Advanced Compression in TimescaleDB

David Kohn

Solutions Architect/Software Engineer, Timescale

david@timescale.com · **github.com/timescale**

OLTP

Primarily UPDATEs

Writes randomly distributed

Transactions to multiple primary keys

Time-series

Primarily INSERTs

Writes to recent time interval

Writes primarily associated with a timestamp

Simplifies stack, accelerates development

Application

RDBMS

NoSQL

VS

Application

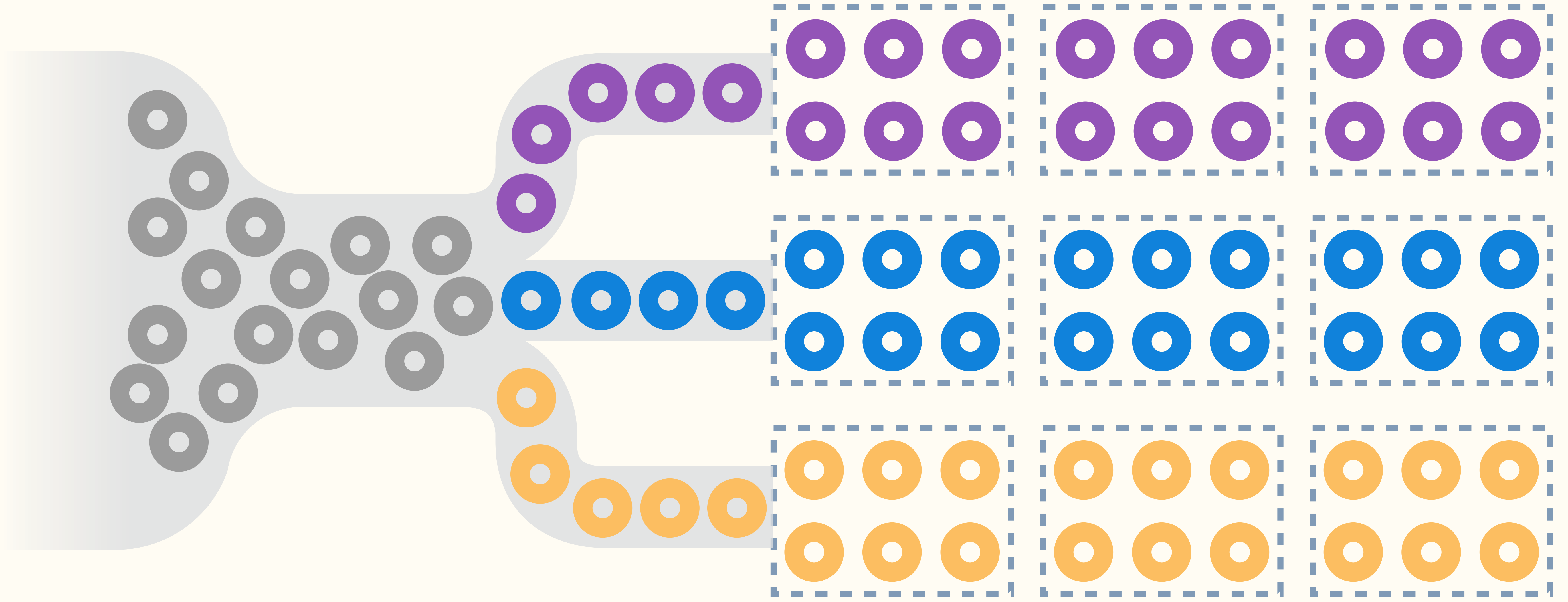
TimescaleDB
(with JOINS)

Older

Older

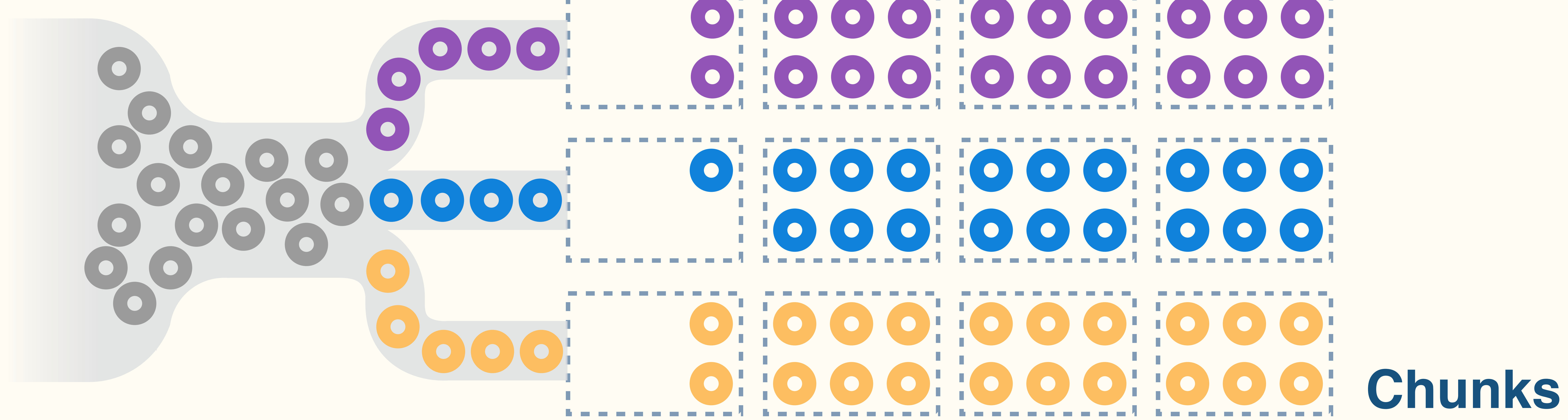
Older

Automatic Space-time Partitioning



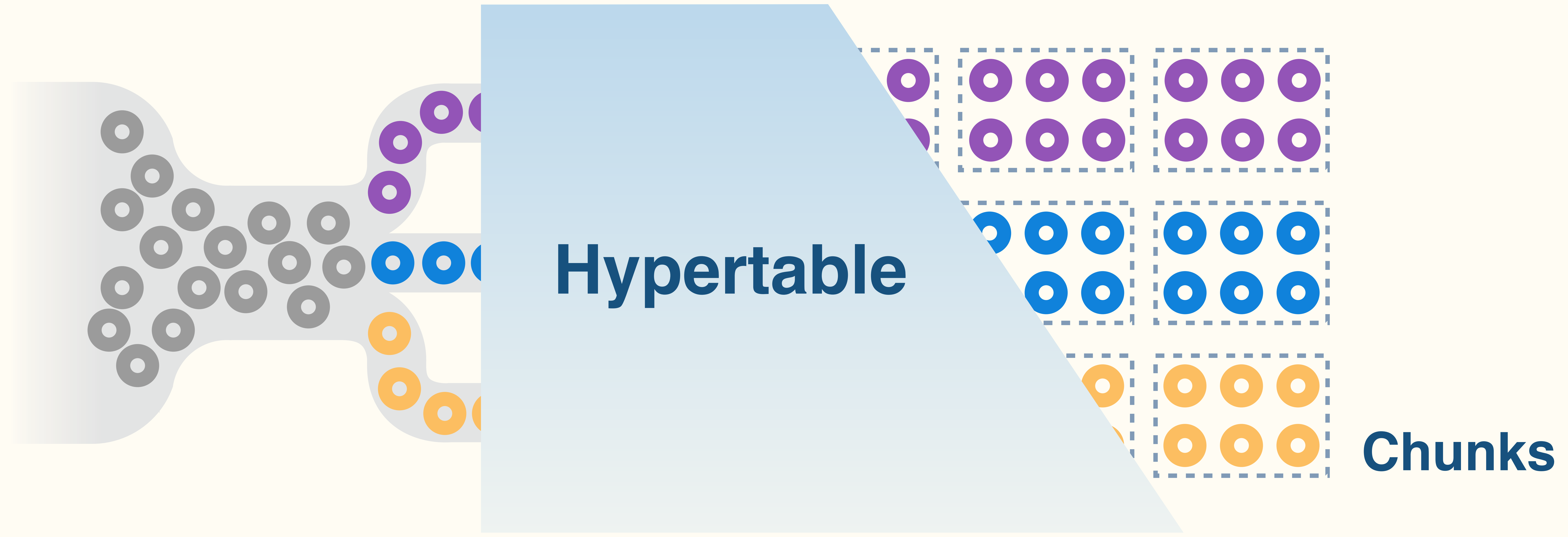
Chunks

Automatic Space-time Partitioning



Automatic Space-time Partitioning

Automatic Space-time Partitioning



Ingest millions of datapoints a second

Scale to 100s of billions of rows

The Extensibility of PostgreSQL

Time-Series Queries **Standard SQL Queries**

Query Planner

Data Management

Storage Layer

PostgreSQL **9.6, 10, 11, …**

[record scratch sound effect]

Automatic Space-time Partitioning

Ingest millions of datapoints a second

Scale to 100s of billions of rows

The Extensibility of PostgreSQL

Time-Series Queries

Standard SQL Queries

Query Planner

Data Management

Storage Layer

PostgreSQL 9.6, 10, 11, …

Yup, it’s pretty big.

Logical Data Definition

A table is a collection of rows each row split into columns

table

row1 col1 col2 col3 •••

row2 col1 col2 col3 •••

row3 col1 col2 col3 •••

Logical Data Definition

Example

Table: Metrics

Columns: time, device_id, sensor1, sensor2

(sensor1 = temp, sensor2 = humidity)

table

row1 col1 col2 col3

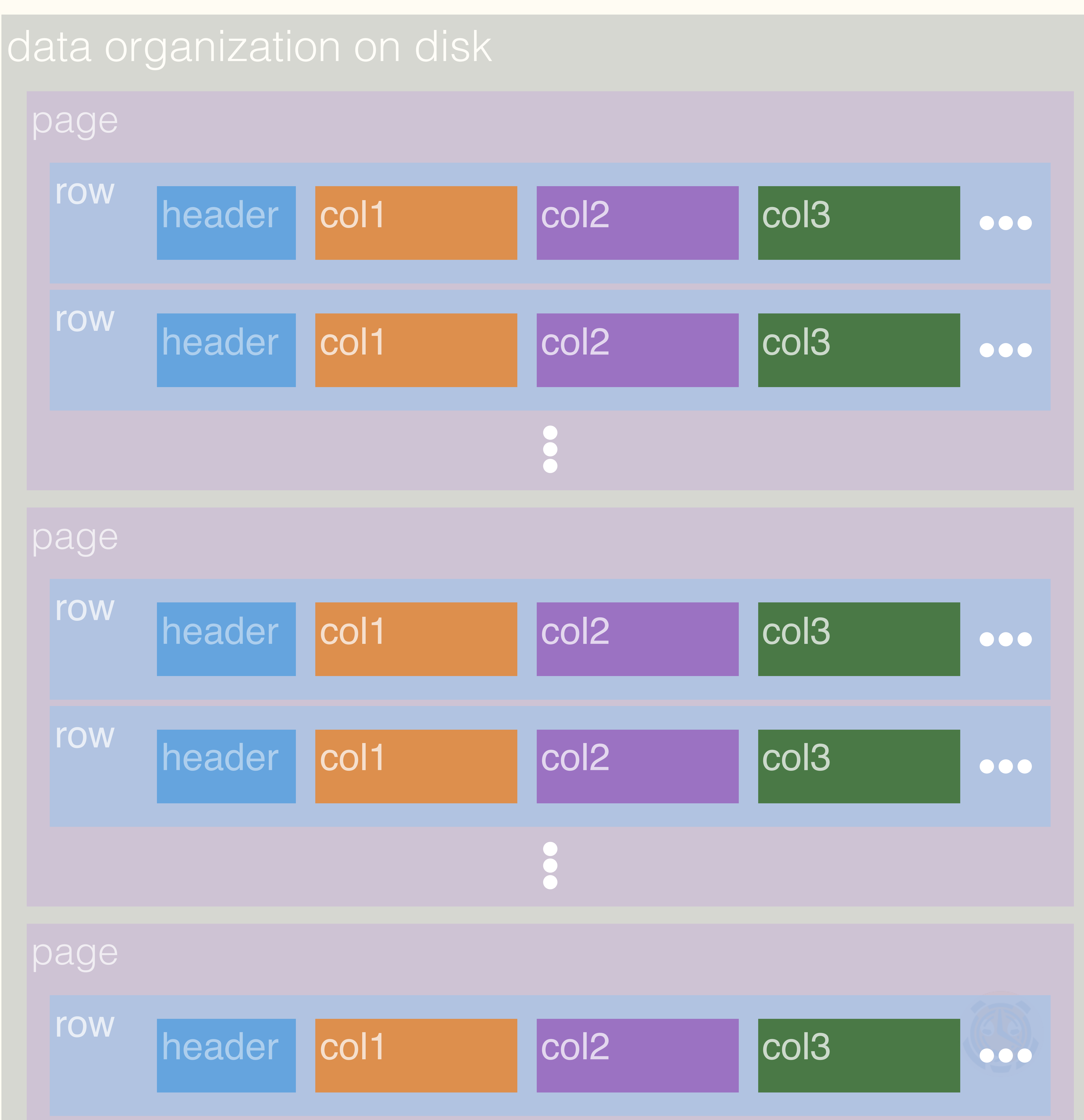
row2 col1 col2 col3

row3 col1 col2 col3

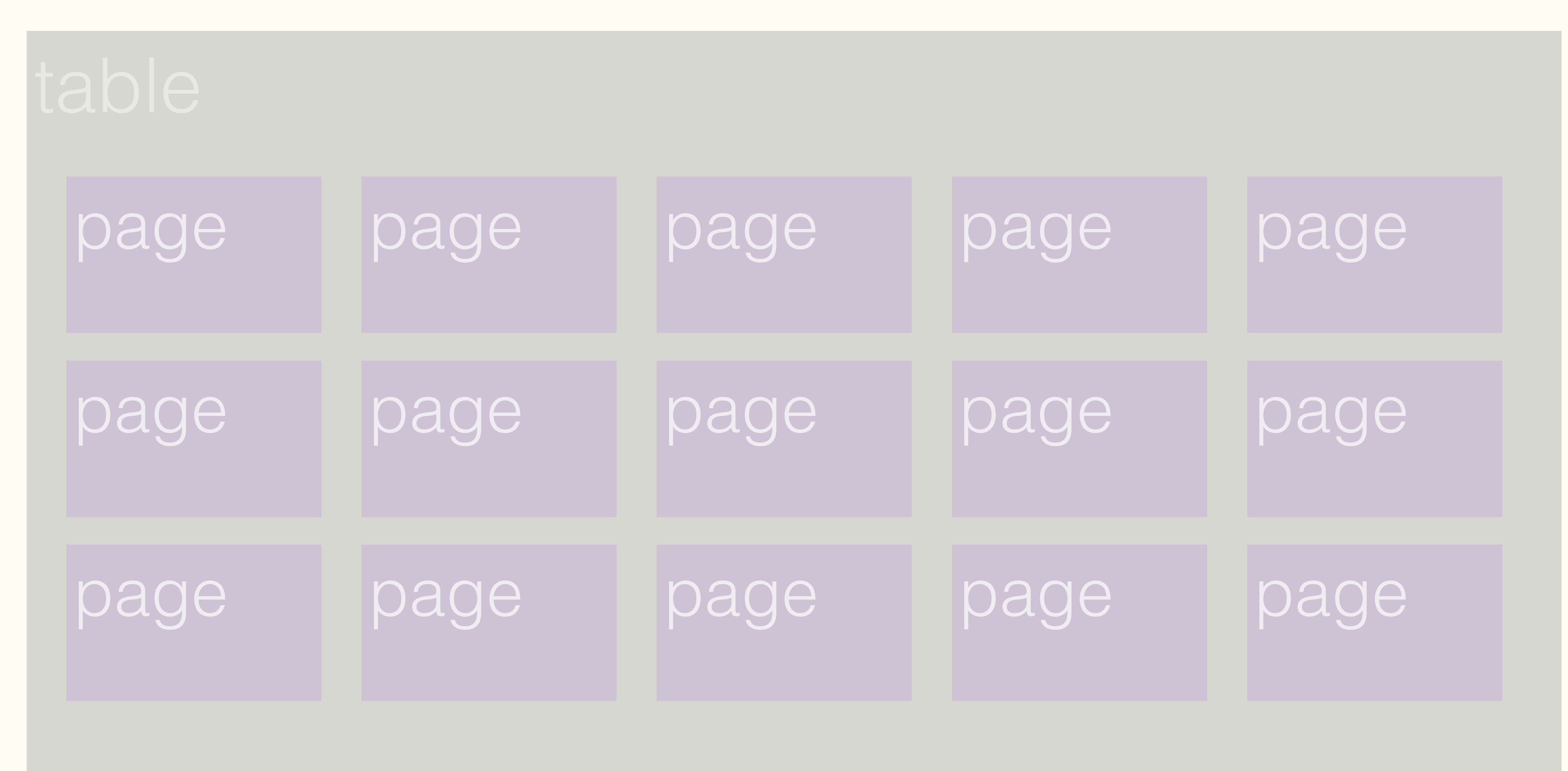
Row Store

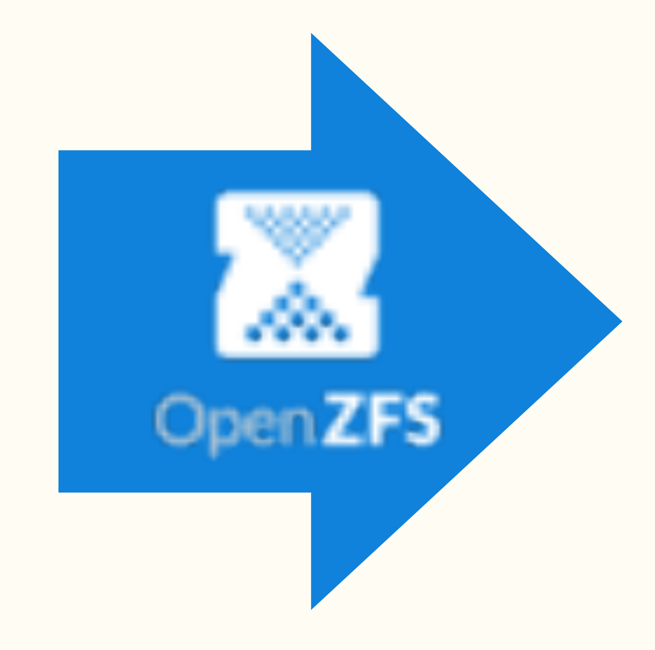
(e.g. Postgres)

- Indexable: get a few rows quickly
- High concurrency for small reads
- Often IO bound on aggregates
- Per row overhead for txn control
- Compresses badly
  - 3-7x using ZFS



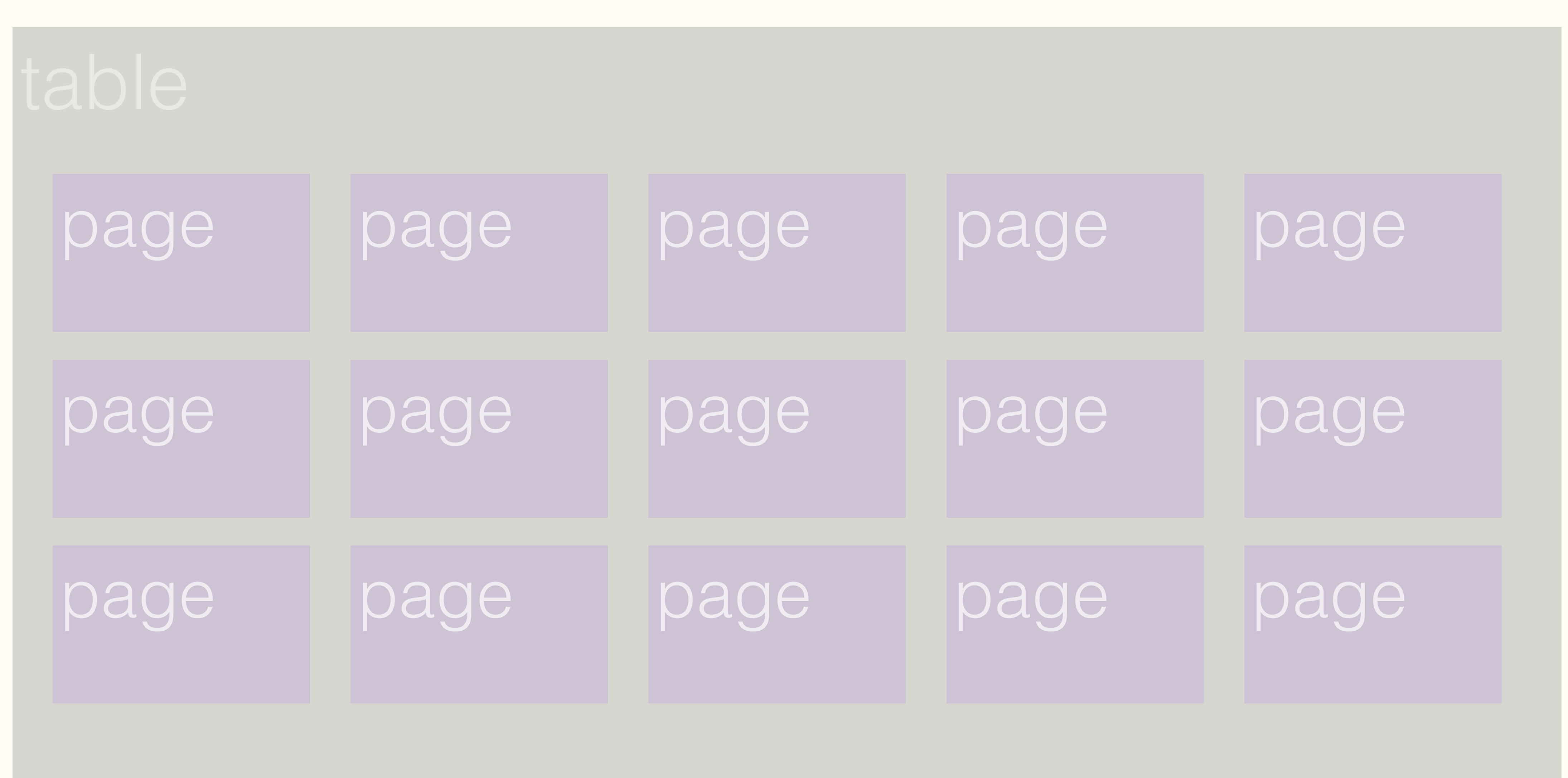
Today: Filesystem Compression

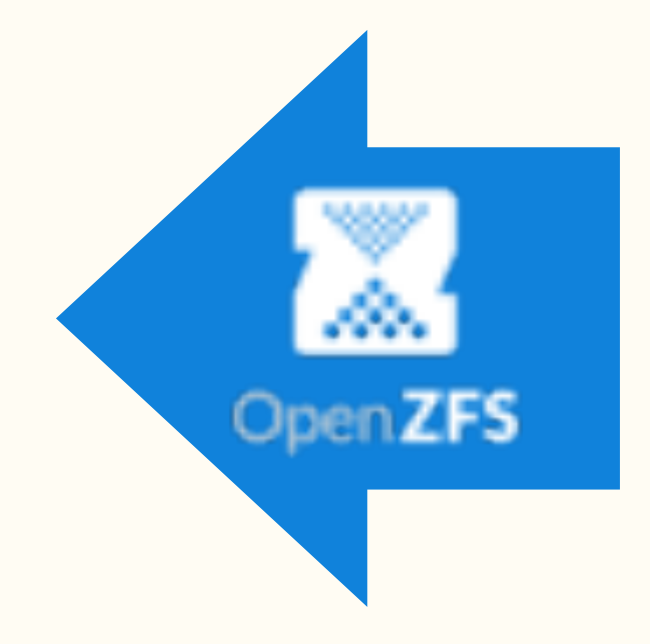




Whole pages are compressed together, no type specificity

Today: Filesystem Compression

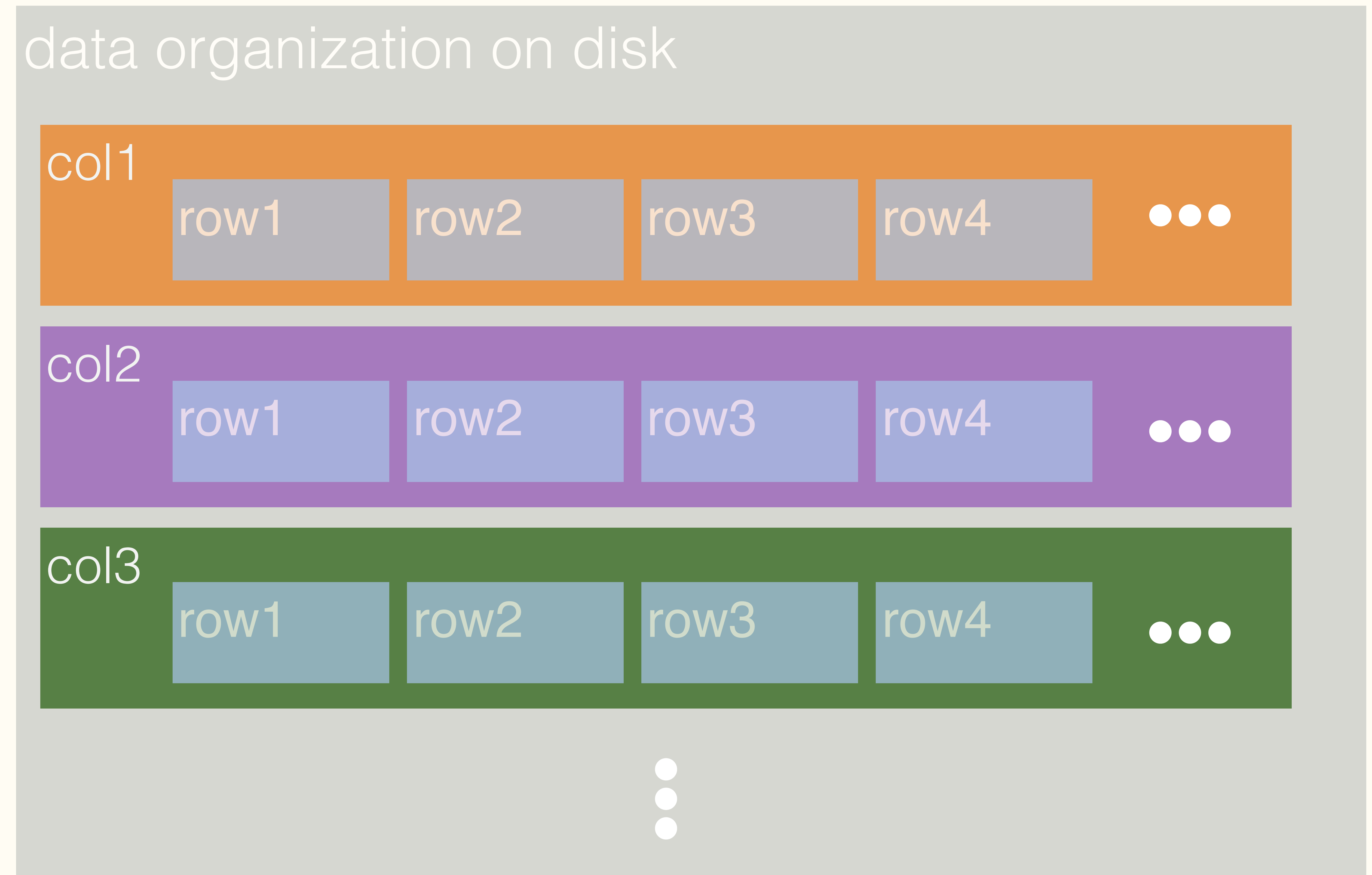




On decompress pages must be accessed with their neighbors.

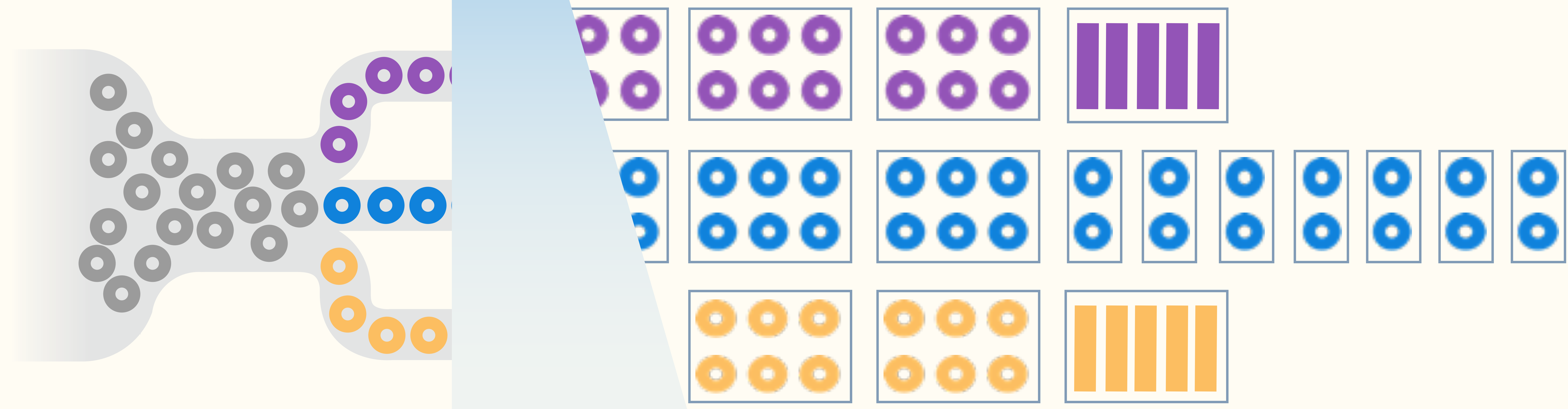
Column Store

- Can scan columns individually
- Efficient aggregates
- Highly compressible
- Reconstructing full row is expensive
  - Per-row updates/deletes expensive
- Indexing is coarse-grained.



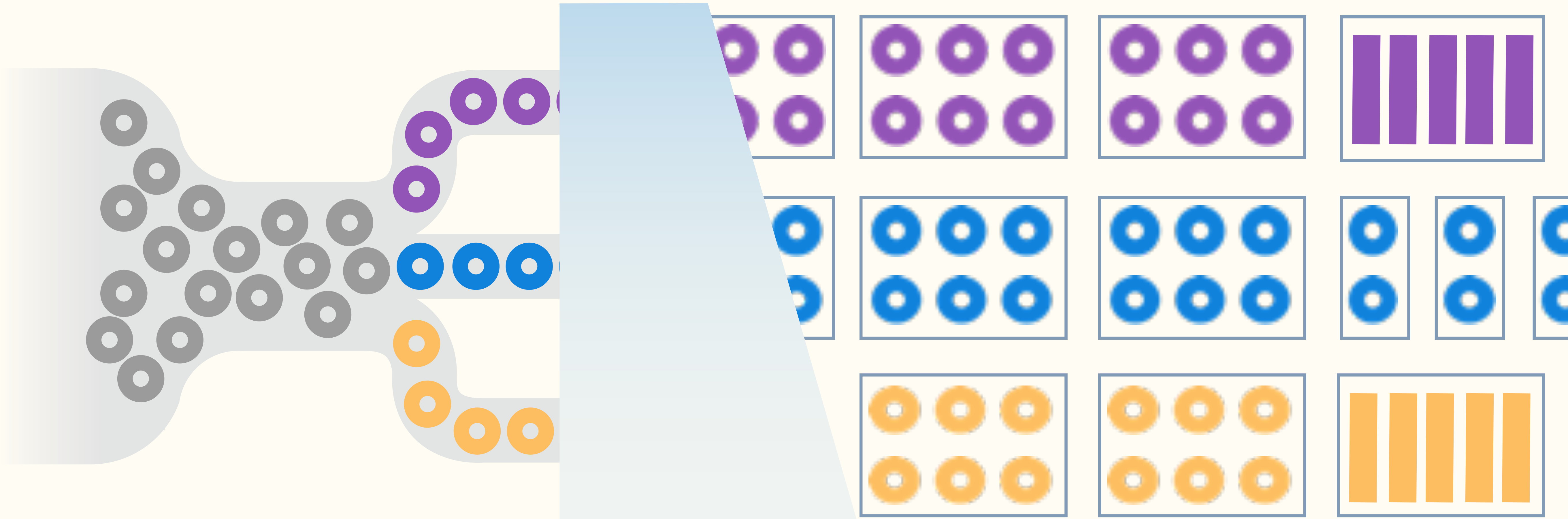
Insight: These properties map well to different needs as data ages

Timescale Transparent Compression

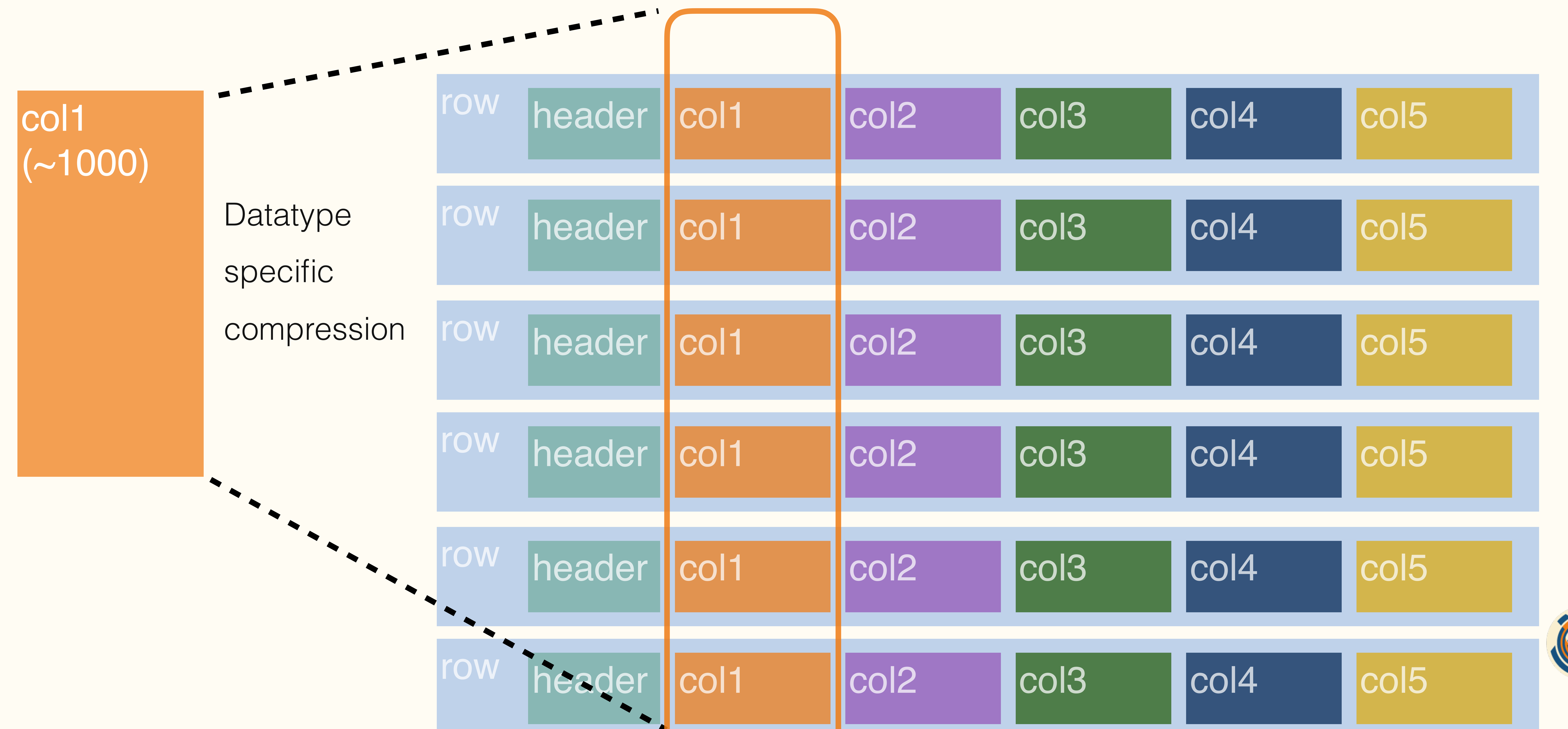


Timescale Transparent Compression: Goals

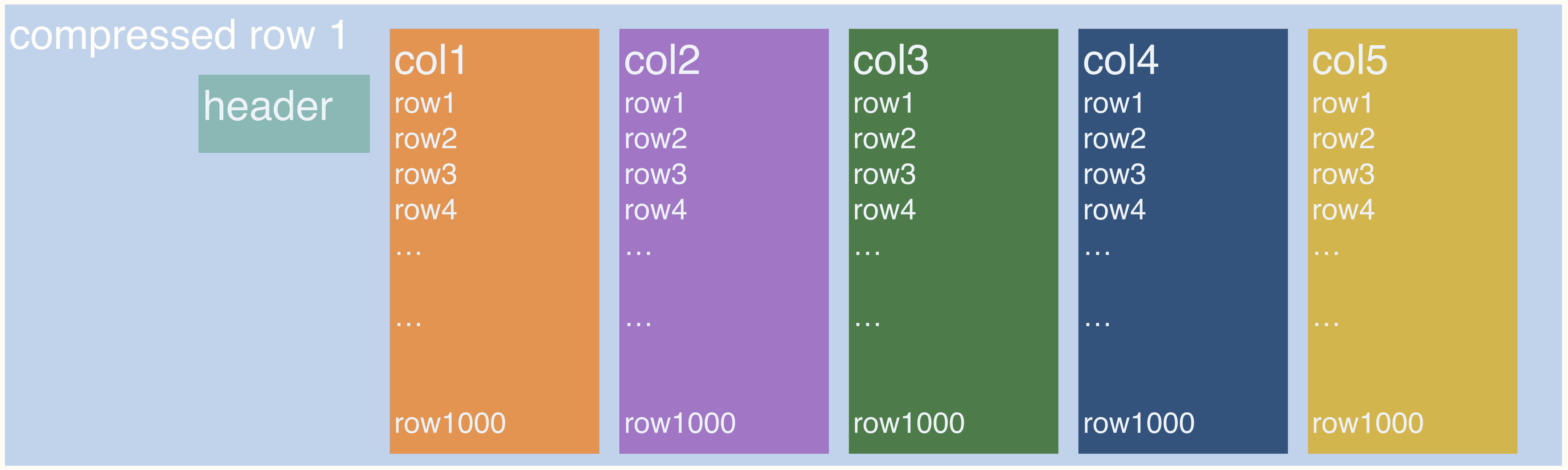
- Reduce storage size
  - TCO + Query Performance
- Maintain ingest rates
- Reuse PG storage layer
- Allow transparent queries over raw and compressed data regions



Hybrid Row - Column Store



Hybrid Row - Column Store



compressed row 2

header

col1
row1001
row1002
row1003
row1004
…
…

col2
row1001
row1002
row1003
row1004
…
…

col3
row1001
row1002
row1003
row1004
…
…

col4
row1001
row1002
row1003
row1004
…
…

col5
row1001
row1002
row1003
row1004
…
…

Example of Compressed Row

Uncompressed

Timestamp	Device_id	temp	humidity
2001-01-01	1	70	40
2001-01-02	1	71	39
2001-01-01	2	170	90
2001-01-02	2	171	91

Compressed

Timestamp	Device_id	temp	humidity
[2001-01-01, 2001-01-02]	[1,1]	[70,71]	[40,39]
[2001-01-01, 2001-01-02]	[2,2]	[170,171]	[90,91]

What happens when columns won't fit on a page?

page

row

header

col1

col2

col3

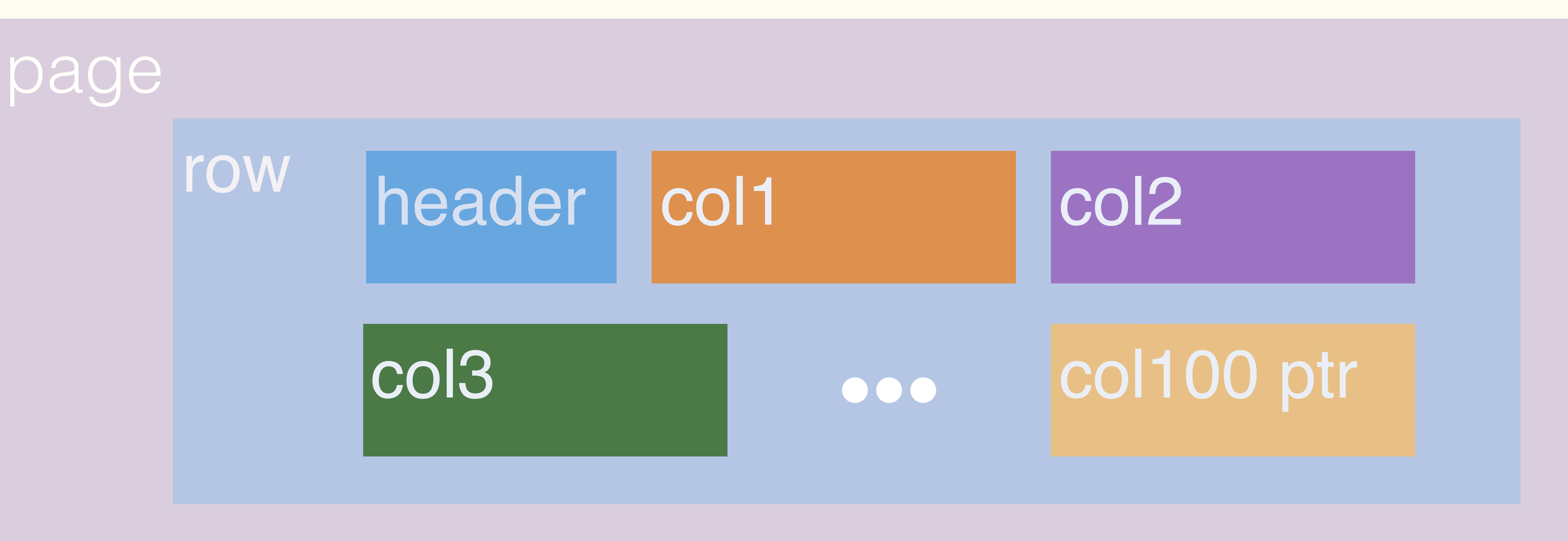
•••

col100

?

col100

The Oversized Attribute Storage Technique



TOAST is a method for storing data out of line

Logical Representation

compressed row 1

header

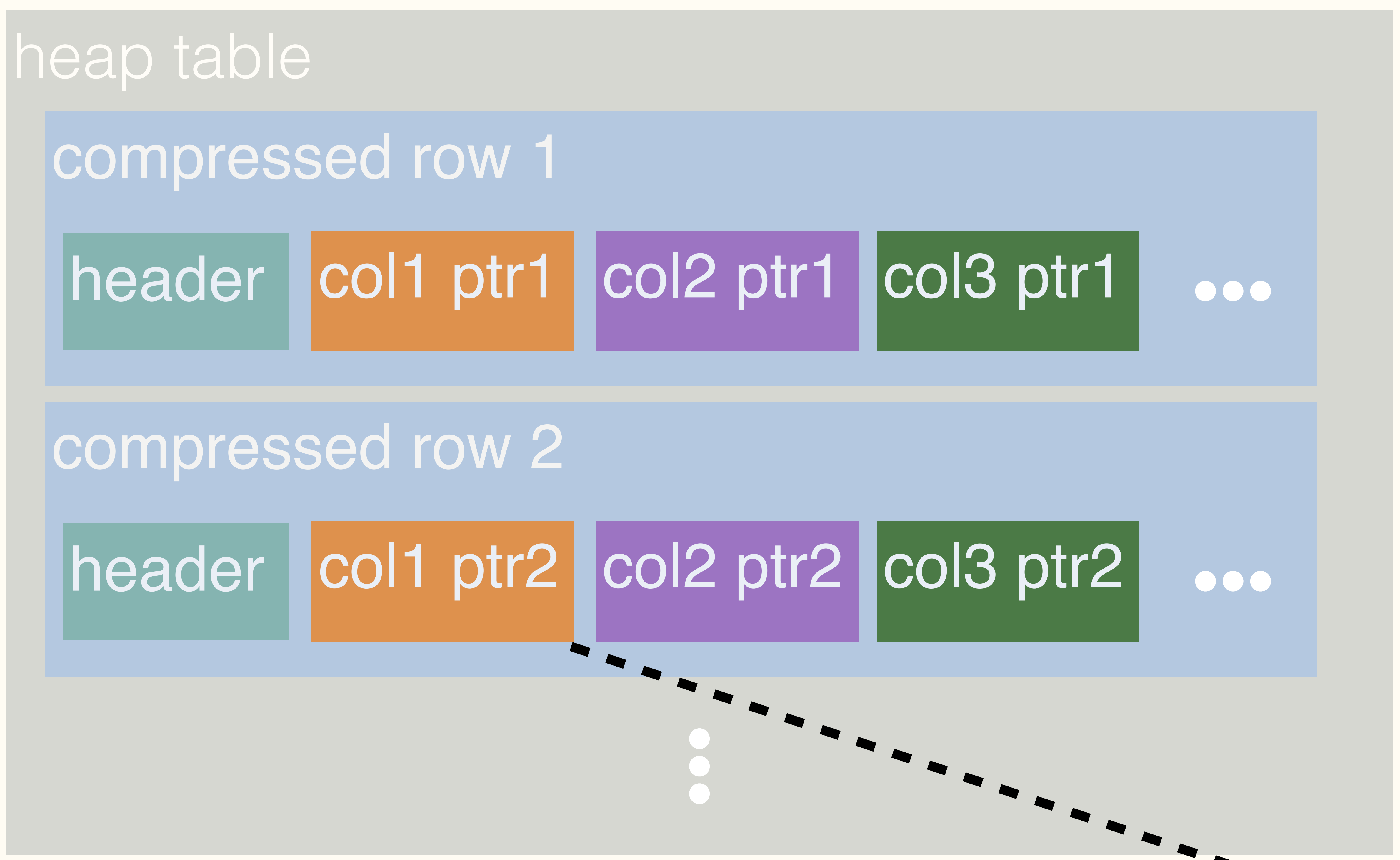
col1	col2	col3	col4	col5
row1	row1	row1	row1	row1
row2	row2	row2	row2	row2
row3	row3	row3	row3	row3
row4	row4	row4	row4	row4
…	…	…	…	…
…	…	…	…	…
row1000	row1000	row1000	row1000	row1000

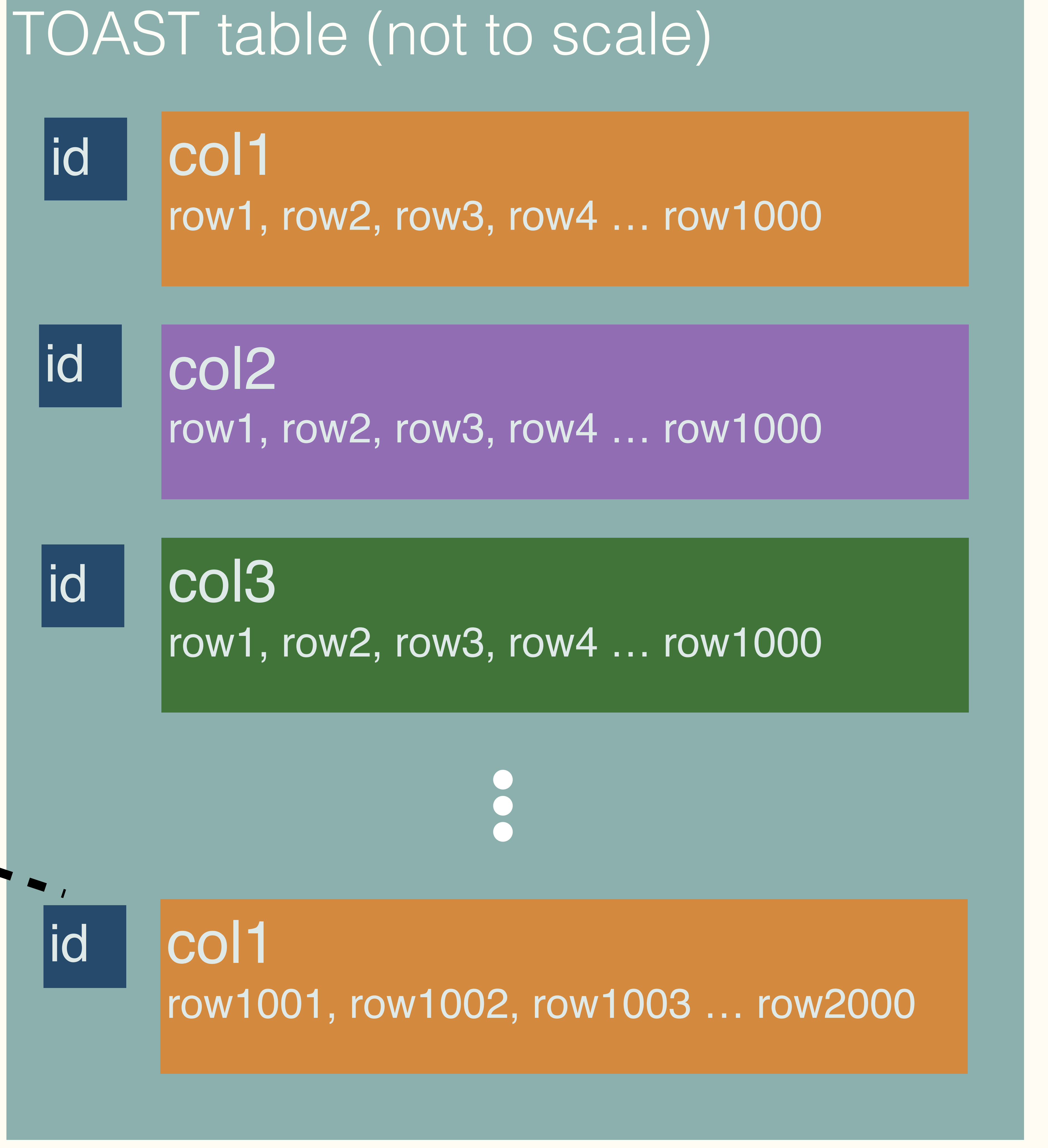
compressed row 2

header

col1	col2	col3	col4	col5
row1001	row1001	row1001	row1001	row1001
row1002	row1002	row1002	row1002	row1002
row1003	row1003	row1003	row1003	row1003
row1004	row1004	row1004	row1004	row1004
…	…	…	…	…
…	…	…	…	…

On Disk Layout





Retrieval of TOASTed values is well indexed and efficient.

Hybrid store is like a column store

- "Heap" is small - consists of mostly pointers to TOAST
- Thus, scanning a single column is efficient

  (Only retrieve TOAST values for that column)
- Compression is efficient - compress items of same type

Data Type Specific Compression

- Integers/timestamps: Delta-Delta + Simple 8B (with RLE)
- Floats: Gorilla
- Everything else: Dictionary/Array + standard PGLZ compression

Automated Compression of Older Chunks

Time

Oldest

Newest

Compression

Chunk

Chunk

```
compress_chunk(chunk);
add_compression_policy(hypertable, older_than);
```

Order of Input Matters

Two devices:

- Device 1: 1,0,1,0,1
- Device 2: 100001, 100002, 100001, 100002

Two orders:

- Order A: 1,100001, 0,100002, 1,100001, 0,10002
- Order B: 1,0,1,0,1,100001, 100002, 100001, 100002

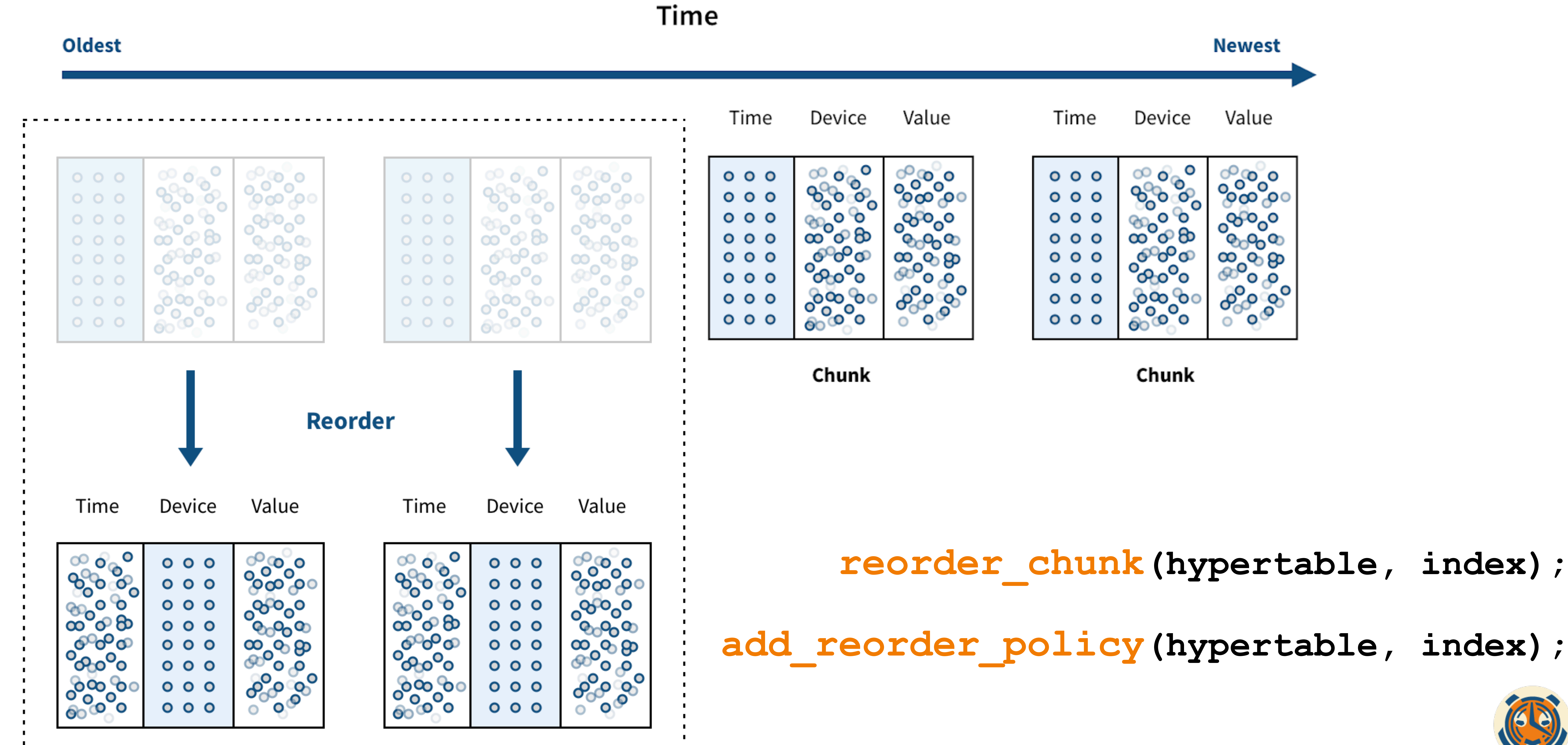
Which one will compress better?

Set Order Before Compressing

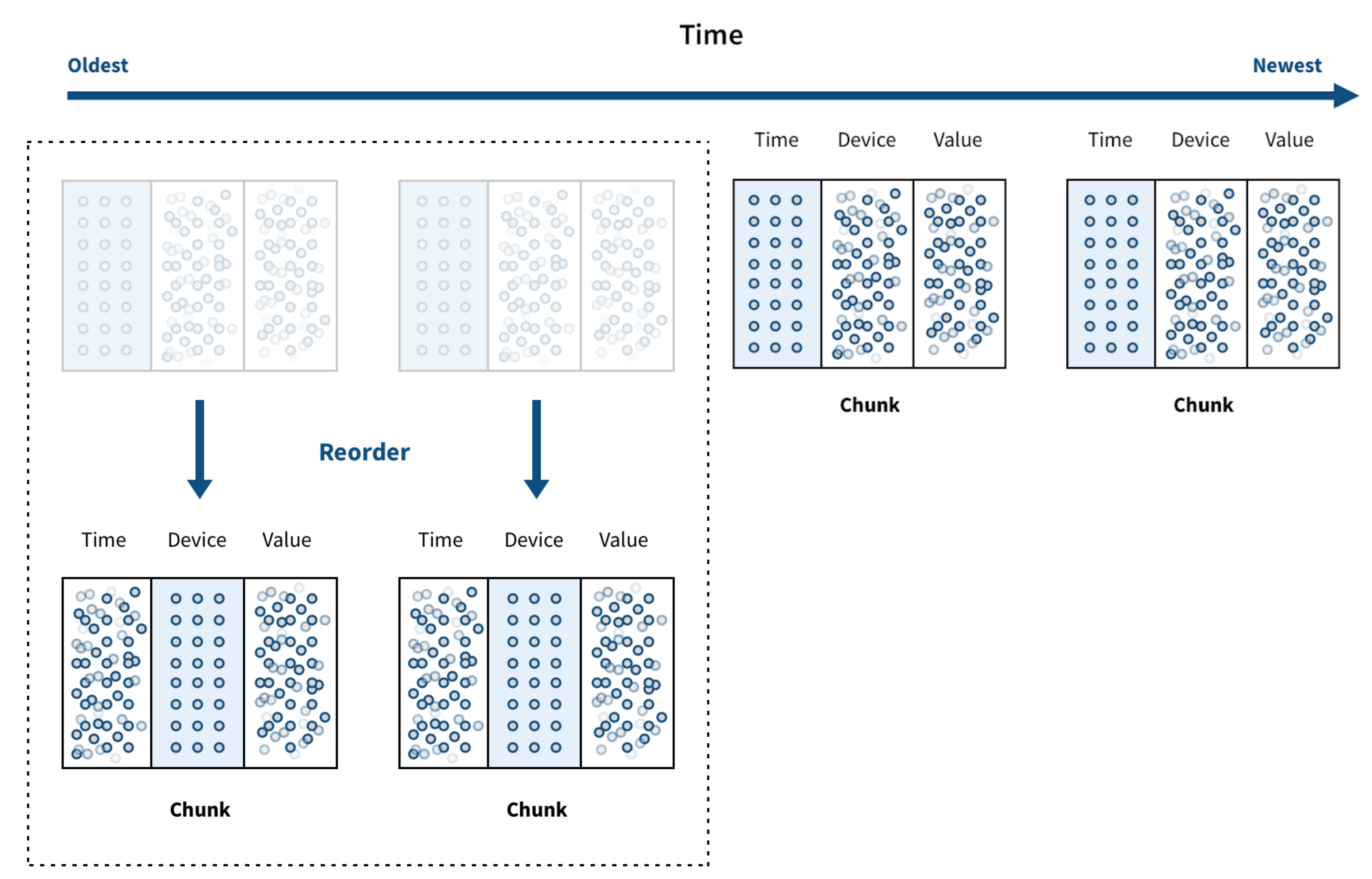
```
=> ALTER TABLE foo SET (
      timescaledb.compress,
      timescaledb.compress_orderby = ‘device_id, time DESC’
      );
```

This also matters for queries!

Automated **data reordering**



Automated **data reordering**



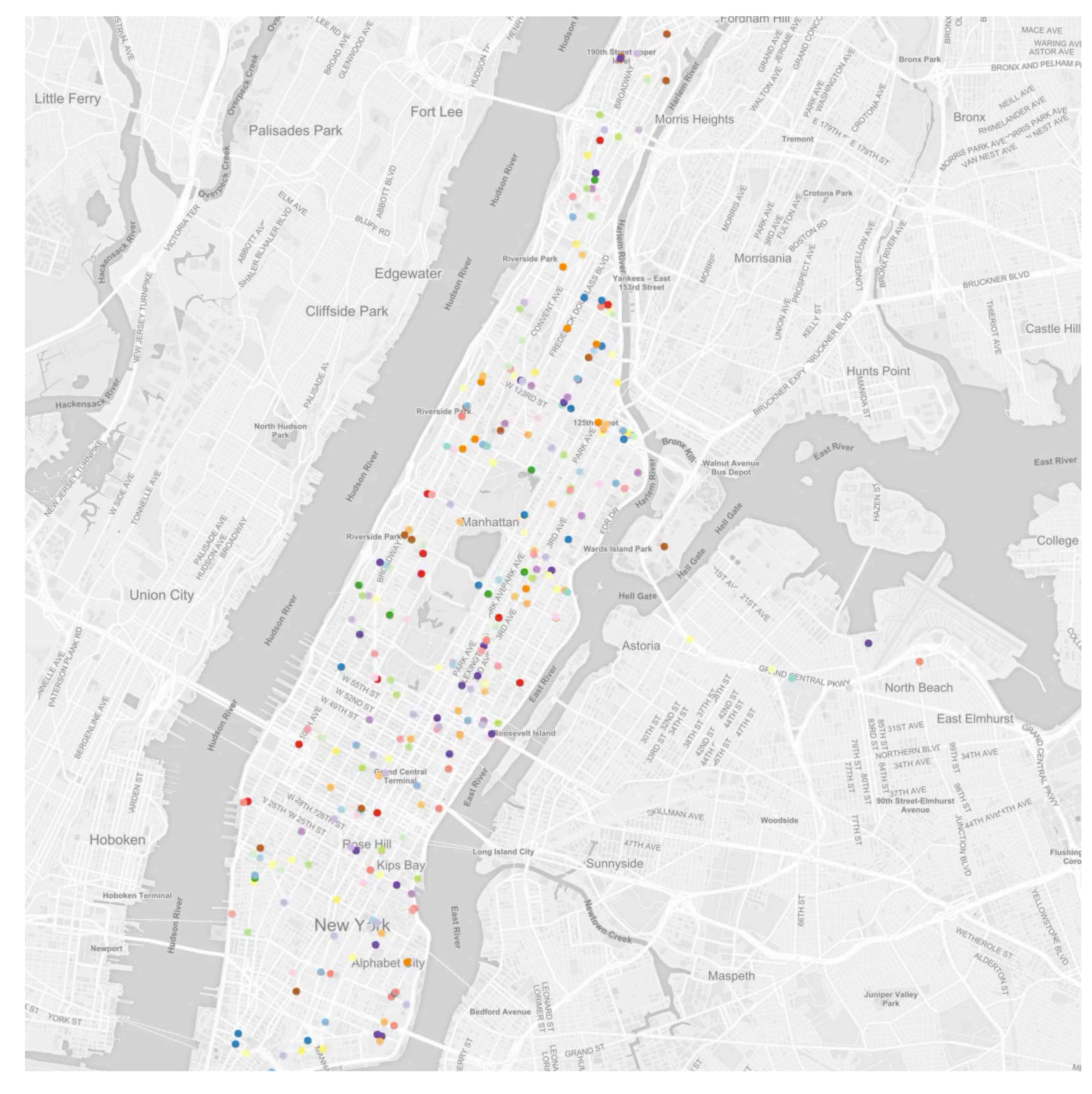

```
=> SELECT * FROM mta WHERE route_id = 'B39';
```

Heap Blocks: exact=20173; Execution Time: 12099 ms

```
=> SELECT reorder_chunk(…,'idx_mta_route');
```

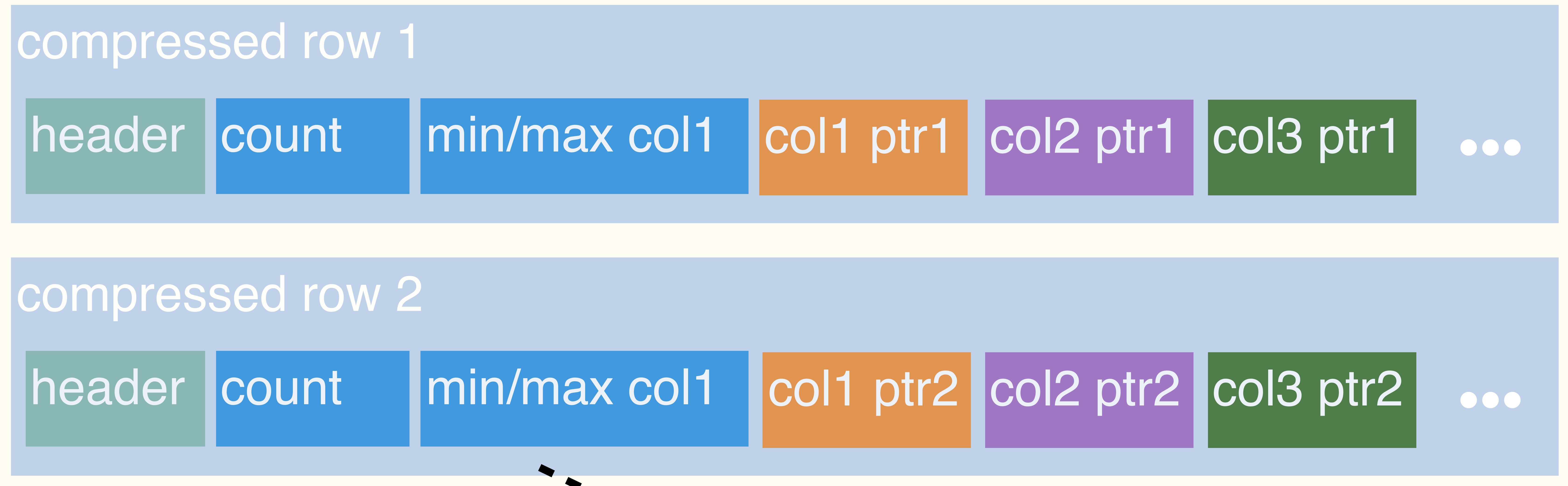
```
=> SELECT * FROM mta WHERE route_id = 'B39';
```

Heap Blocks: exact=250; Execution Time: 3.690 ms



https://github.com/timescale/mta-timescale

Metadata Also Helps with Queries



Don’t fetch and unpack if where clause outside of min/max range.

Segment By A Column

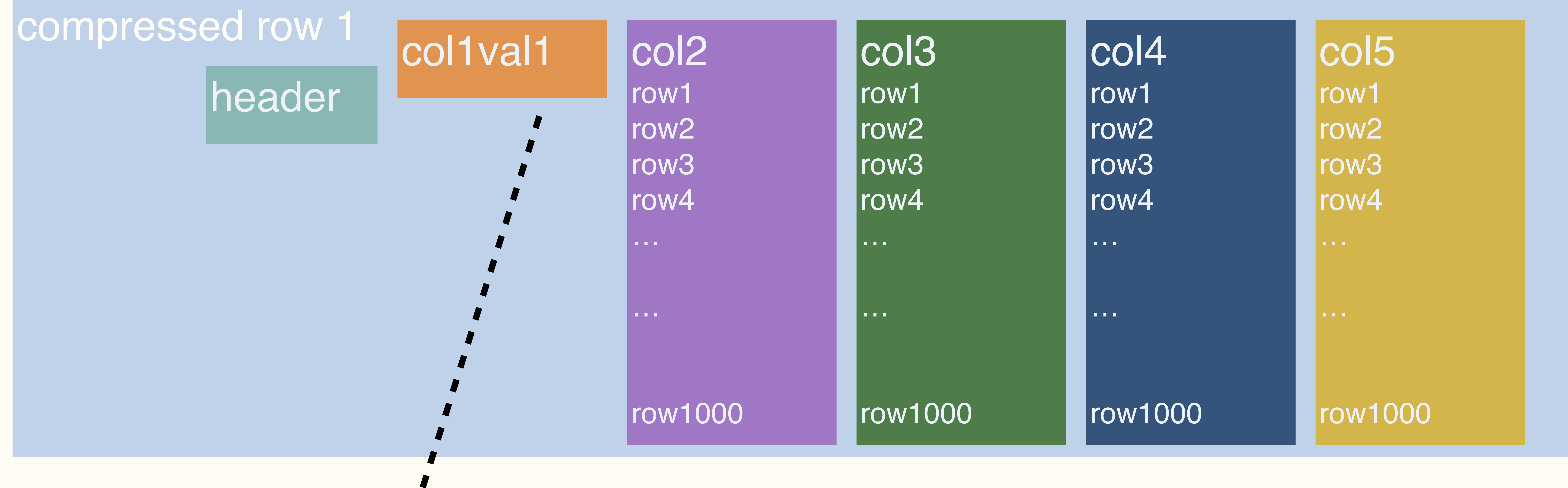
compressed row 1

header

col1	col2	col3	col4	col5
row1	row1	row1	row1	row1
row2	row2	row2	row2	row2
row3	row3	row3	row3	row3
row4	row4	row4	row4	row4
…	…	…	…	…
…	…	…	…	…
row1000	row1000	row1000	row1000	row1000

```
=> ALTER TABLE foo SET (
      timescaledb.compress,
      timescaledb.compress_segmentby = ‘col1’
      );
```

Segment By A Column



Can index/join on segment by columns more easily

Example of Segment By

Uncompressed

Timestamp	Device_id	temp	humidity
2001-01-01	1	70	40
2001-01-02	1	71	39
2001-01-01	2	170	90
2001-01-02	2	171	91

Compressed

Timestamp	Device_id	temp	humidity
[2001-01-01, 2001-01-02]	1	[70,71]	[40,39]
[2001-01-01, 2001-01-02]	2	[170,171]	[90,91]

(VERY PRELIMINARY) Benchmarks

Compression Ratio: 26x

cpu-max-all	44x
cpu-max-all-8	9x
groupby-orderby-limit	0.007x
high-cpu-1	6x
high-cpu-all	1.2x
single-group-by-1-1-2	10x
single-group-by-1-1-1	5x
single-group-by-1-8-1	15x
single-group-by-5-1-12	3x
single-group-by-5-1-1	2x
single-group-by-5-8-1	3x
double-group-by-1	1.5x
double-group-by-5	1.2x
double-group-by-all	1.5x
lastpoint	<1

MVP Release

- No Updates/Deletes/Inserts to chunks that have been compressed
- No ALTER TABLE to hypertables with compressed chunks

Both of these limitations will be relaxed in future releases

- Public beta available now!
- Full MVP release scheduled for October with more features following.

Source code

- github.com/timescale/timescaledb

Join the Community

- slack.timescale.com

TIMESCALE
Powered by aiven

30 days remaining in your free trial | Credits $300 | Billing $0.00 | Welcome Michael Freedman

Help

Logout

Current project:
mike-cf3c

- mike-cf3c
- + Create new project

- Services
- Events
- Members
- VPC
- Billing

Current services

+ Create a new service

Service	Plan	Cloud	Created
tsdb-ha-pair-google-cloud-1 TimescaleDB • Running	Timescale-pro-1024-io-optimized 4 CPU / 15 GB RAM / 1024 GB storage - high availability pair	Timescale / GCP: google-europe-west1 Europe, Belgium	10 minutes ago
grafana-aws-1 Grafana • Running	Dashboard-1 2 CPU / 1 GB RAM	Timescale / AWS: aws-us-east-2 United States, Ohio	19 minutes ago
	escale-basic-512-io-optimized U / 15 GB RAM / 512 GB storage	Timescale / AWS: aws-us-east-2 United States, Ohio	19 minutes ago
	escale-basic-512-io-optimized U / 8 GB RAM / 512 GB storage	Timescale / GCP: google-us-central1 United States, Iowa	20 minutes ago

timescale.com/cloud-promo

TIMESCALE