

## Lec 09 Data Compression

对于Disk-Oriented DB, IO操作成为瓶颈。

对于In-Memory DB, 关键trade-off在于 **speed** 与 **compression ratio**

所以关键在于减少DRAM的处理开销, In-Memory通常倾向于追求速度, 一些性能比较好的压缩算法可能非常 **computationally expensive**

### Goal

- Goal1

Must produce **fixed-length** values. (Exception is var-length data)

- Goal2

**late materialization** 在执行query时候尽可能推迟decompression

- Goal3

**Lossless** 不能损失数据.

## DATA SKIPPING

- Approach1: **Approximate Queries**

Execute queries on sampled subset of the entire table to produce **approximate results**.

- Approach2: **Zone Maps**

Pre-compute columnar aggregates per block that allow DBMS to check whether queries need to access it.

Pre-computed aggregates for blocks of data.  
DBMS can check the zone map first to decide whether it wants to access the block.

```
SELECT * FROM table  
WHERE val > 600
```

*Original Data*

val
100
200
300
400
400



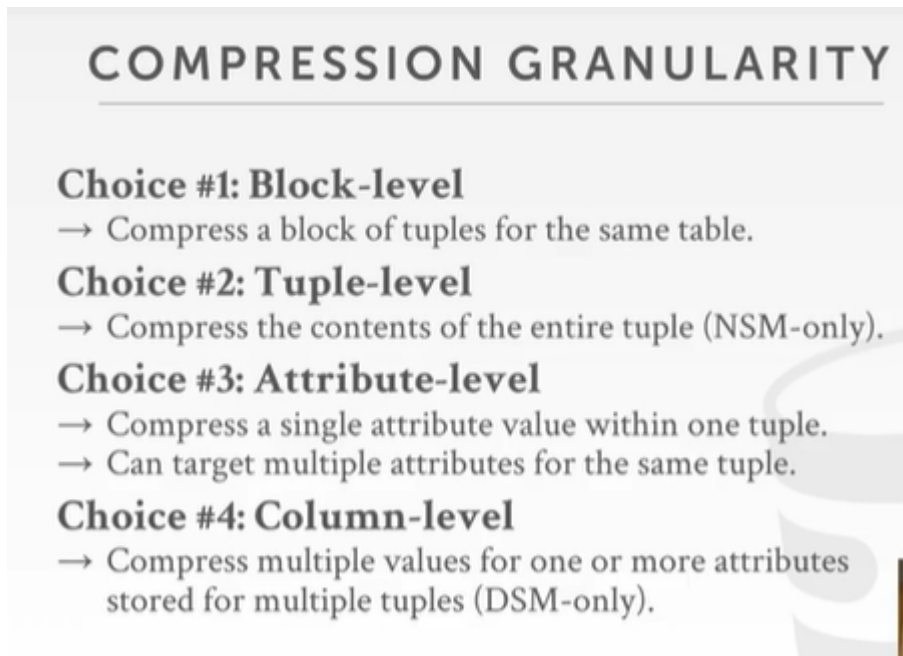
*Zone Map*

type	val
MIN	100
MAX	400
AVG	280
SUM	1400
COUNT	5

大意是对某个block的data计算其aggregation function,如MIN,MAX,AVG,SUM,COUNT,记录在Zone Map上。

然后query到达的时候,先检查zone map看这个block是否有符合其要求的数据。如果没有的话直接skip掉这个block。例如上面的语句 $val > 600$ , 然而 $MAX = 400$ ,说明这个block不可能包含query所需要的数据。

## 压缩的粒度



**COMPRESSION GRANULARITY**

- Choice #1: Block-level**
  - Compress a block of tuples for the same table.
- Choice #2: Tuple-level**
  - Compress the contents of the entire tuple (NSM-only).
- Choice #3: Attribute-level**
  - Compress a single attribute value within one tuple.
  - Can target multiple attributes for the same tuple.
- Choice #4: Column-level**
  - Compress multiple values for one or more attributes stored for multiple tuples (DSM-only).

## NAIVE COMPRESSION

考虑常规的数据压缩算法snappy,gzip等

考虑因素

- 计算开销
- 压缩/解压速率

## COLUMNAR COMPRESSION

### Null Suppression

连续的0或者空白用**这个位置有多少个0或者空白来代替**

### Run-length Encoding

## RUN-LENGTH ENCODING

Compress runs of the same value in a single column into triplets:

- The value of the attribute.
- The start position in the column segment.
- The # of elements in the run.

Requires the columns to be sorted intelligently to maximize compression opportunities.

### Original Data

id	sex
1	M
2	M
3	M
4	F
6	M
7	F
8	M
9	M



### Compressed Data

id	sex
1	(M,0,3)
2	(F,3,1)
3	(M,4,1)
4	(F,5,1)
6	(M,6,2)
7	<b>RLE Triplet</b> - Value - Offset - Length
8	
9	

由上图所示，可以压缩成 (value,offset,length)

After sorted: 压缩效果更好

## Sorted Data

id	sex
1	M
2	M
3	M
6	M
8	M
9	M
4	F
7	F



## Compressed Data

id	sex
1	(M,0,6)
2	(F,7,2)
3	
6	
8	
9	
4	
7	

## Bitmap Encoding

Only a good idea when the cardinality is small.

## Original Data

id	sex
1	M
2	M
3	M
4	F
6	M
7	F
8	M
9	M

$9 \times 8\text{-bits} = 72\text{ bits}$

## Compressed Data

id	sex	
	M	F
1	1	0
2	1	0
3	1	0
4	0	1
6	1	0
7	0	1
8	1	0
9	1	0

$2 \times 8\text{-bits} = 16\text{ bits}$

$9 \times 2\text{-bits} = 18\text{ bits}$

With the following circumstances:

THIS IS TERRIBLE:

## BITMAP ENCODING: EXAMPLE

```
CREATE TABLE customer_dim (  
  id INT PRIMARY KEY,  
  name VARCHAR(32),  
  email VARCHAR(64),  
  address VARCHAR(64),  
  zip_code INT  
);
```

Assume we have 10 million tuples.  
43,000 zip codes in the US.

→  $10000000 \times 32\text{-bits} = 40 \text{ MB}$

→  $10000000 \times 43000 = 53.75 \text{ GB}$

Every time a txn inserts a new tuple, the DBMS must extend 43,000 different bitmaps.

## Delta Encoding

适用于变化比较小的value的压缩，可以结合run-time encoding 达到更好效果

Recording the difference between values that follow each other in the same column.

→ Store base value **in-line** or in a separate **look-up table**.

→ Combine with RLE to get even better compression ratios.

### Original Data

time	temp
12:00	99.5
12:01	99.4
12:02	99.5
12:03	99.6
12:04	99.4

$5 \times 32\text{-bits}$   
 $= 160 \text{ bits}$

### Compressed Data

time	temp
12:00	99.5
+1	-0.1
+1	+0.1
+1	+0.1
+1	-0.2

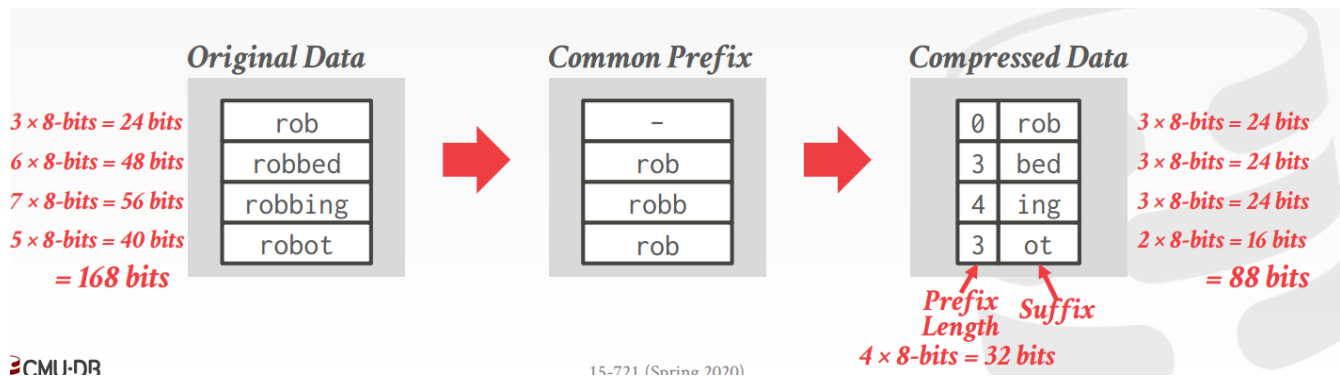
$32\text{-bits} + (4 \times 16\text{-bits})$   
 $= 96 \text{ bits}$

### Compressed Data

time	temp
12:00	99.5
(+1, 4)	-0.1
	+0.1
	+0.1
	-0.2

$32\text{-bits} + (2 \times 16\text{-bits})$   
 $= 64 \text{ bits}$

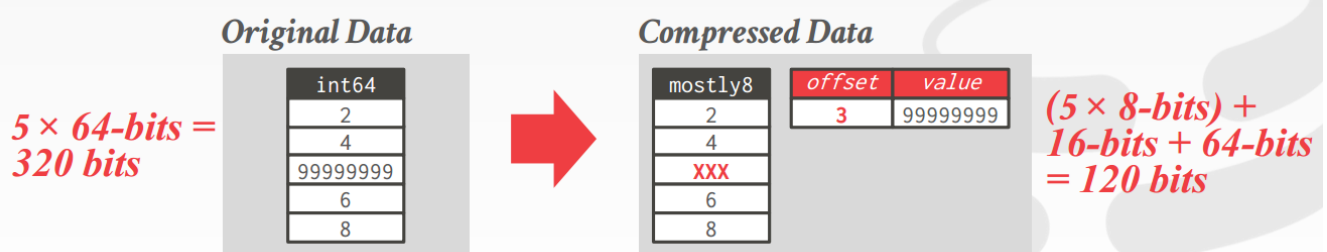
## Incremental Encoding



## Mostly Encoding

When values for an attribute are “mostly” less than the largest size, store them as smaller data type.

→ The remaining values that cannot be compressed are stored in their raw form.



## Dictionary Encoding

### Choice #1: Block-level

- Only include a subset of tuples within a single table.
- Potentially lower compression ratio, but can add new tuples more easily.

### Choice #2: Table-level

- Construct a dictionary for the entire table.
- Better compression ratio, but expensive to update.

### Choice #3: Multi-Table

- Can be either subset or entire tables.
- Sometimes helps with joins and set operations.

- Data Structures

## Choice #1: Array

- One array of variable length strings and another array with pointers that maps to string offsets.
- Expensive to update.

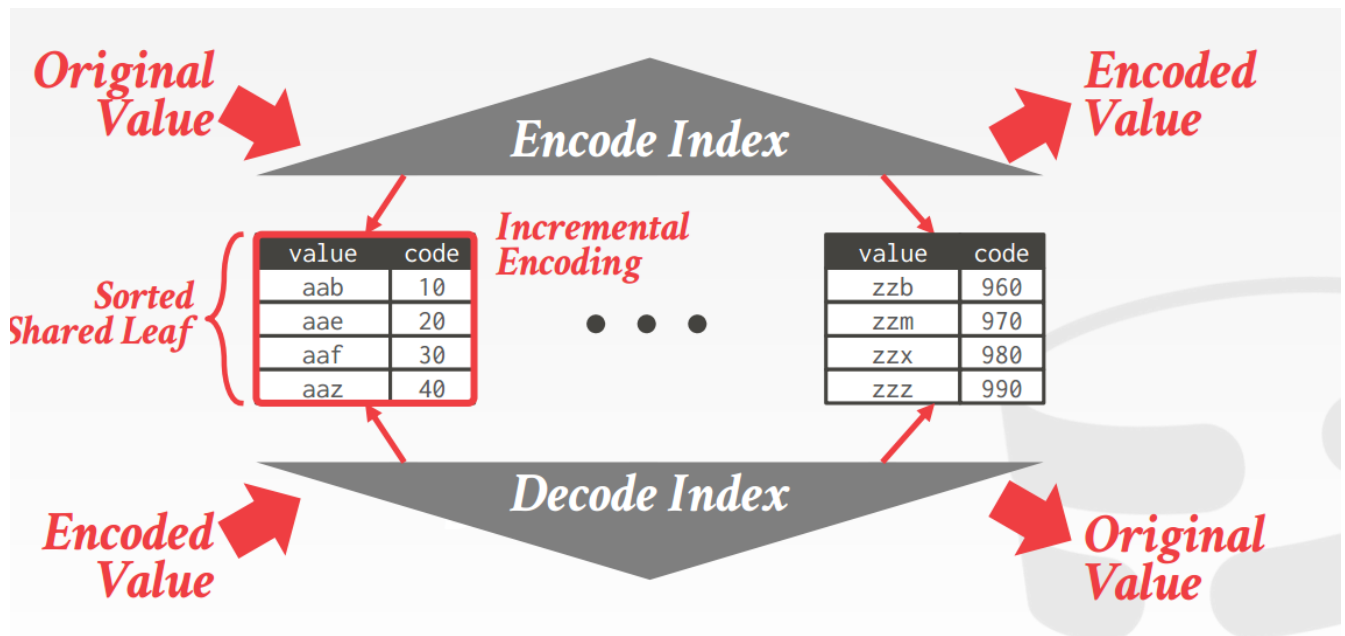
## Choice #2: Hash Table

- Fast and compact.
- Unable to support range and prefix queries.

## Choice #3: B+Tree

- Slower than a hash table and takes more memory.
- Can support range and prefix queries.

- B+Tree Shard Index

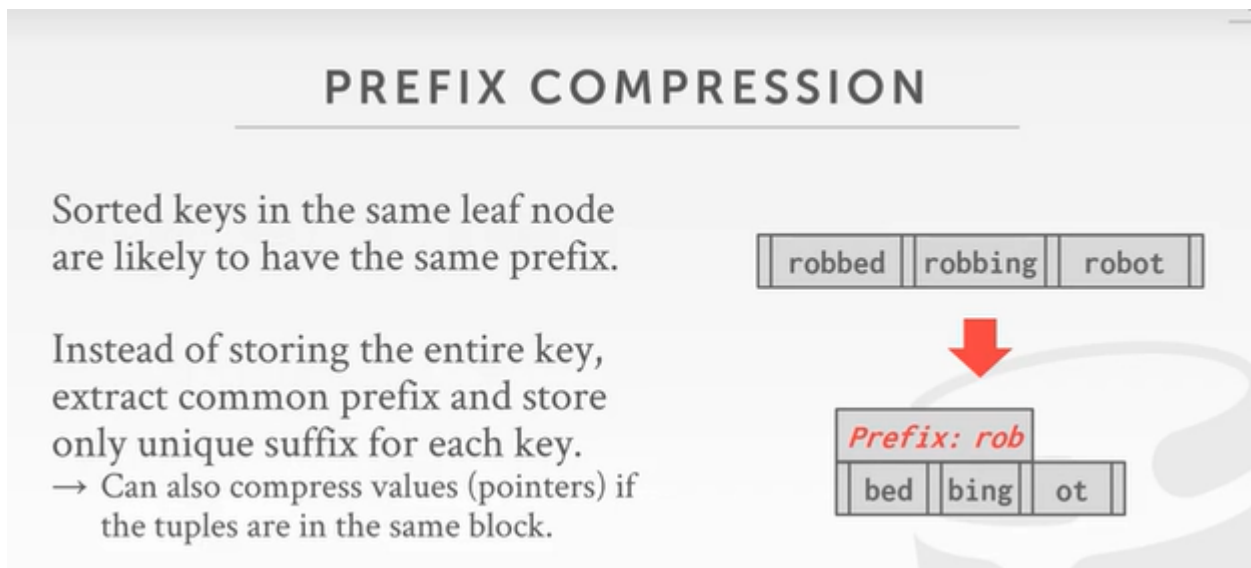


到此为止。我们讨论的所有压缩trick都只是适用于OLAP，OLTP是不能够使用这些trick的，因为OLTP需要支持快速的随机访问。而在OLTP中索引占用了很大一部分的内存开销。

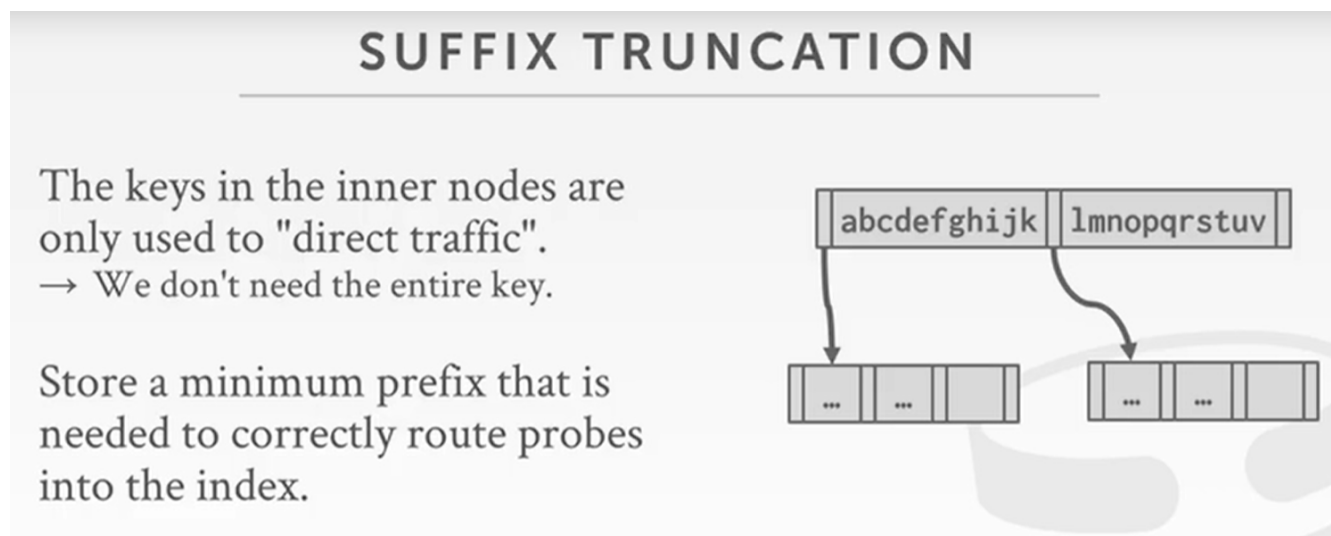


接下来讨论OLTP的压缩trick

- Prefix Compression 压缩前缀



- Suffix Truncation 后缀截断



- Hybrid Index:



# HYBRID INDEXES

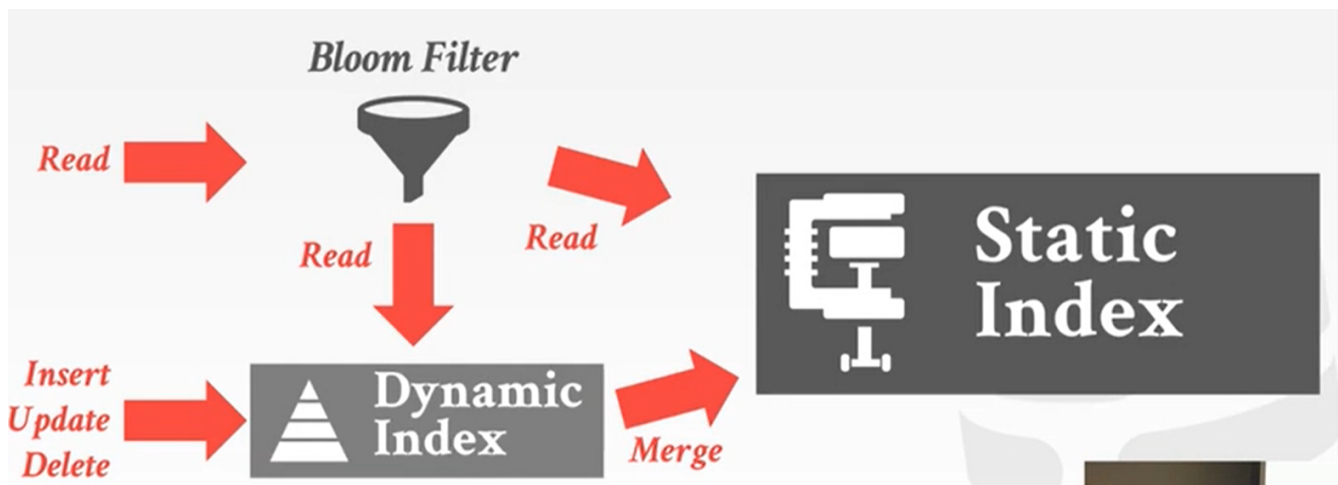
Split a single logical index into two physical indexes. Data is migrated from one stage to the next over time.

→ **Dynamic Stage:** New data, fast to update.

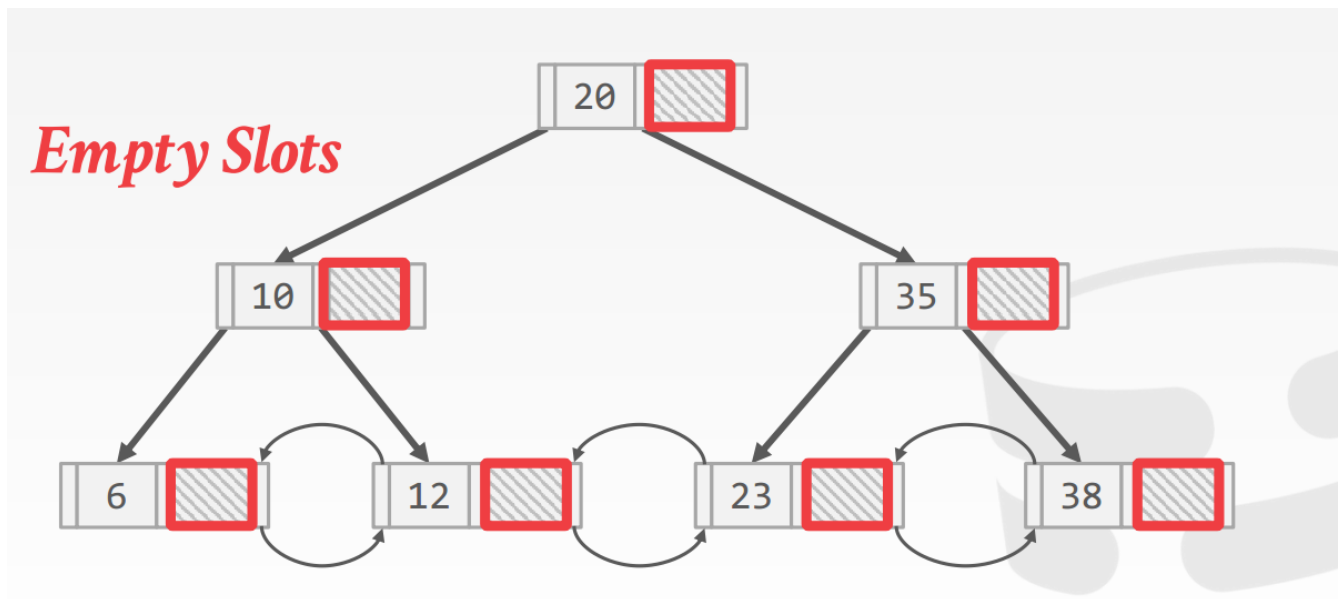
→ **Static Stage:** Old data, compressed + read-only.

All updates go to dynamic stage.

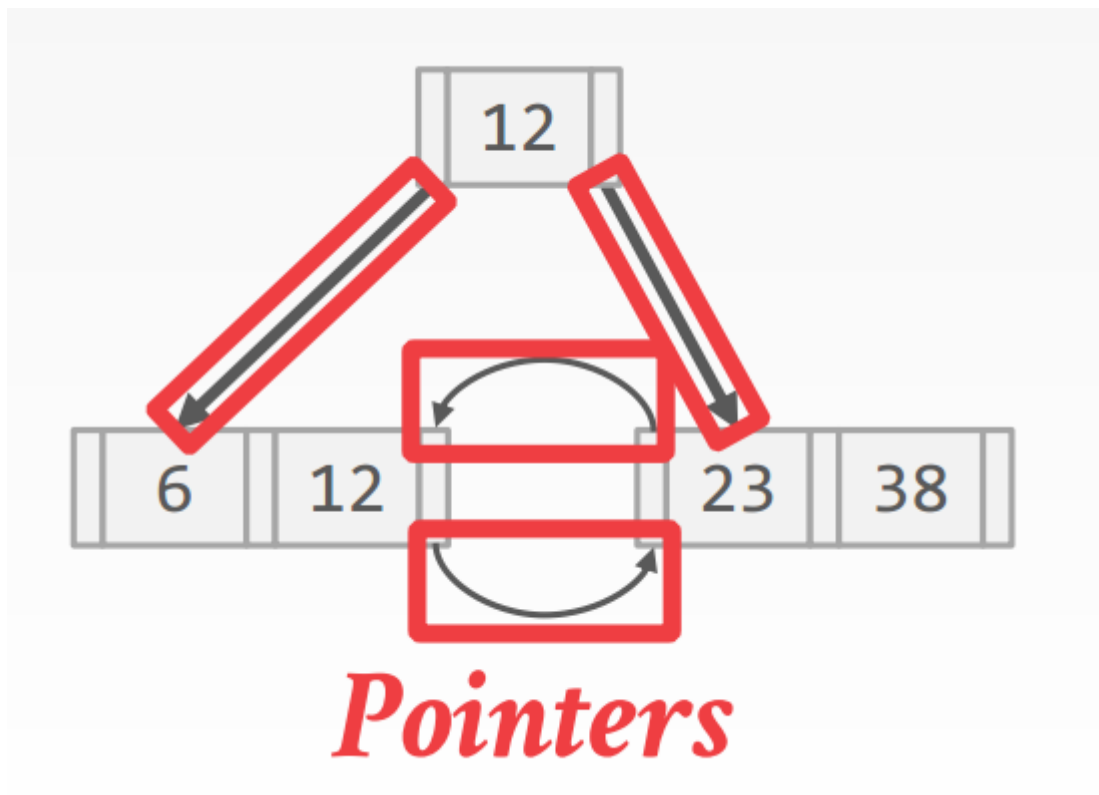
Reads may need to check both stages.



- Compaction of B+Tree



remove the empty slots, and instead of saving the pointers, you can just save the computed offset.



*Computed Offset*

