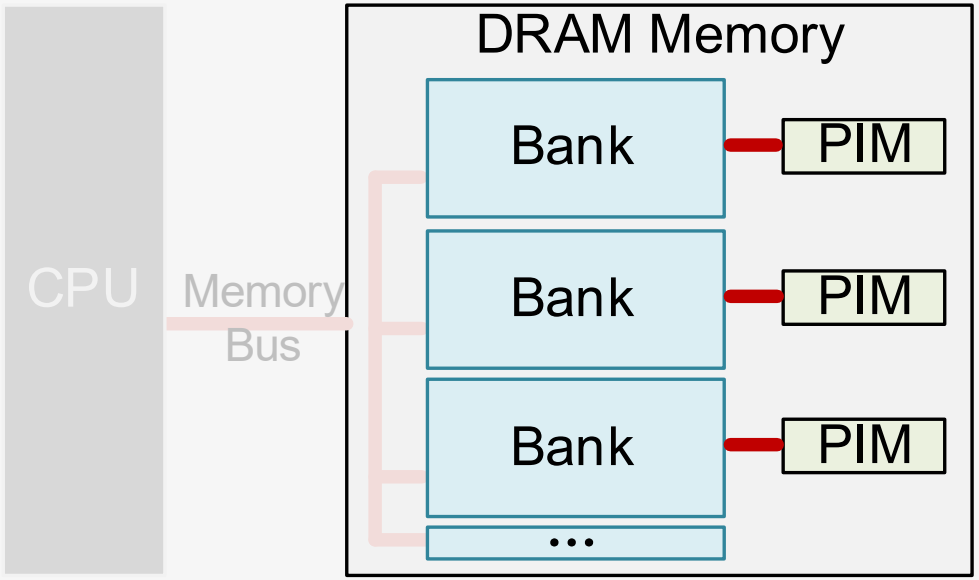
Conventional Von Neumann Architecture



Through memory bus
-- suffer from **limited bandwidth**

Processing-in-memory (PIM):

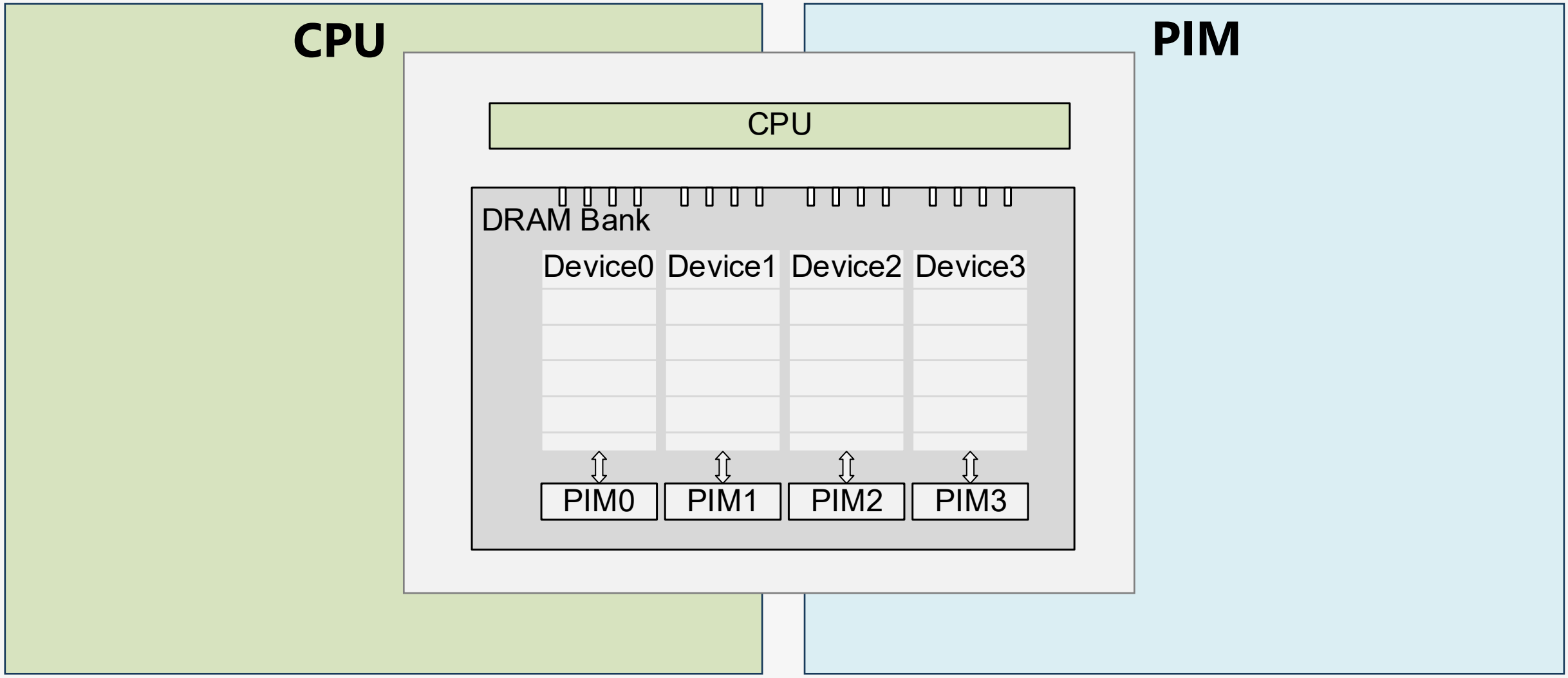
Integrate Computing Units in banks/devices



Parallel access of thousands of PIM -
-- provide **massive bandwidth**

Fit for memory-intensive OLAP

Access Divergence in CPU-PIM System



Access Divergence in CPU-PIM System

CPU

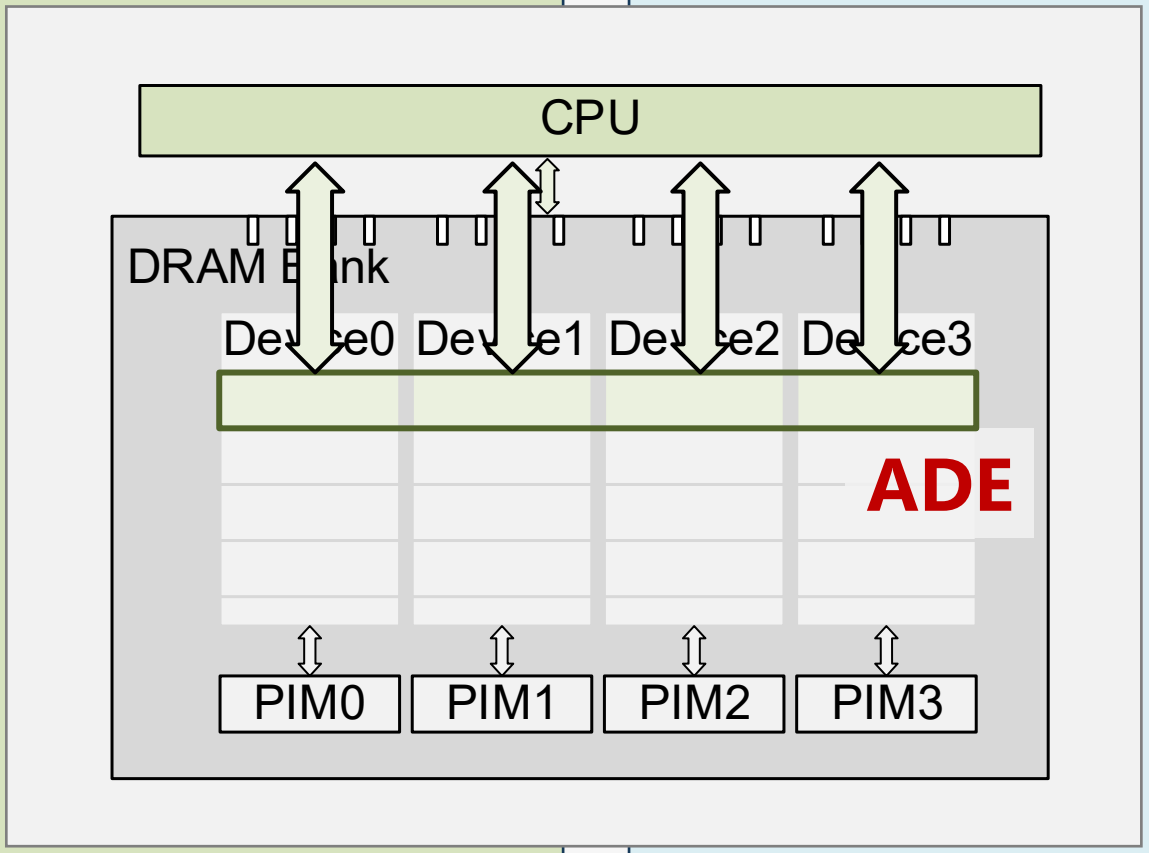
PIM

Interleaved Access

for high bandwidth

→ **Across devices**

(ADE) Access



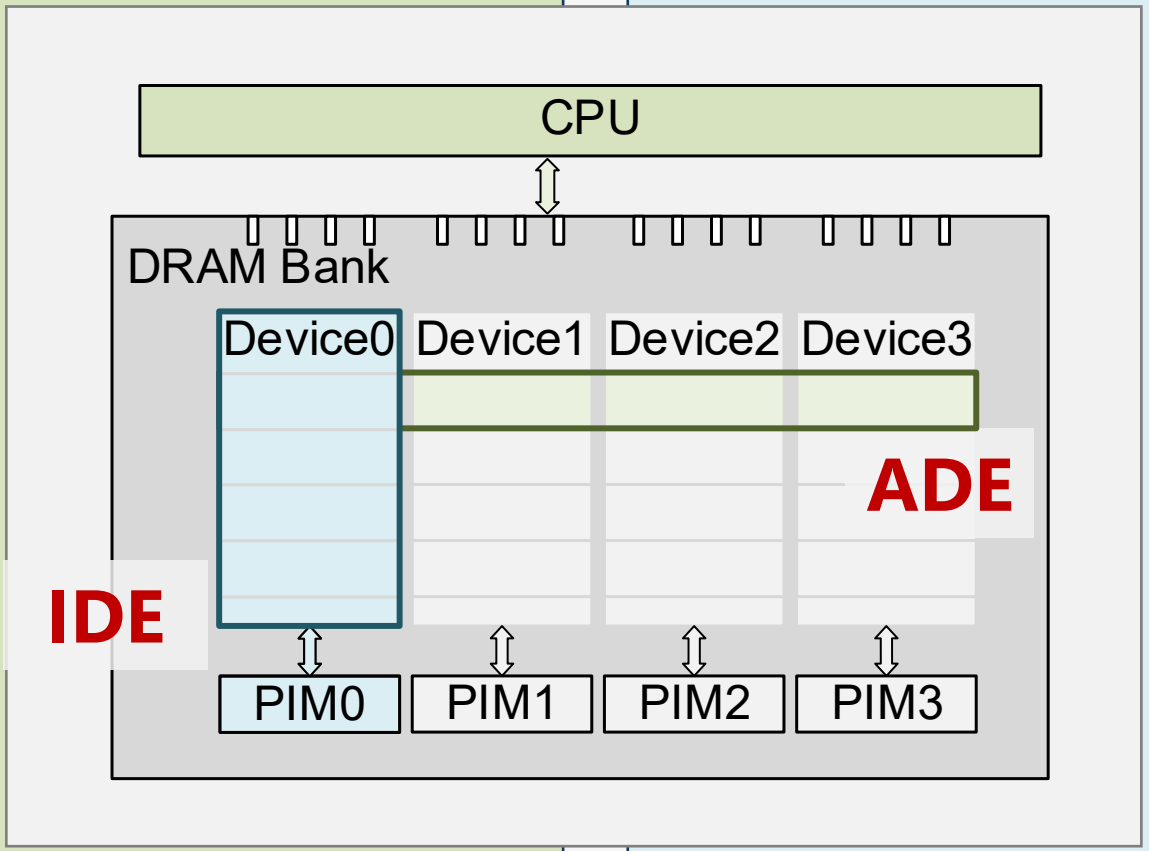
Access Divergence in CPU-PIM System

CPU

Interleaved Access for high bandwidth

→ **Across devices**

(ADE) Access



PIM

Contiguous access because PIM only visible to local device

→ **Inside device**

(IDE) Access

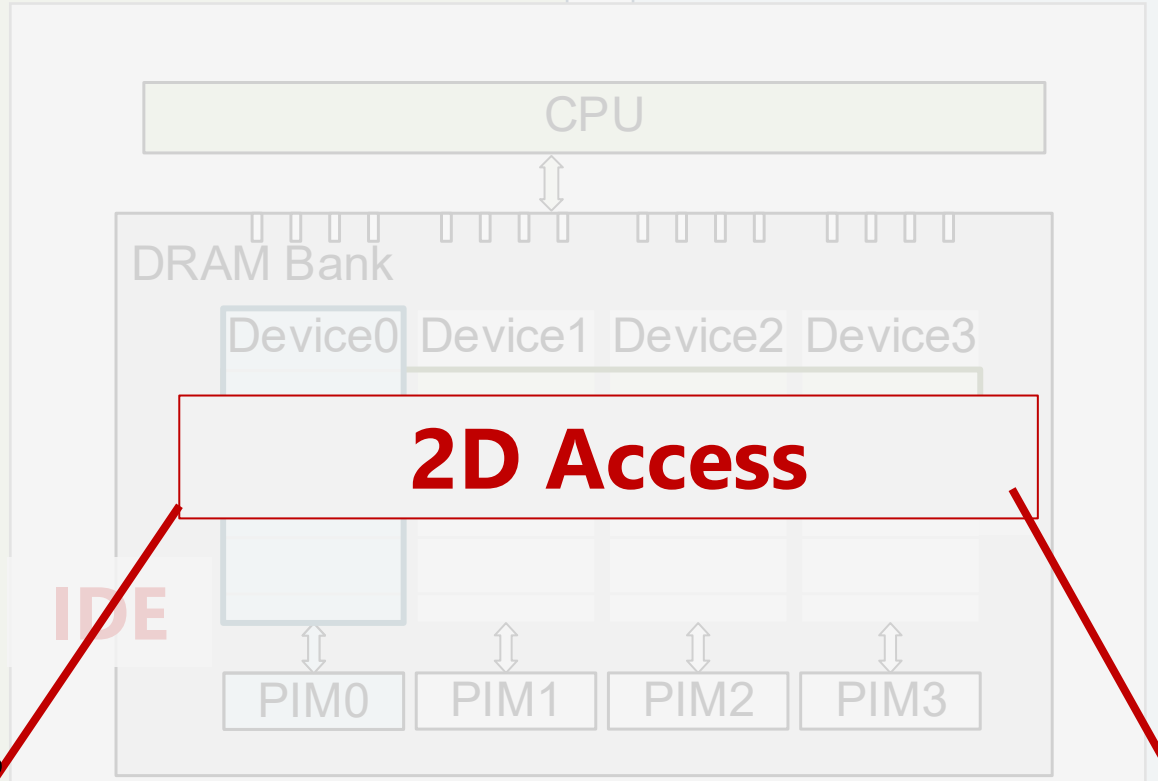
Access Divergence in CPU-PIM System

CPU

Interleaved Access for high bandwidth

→ Across devices

(ADE) Access



2D Access

PIM

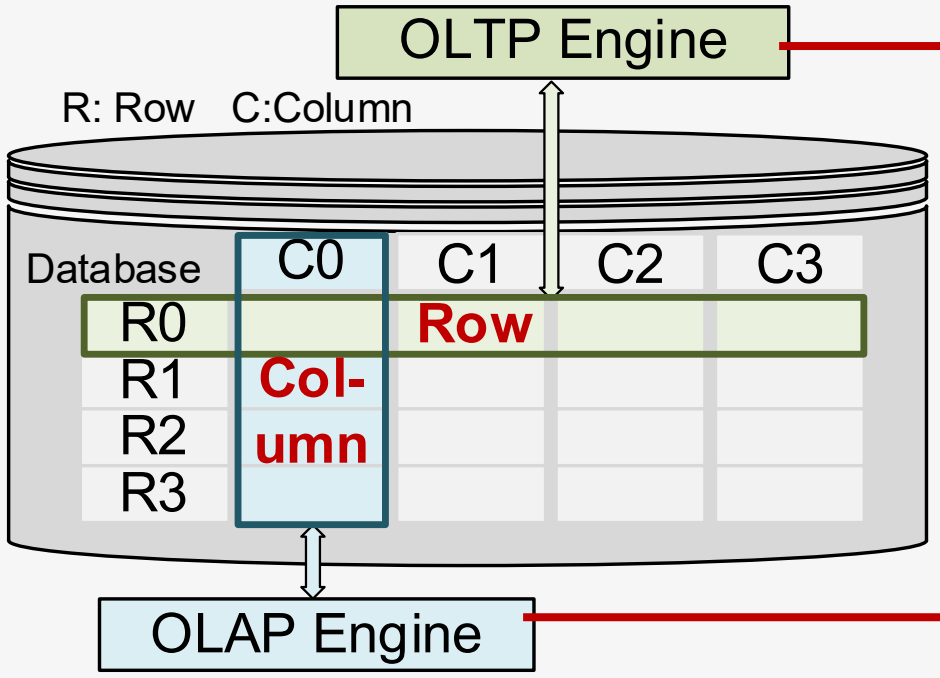
Contiguous access because PIM only visible to local device

→ Inside device

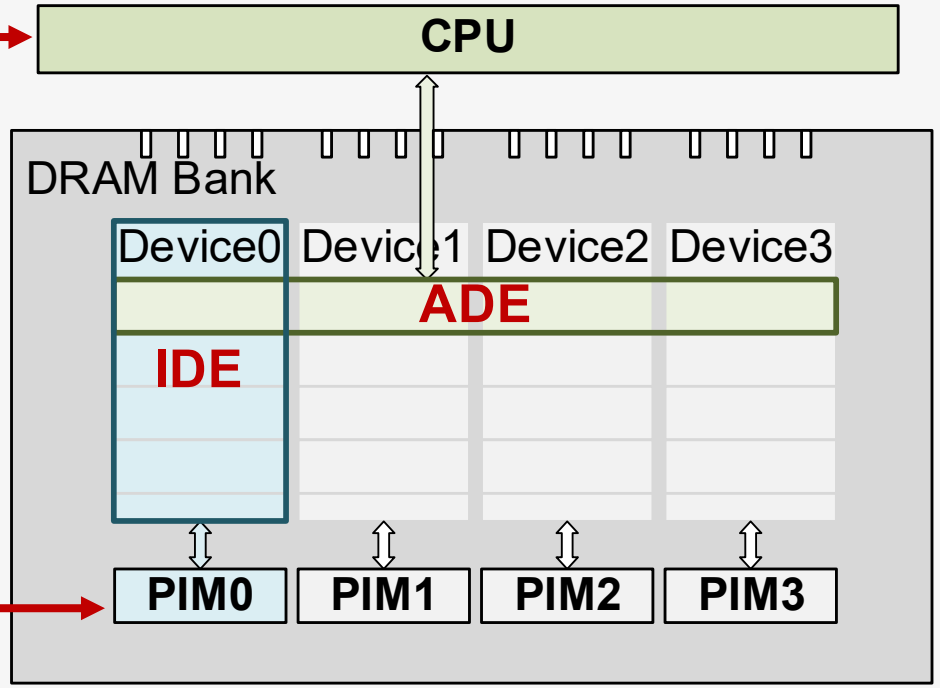
(IDE) Access

Key Insight: Format Conflict vs. Access Divergence

HTAP					PIM
OLTP	Row-wise	→	Across Device (ADE)	CPU	
OLAP	Column-wise	→	Inside Device (IDE)	PIM	



Format Conflicts



2D Access

Objective: Single Instance, Unified Format

-- Satisfies both row & column operations, simultaneously achieve 3 goals

- Workload-specific Optimization:	*Unified format -> both row & column access*
- Performance Isolation:	*No consistency (merge/rebuild) overhead*
- Data Freshness:	*Processed on the same instance*

The PUSHtap Vision: Hardware-Software Co-alignment

Objective: Single Instance, Unified Format

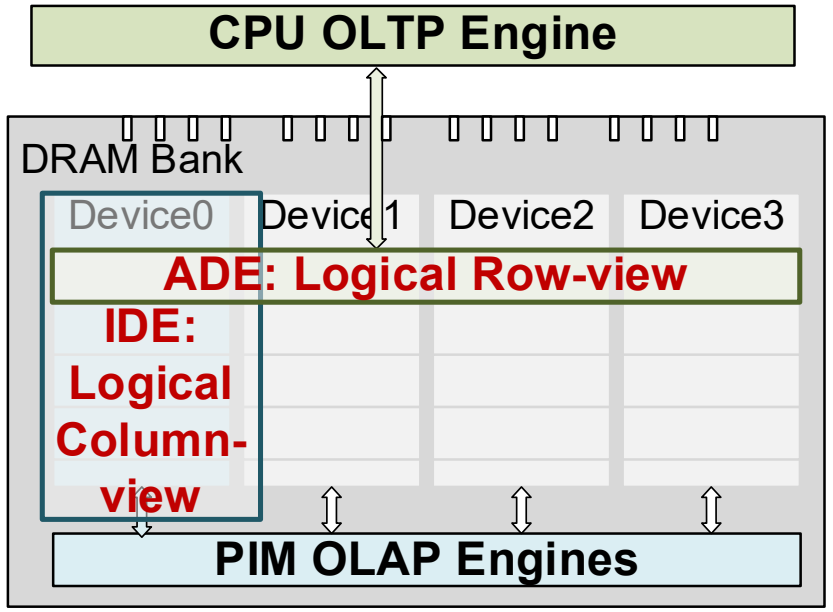
-- Satisfies both row & column operations, simultaneously achieve 3 goals

- *Workload-specific Optimization:* — *Unified format -> both row & column access*
- *Performance Isolation:* — *No consistency (merge/rebuild) overhead*
- *Data Freshness:* — *Processed on the same instance*

Core Approach: **Unified Data Format**

Logical Rows: **Aligned** to ADE

Logical Columns: **Aligned** to IDE



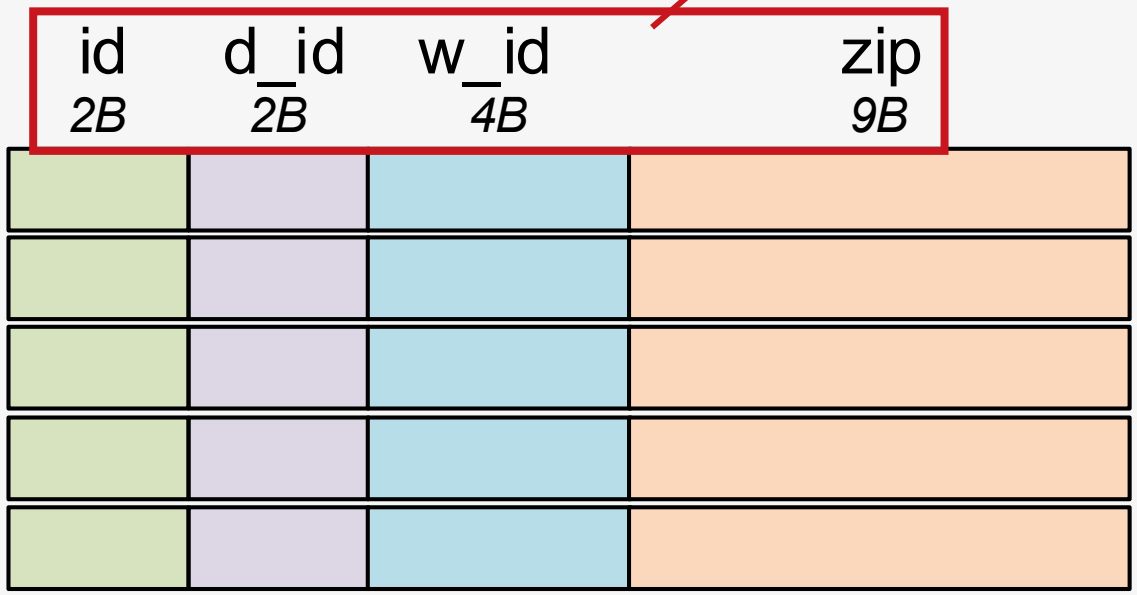
Challenge 1: Data Format Challenge (1)

Our goal requires to align data in the IDE & ADE direction

Format challenge:

Software:
Diverse column widths

2~9 Bytes

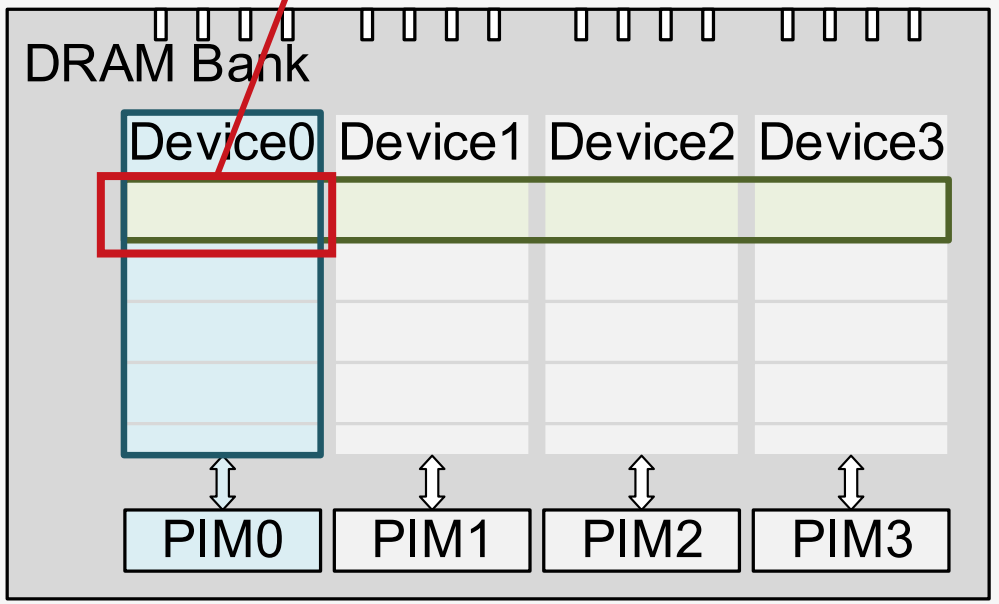


An Example of Table

vs.

Hardware:
Fixed interleaving granularity

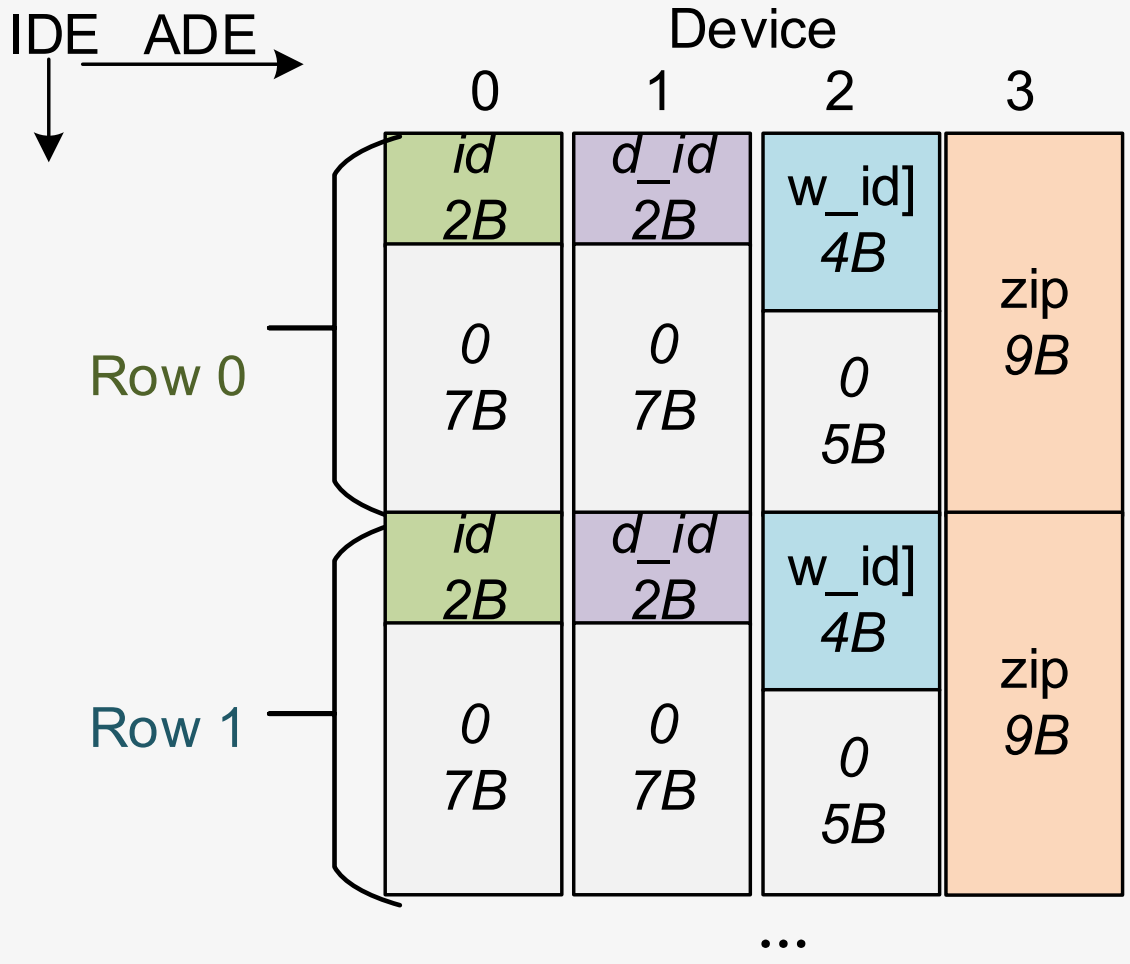
Typically 4 Bytes



A DRAM Rank

Challenge 1: Data Format Challenge (2)

One Naïve Solution: padding 0s



Aligned on both dimensions √

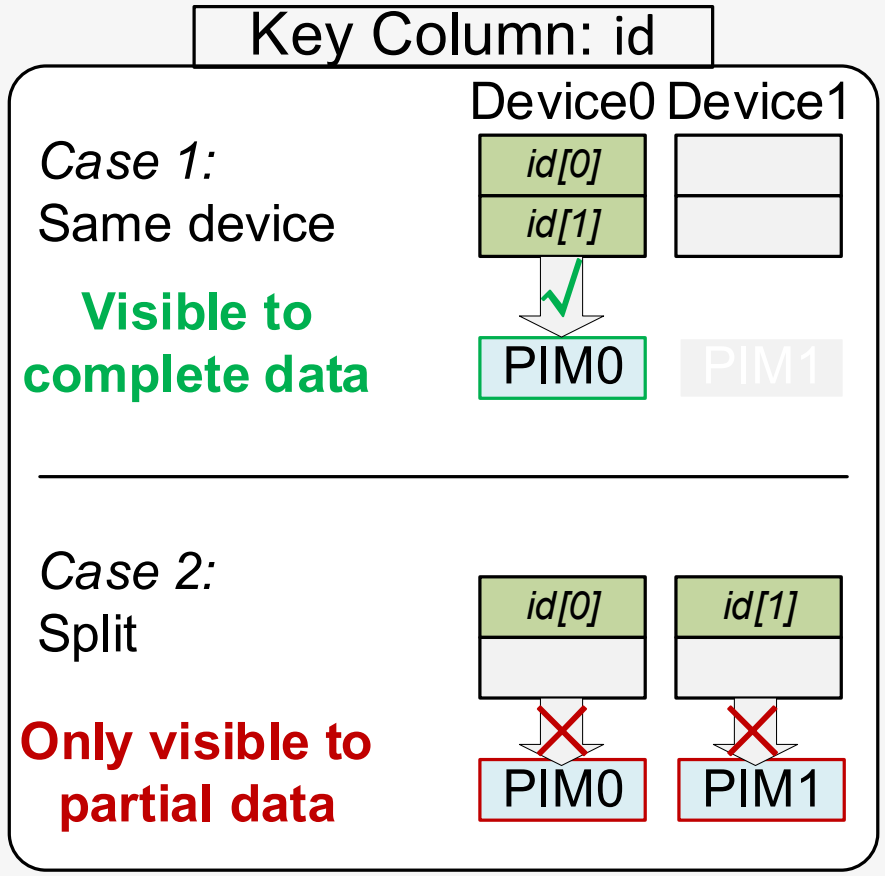
Too many 0s (51.2%)×

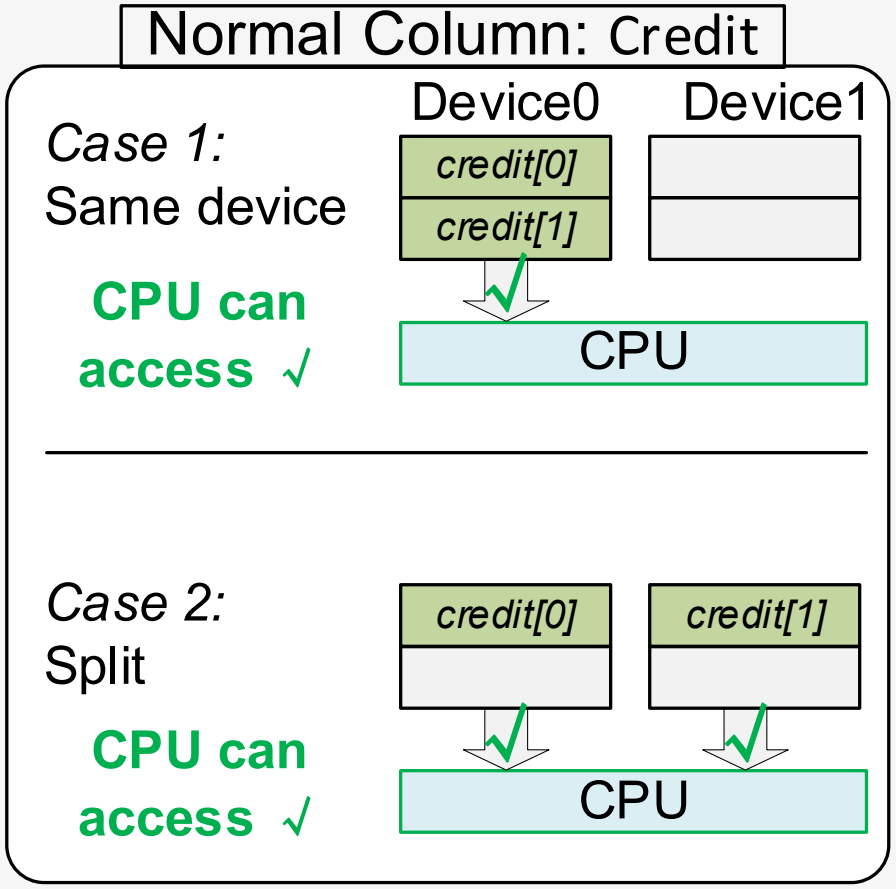
How to reduce the 0s?

Solution 1: Compact Aligned Format (1)

Row Categorization:

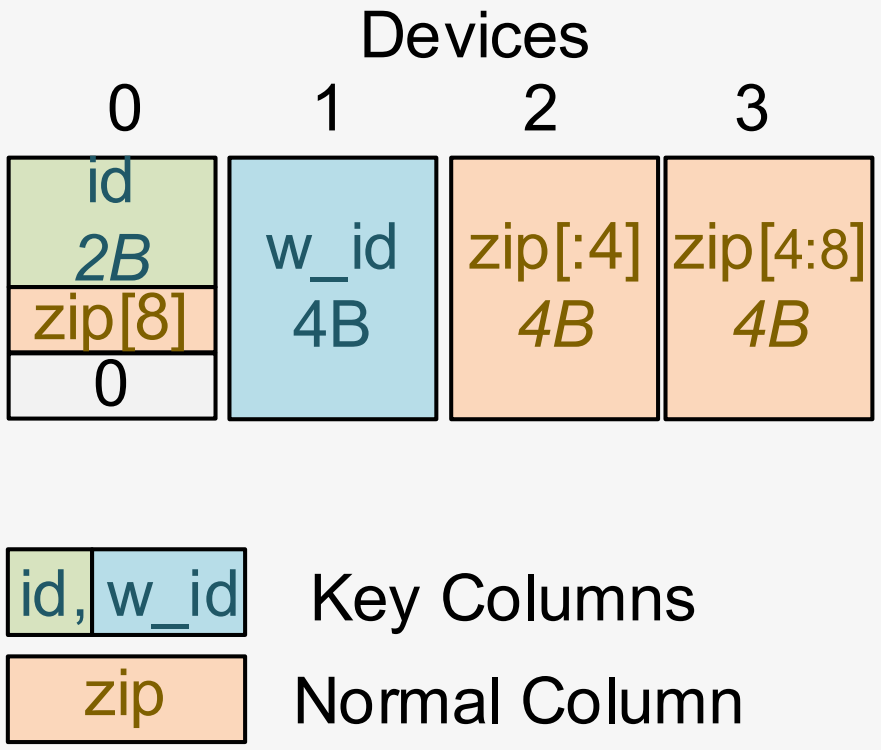
- ***Key columns***: Frequently scanned by OLAP → Should be on the same device
- ***Normal columns***: Never scanned by OLAP → Can be split to multiple devices





Solution 1: Compact Aligned Format (2)

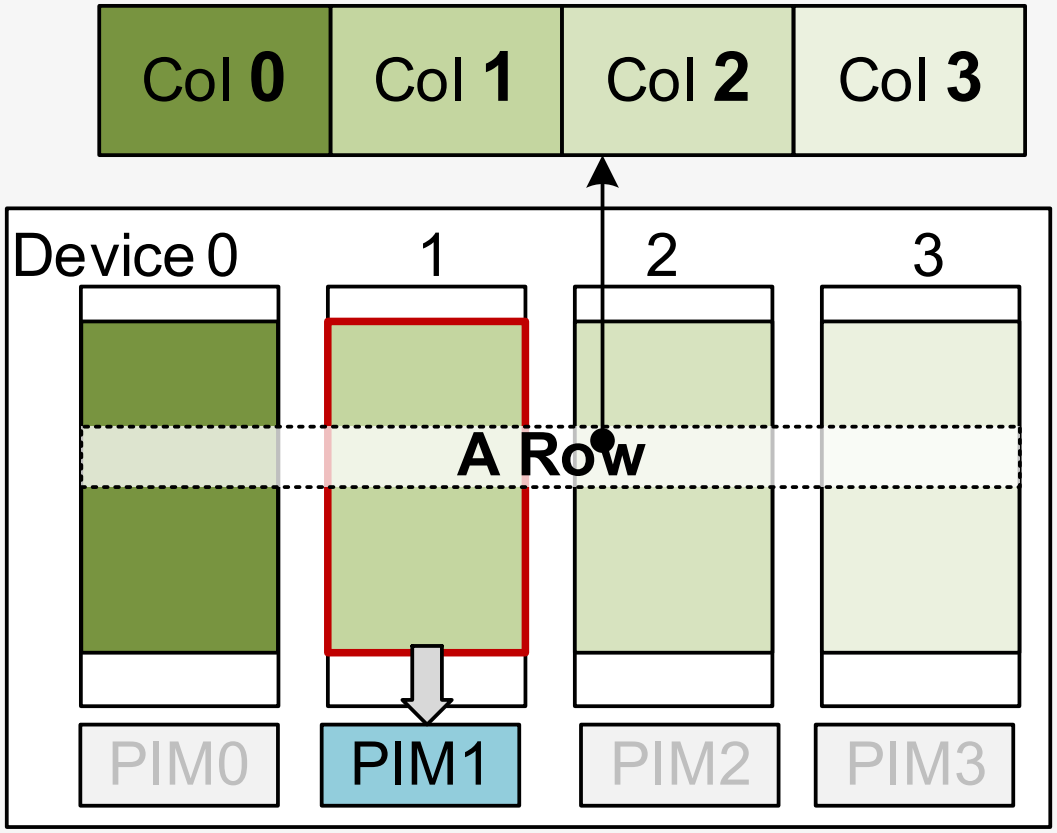
Compact Aligned Format:



Use *normal columns* (e.g., zip) for padding instead of "0"

→ **Fewer "0"s** (6.25%)

If we only consider the alignment:

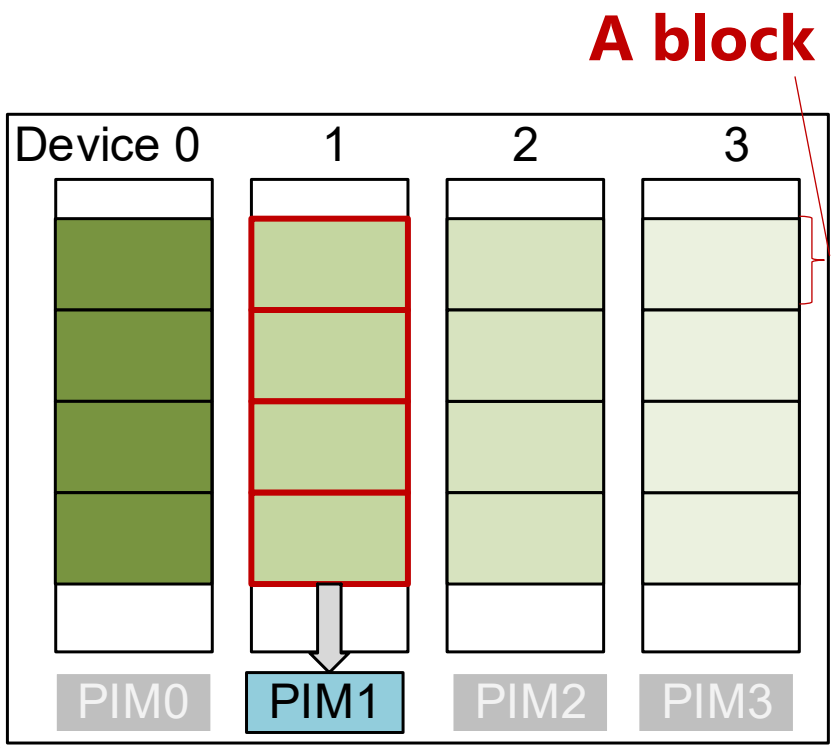


PIM Underutilization Challenge:
Scanning a column activates **only 1 PIM**.

Solution 2: Block-circulant Data Placement

Distribute columns across PIM units while preserving memory alignment.

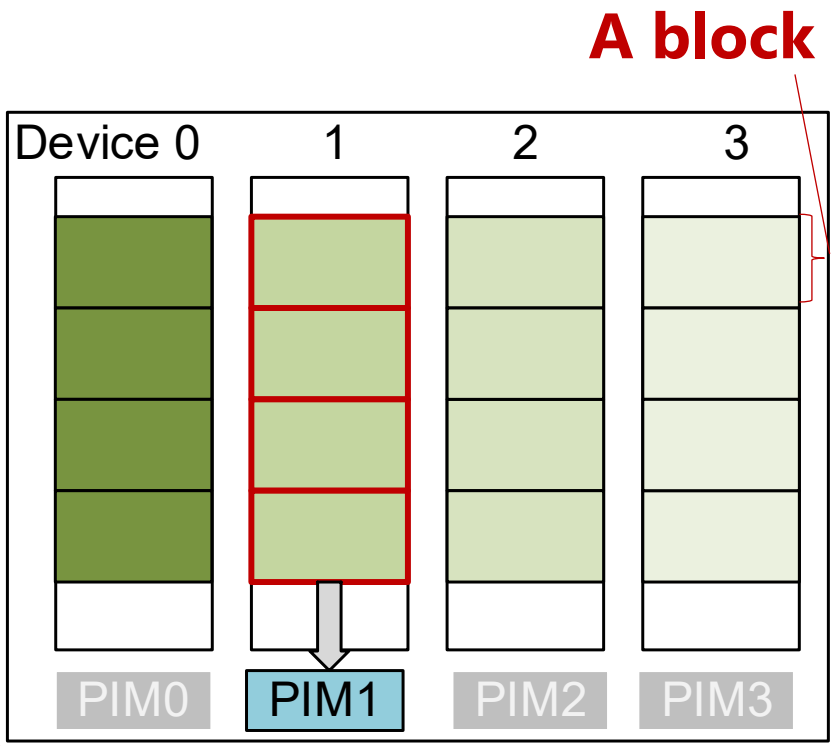
1 Divide into column blocks.



Solution 2: Block-circulant Data Placement

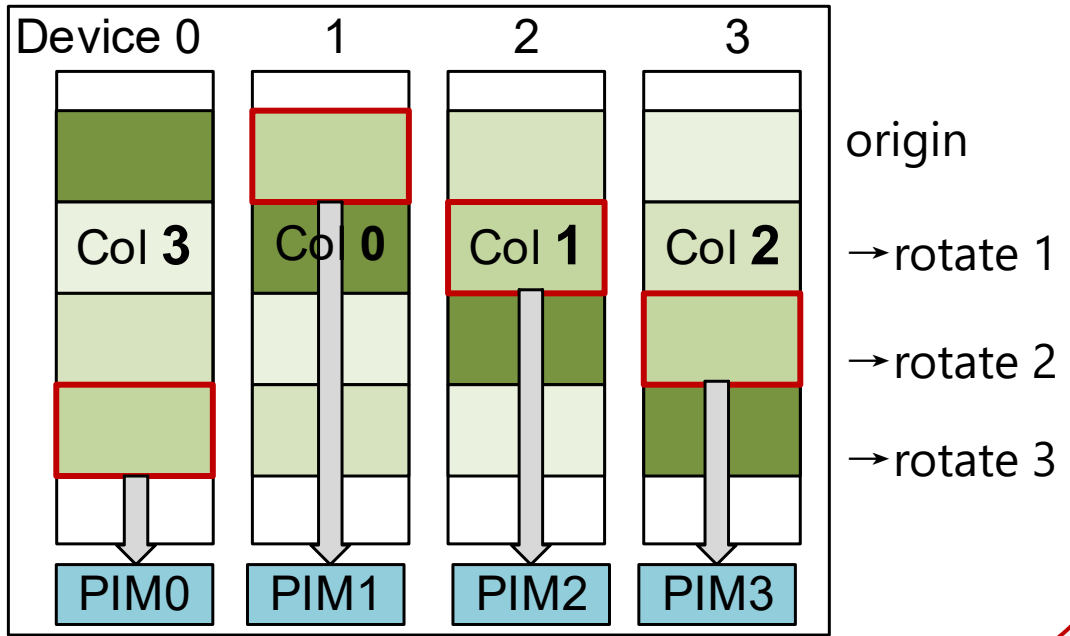
Distribute columns across PIM units while preserving memory alignment.

1 Divide into column blocks.



2 Rotate rows in each block

- Still aligned
- **Ful PIM parallelism**



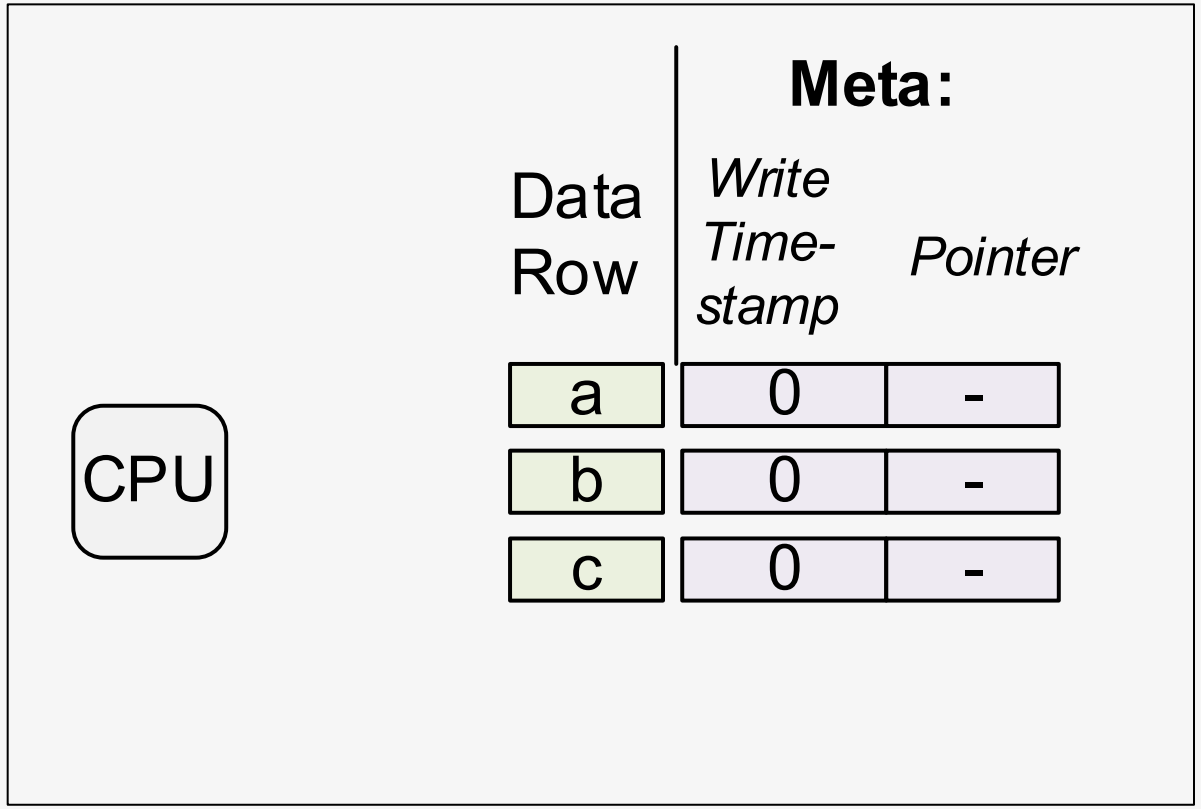
Background for Challenge 3: MVCC and Version Chain

Concurrency control: Enables consistent snapshots without blocking writers
→ Essential for Single-instance database

Background for Challenge 3: MVCC and Version Chain

Concurrency control: Enables consistent snapshots without blocking writers
→ Essential for Single-instance database

Multi-version Concurrency Control (MVCC):

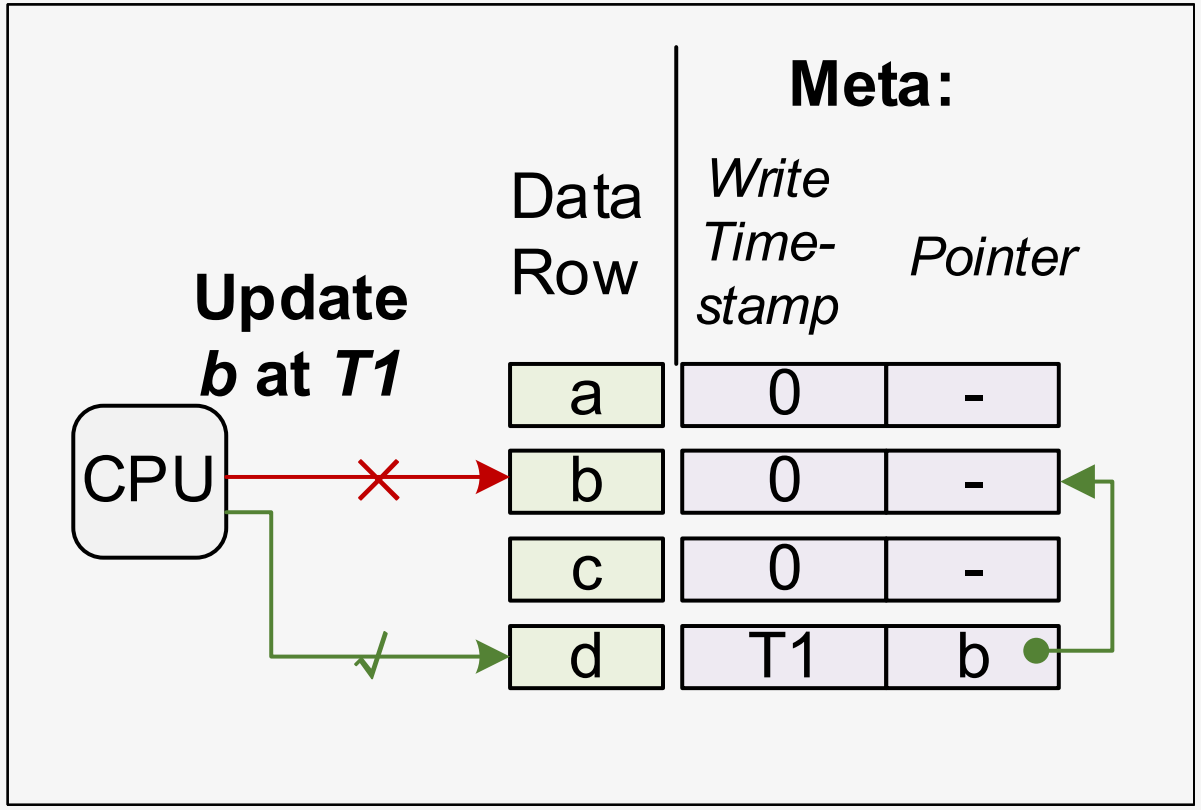


Each data row maintains a *meta*

Background for Challenge 3: MVCC and Version Chain

Concurrency control: Enables consistent snapshots without blocking writers
→ Essential for Single-instance database

Multi-version Concurrency Control (MVCC):



Each data row maintains a *meta*

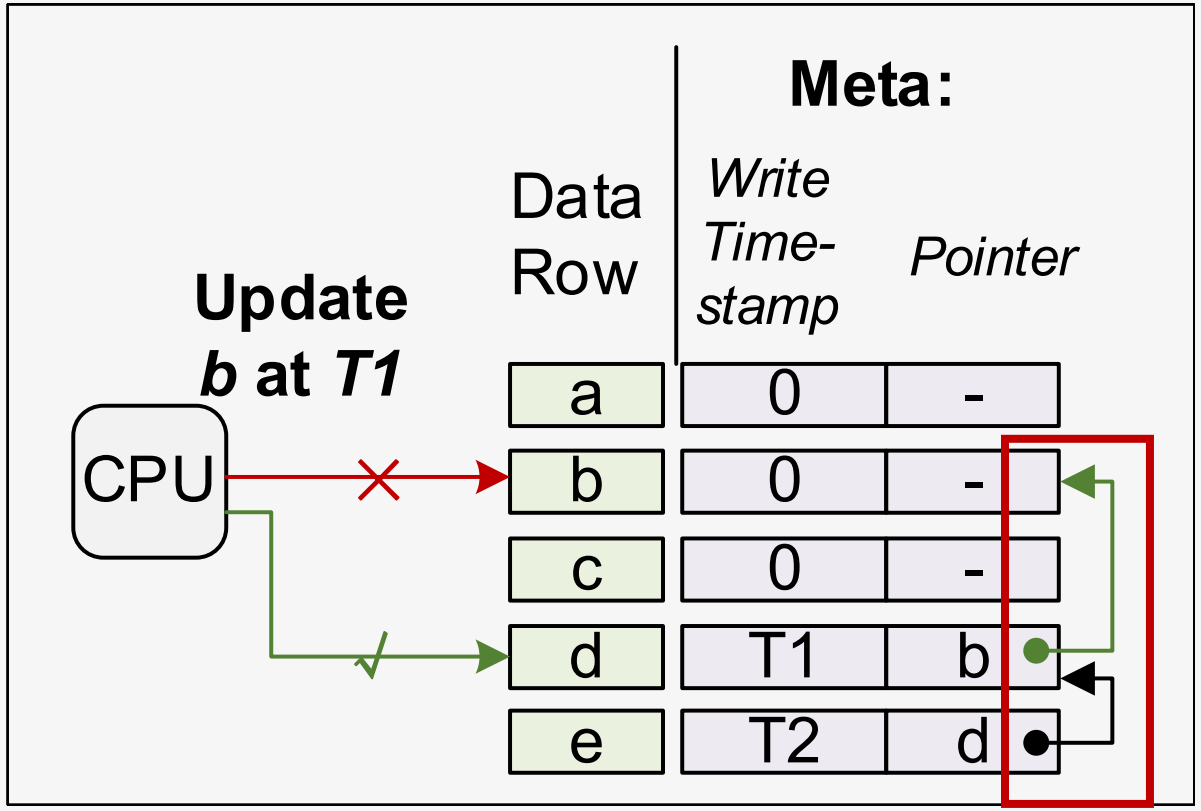
~~In-place update~~ → Append a new row

Pointer: point to the previous version

Background for Challenge 3: MVCC and Version Chain

Concurrency control: Enables consistent snapshots without blocking writers
→ Essential for Single-instance database

Multi-version Concurrency Control (MVCC):



Each data row maintains a *meta*

~~In-place update~~ → Append a new row

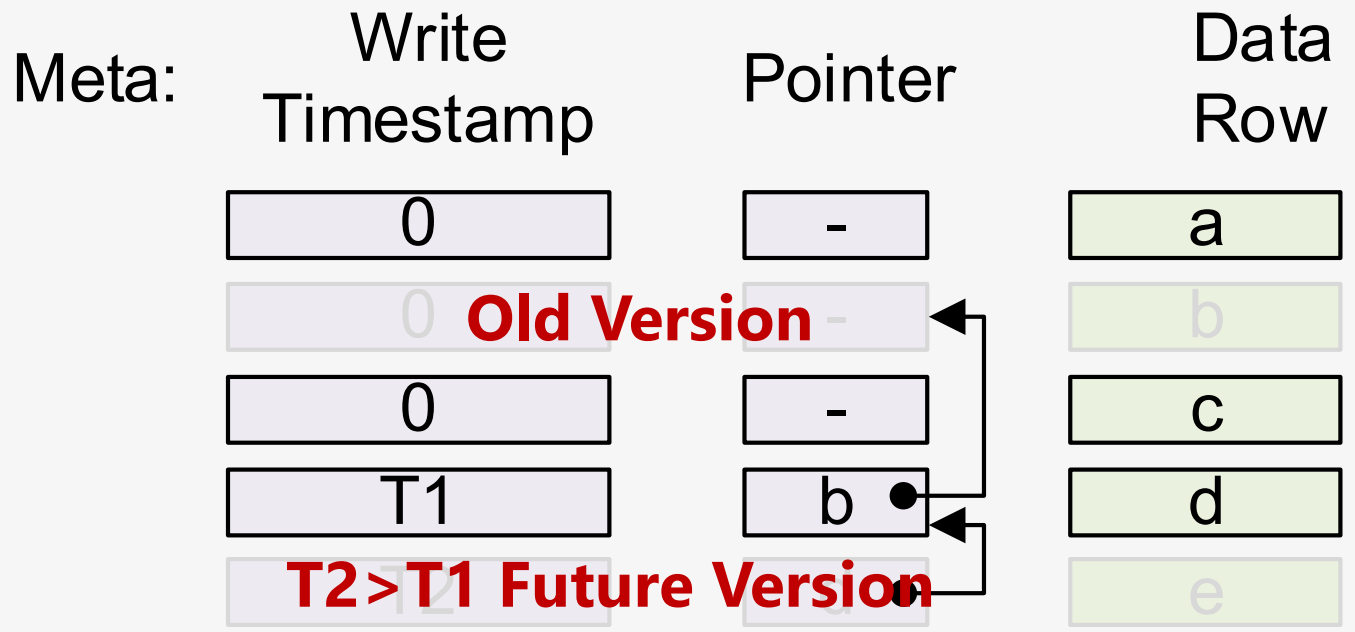
Pointer: point to the previous version

→Forms a ***version chain*** for each record (e-d-b)

Background for Challenge 3: Snapshot in MVCC

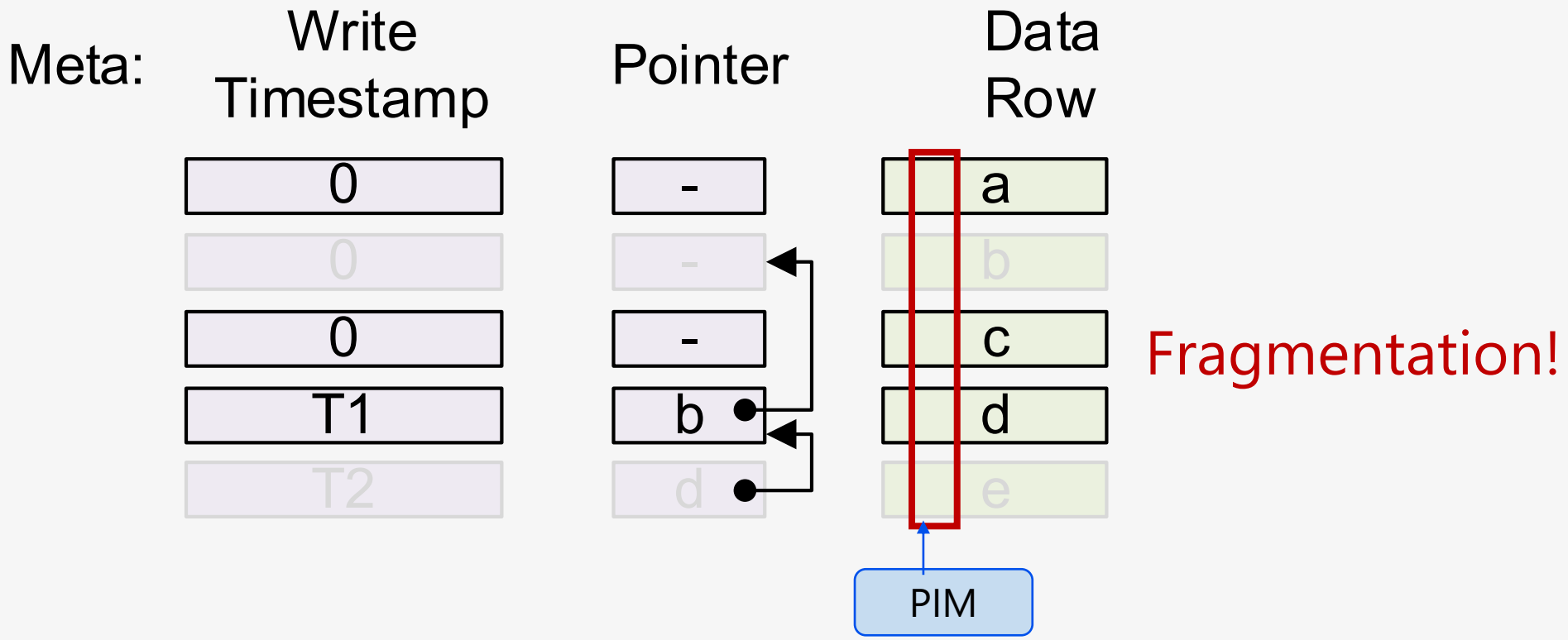
Snapshot: A consistent view at a time point

example: A snapshot at $T1 < t < T2$ -- row a,c,d



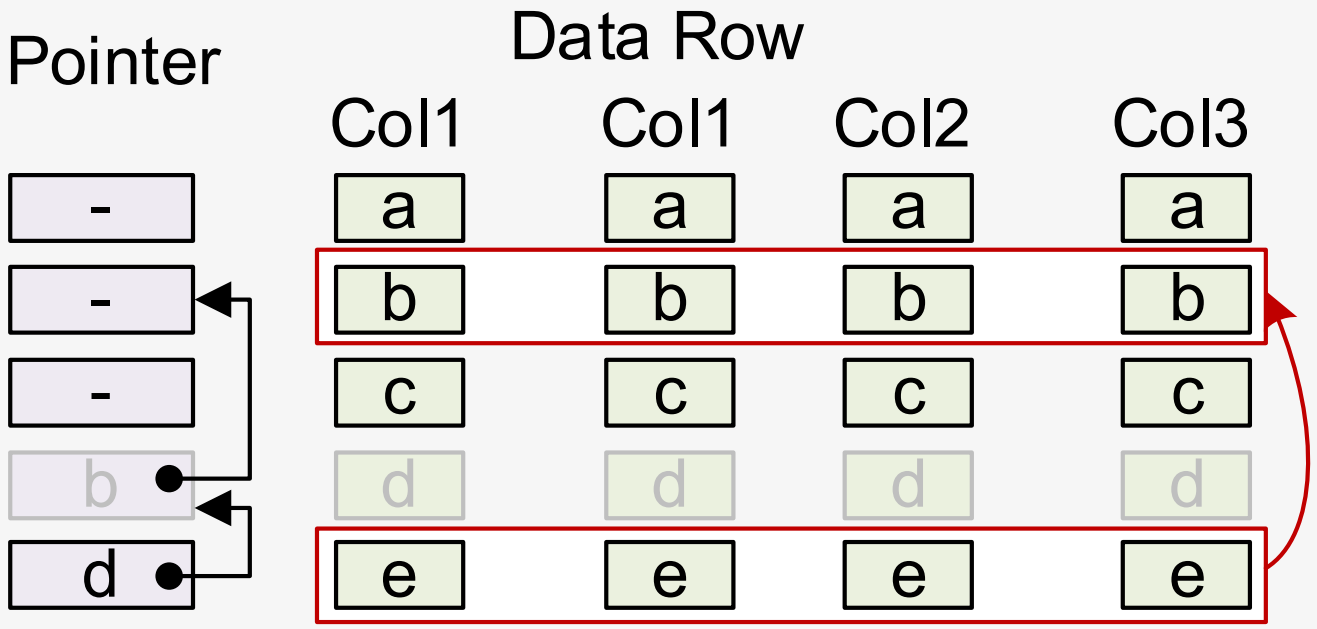
Fragmentation Challenge in PUSHtap:

PIM faces sparse access on fragmented storage



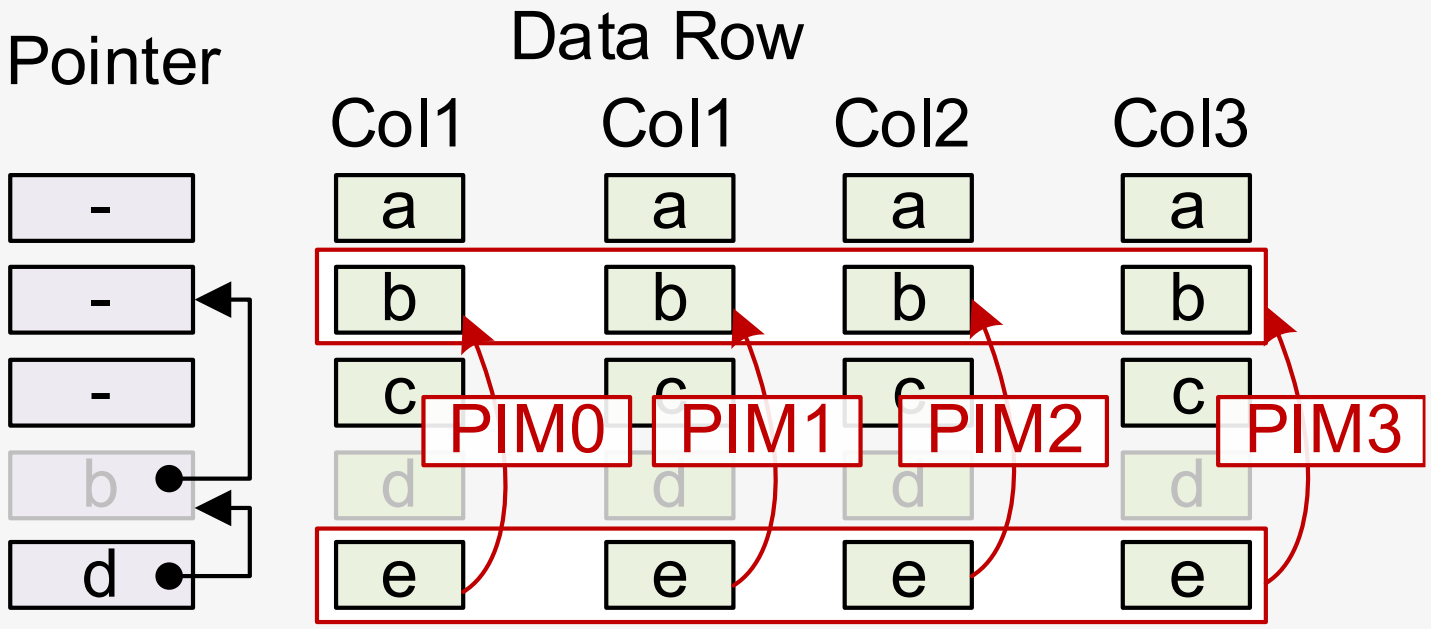
Solution 3: Defragmentation

→**Solution**: periodically **defragmentation**
(override origin version with the newest)



Solution 3: Defragmentation

→**Solution**: periodically **defragmentation**
(override origin version with the newest)

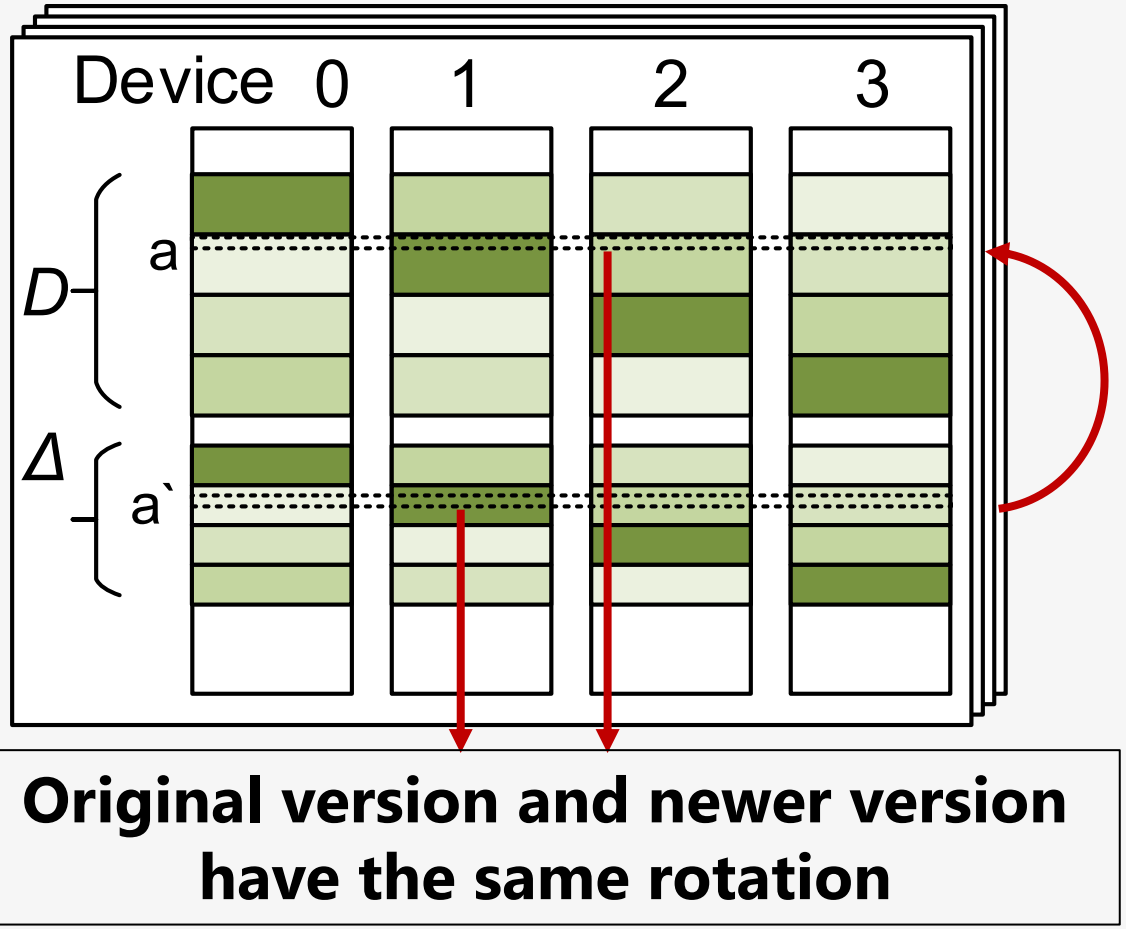


Defragmentation is a data copy → Utilize the high-bandwidth of PIM

Format-aware MVCC: to enable PIM do defragmentation

Data region (D):
the original version

Delta region (Δ):
the newer versions



→ **Enable PIM do direct in-place overwrite**

Solution 3: Defragmentation – Data Copy

Defragmentation: Two Options

- Option 1: Defragmentation with PIM

  Step 1: CPU read meta and broadcast to all PIMs

  Step 2: PIM move data

$$communication = \frac{dmn + 2npdw}{bdw_{PIM}} + \frac{mn + dmn}{bdw_{CPU}}$$

d: device num
m: meta size
w: row size
n: row num in Δ
p: ratio of newest version in Δ
$bdw_{CPU/PIM}$: bandwidth of CPU/PIM

mn: total meta size
$npdw$: total moved data size

Solution 3: Defragmentation – Data Copy

Defragmentation: Two Options

- Option 1: Defragmentation with PIM

  Step 1: CPU read meta and broadcast to all PIMs

  Step 2: PIM move data

$$communication = \frac{dmn + 2npdw}{bdw_{PIM}} + \frac{mn + dmn}{bdw_{CPU}}$$

- Option 2: Defragmentation with CPU

  Step 1: CPU read meta

  Step 2: CPU move data

$$communication = \frac{mn + 2npdw}{bdw_{CPU}}$$

d: device num
m: meta size
w: row size
n: row num in Δ
p: ratio of newest version in Δ
$bdw_{CPU/PIM}$: bandwidth of CPU/PIM

mn: total meta size
$npdw$: total moved data size

Solution 3: Defragmentation – Data Copy

Defragmentation: Two Options

- Option 1: Defragmentation with PIM

  Step 1: CPU read meta and broadcast to all PIMs

  Step 2: PIM move data

$$communication = \frac{dmn + 2npdw}{bdw_{PIM}} + \frac{mn + dmn}{bdw_{CPU}}$$

- Option 2: Defragmentation with CPU

  Step 1: CPU read meta

  Step 2: CPU move data

$$communication = \frac{mn + 2npdw}{bdw_{CPU}}$$

PIM is better

Large w
(Total column width)

Small w

CPU is better

d: device num
m: meta size
w: total column width
n: row num in Δ
p: ratio of newest version in Δ
$bdw_{CPU/PIM}$: bandwidth of CPU/PIM

mn: total meta size
$npdw$: total moved data size

e.g. $bdw_{CPU} : bdw_{PIM} = 1:3, m = 16, p \approx 1$
$w > 16$ → use PIM
$w < 16$ → use CPU

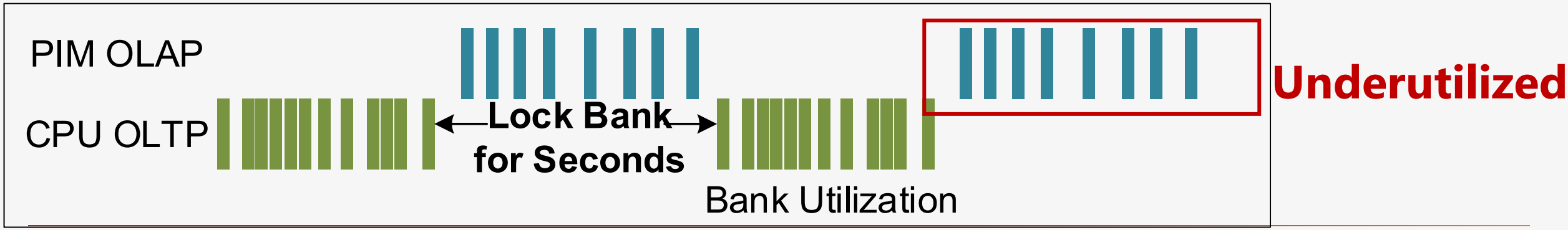
Challenge 4: CPU-PIM Bank Contention

Conventional PIM program: (e.g. on UPMEM):

```
int Sum(){
  mram_read(a, data[0]);   // DRAM->PIM buffer Xfer
  sum += a;                // Compute
  mram_read(a, data[1]);   // DRAM->PIM buffer Xfer
  sum += a;                // Compute
  mram_read(a, data[2]);   // DRAM->PIM buffer Xfer
  sum += a;                // Compute
  ...
```

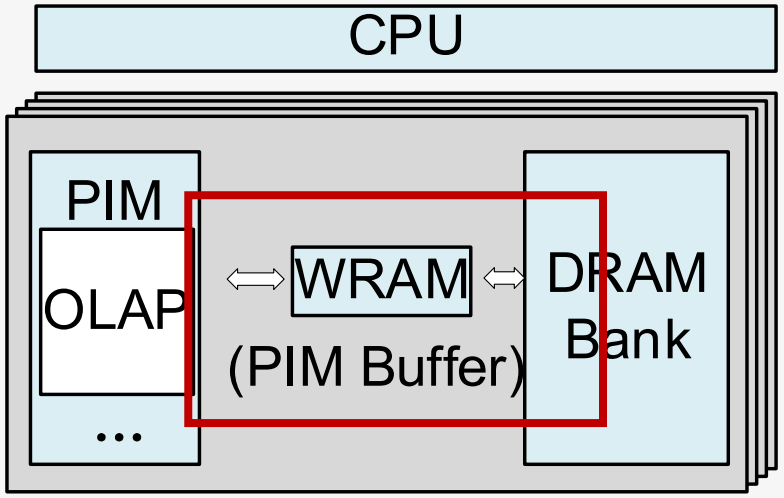
Interleaved Access & Computation

CPU access is prevented during the whole program



Solution 4: Architecture Support -- Two-phase Execution

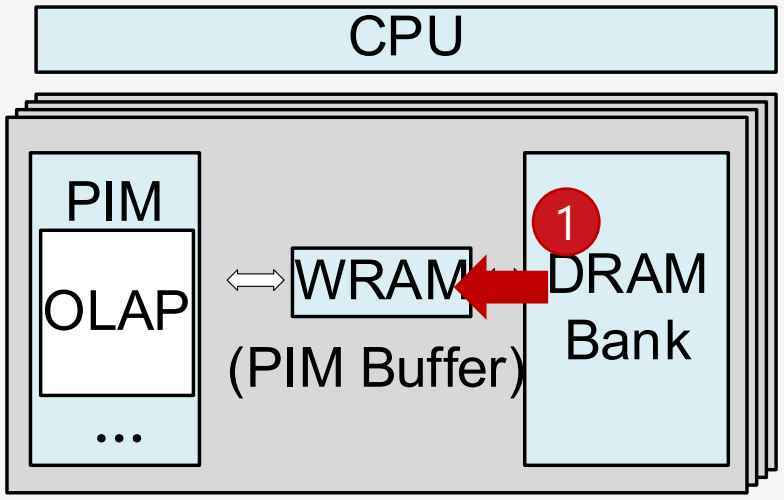
Decouple access and computation in PIMs by leveraging PIM buffers.



Solution 4: Architecture Support -- Two-phase Execution

Decouple access and computation in PIMs by leveraging PIM buffers.

PIM Buffer ~32kB
->30us block in phase 1



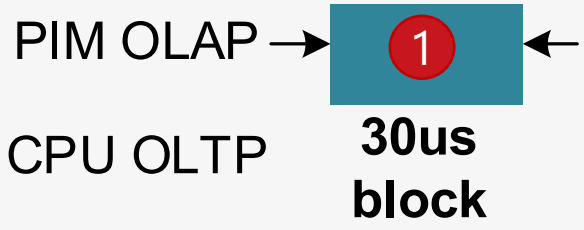
1 Load Phase: Bank->Buffer, Block CPU

PIM Program:

mram_read(a[0], data[0]); // **DRAM->PIM buffer**

mram_read(a[1], data[1]); // **DRAM->PIM buffer**

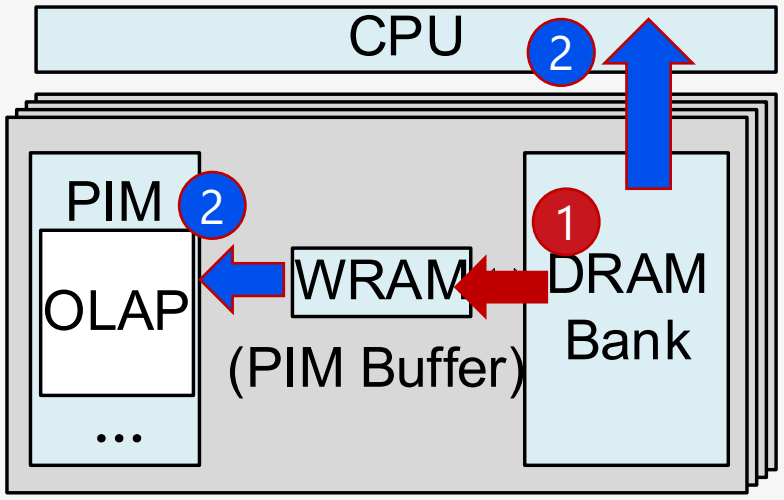
Only Access



Solution 4: Architecture Support -- Two-phase Execution

Decouple access and computation in PIMs by leveraging PIM buffers.

PIM Buffer ~32kB
->30us block in phase 1



1 Load Phase: Bank->Buffer, Block CPU

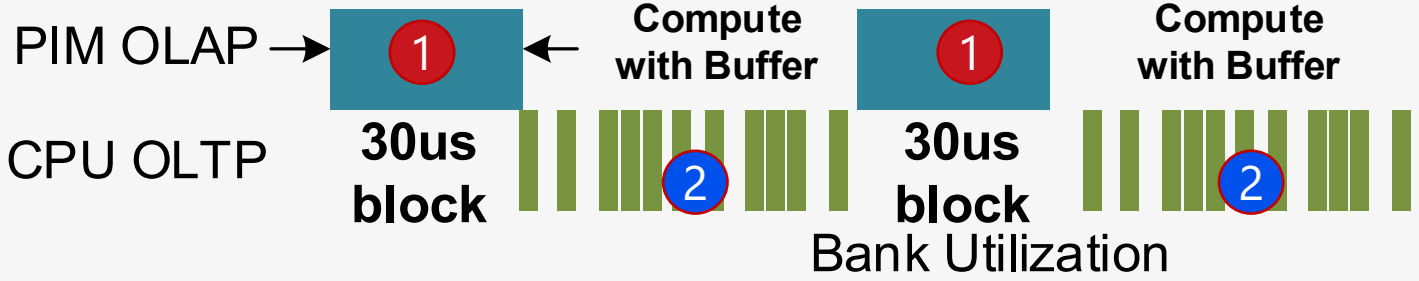
```
PIM Program:
  mram_read(a[0], data[0]); // DRAM->PIM buffer
  mram_read(a[1], data[1]); // DRAM->PIM buffer
```

Only Access

2 Compute Phase: Buffer->PIM, CPU access

```
PIM Program:
  sum += a[0];   // Compute
  sum += a[1];   // Compute
```

Only Compute



Better Utilized

More in the Paper

- Unified data format:
  - Algorithm for generating the unified data format
- Format-aware MVCC:
  - Communication optimization during snapshotting
- Architecture support:
  - CPU-PIM switch overhead optimization

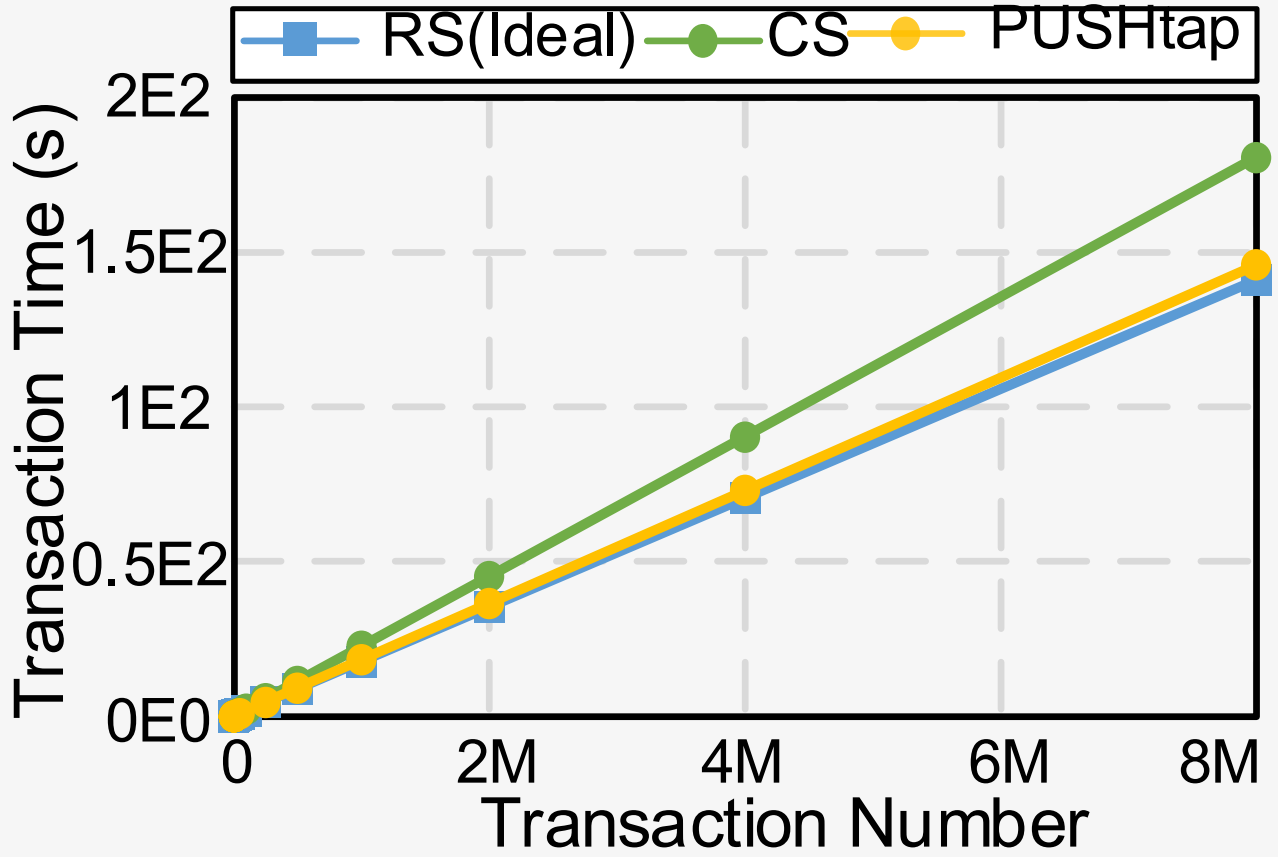
Experimental Setup

- HTAP Benchmark:
  - CH-benchmark (TPC-C+TPC-H), implemented based on DBx1000
  - size: 20GB
- Baselines:
  - ***RS:*** Row-store
  - ***CS:*** Column-store
  - ***MI:*** Multi-instance, row-store on CPU side, column-store on PIM side [ICDE'22], ([ICDE'22] is based on HBM, we fairly adapted it to DDR5)
- Simulation:
  - CPU model: zsim
  - DRAM model: Ramulator2

System Configuration	
CPU	16@3.2Hz
DRAM	DDR5-3200, {4 PIM Channel,4 CPU Channel}*4 Ranks
PIM	500MHz, 64 per Rank

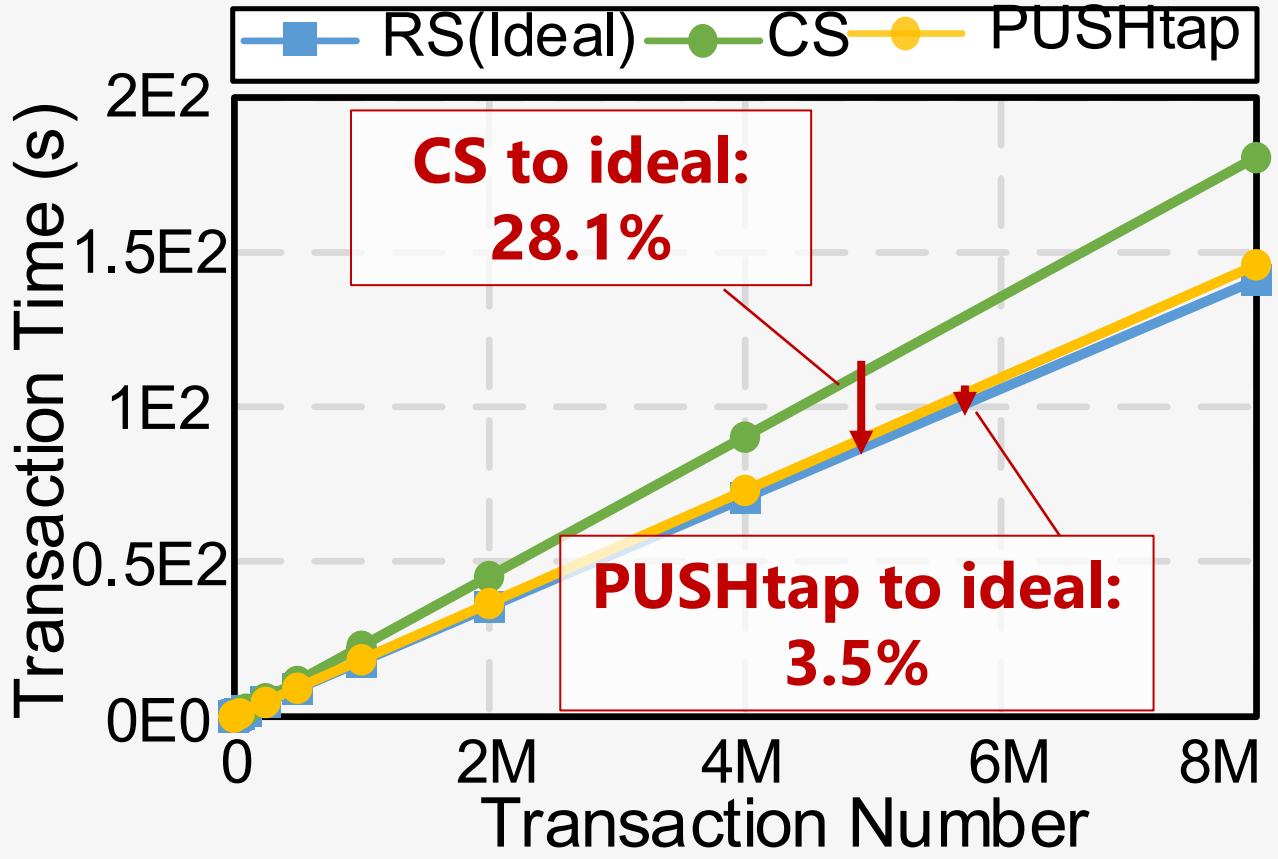
Result: Standalone Performance -- OLTP

OLTP Performance:



Result: Standalone Performance -- OLTP

OLTP Performance:

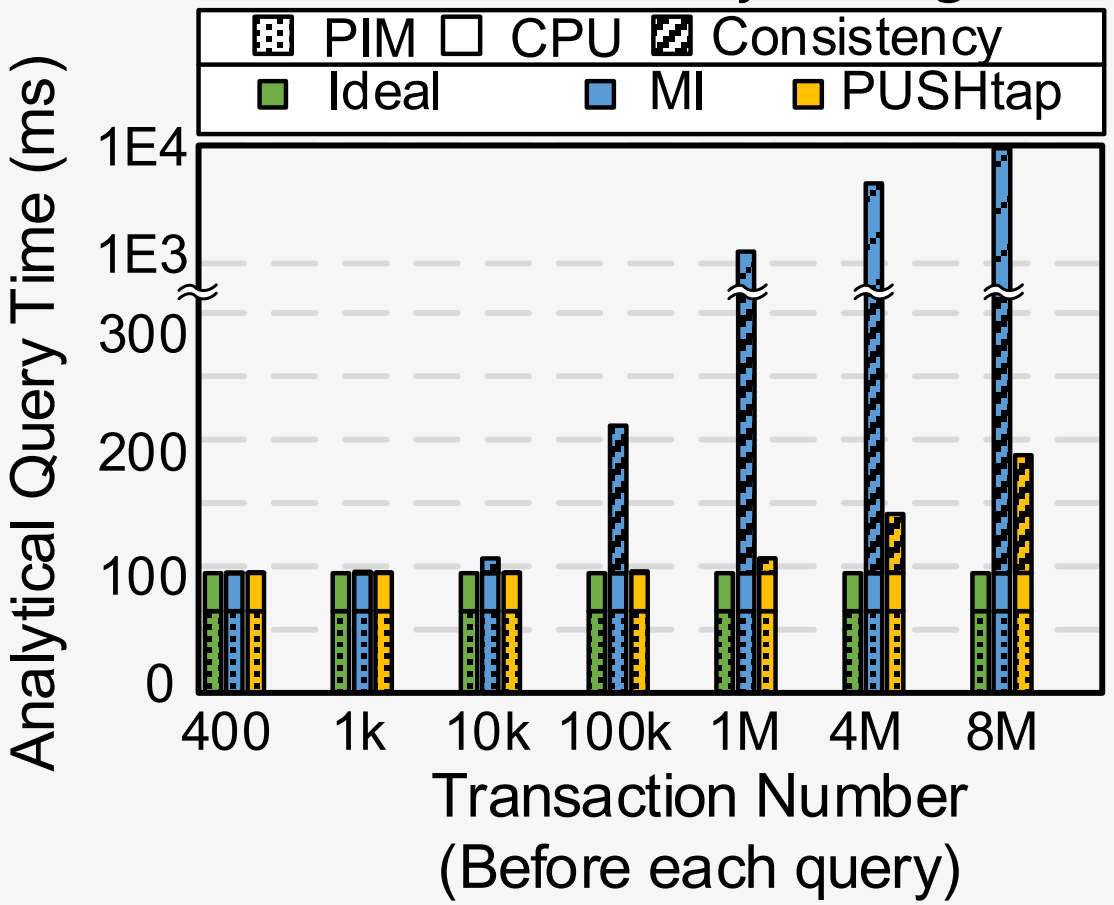


- Compared to *Row-store*(*RS*, *ideal*):
- *Column-store (CS)* is 28.1% worse than *ideal*
- ***PUSHtap*** **only incurs 3.5% degradation**

Result: Standalone Performance -- OLAP

OLAP Performance:

ideal: with zero-latency defragmentation



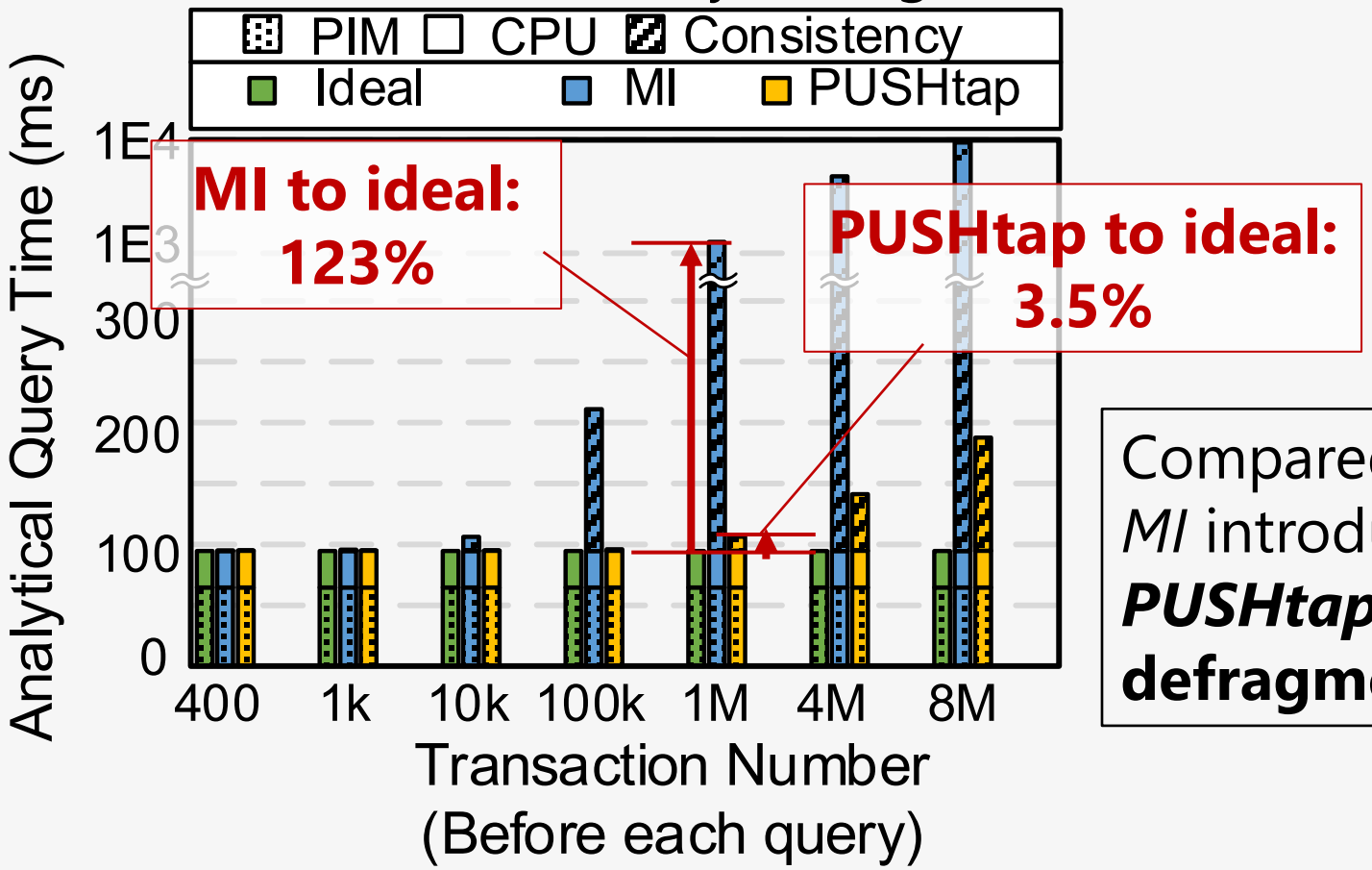
Consistency includes defragmentation/ rebuilding

Result: Standalone Performance -- OLAP

OLAP Performance:

Consistency includes defragmentation/ rebuilding

ideal: with zero-latency defragmentation



Compared to *ideal*: with 1M transactions, *MI* introduce 123% rebuilding overhead

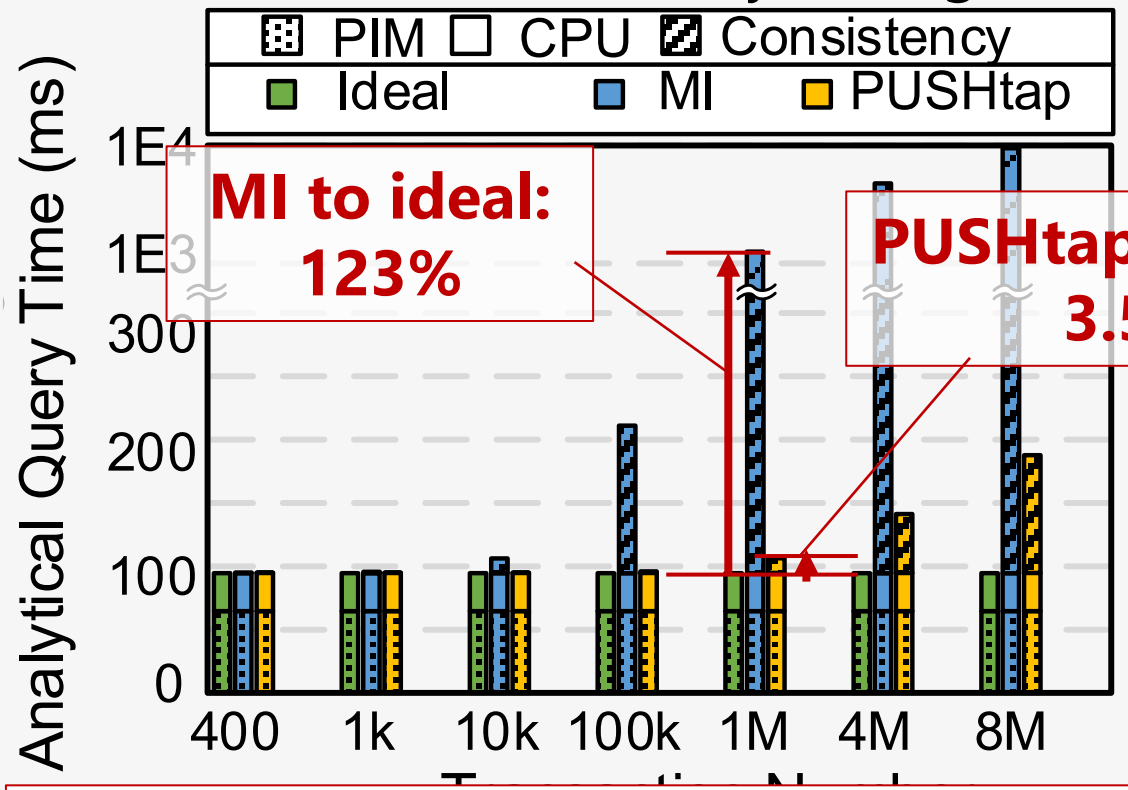
***PUSHtap* only incurs 1.5% defragmentation overhead**

Result: Standalone Performance -- OLAP

OLAP Performance:

Consistency includes defragmentation/ rebuilding

ideal: with zero-latency defragmentation

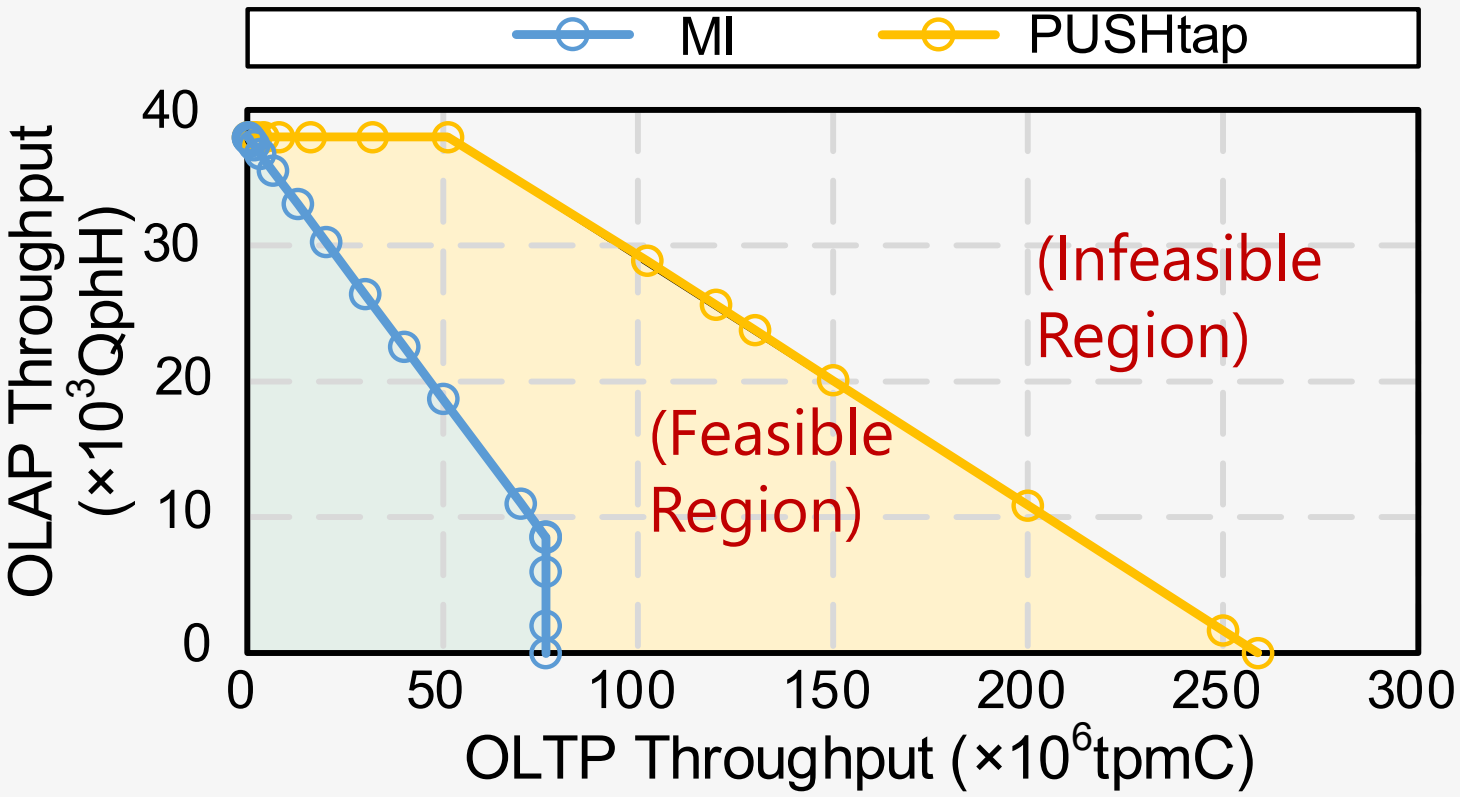


Compared to *ideal*: with 1M transactions, *MI* introduce 123% rebuilding overhead

PUSHtap **only incurs 1.5% defragmentation overhead**

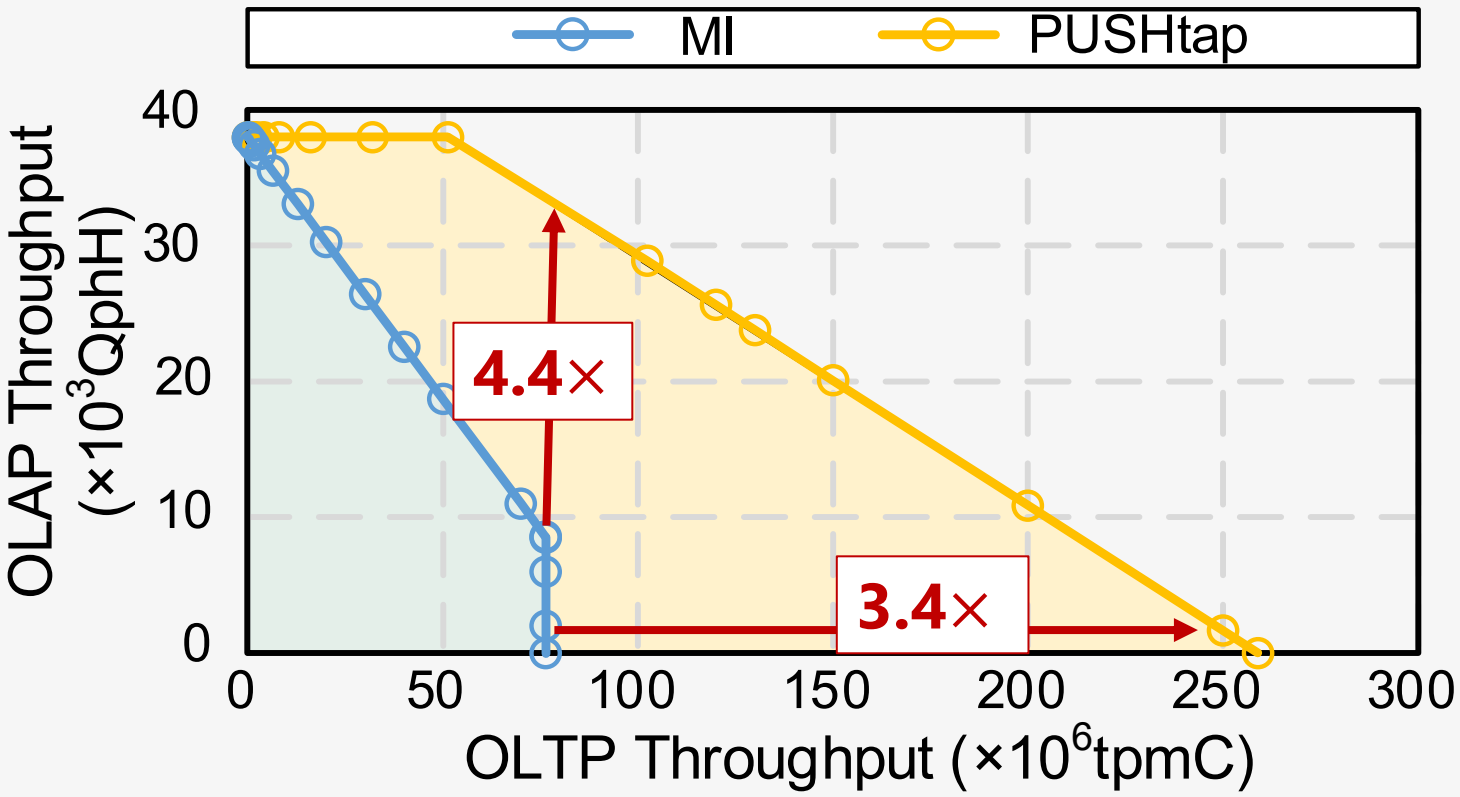
PUSHtap performance is closed to ideal on both OLTP and OLAP -- Workload-specific optimization

Result: Performance Isolation



The line: **throughput frontier**
Left/Bottom: feasible
Right/Top: infeasible due to system constrains

Result: Performance Isolation



The line: **throughput frontier**
Left/Bottom: feasible
Right/Top: infeasible due to system constrains

Compared to *MI*, *PUSHtap* has **3.4× peak OLTP throughput**
When *MI* OLTP reaches peak, PUSHtap can still have **4.4× OLAP throughput**

PUSHtap have better performance isolation

More Results in the Paper

- The effectiveness of unified data format algorithm
- Defragmentation overhead analyzation
- Defragmentation method selection
- Area overhead

Summary

Exploiting CPU-PIM access divergence, enables a single unified data format for both OLTP and OLAP in HTAP.

Critical contributions:

- Unified Data Format: Compact Aligned Format + Block-circulant Placement.
- Concurrency Control: Format-aware MVCC & Defragmentation.
- Architecture Support: Two-phase Execution for eliminating CPU-PIM stalls.

Results:

- 3.4× peak OLTP throughput
- 4.4× OLAP throughput under peak OLTP (vs. MI baseline)

PUSHtap: PIM-based In-Memory HTAP with Unified Data Storage Format

Yilong Zhao, Mingyu Gao, Huanchen Zhang, Fangxin Liu, Gongye Chen, He Xian, Haibing Guan, and Li Jiang

sjtuzyl@sjtu.edu.cn, ljiang_cs@sjtu.edu.cn

Shanghai Jiao Tong University, Shanghai Qi Zhi Institute, Tsinghua University

ASPLOS' 2025

March 24, 2026

上海交通大學

SHANGHAI JIAO TONG UNIVERSITY

Thank you

饮 水 思 源　爱 国 荣 校

上海交通大學
SHANGHAI JIAO TONG UNIVERSITY

Backup Slides

饮 水 思 源　爱 国 荣 校

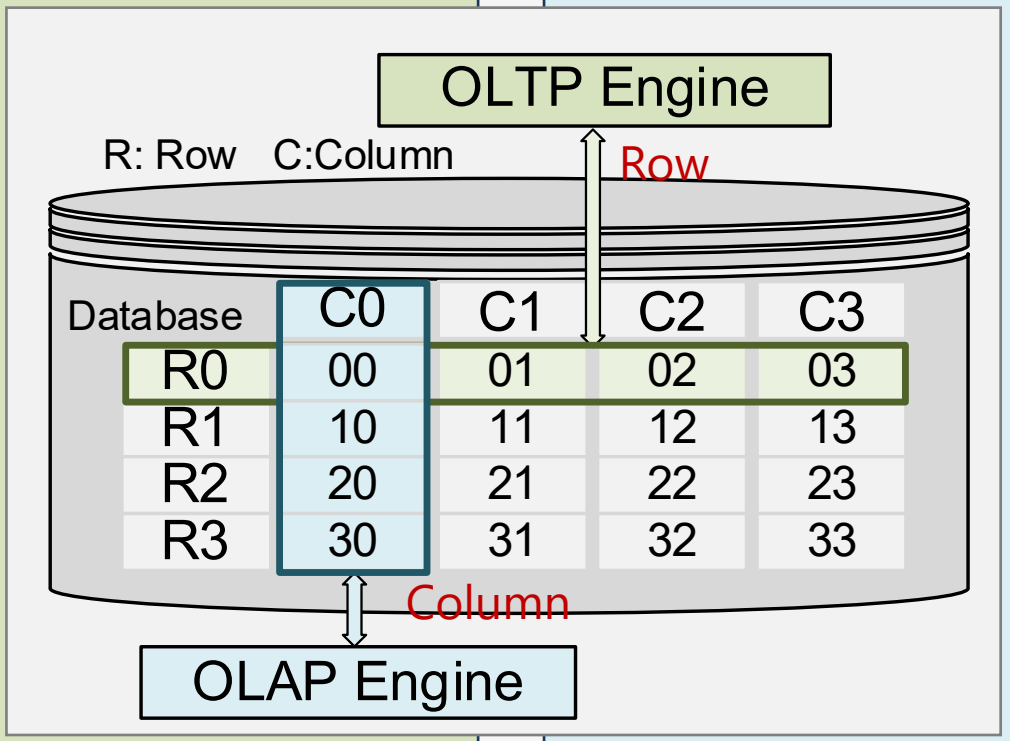
HTAP: The Convergence of Conflicting Workloads

Hybrid transaction/analytical processing (HTAP) database, 2 processing sets:

Online transaction processing (OLTP):

Access Pattern:
Row-wise, random
Operation:
Frequent updates
Bottleneck:
CPU-intensive

Ideal Format:
Row-store



Online analytical processing (OLAP):

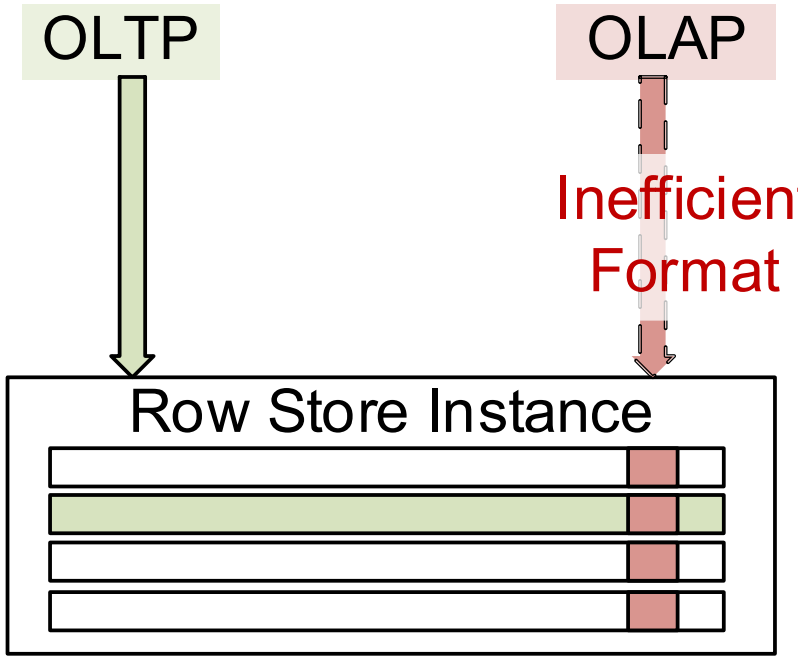
Access Pattern:
Column-wise, sequential
Operation:
Massive read-only
Bottleneck:
Memory-intensive

Ideal Format:
Column-store

Current solution 1 - Single-instance, Single-format

Data Format:

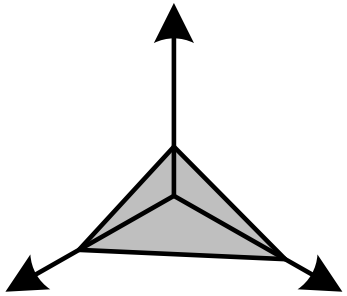
either row/column-store
(e.g. row-store)



Goals:

Workload-specific
Optimization
(Low)

(Inefficient
Format)



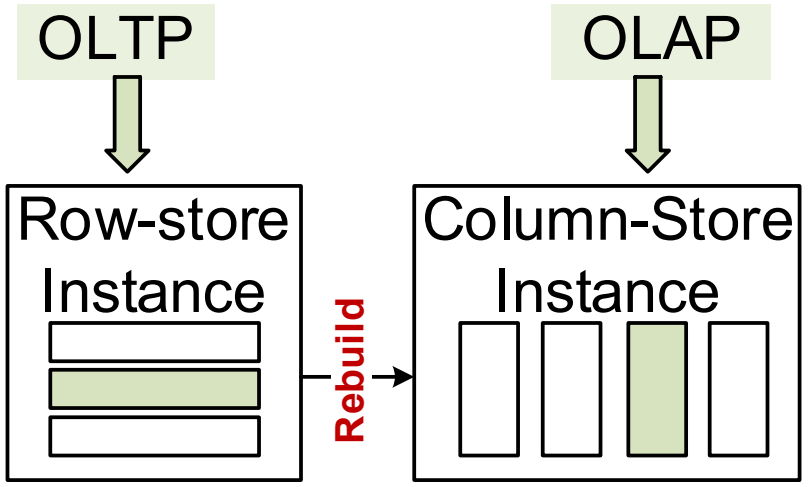
(No Consistency
Overhead)
Performance
Isolation
(High)

(Single
Instance)
Data
Freshness
(High)

Current solution 2 - Multi-instance, Multi-format

Data Format:

A row-store instance +
a column-store instance



Can only periodically rebuild due to large overhead

Goals:

Workload-specific Optimization (High)

(Efficient Format)

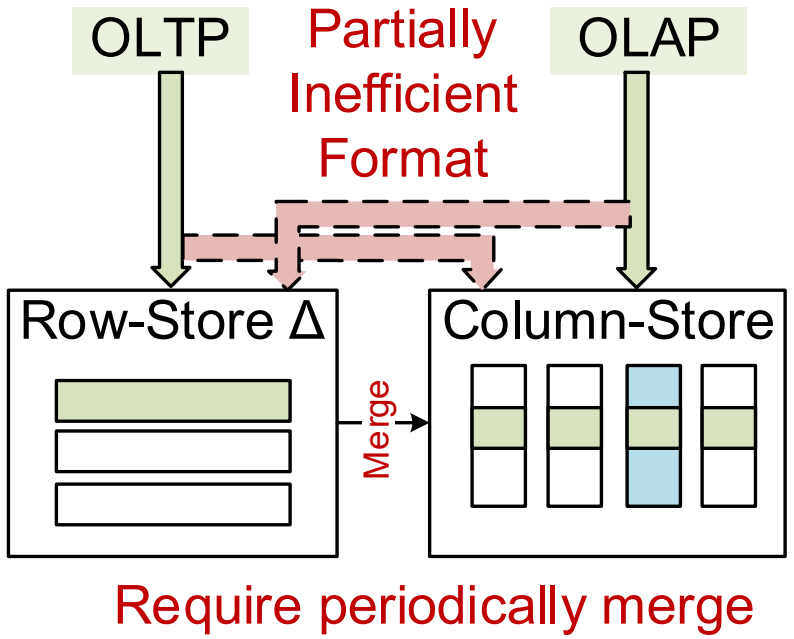
(Two Instances)
Performance Isolation (High)

(Periodically rebuild)
Data Freshness (Low)

Current solution 3 - Single-instance, Mixed-format

Data Format:

Primary column-store + row-store Δ



Require periodically merge

Goals:

Workload-specific Optimization (Medium)

(Partially Inefficient Format)

(Periodically Merge)

Performance Isolation (Medium)

(Single Instance)

Data Freshness (High)

Summary: Limitations of Current HTAP Solutions

Format	Performance Isolation	Data Freshness	Workload-Specific Optimization
Single-instance	H	H	L
Multi-instance	H	L	H
Mixed-format	M	H	M

Summary: Data format conflicts prevent CPU-only systems from simultaneously achieving all three goals

Summary: Limitations of Current HTAP Solutions

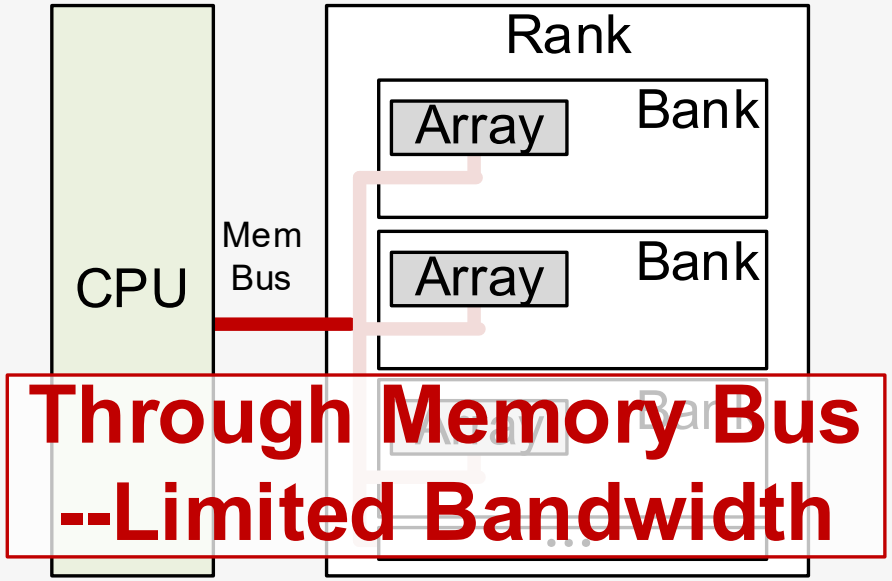
Format	Performance Isolation	Data Freshness	Workload-Specific Optimization
Single-instance	H	H	L
Multi-instance	H	L	H
Mixed-format	M	H	M
Ideal PUSHtap?	**H**	**H**	**H**

Resolve the data format conflict?

Summary: Data format conflicts prevent CPU-only systems from simultaneously achieving all three goals

Background: Processing-in-memory (PIM)

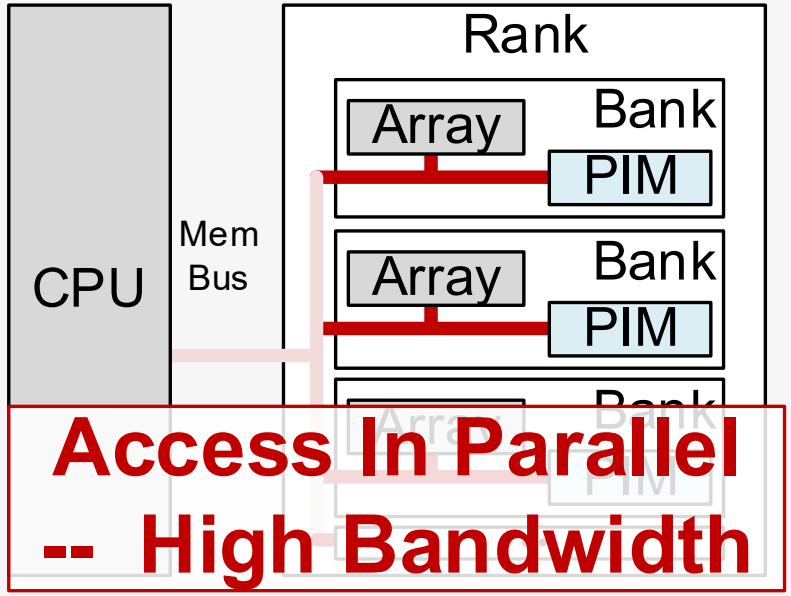
Conventional Von Neumann Architecture



Suffer from limited bandwidth between CPU and Memory

Processing-in-memory (PIM):

Integrate Computing Units in banks/devices

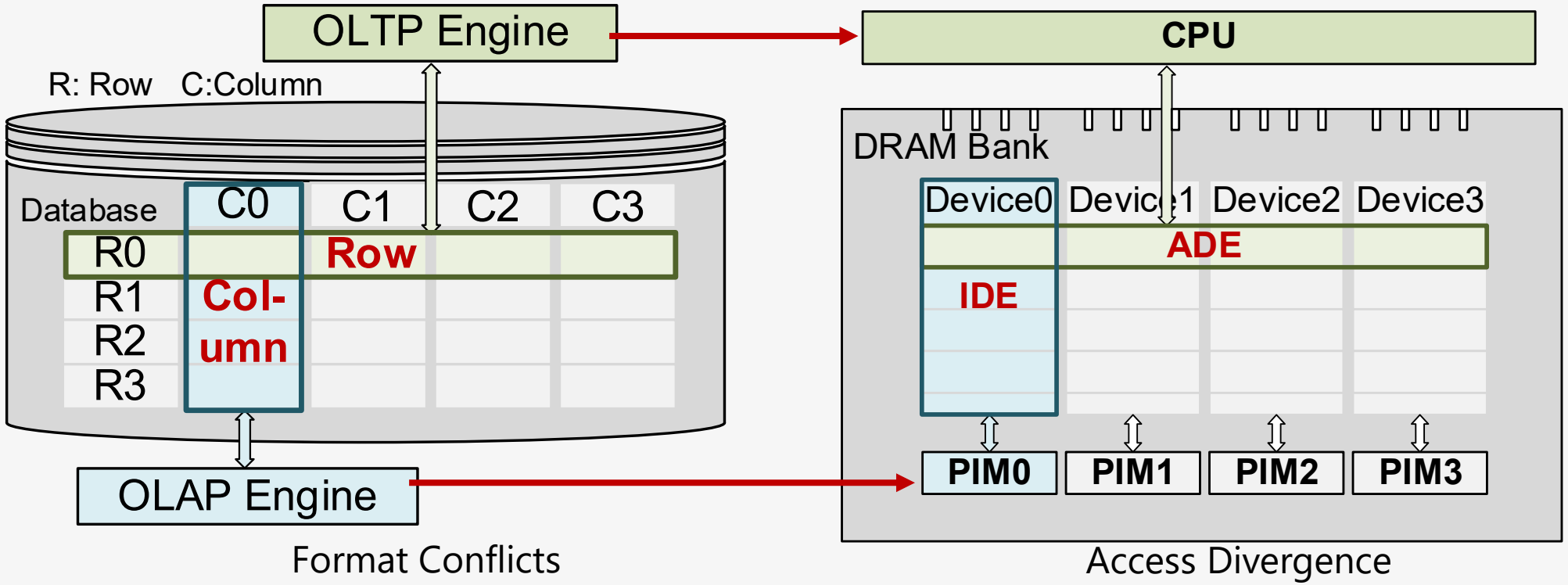


Concurrent access of thousands of PIM units provide massive bandwidth

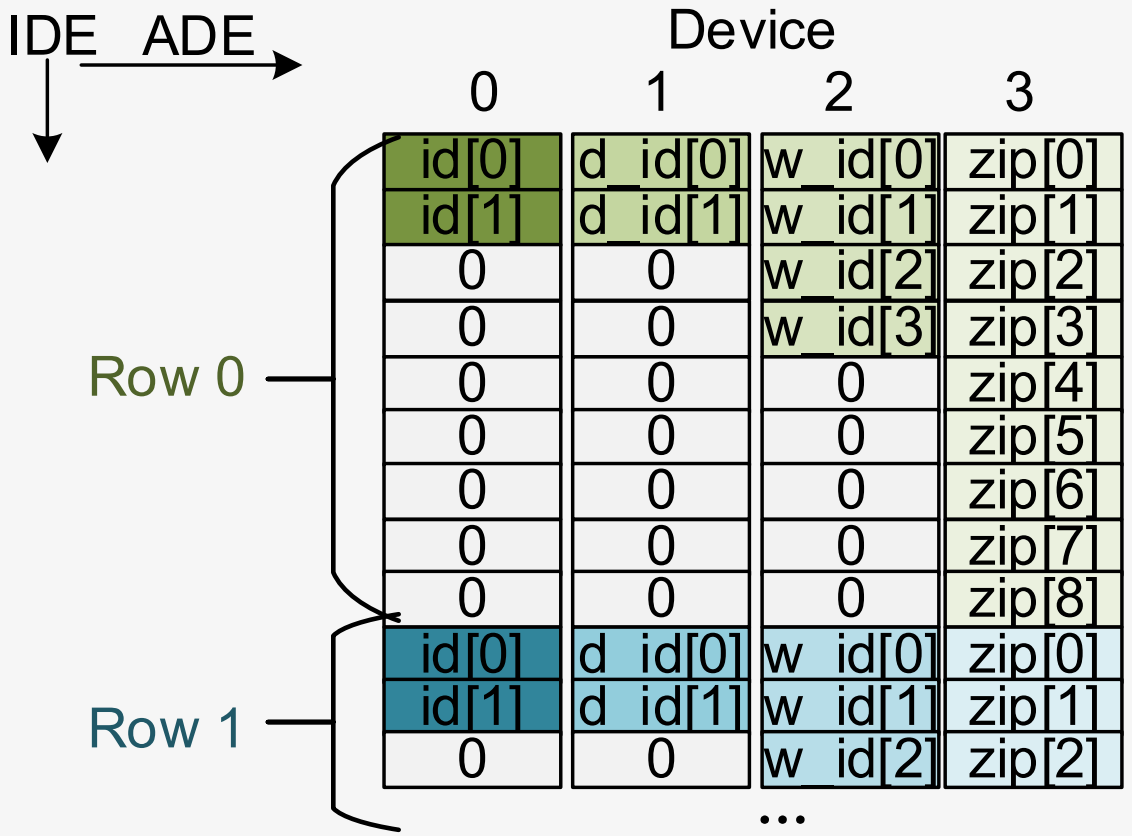
PIM is fit for memory-intensive OLAP

Key Insight: Format Conflict vs. Access Divergence

HTAP				PIM
OLTP	Row-wise Computation Intensive	→	Across Device (ADE) High Computational Power	CPU
OLAP	Column-wise Memory Intensive	→	Inside Device (IDE) High Bandwidth	PIM

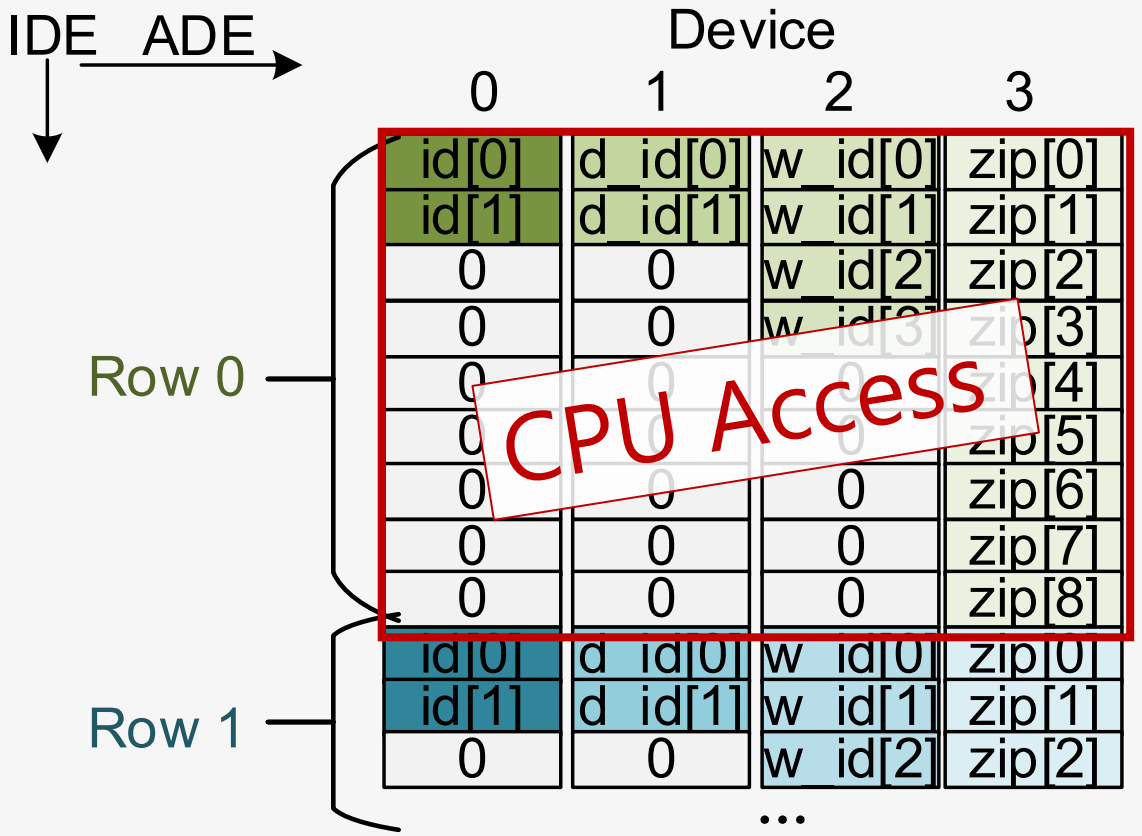


One Naïve Solution: padding 0s



Aligned on both dimensions √

One Naïve Solution: padding 0s



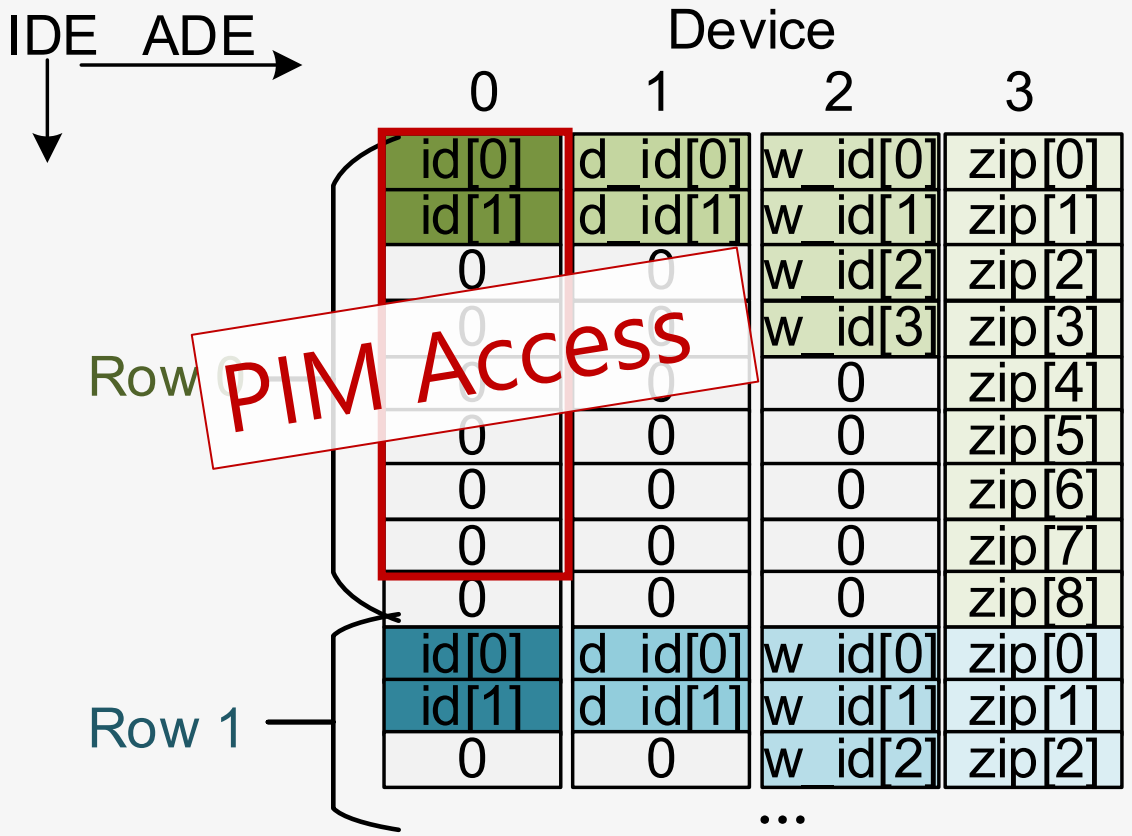
Aligned on both dimensions √

Bandwidth Waste ×
(Both CPU & PIM)

CPU: Interleaved Access
<0.5 effective bandwidth

Challenge 1: Data Format Challenge (2)

One Naïve Solution: padding 0s

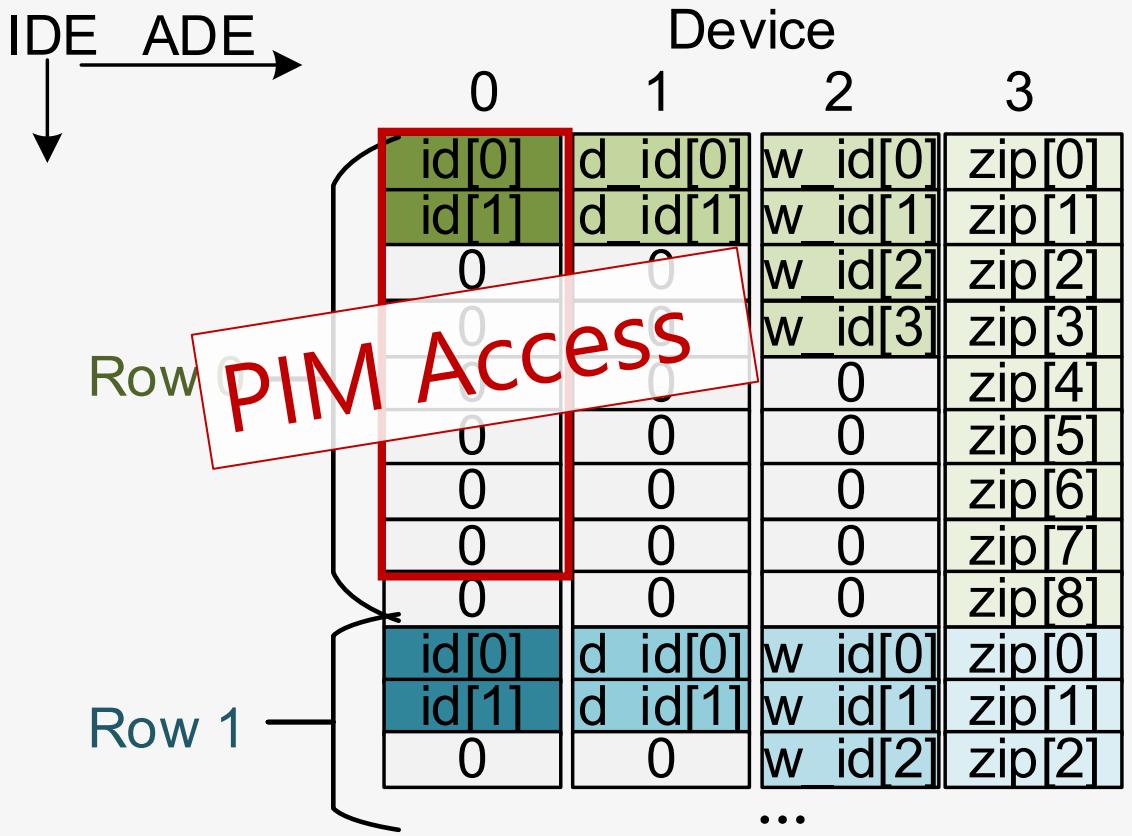


Aligned on both dimensions √

Bandwidth Waste ×
(Both CPU & PIM)

PIM: 8 Byte Burst (UPMEM)
<25% effective bandwidth

One Naïve Solution: padding 0s



Aligned on both dimensions √

Bandwidth Waste ×
(Both CPU & PIM)

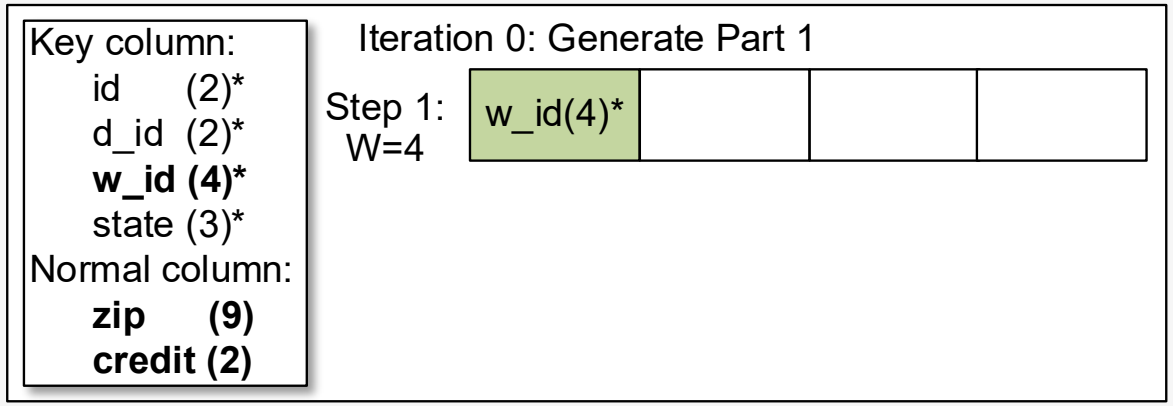
How to reduce the 0s?

Generate Compact Aligned Format (1)

Generating the Compact Aligned Format:

In each iteration:

Step 1: Select the widest *key column, width W*



Generate Compact Aligned Format (1)

Generating the Compact Aligned Format:

In each iteration:

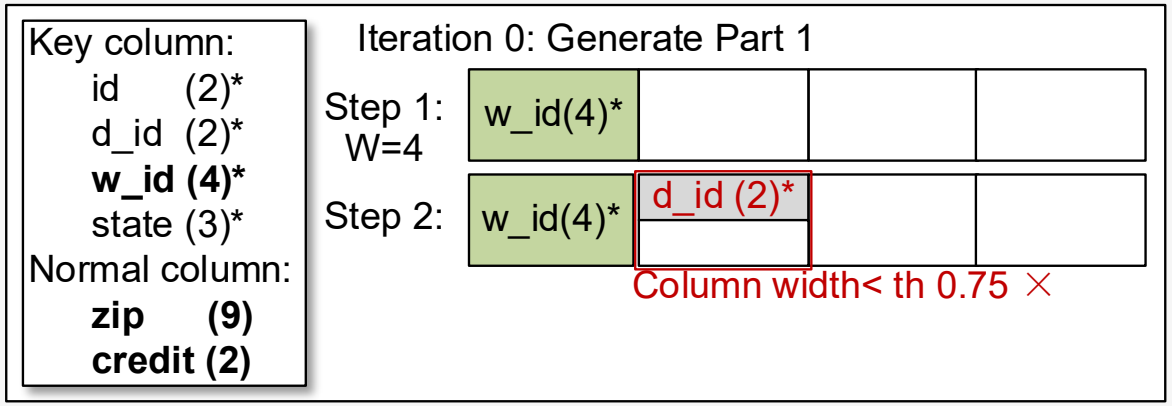
Step 1: Select the widest *key column, width* W

Step 2: Select from the remaining *key columns*

that is wider than $W \cdot th$

(th is a predefined threshold, e.g., 0.75)

d_id (width =2, $< W \cdot th = 3$) ×



Generate Compact Aligned Format (1)

Generating the Compact Aligned Format:

In each iteration:

Step 1: Select the widest *key column, width* W

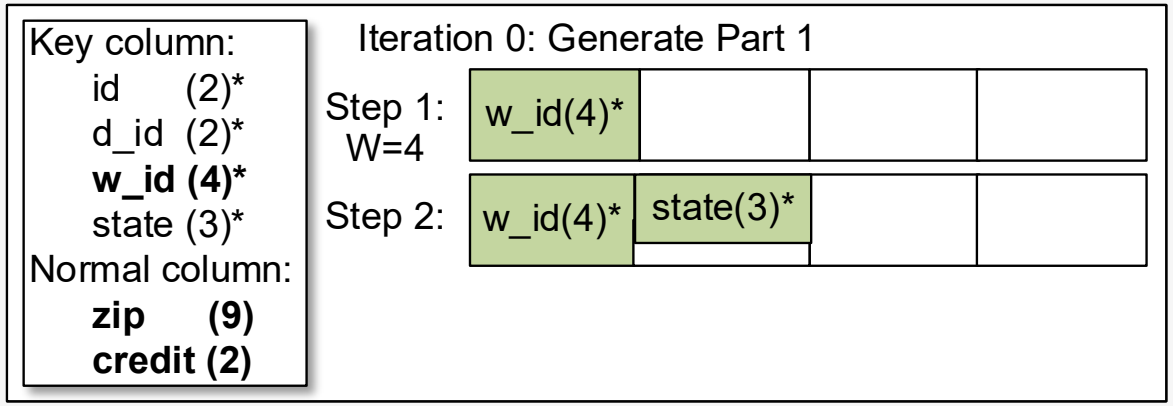
Step 2: Select from the remaining *key columns*

that is wider than $W \cdot th$

(th is a predefined threshold, e.g., 0.75)

d_id (width =2, $< W \cdot th = 3$) ×

state(width =3, $\geq W \cdot th = 3$) √



Generate Compact Aligned Format (1)

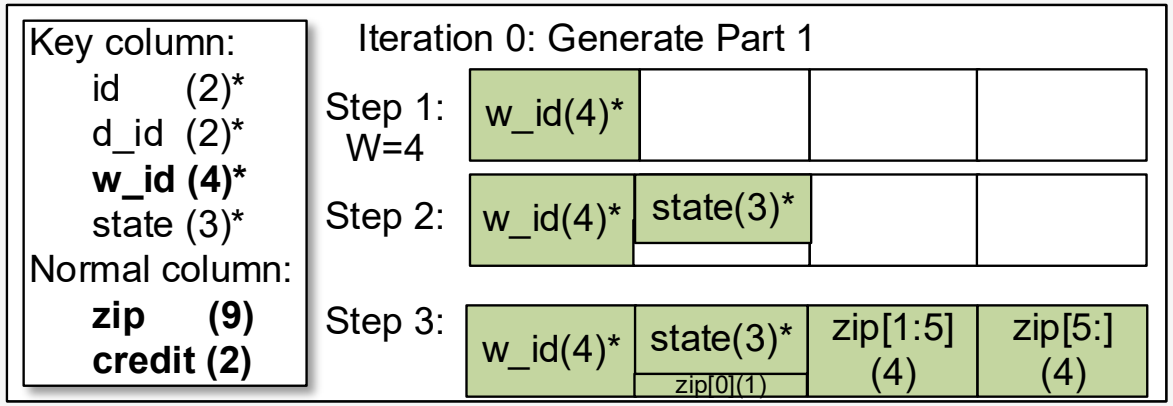
Generating the Compact Aligned Format:

In each iteration:

Step 1: Select the widest *key column, width* W

Step 2: Select from the remaining *key columns* that is wider than $W \cdot th$ (th is a predefined threshold, e.g., 0.75)

Step 3: Fill the rest of the bytes with *normal columns*



Generate Compact Aligned Format (1)

Generating the Compact Aligned Format:

In each iteration:

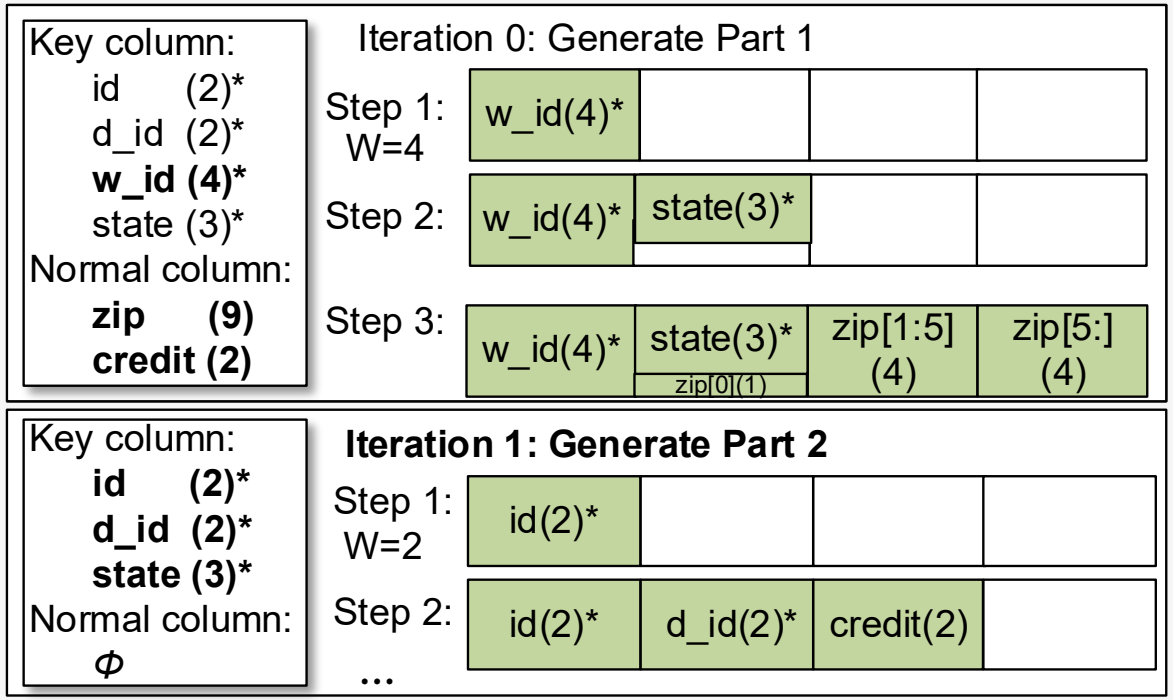
Step 1: Select the widest *key column, width* W

Step 2: Select from the remaining *key columns* that is wider than $W \cdot th$ (th is a predefined threshold, e.g., 0.75)

Step 3: Fill the rest of the bytes with *normal columns*

Each iteration generate one part of table

The parts are located on different banks for efficient CPU access



Generate Compact Aligned Format (2) -- Trade-off with th

CPU-PIM effective bandwidth trade-off with th:

Generate Compact Aligned Format (2) -- Trade-off with th

CPU-PIM effective bandwidth trade-off with th:

If th is too small: PIM bandwidth is underutilized

$th = 0.5$: *d_id* can fit in part 1

Generated Format:

Part 1:

w_id(4)*	d_id (2)*	id (2)*	state (3)*
	credit(2)	zip[0:2] (2)	zip[2] (1)

Part 2:

zip[3:5] (2)	zip[5:7] (2)	zip[7:] (2)	0

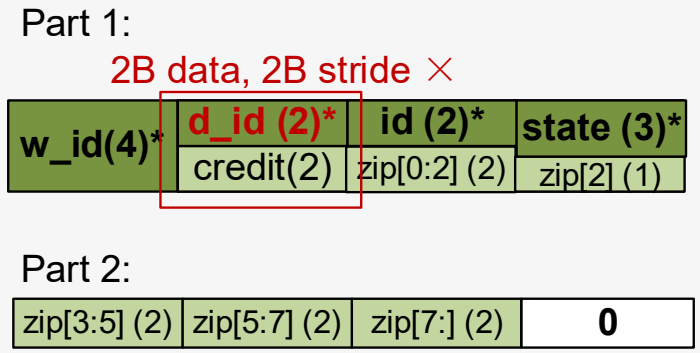
Generate Compact Aligned Format (2) -- Trade-off with th

CPU-PIM effective bandwidth trade-off with th:

If th is too small: PIM bandwidth is underutilized

$th = 0.5$: *d_id* can fit in part 1

Generated Format:



Typical PIM access granularity: 8 byte (UPMEM)

PIM scans *d_id*: 2B data + 2B stride

→ 50% PIM bandwidth utilization ×

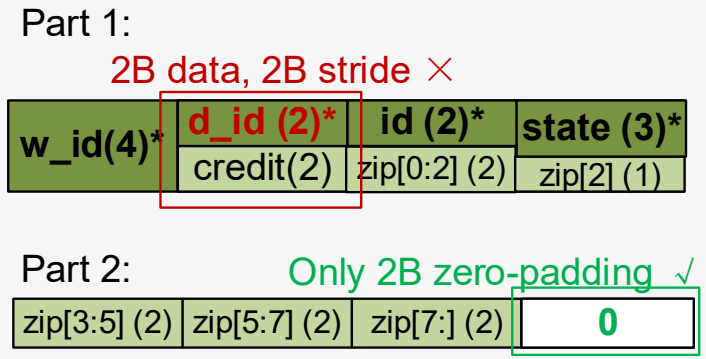
Generate Compact Aligned Format (2) -- Trade-off with th

CPU-PIM effective bandwidth trade-off with th:

If th is too small: PIM bandwidth is underutilized

$th = 0.5$: *d_id* can fit in part 1

Generated Format:



Typical PIM access granularity: 8 byte (UPMEM)

PIM scans *d_id*: 2B data + 2B stride

→ 50% PIM bandwidth utilization ×

CPU access the row: Only 2B zero-padding

→ 90% CPU bandwidth utilization √

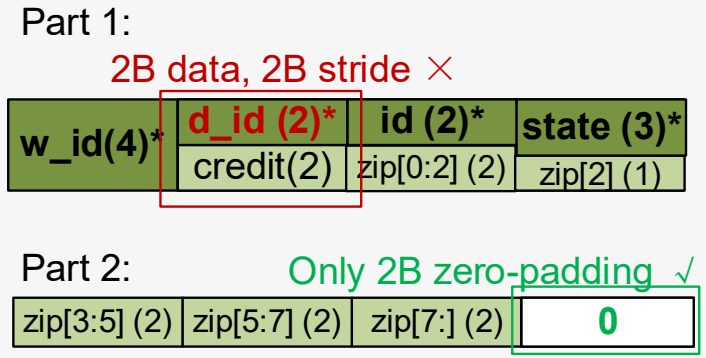
Generate Compact Aligned Format (2) -- Trade-off with th

CPU-PIM effective bandwidth trade-off with th:

If th is too small: PIM bandwidth is underutilized

$th = 0.5$: *d_id* can fit in part 1

Generated Format:



Typical PIM access granularity: 8 byte (UPMEM)

PIM scans *d_id*: 2B data + 2B stride

→ 50% PIM bandwidth utilization ×

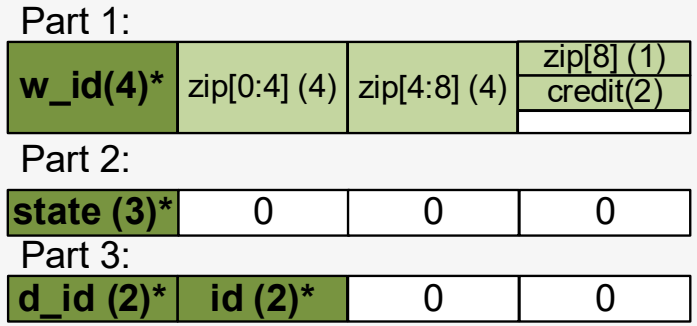
CPU access the row: Only 2B zero-padding

→ 90% CPU bandwidth utilization √

If th is too large: CPU bandwidth is underutilized

$th = 1$: *state* can not fit in part 1 -- arranged in a new part

Generated Format:



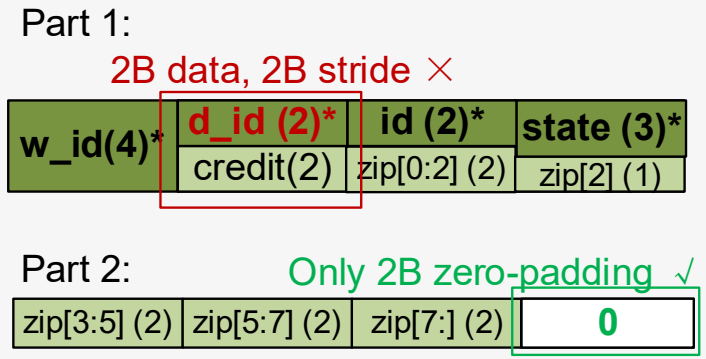
Generate Compact Aligned Format (2) -- Trade-off with th

CPU-PIM effective bandwidth trade-off with th:

If th is too small: PIM bandwidth is underutilized

$th = 0.5$: *d_id* can fit in part 1

Generated Format:



Typical PIM access granularity: 8 byte (UPMEM)

PIM scans *d_id*: 2B data + 2B stride

→ 50% PIM bandwidth utilization ×

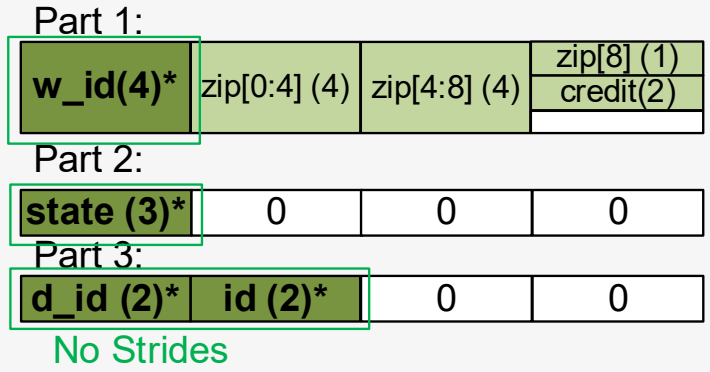
CPU access the row: Only 2B zero-padding

→ 90% CPU bandwidth utilization √

If th is too large: CPU bandwidth is underutilized

$th = 1$: *state* can not fit in part 1 -- arranged in a new part

Generated Format:



PIM scans all the *key columns*: No strides

→ 100% PIM bandwidth utilization √

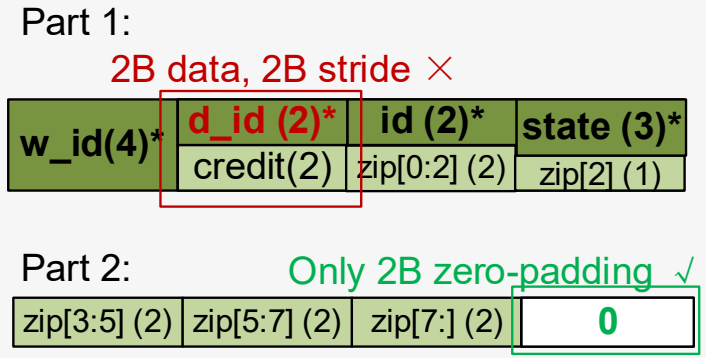
Generate Compact Aligned Format (2) -- Trade-off with th

CPU-PIM effective bandwidth trade-off with th:

If th is too small: PIM bandwidth is underutilized

$th = 0.5$: *d_id* can fit in part 1

Generated Format:



Typical PIM access granularity: 8 byte (UPMEM)

PIM scans *d_id*: 2B data + 2B stride

→ 50% PIM bandwidth utilization ×

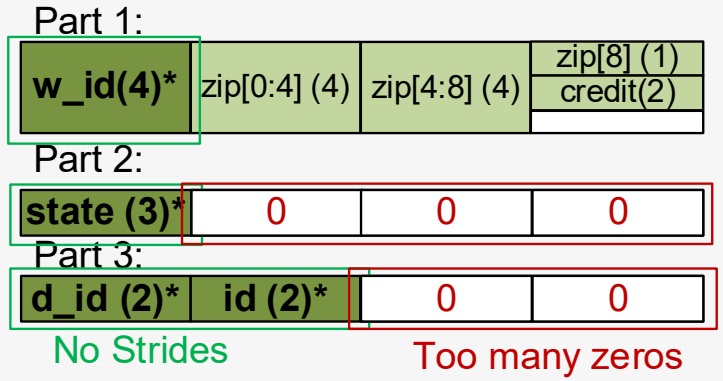
CPU access the row: Only 2B zero-padding

→ 90% CPU bandwidth utilization √

If th is too large: CPU bandwidth is underutilized

$th = 1$: *state* can not fit in part 1 -- arranged in a new part

Generated Format:



PIM scans all the *key columns*: No strides

→ 100% PIM bandwidth utilization √

CPU access the row: More 0 padding

→ 60% CPU bandwidth utilization ×

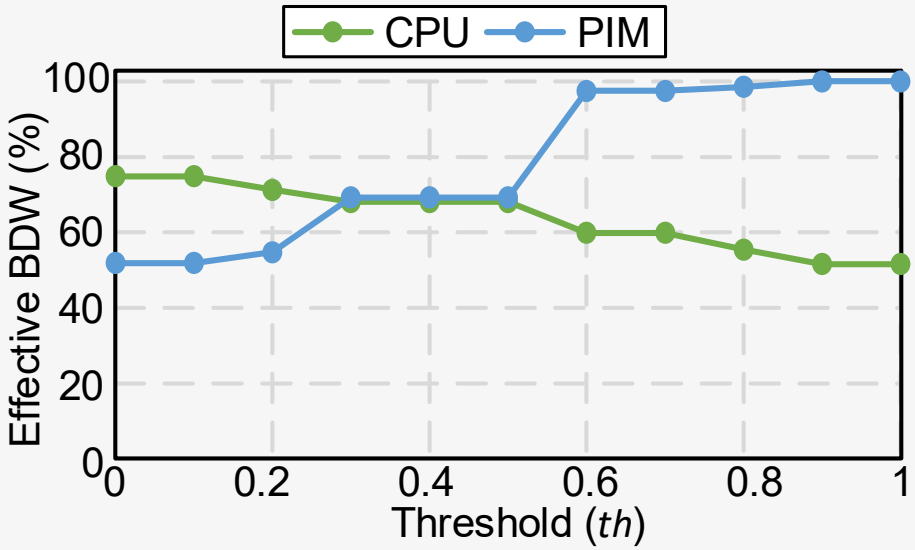
System Configuration

Table 1. System Configuration

Host CPU	
Processor	16 × O3CPU @3.2GHz
L1I/L1D	32kB / 32kB, Assoc: 8
L2 / L3	1MB Assoc: 16 / 22MB, Assoc: 22
Cache Line	64 B
DRAM DIMM	
DRAM	DDR5-3200, 8×8, 8GB/Rank
Ba / De / Ro / Co	8 / 8 / 131072 / 1024
Timing Param.	tBURST=2.5ns tRCD=tCL=tRP=7.5ns tRAS=16.3ns tRRD=2.5ns tRFC=121.9ns tWR=15.0ns tWTR=11.2ns tRTP=3.75ns tRTW=tCS=4.4ns tREFI=3.9us
PIM Units	
PIM Unit	500MHz, 16 tasklets, 1GB/s bandwidth [11] 64kB WRAM, 64-bit PIM-DRAM wire width
Num	64 per Rank, at Bank level inside Devices
System Configuration	
CPU System	4 Channels ×4 Ranks normal DRAM 4 Channels ×4 Ranks with PIM units

Result: Data Format

PIM-CPU bandwidth trade-off with th:



th=0.6

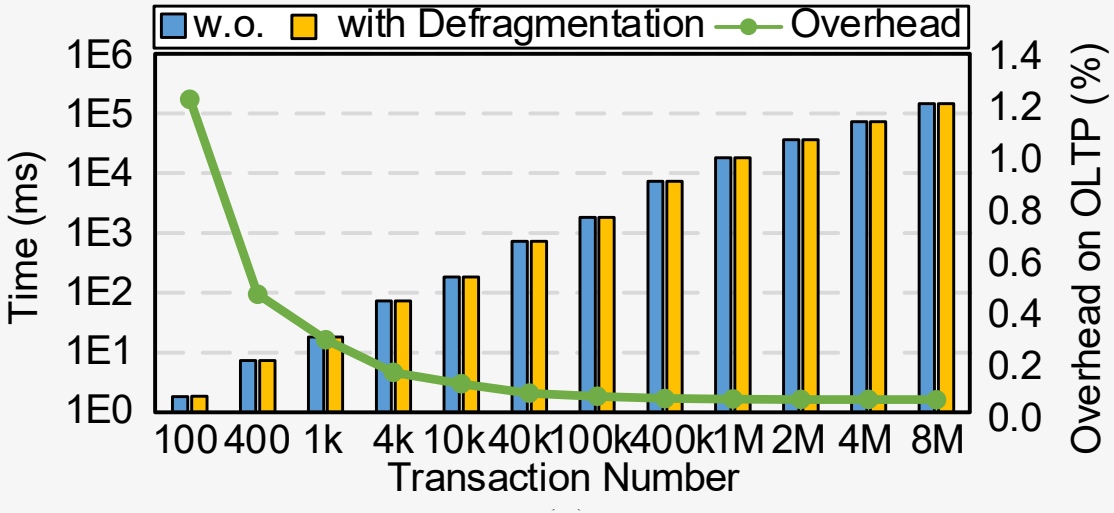
Item	Storage
Data	96.9%
Padding 0	0.8%
Snapshot	2.3%

OLTP is Compute-Bounded

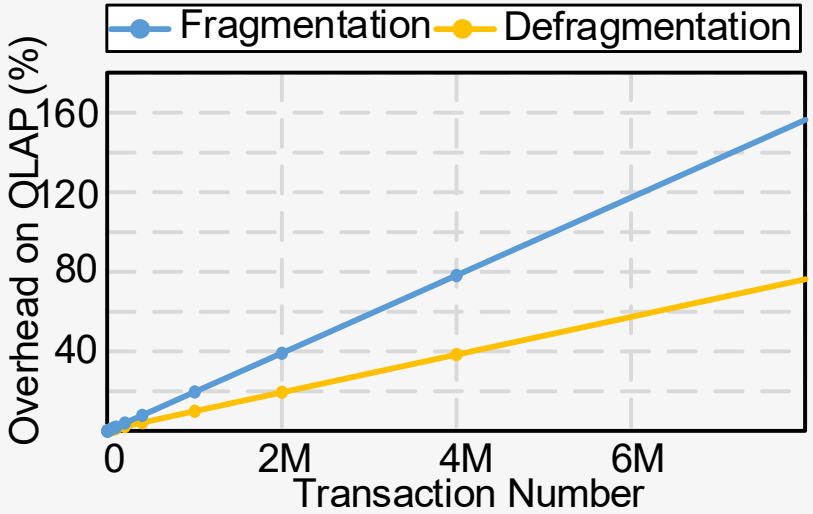
→ Can sacrifice some OLTP bandwidth to favor OLAP.

Decision: $th = 0.6$
PIM effective bandwidth: 97.4%;
CPU effective bandwidth: 59.8% (only 15% lower than peak)

Result: Defragmentation Overhead

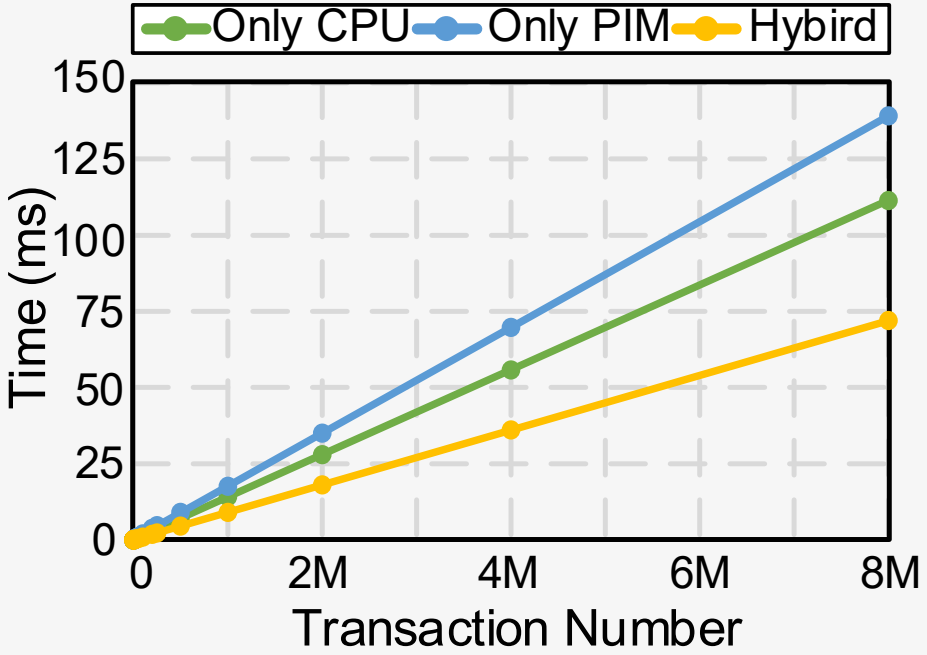


Defragmentation only incur **<1.5%** overhead on OLTP



Defragmentation improves OLAP performance by **2.05**×

Result: Defragmentation Method



Only CPU: Defragmentation with CPU
Only PIM: Defragmentation with PIM
Hybrid: Choose CPU/PIM according to column width

Column width varies from 2 bytes to 20 bytes

Hybrid achieves the best efficiency