“Better”

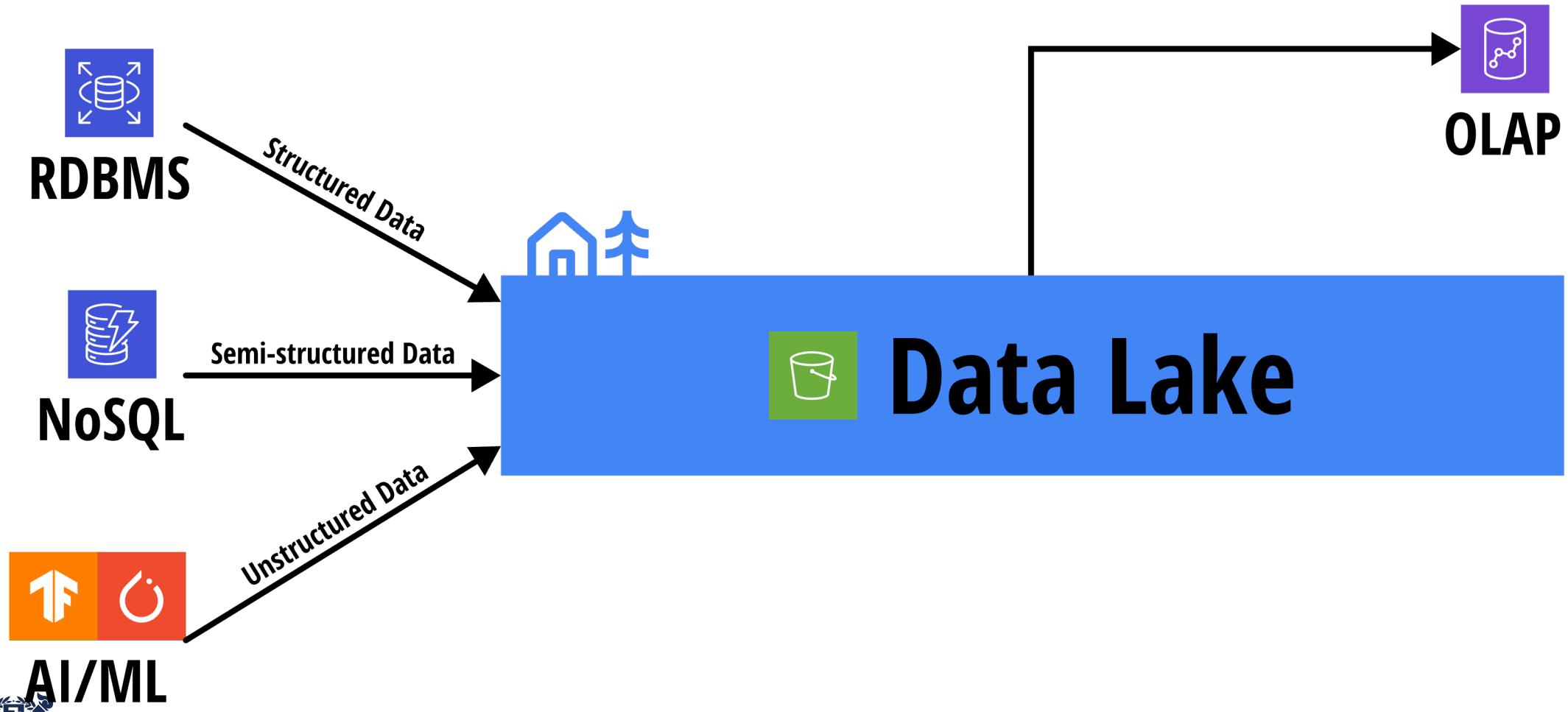
BtrBlocks: Efficient Columnar Compression for Data Lakes

Maximilian Kuschewski, David Sauerwein, Adnan Alhomssi, *Viktor Leis*

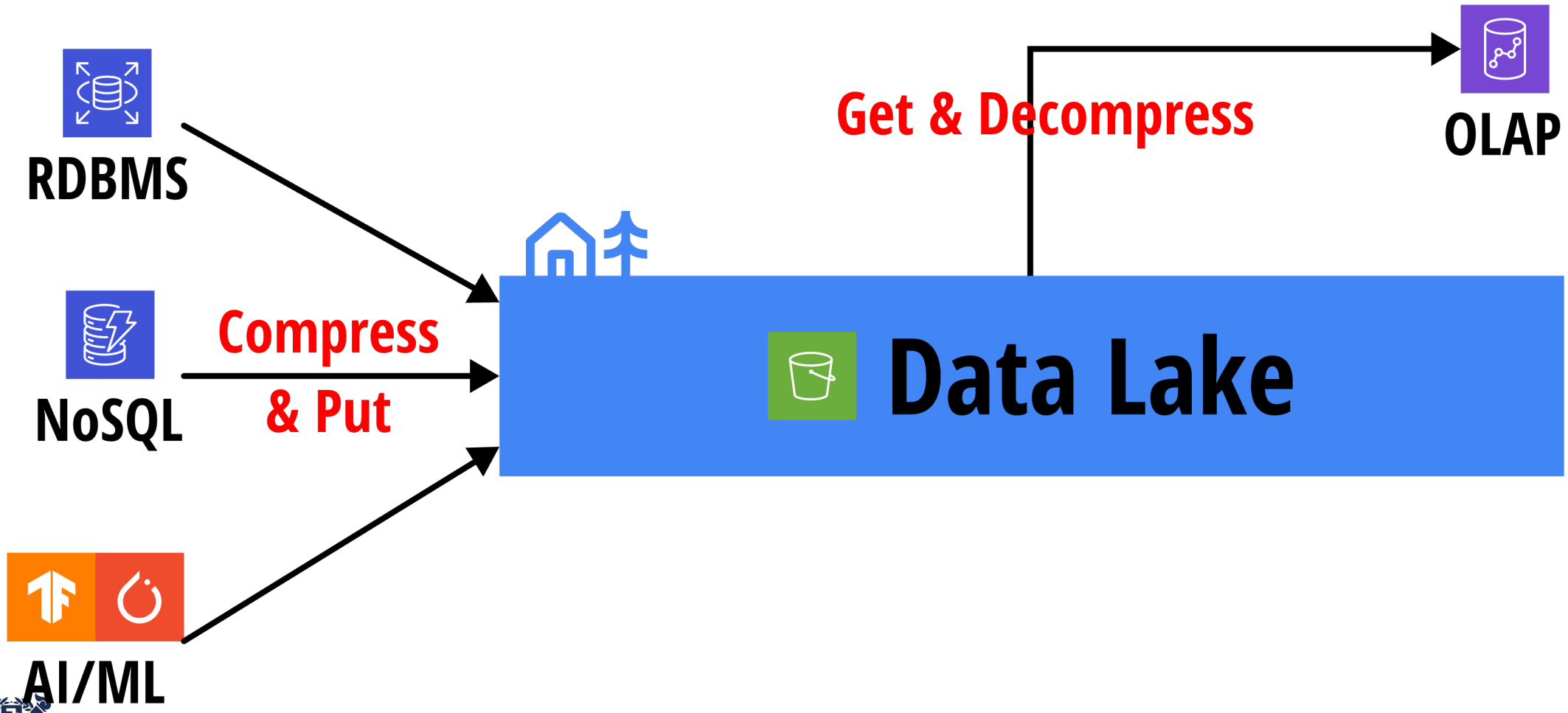
SIGMOD’23

Presenter: Haeram Kim / Aug 10, 2024

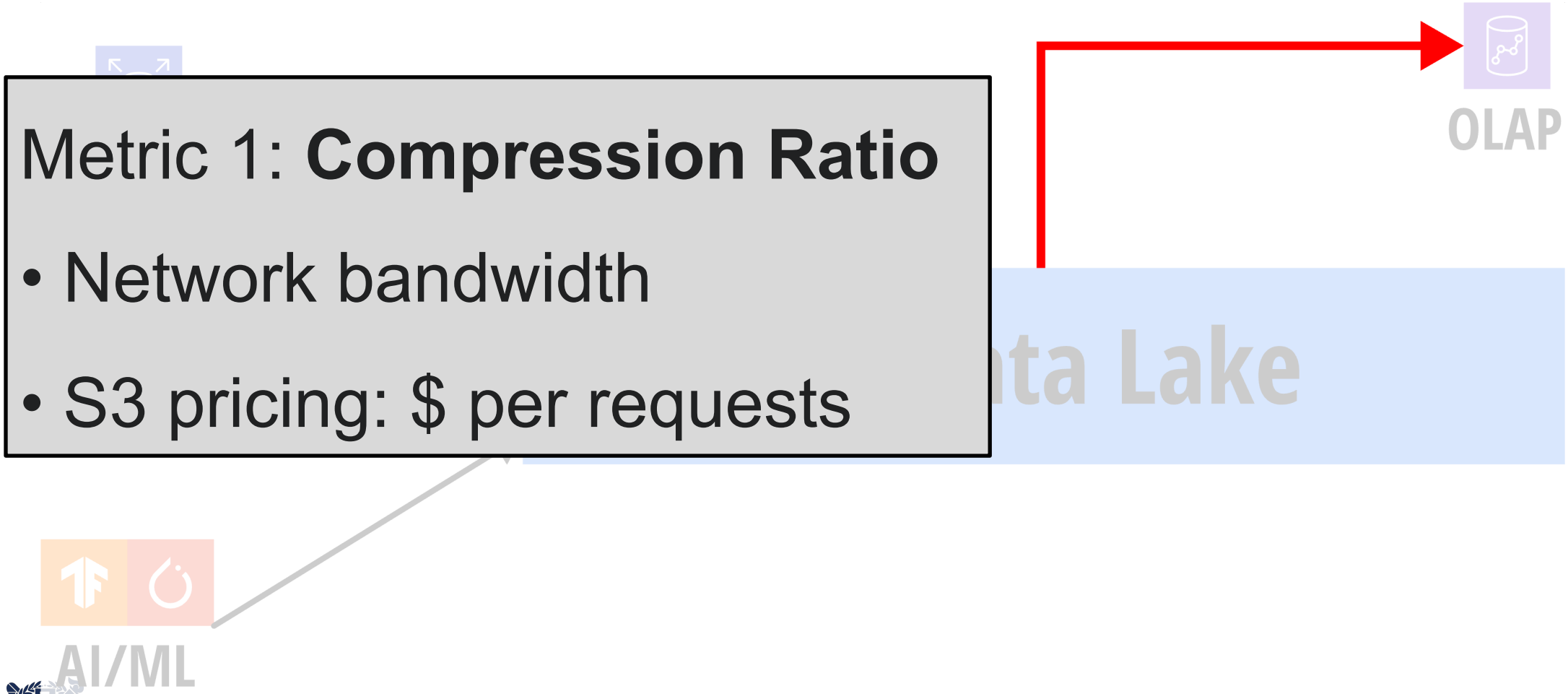
Metrics for the “Better” Compression



Metrics for the “Better” Compression



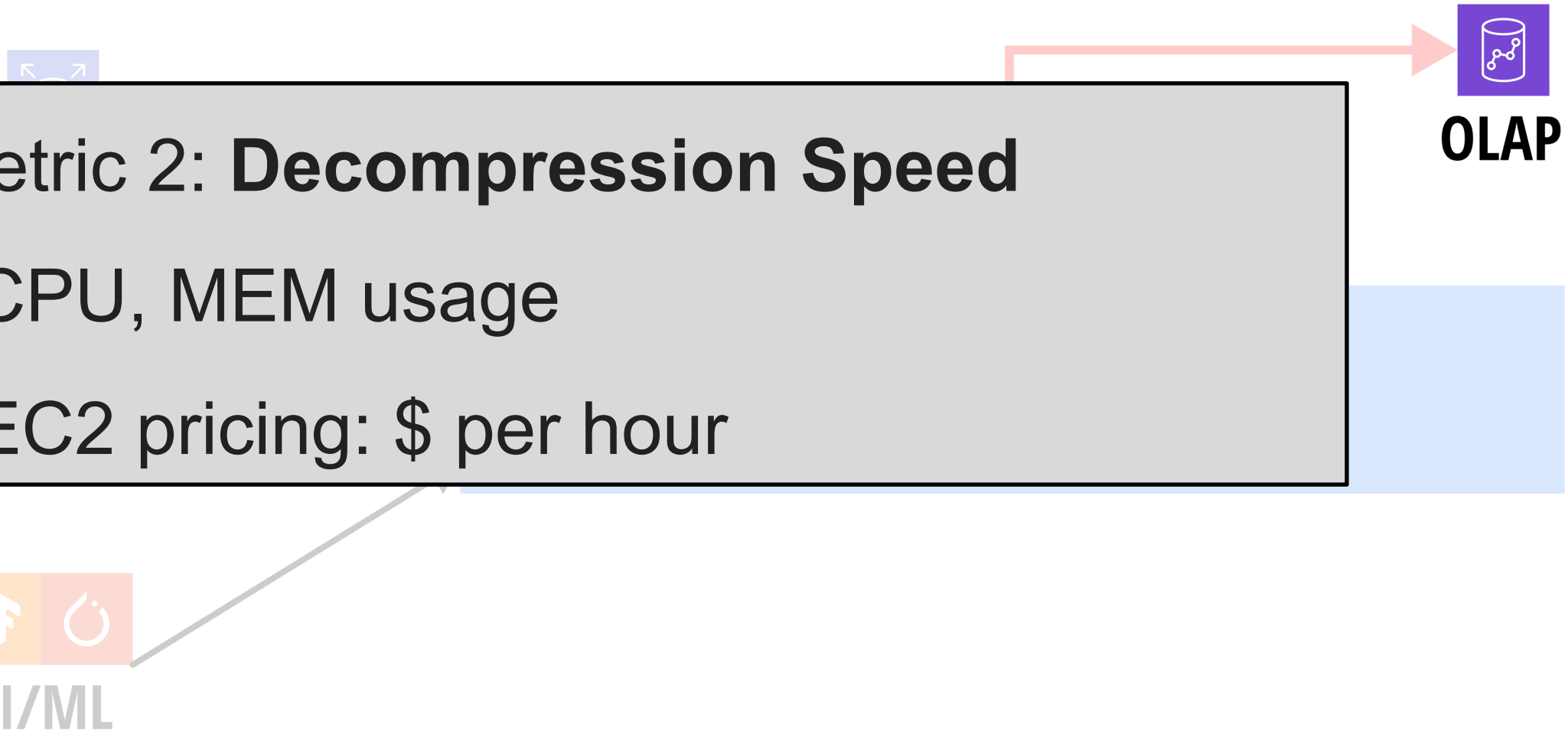
Metrics for the “Better” Compression



Metric 1: **Compression Ratio**

- Network bandwidth
- S3 pricing: $ per requests

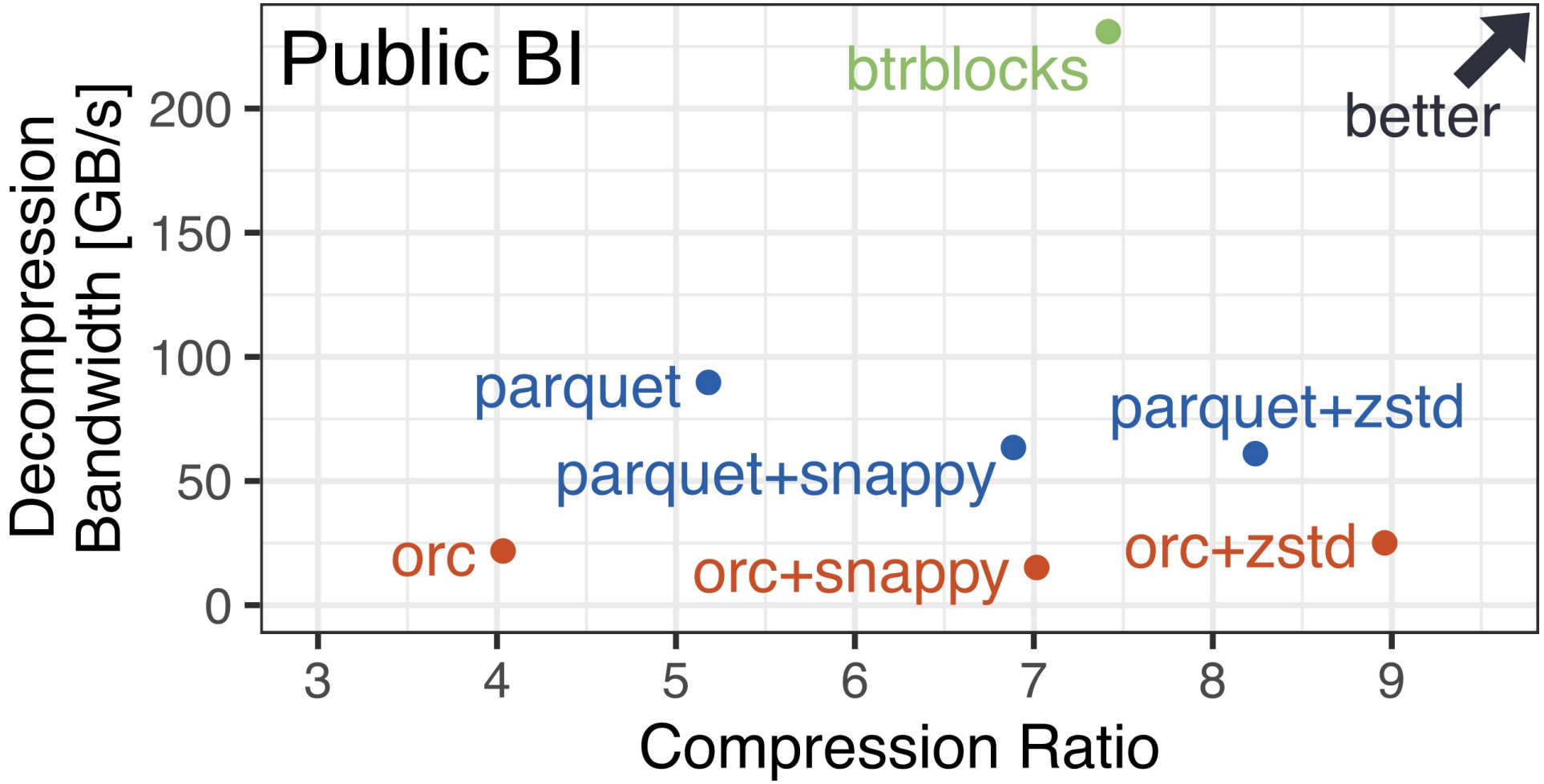
Metrics for the “Better” Compression



Metric 2: **Decompression Speed**

- CPU, MEM usage
- EC2 pricing: $ per hour

Is BtrBlocks “Better”?

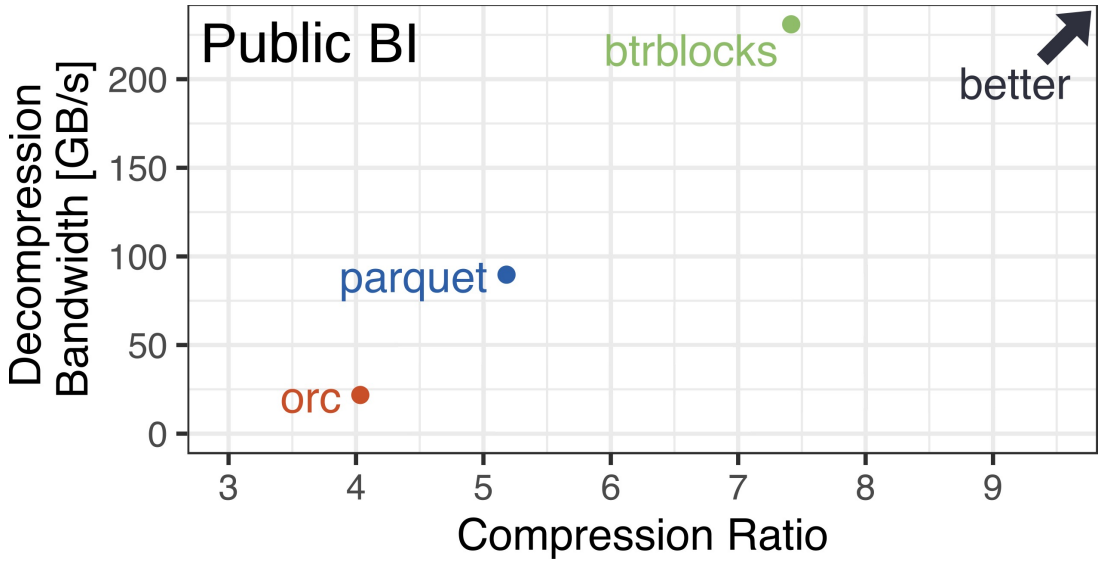


Is BtrBlocks “Better”?

Open-source columnar data format

(**Parquet**, **ORC**)

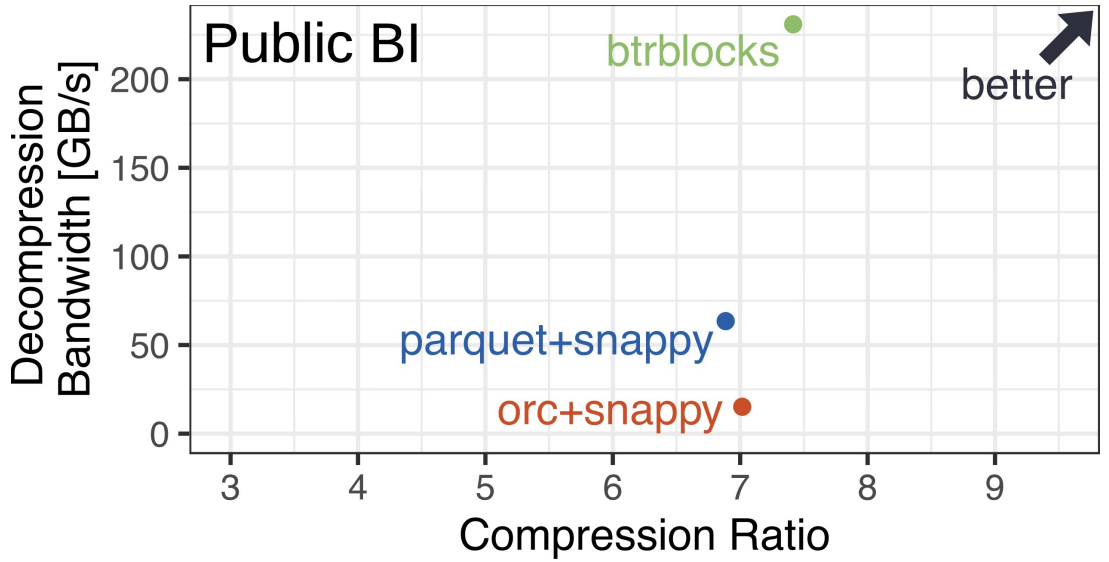
- Low compression ratio
- Low decompression speed



Is BtrBlocks “Better”?

+ General purpose compression scheme (**Snappy**)

- Similar compression ratio
- Low decompression speed



Is BtrBlocks “Better”?

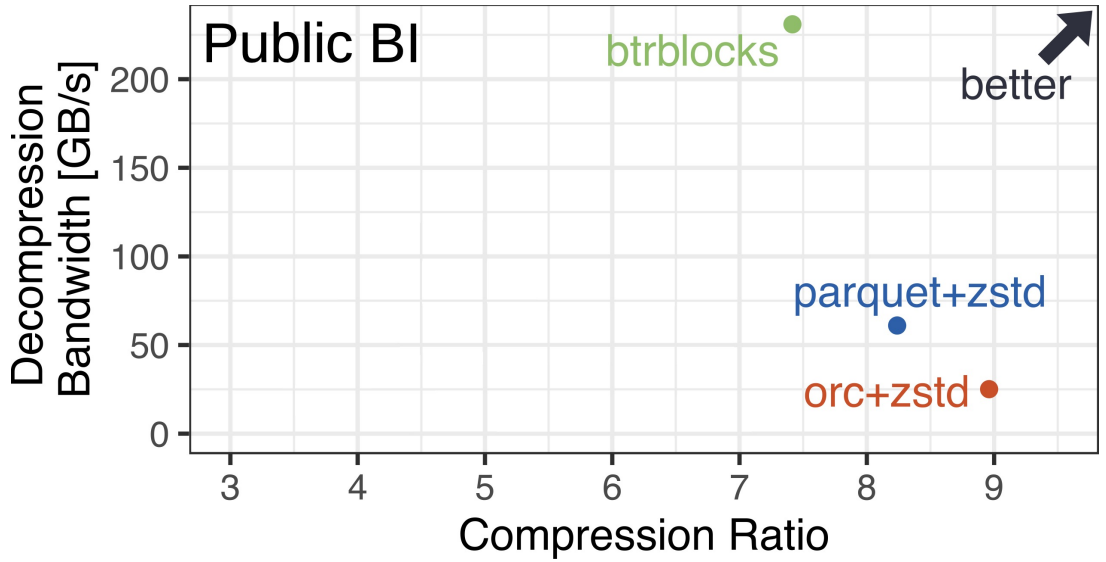
+ Ratio-optimized compression scheme (**Zstandard**)

- High compression ratio
- Low decompression speed

Trade-off between ratio / speed

Expensive in terms of cloud cost

(explained in evaluation)



What is BtrBlocks?

- Compressed **columnar data** format
- **High compression** ratio
- **Fast decompression** speed

Key Idea 1: Scheme Pool

- **Scheme pool** consists of 8 compression schemes (7 existing + **1 new**)
  - **Fast** decompression speed
  - **Conditional** compression ratio (depends on data characteristic)

Key Idea 2: Scheme Selection + Cascade

- **Select scheme** among scheme pool based on **sampling**
- **Recursively** compress data (= *Cascading*) with each selected scheme

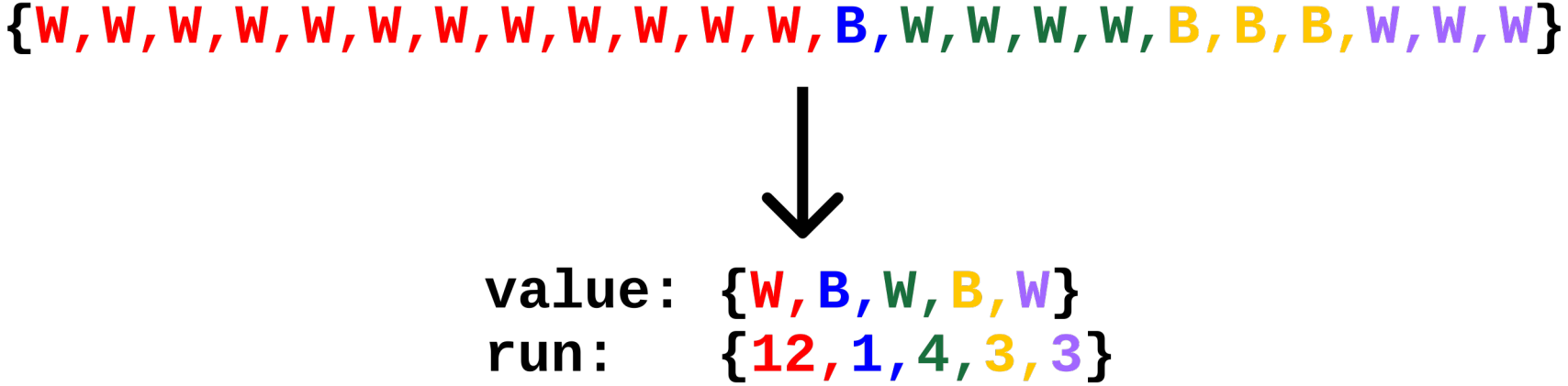
Scheme Pool

No.	Scheme	Type
1	Run Length Encoding (RLE)	*
2	One Value	*
3	Dictionary	*
4	Frequency	*
5	Fast Static String Table (FSST)	String
6	Patched Frame of Reference (PFOR) + Bit Packing (BP)	Integer
7	Roaring Bitmap	Bitmap
8	**Pseudodecimal Encoding (PDE) - *new***	**Float**

Scheme Pool: RLE

RLE
One Value
Dictionary
Frequency
FSST
PFOR + BP
Roaring
PDE

- Run Length Encoding (RLE): Use **value** and the **run length** instead of repeating-value seqeuence



Scheme Pool: One Value

RLE
One Value
Dictionary
Frequency
FSST
PFOR + BP
Roaring
PDE

- One Value: Special case of RLE when all values are **identical**

`{W,W,W,W,W,W,W,W,W,W,W,W,W,W,W,W,W,W,W,W}`

`one-value: W`

- When to use?
  - Best when: All values are identical
  - Worst when: Except for the “Best case”

Scheme Pool: RLE, One Value

RLE
One Value
Dictionary
Frequency
FSST
PFOR + BP
Roaring
PDE

- Decompression: **Copy** the value for ”run length” times
- Compression ratio: Depends on **avg. run length**
  - E.g. All the values are unique (1,2,3,4,5,6,…)
  - E.g. Limited number of values are repeated (1,2,3,1,2,3,…)

Scheme Pool: Dictionary

RLE
One Value
Dictionary
Frequency
FSST
PFOR + BP
Roaring
PDE

- Dictionary: Build a non-duplicated array (*Dictionary*) with the values and replace values with index (*Code*)

```
{USA,USA,USA,USA,Mexico,Canada,Mexico,Mexico,Mexico,Argentina}
```

```
dictionary: {USA,Mexico,Canada,Argentina}
code:       {0,0,0,0,1,2,1,1,1,3}
```

Scheme Pool: Frequency

RLE
One Value
Dictionary
Frequency
FSST
PFOR + BP
Roaring
PDE

- Frequency Encoding: Similar to *Huffman Coding* – Use small bit code for frequent value

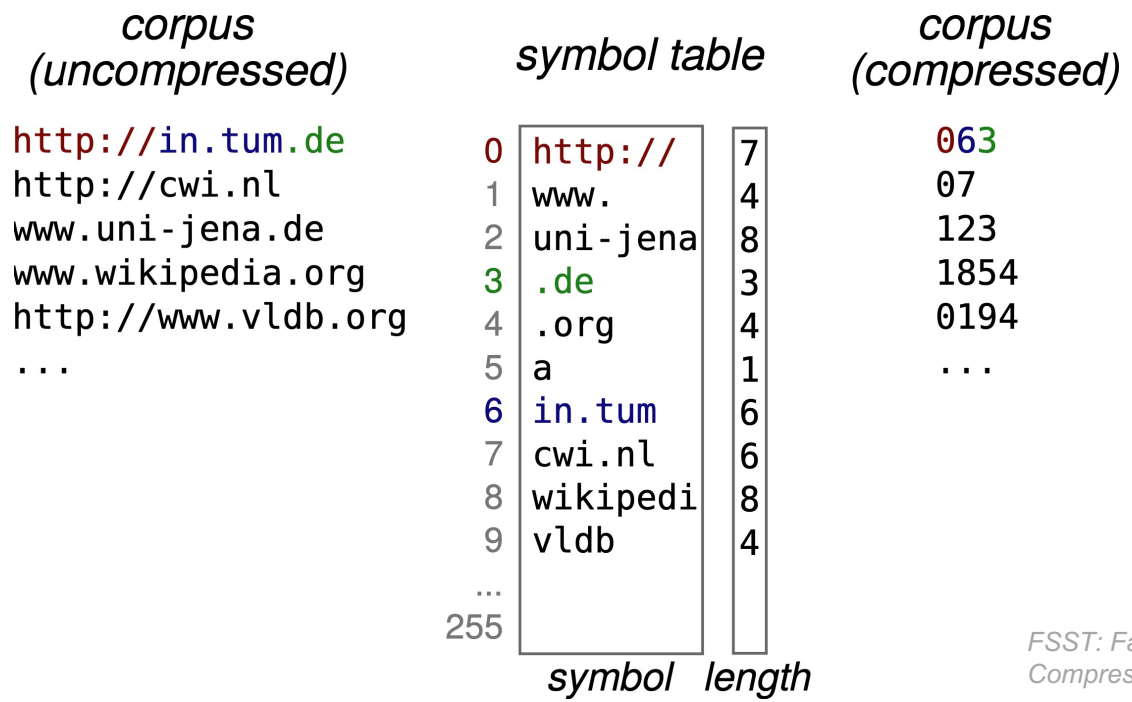
```
{USA,USA,USA,USA,Mexico,Canada,Mexico,Mexico,Mexico,Argentina}
```

```
table: {USA: 0,Mexico: 10,Canada: 110,Argentina: 111}
code:  00001011010101010111
```

Scheme Pool: FSST

- RLE
- One Value
- Dictionary
- Frequency
- **FSST**
- PFOR + BP
- Roaring
- PDE

- Fast Static String Table (FSST): Replace top-255 frequent substring (*Symbol*) with code



FSST: Fast Random Access String Compression (VLDB'20)

Scheme Pool: Dict., Freq., FSST

RLE
One Value
Dictionary
Frequency
FSST
PFOR + BP
Roaring
PDE

- Decompression: **Replace** code with value
- Compression ratio: Depends on **unique value ratio**
  - E.g. Continuously increasing value (1,2,3,4,5,6,…)
  - Except for *FSST*

Scheme Pool: PFOR + BP

RLE
One Value
Dictionary
Frequency
FSST
PFOR + BP
Roaring
PDE

- Frame of Reference (FOR): Save **difference** from the **base**
- Bit Packing (BP): Use **fewer bits** that fits to the value

Scheme Pool: PFOR + BP

- RLE
- One Value
- Dictionary
- Frequency
- FSST
- **PFOR + BP**
- Roaring
- PDE

```
{107,108,110,115,120,125,132,132,131,135}: 10 * int8 = 80bits total
```

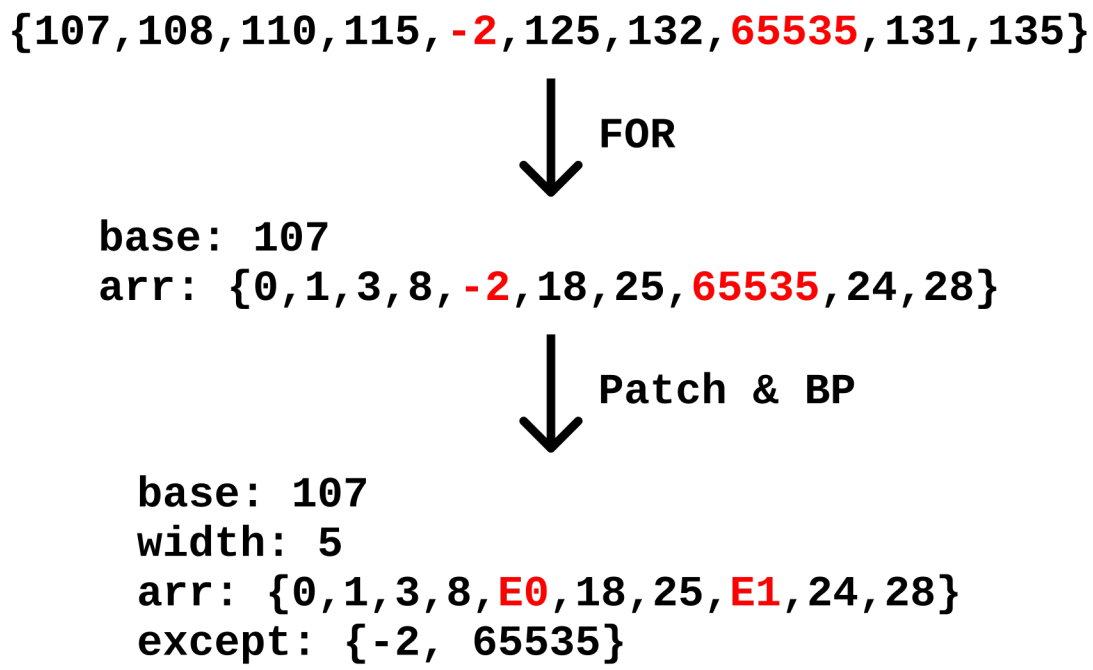

```
base: 107                                : 1 * int8
arr: {0,1,3,8,13,18,25,25,24,28}: 10 * int8 = 88bits total
```


```
base: 107                                : 1 * int8
width: 5                                 : 1 * 3bit
arr: {0,1,3,8,13,18,25,25,24,28}: 10 * 5bit = 61bits total
```

Scheme Pool: PFOR + BP

RLE
One Value
Dictionary
Frequency
FSST
PFOR + BP
Roaring
PDE

- *Patched* FOR: Split *Exceptions* (e.g. negative or super-big value)



Scheme Pool: PFOR + BP

RLE
One Value
Dictionary
Frequency
FSST
PFOR + BP
Roaring
PDE

- Decompression: **Add** *base* to each *value*
- Compression ratio: Depends on **max. – min.**
  - E.g. Continuously increasing value (123456…)

Scheme Pool: Roaring Bitmap

RLE
One Value
Dictionary
Frequency
FSST
PFOR + BP
Roaring
PDE

- Traditional bitmap: Inefficient when the set is sparse
  - Storing 8,388,608 (2^23) to the empty set:
    - Normal set: store only one integer (= 32bit)
    - Traditional bitmap: require 2^23bit (= 1MiB)
- Roaring bitmap: **Partition** and **adoptively switch** normal set and traditional bitmap for each part based on the cardinality

Scheme Pool: Roaring Bitmap

RLE
One Value
Dictionary
Frequency
FSST
PFOR + BP
Roaring
PDE

- When to use: encoding **NULL**s

```
{2,4,8,NULL,16,32,NULL,NULL,64,NULL,...}
```

```
data:    {2,4,8,16,32,64,...}
nullmap: {0,0,0,1,0,0,1,1,0,1,...}
         (Encode nullmap with Roaring Bitmap)
```

Scheme Pool: PDE *(new)*

RLE
One Value
Dictionary
Frequency
FSST
PFOR + BP
Roaring
PDE

- Pseudodecimal Encoding (PDE): **Integer tuple** as a floating point number

```
        Significant          Exponent       [Sign,Exp]
3.25 =  325          * 10^(-2         ) -> [325, 2  ]
```

Scheme Pool: PDE *(new)*

RLE
One Value
Dictionary
Frequency
FSST
PFOR + BP
Roaring
PDE

- Not all values are encodable:
  - Infinite, Not-a-Number (NaN)
  - Precision out of range (max exponent 22)
- These are stored as *Exception*
  - *Exception* value is marked using reserved exponent 23

Scheme Pool: PDE *(new)*

- RLE
- One Value
- Dictionary
- Frequency
- FSST
- PFOR + BP
- Roaring
- **PDE**

- Complete example

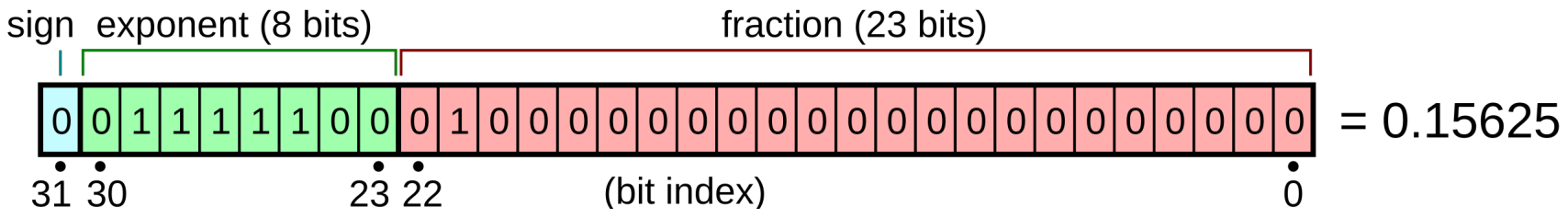
```
{0.99,3.25,+INF,3.14,5.5e-42,18.264,...}
```

```
significant: {99,325,0,314,0,18264,...}
exponent:    {2,2,23,2,23,3,...}
exception:   {+INF,5.5e-42}
```

Scheme Pool: PDE *(new)*

- RLE
- One Value
- Dictionary
- Frequency
- FSST
- PFOR + BP
- Roaring
- **PDE**

- Why PDE? Integer has more chance to be compressed than floating point
  - E.g. exponent bits are located at higher position (index 23 ~ 30 for float type), and it makes BP hard to be applied



https://en.wikipedia.org/wiki/IEEE_754

Scheme Pool: PDE *(new)*

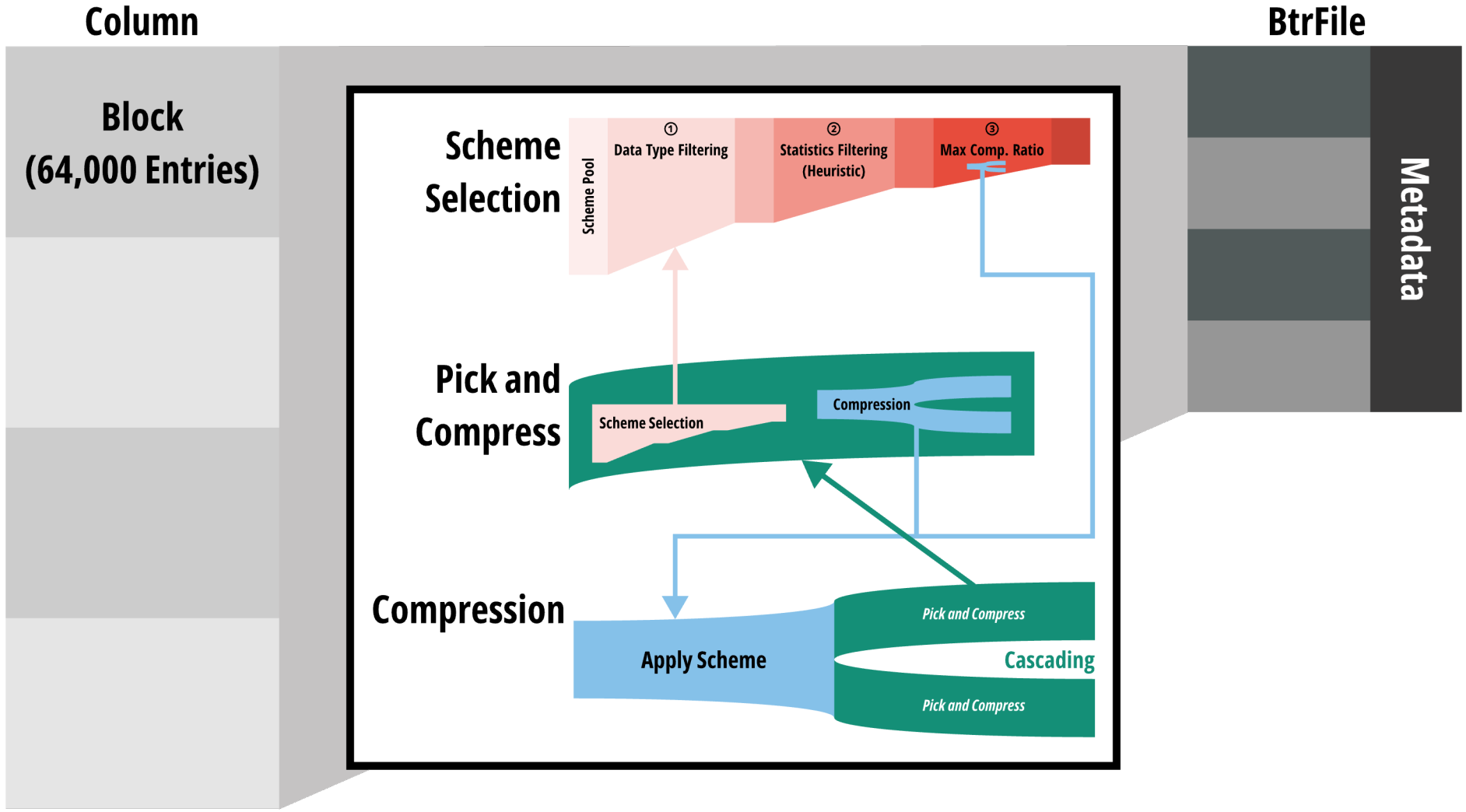
RLE
One Value
Dictionary
Frequency
FSST
PFOR + BP
Roaring
PDE

- Decompression speed: Depends
  - Fast for normal cases: **Multiplying** *significant* with 10^*exponent*
  - Slow when **exceptions are too many** : Exception handling overhead
- Compression ratio: Depends on **next integer compression scheme**
  - PDE itself is not compression (64bit double -> 32bit integer * 2)
  - Must be used with another integer compression scheme

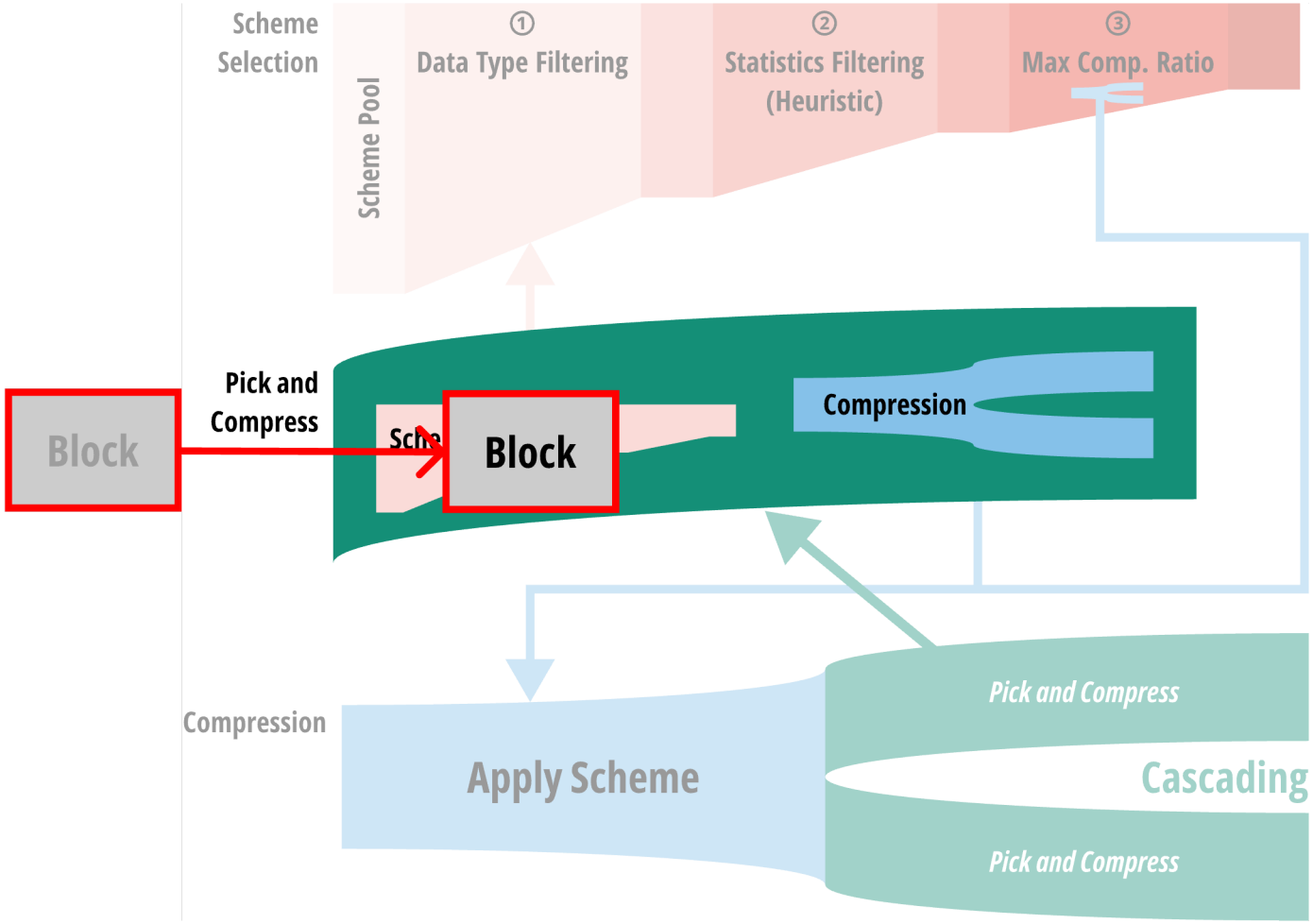
Scheme Pool Summary

- All the schemes are commonly:
  - Simple decompression logic -> **Fast decompression**
  - Compression ratio is depends on data characteristic -> **Scheme selection is the key for overall compression ratio**
  - Encoding result is another data -> **Can encode recursively**

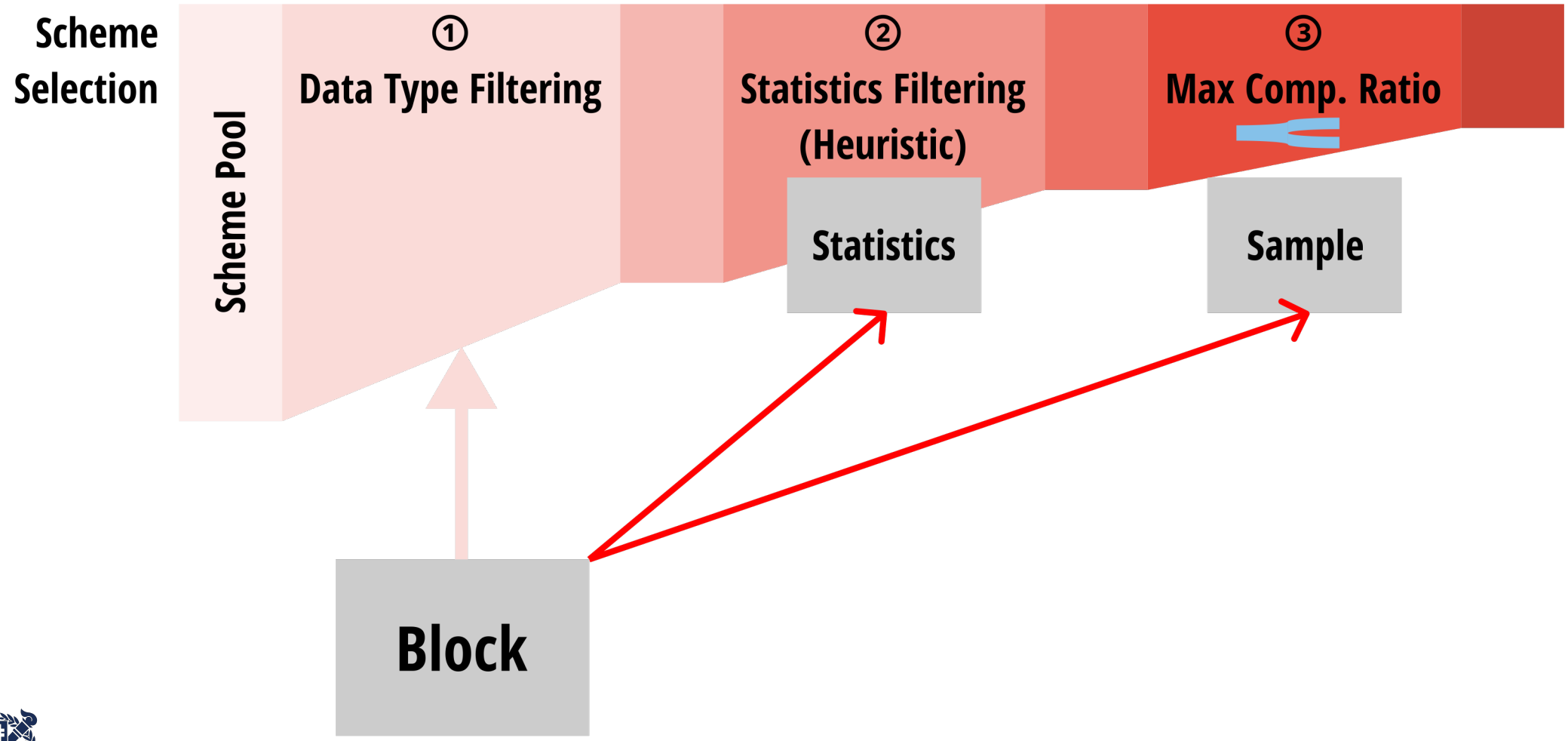
Compression Overview



Compression



Scheme Selection



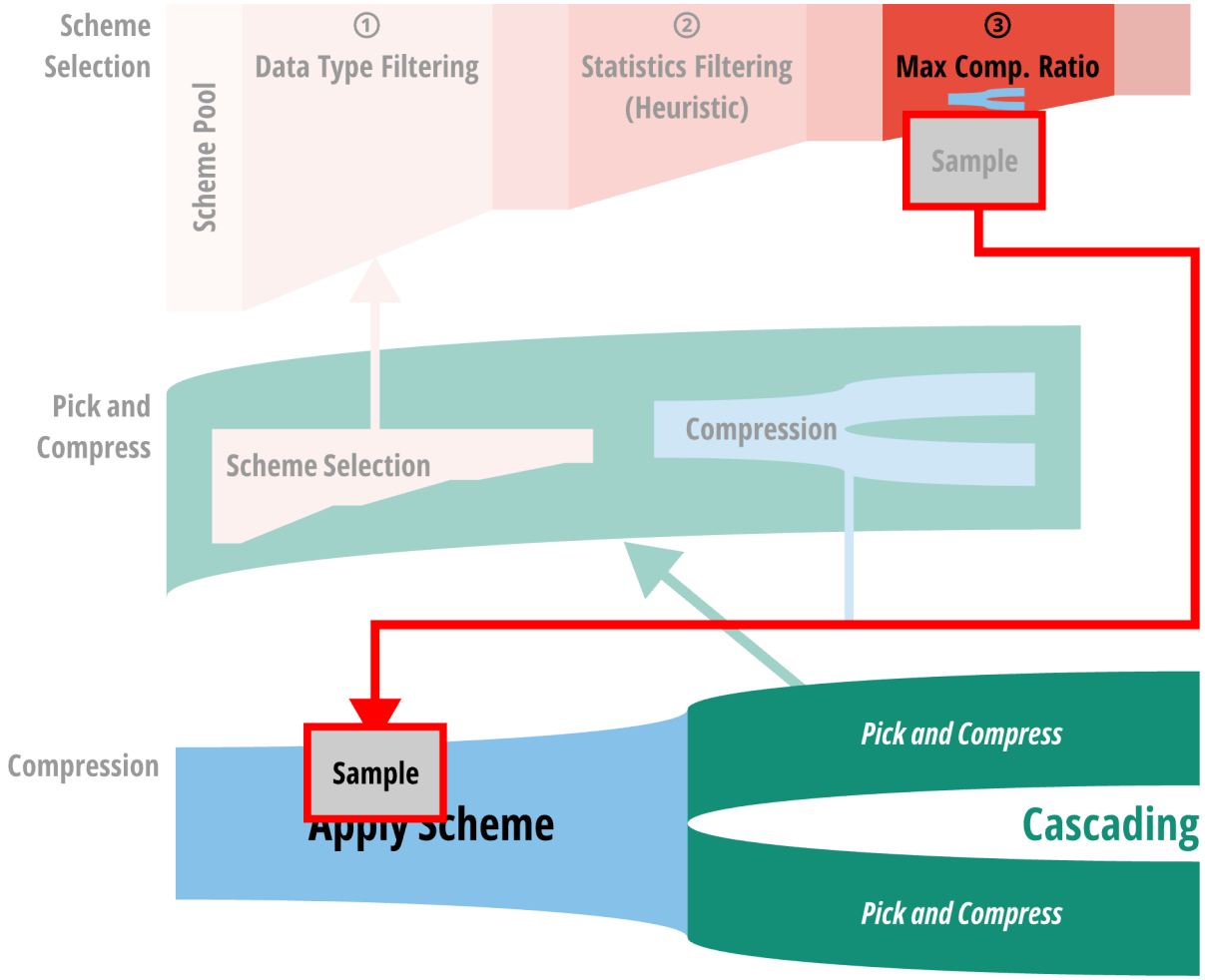
Statistics + Heuristic Filtering

- Need for *Stat.* + *Heuristic Filtering*: To reduce overhead in compressing the samples
- Step 1: Gather the **statistics**
  - Min, Max, Cardinality (# of unique), Avg. run length, etc.
- Step 2: Filter the scheme based on **heuristic**
  - E.g. Exclude RLE when (avg. run length < 2)
  - E.g. Exclude Frequency Encoding when (ratio of unique values > 50%)

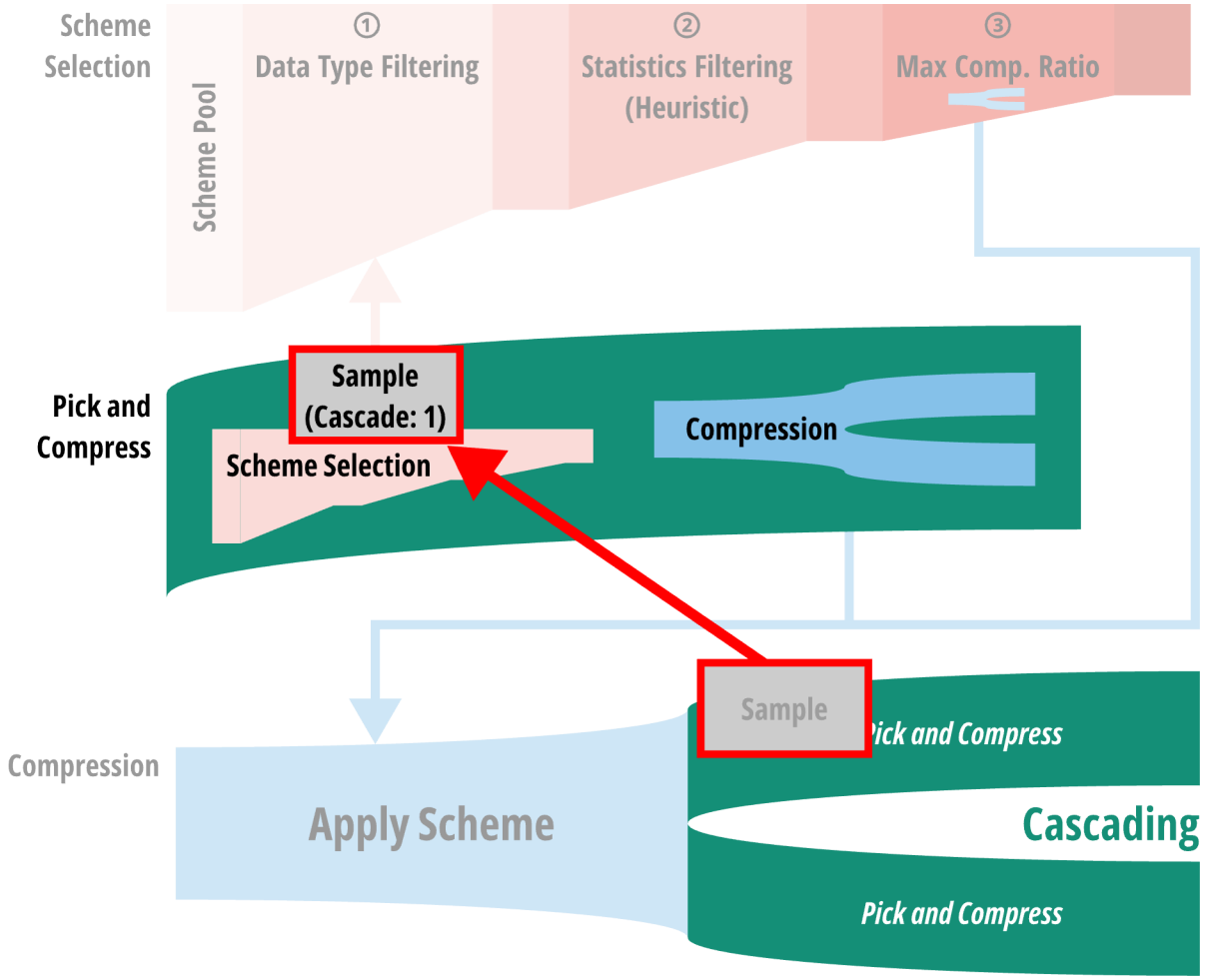
Max. Compression Ratio

- Make **sample** for a **block** (how: explained later)
  - If input data is sample (not a block), do not make sample
- For all remaining schemes:
  - Step 1: **Compress** the sample
  - Step 2: Update max-ratio scheme if compression ratio is higher

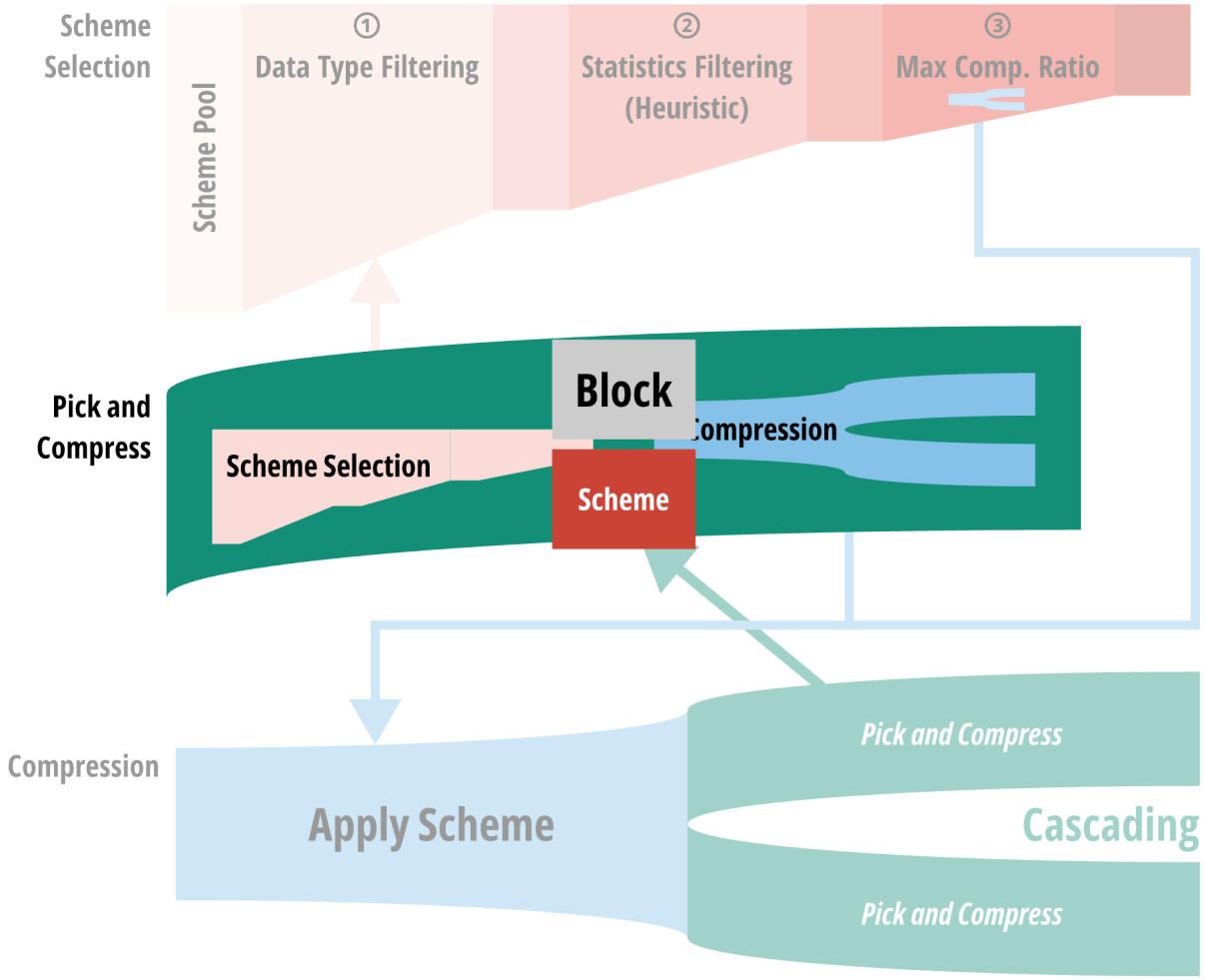
Scheme Selection Loop



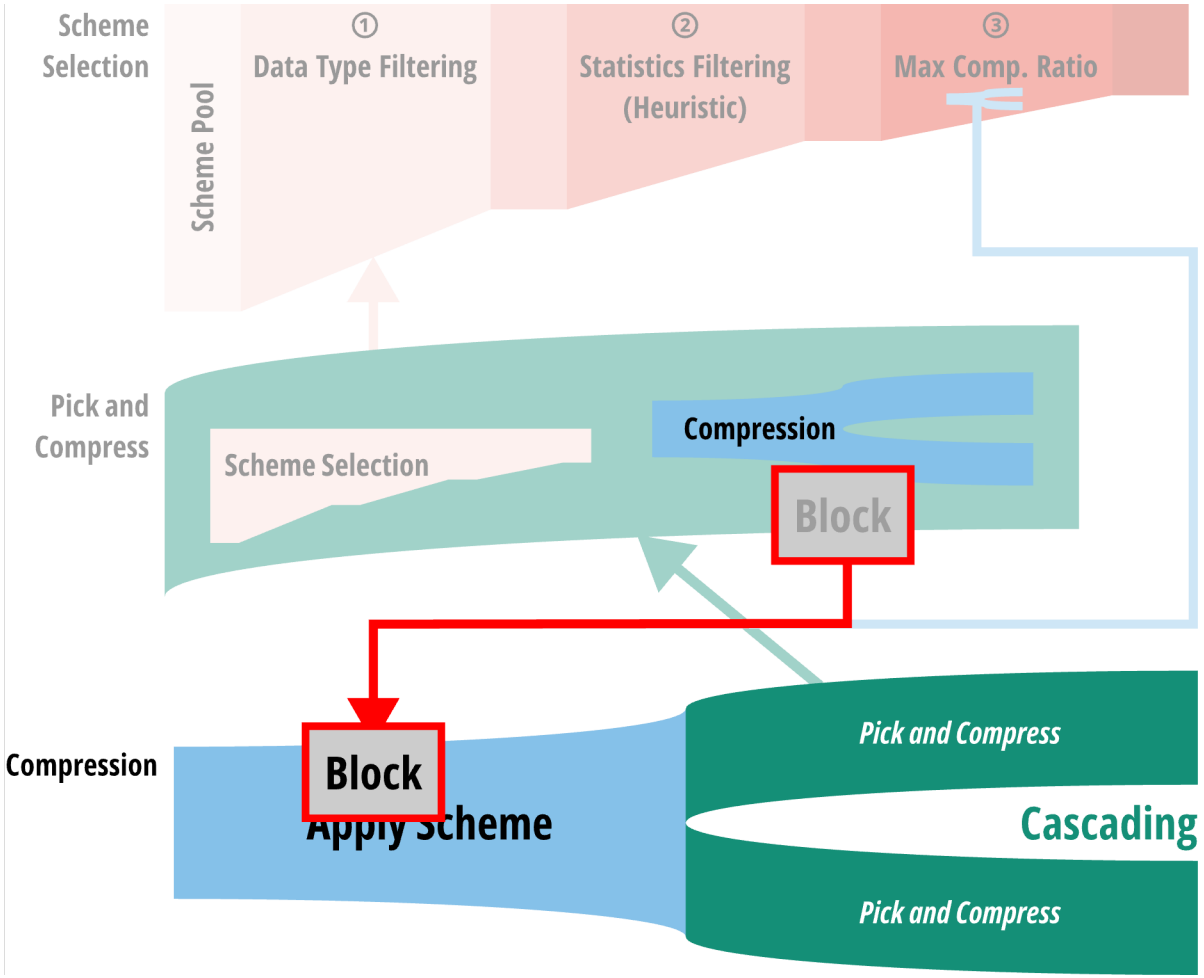
Scheme Selection Loop



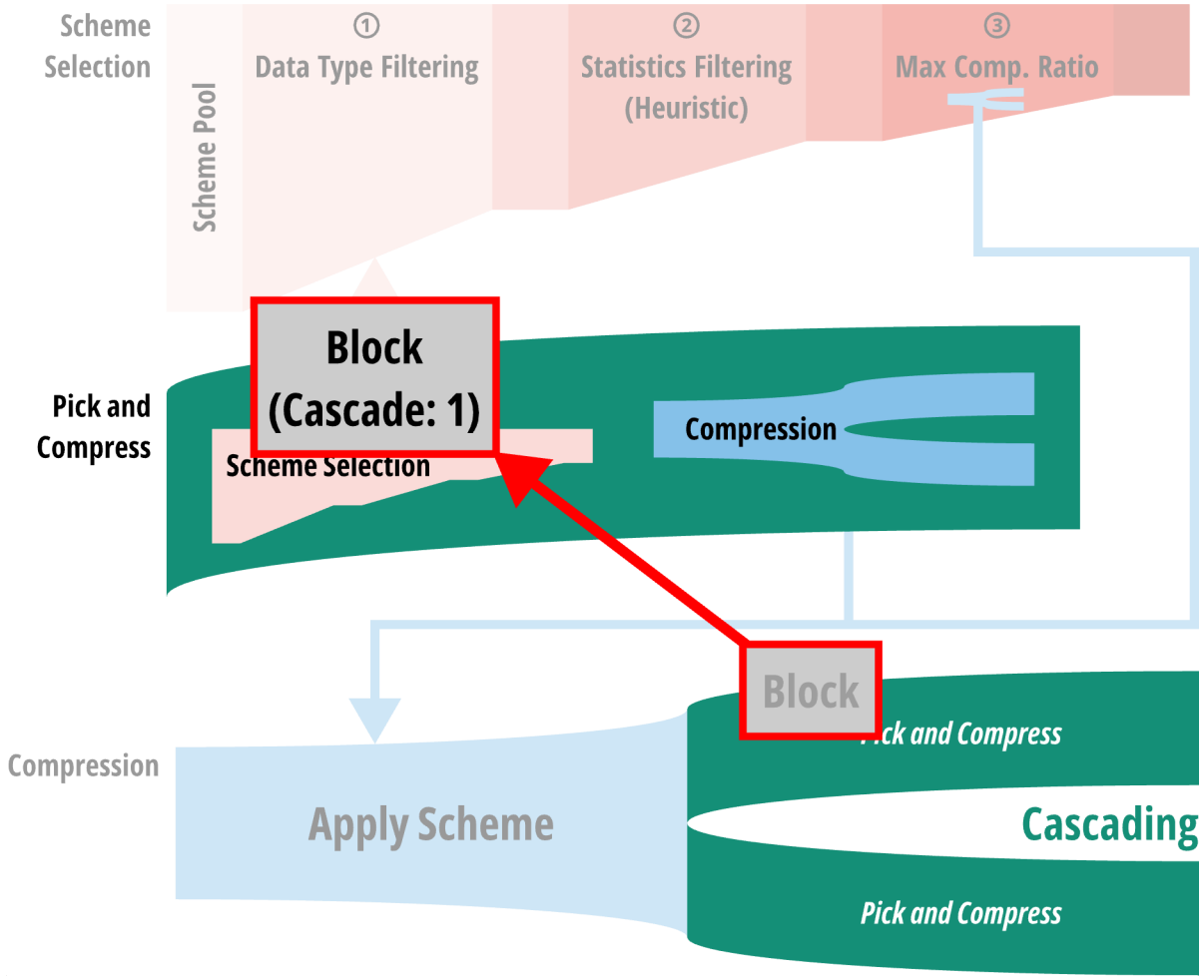
Compression Cascading



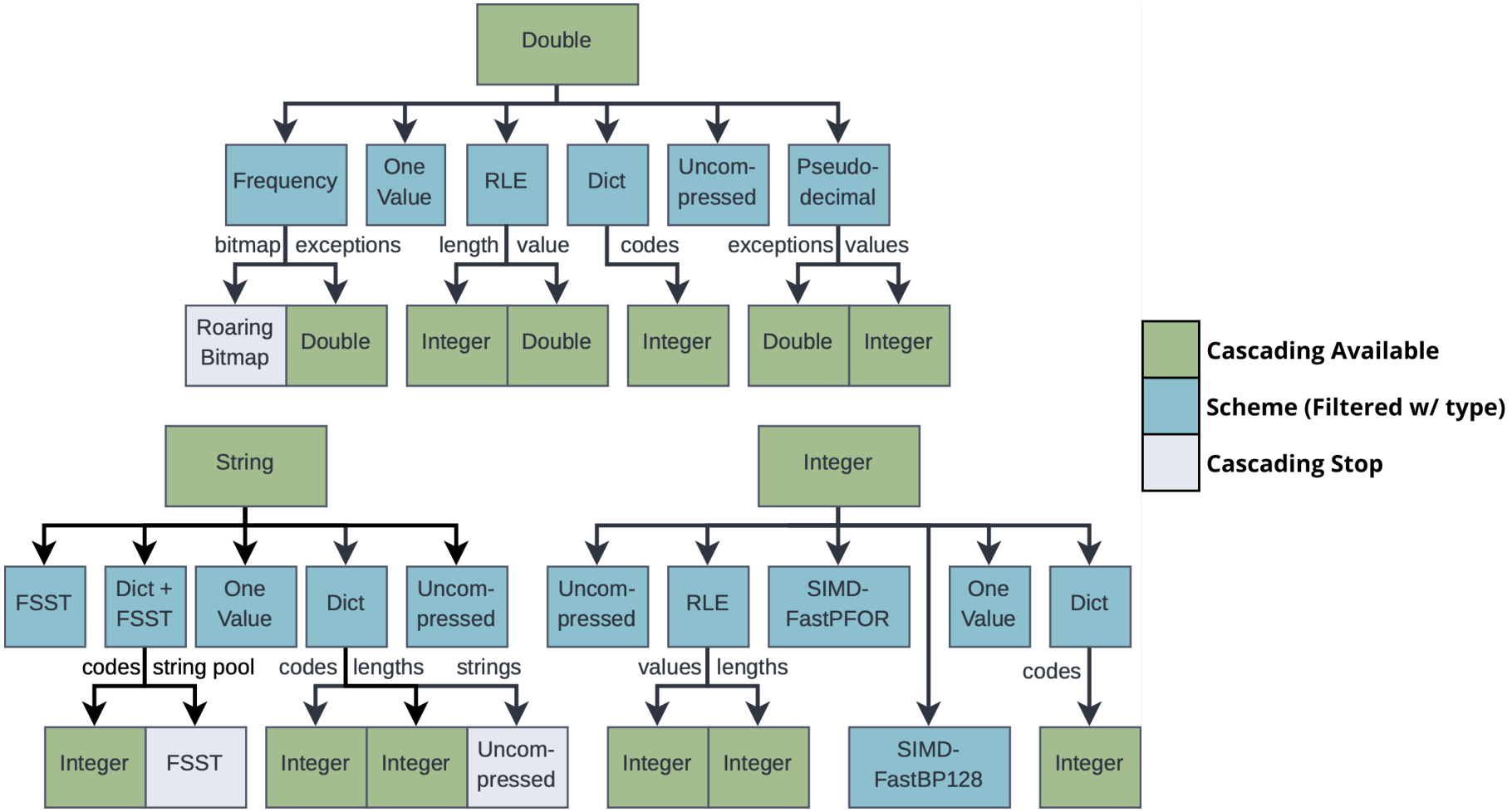
Compression Cascading



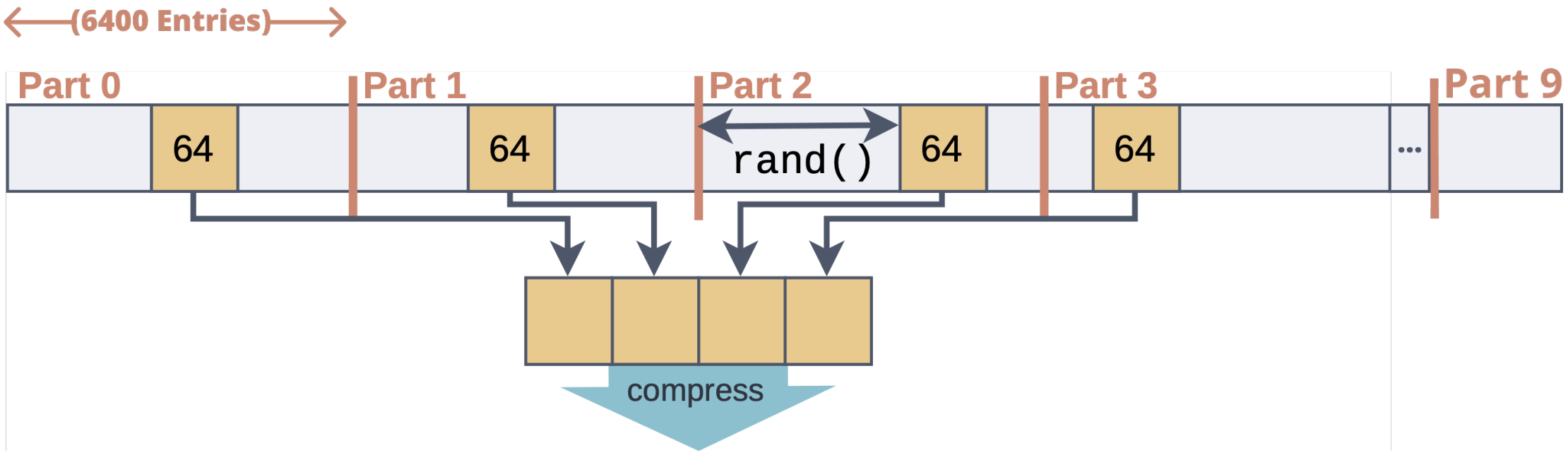
Compression Cascading



Filtering & Cascading by Data Types

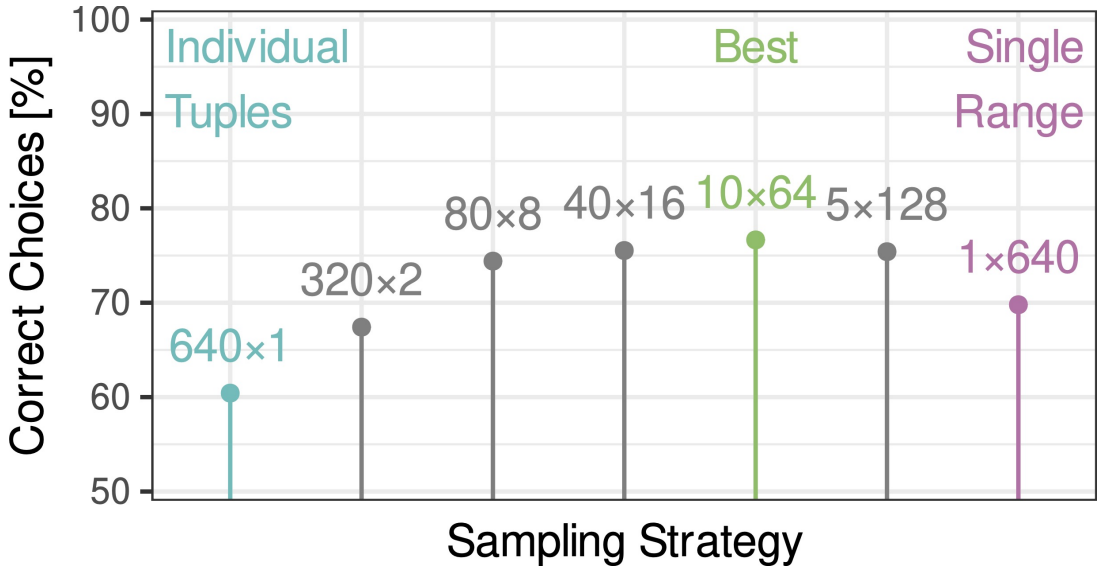


How to Take Sample from the Data



How to Take Sample from the Data

- But, why 10x64?
- 640x1 (Individual tuples on 640 parts): **Worse**
  - Can't utilize runs
- 1x640 (Use only one part): **Not good**
  - Uneven data distribution

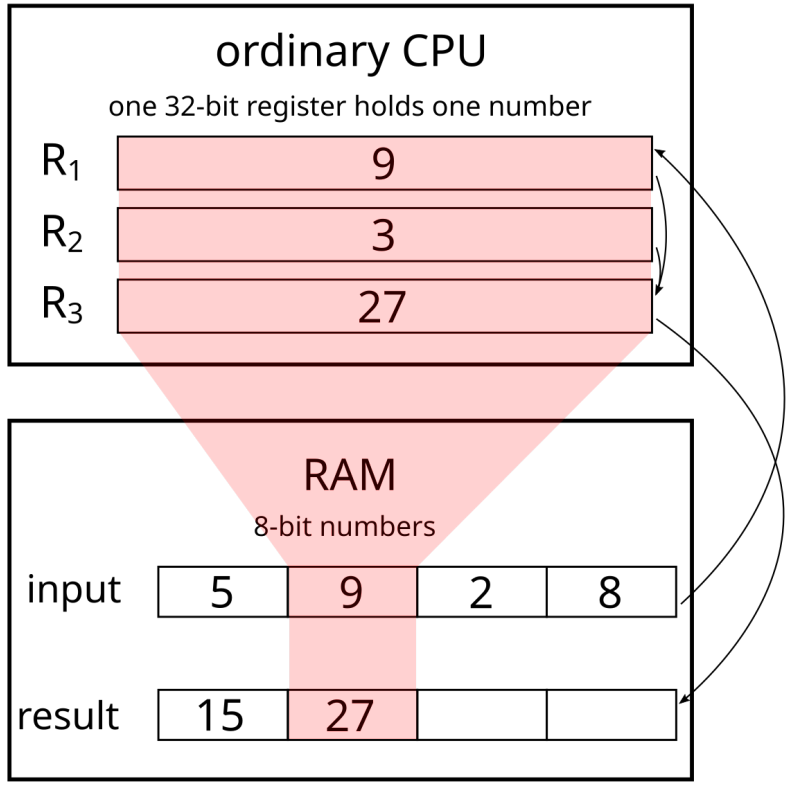


Fast Decompression

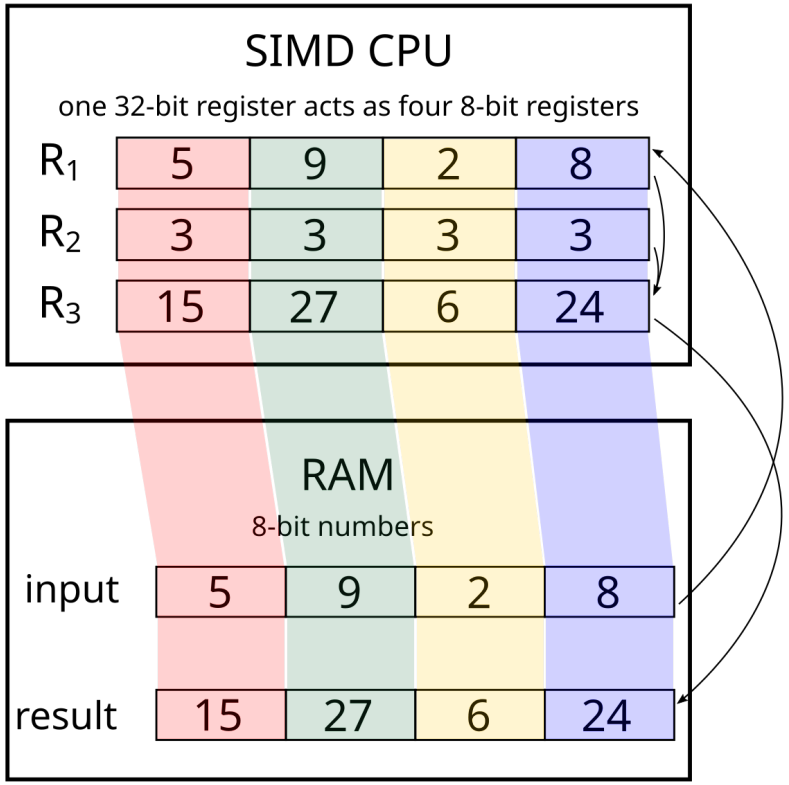
- As each schemes are fast, overall decompression speed is also fast
  - Decompression is faster 2.3x compared to the Parquet + other schemes (Snappy, Zstd, etc.)

- Additional optimization: Using SIMD (17% **more** faster w/o SIMD)

Single Instruction Multiple Data (SIMD)



Operation count:
4 loads, 4 multiplies, and 4 saves



Operation count:
1 load, 1 multiply, and 1 save

https://en.wikipedia.org/wiki/Single_instruction,_multiple_data

Decompression with SIMD

- Decompressing a value is independent with other values
  - RLE, One Value: *Copy* the value n-times
  - Dictionary, Frequency, FSST: *Replace* each code
  - PFOR: *Add* base to each packed value
- Can be optimized with **SIMD vectorization**
  - Instruction set: AVX2 (256bit register)
    - 8x for 32bit type (integer), 4x for 64bit type (double)

Evaluation

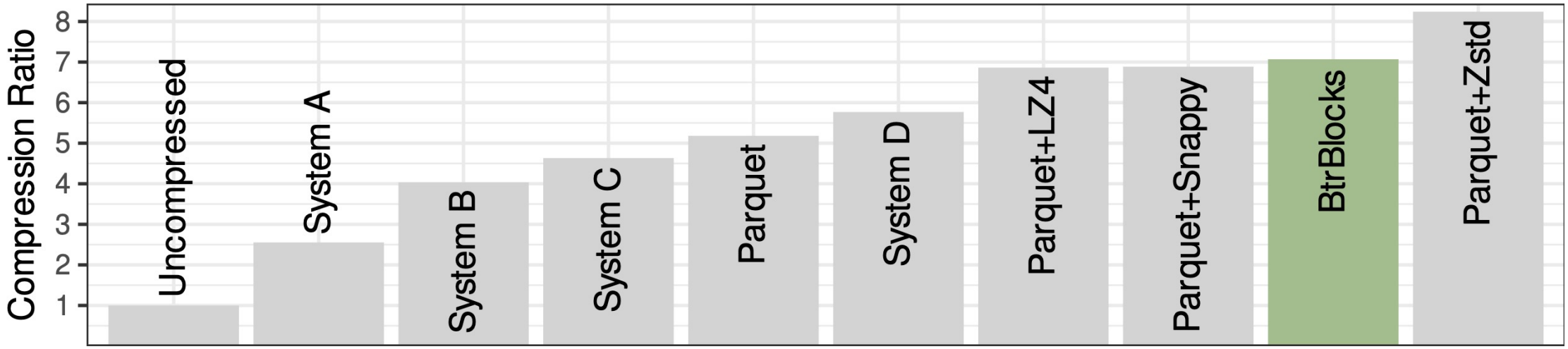
- HW: (AWS EC2 c5n.18xlarge)
  - CPU: 36C 72T (Intel Xen Pantium 8000, Skylake)
  - Memory: 192G
  - Network BW: **100Gbps**
- Dataset:
  - **Tabeau PBI**
  - TPC-H
- Parquet, ORC setup: Spark + Arrow

Public Business Intelligence Dataset

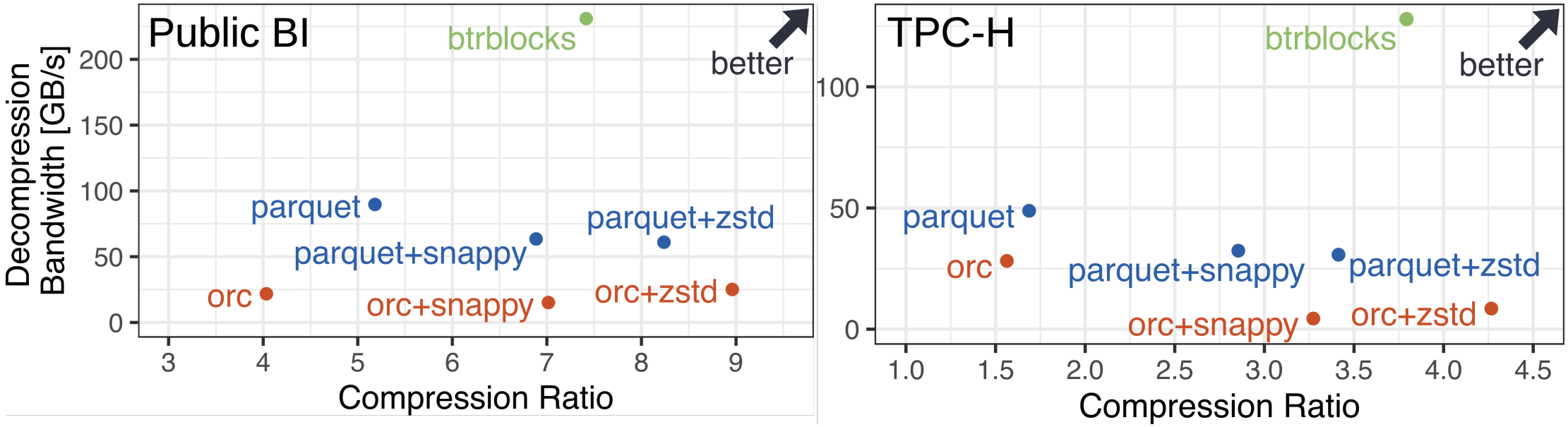
- TPC dataset is **too perfect**
  - E.g. Even data distribution, Perfectly normalized relation
- Real-world data is **not perfect**
  - E.g. Type misuse (represent double as string), Duplicated data (less normalized)
- To gather real-world data (Public BI Dataset), use 46 **Tableau's public workbooks** (https://public.tableau.com).

Compression Ratio

- System A~D: Proprietary system (Not mentioned)



Compression Ratio & Decompression Speed



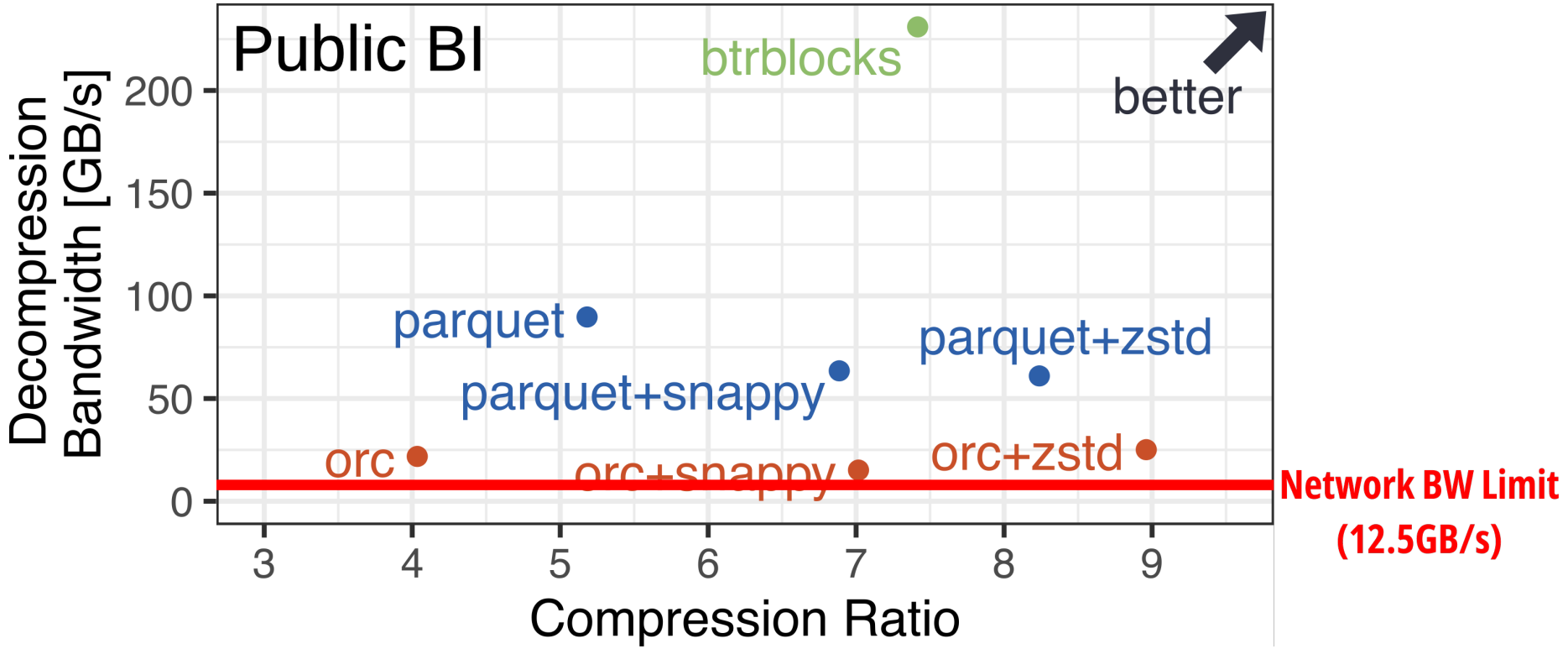
PDE Evaluation

- Compression ratio comparison: PDE+PFOR and others

Column	FPC	Gorilla	Chimp	$Chimp_{128}$	**PDE**
CommonGov./10	1.2	1.1	1.5	**1.9**	1.8
CommonGov./26	15.1	48.0	28.0	6.9	**75.0**
CommonGov./30	6.4	7.0	7.6	5.0	**7.8**
CommonGov./31	9.3	14.3	13.3	5.6	**23.4**
CommonGov./40	14.3	38.0	25.0	6.7	**54.6**
Arade/4	.95	1.1	1.2	1.6	**1.9**
NYC/29	1.5	2.1	**2.5**	1.7	1.0
CMSProvider/1	1.5	1.7	1.8	**2.4**	1.6
CMSProvider/9	2.7	2.3	3.4	2.4	**6.6**
CMSProvider/25	.98	.98	1.1	**1.2**	1.0
Medicare/1	1.2	1.4	1.5	**2.0**	1.5
Medicare/9	2.6	2.3	3.4	2.3	**6.3**

Cloud Cost Evaluation: Scan Throughput

- Question: Isn't **network BW** (100Gbps = 12.5GB/s) bottleneck, not CPU?



Cloud Cost Evaluation : Scan Throughput

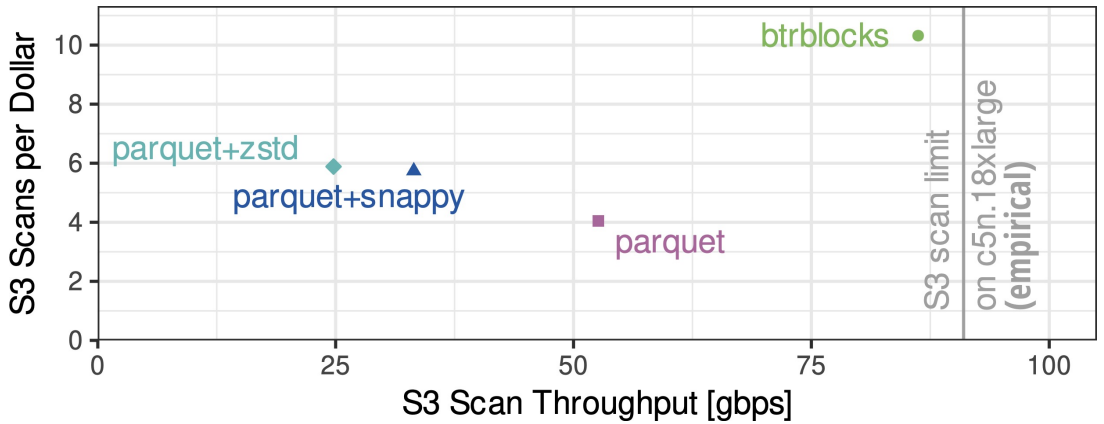
- Decompression throughput: based on **uncompressed data size**
  - $Tu = UncompDataSize / DecompTime$
- Network bandwidth: based on **transferred compressed data size**
  - $Bw = TransferredCompDataSize / TransferTime$
- New metric: Scan throughput (Tc)
  - $Tc = TransferedCompDataSize / DecompTime$

Cloud Cost Evaluation: Scan Cost

- AWS pricing policy:
  - S3 pricing: **GET req. count** for fixed-size chunk (Recommended = 16MB)
  - EC2 pricing: **EC2 usage hour**
- Scan cost ($):
  - $S3Cost + EC2Cost$
  - $= (CompSize/ChunkSize * S3Pricing) + (DecompTime * EC2Pricing)$

Cloud Cost Evaluation

Format	S3 T_u [GB/s]	S3 T_c [Gbit/s]	Scan cost [$]	**Normalized Cost** [×]
BtrBlocks	**174.6**	**86.2**	**0.97**	**1.00**
Parquet	56.1	52.6	2.47	2.61
+Snappy	77.6	33.2	1.74	1.84
+Zstd	78.6	24.8	1.70	1.77

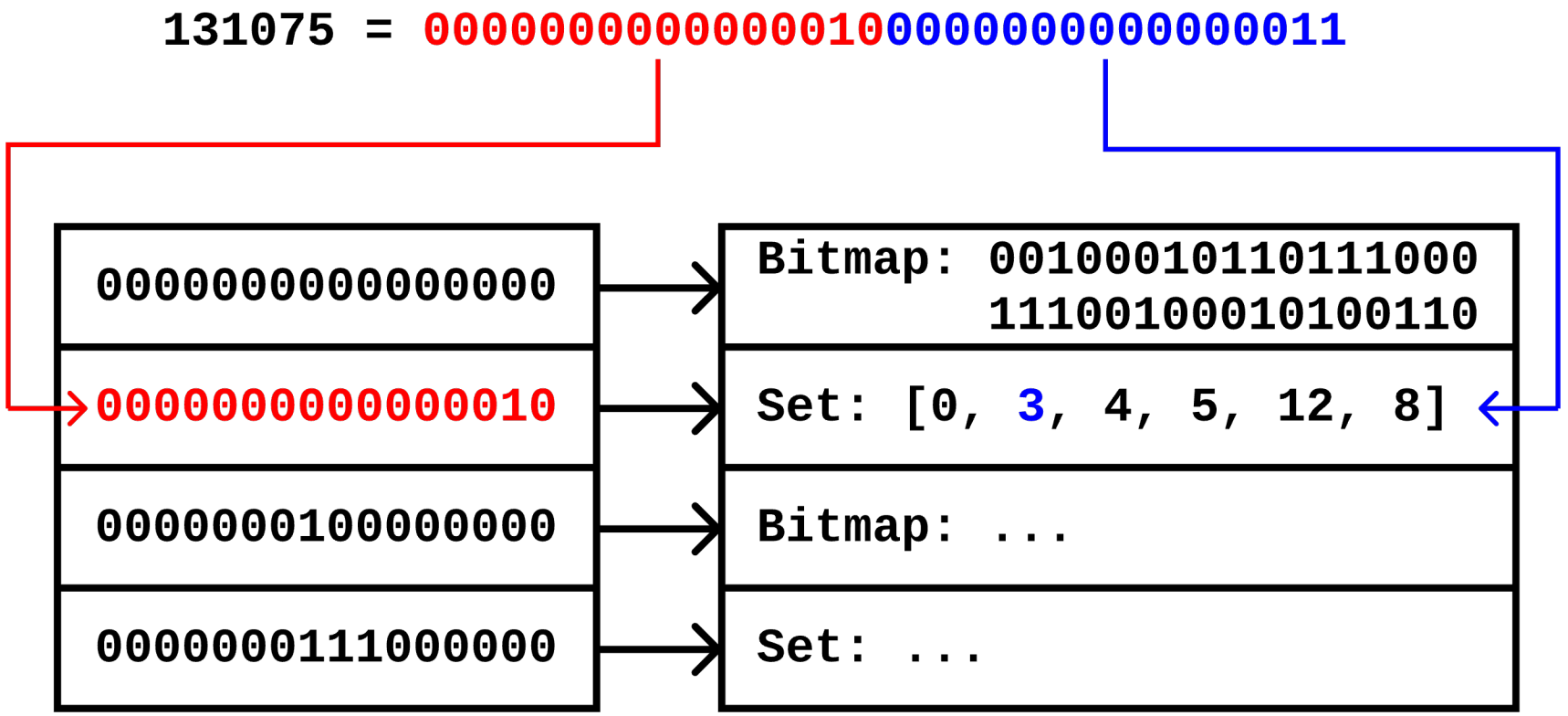


Q/A

Appendix

Building Blocks: Roaring Bitmap

- Example for storing 131075 in a Roaring Bitmap:



Building Blocks: PFOR

- PFOR data format

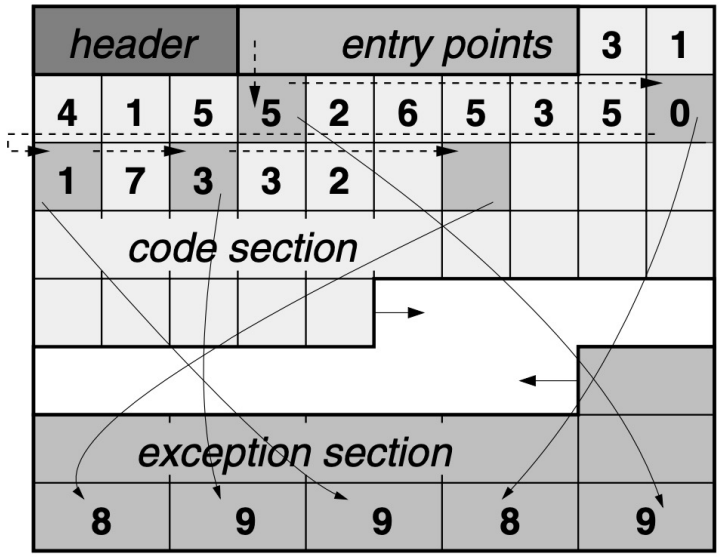


Figure 3. Compressed Segment Layout (encoding the digits of π: 3141592653589793 2 using 3-bit PFOR compression).

Decomp. Optimization: SIMD result

- SIMD-RLE: 128% (integer), 14% (double), 76% (E2E)
- *Code* (in Dict. Encoding) + SIMD-RLE: 78%
- Dict: 18% (integer), 8% (double)

Decomp. Opmization: Fusing Dict + RLE

- Dict + RLE: Cascade *code sequence* with RLE
- Decompression:
  - Step 1: decompress RLE (runs of code -> code sequence)
  - Step 2: decompress Dict (code sequence -> value sequence)
- Fusing: Eleminate **intermediate** result (code sequence)
  - Step 1: decompress Dict (runs of code -> runs of value):
  - Step 2: decompress RLE (runs of value -> value sequence)

Decomp. Opmization : Offset Copy for String Dictionary

- Copy *offset* (inside of string dict.) and *string length* instead of copying each character
  - Ordinary string copy: O(n)
  - Offset copy: O(1)

Decomp. Optimization: SIMD-PDE

- Problem: *Exceptions* inside of PDE make unable to apply vectorization
- Solve:
  - Make positions for the exceptions to **Roaring Bitmap**
  - Use SIMD only when Roaring Bitmap is empty

Evaluation: PBI vs TPC-H

- TPC-H contains more numeric values than PBI

datatype	String				Double				Integer				Combined	
dataset	PBI		TPC-H		PBI		TPC-H		PBI		TPC-H		PBI	TPC-H
metric	sh [%]	cr [×]	sh [%]	cr [×]	sh [%]	cr [×]	sh [%]	cr [×]	sh [%]	cr [×]	sh [%]	cr [×]	cr [×]	cr [×]
Binary	71.5	—	61.7	—	14.4	—	19.5	—	14.1	—	18.7	—	—	—

Evaluation: PBI vs TPC-H

- PBI and TPC-H shows different benchmark results

datatype	String				Double				Integer				Combined	
dataset	PBI		TPC-H		PBI		TPC-H		PBI		TPC-H		PBI	TPC-H
metric	sh [%]	cr [×]	sh [%]	cr [×]	sh [%]	cr [×]	sh [%]	cr [×]	sh [%]	cr [×]	sh [%]	cr [×]	cr [×]	cr [×]
+Zstd	33.6	**17.13**	40.0	**5.27**	50.1	2.30	23.3	2.87	16.3	**6.97**	36.7	1.74	**6.05**	3.41
BtrBlocks	43.6	11.32	54.9	4.26	41.9	**2.36**	16.2	**4.58**	14.5	6.70	28.9	**2.46**	5.28	**3.79**

Evaluation: Compression Speed

- From CSV: Start from file IO
- From binary: CSV file is loaded to memory beforehand

	From CSV	From binary	Compression Factor
BtrBlocks	**38.2 MB/s**	**75.3 MB/s**	7.06×
Parquet+Snappy	38.0 MB/s	41.9 MB/s	6.88×
Parquet+Zstd	37.3 MB/s	41.0 MB/s	**8.24×**