



**Dhirubhai Ambani
University**

CUCKOO INDEXING

B.Tech Mini Project

Mausam Kamdar

202201372

Under the supervision of:

Prof. Minal Bhise

May 2025

Acknowledgements

I would like to express my sincere gratitude to Prof. Minal Bhise, for her continuous support, guidance and motivation throughout the project and for providing me with this opportunity to expand my knowledge base.

I am also grateful to Ms.Prachi Singh for her invaluable assistance during my B.Tech Mini Project. I would also like to thank my family and friends for continued support.

Abstract

In this B.Tech Mini Project, I explore the concept of Cuckoo Indexing and implement it using Python and PostgreSQL. First, I studied and analyzed the paper "*Cuckoo Index: A Lightweight Secondary Index Structure*", diving deep into concepts like fingerprints, scan rate optimization, collision handling, and block bitmaps.

Upon completion of the theoretical study, I implemented Cuckoo Indexing practically using PostgreSQL and Python, initially testing it on a dummy dataset and later applying it to the `l_orderkey` column of the `lineitem` table (TPC-H benchmark).

The goal was to design a secondary index structure that minimizes scan rates and reduces data access costs while preserving accuracy, thus providing an efficient solution for modern data warehousing.

Problem Statement

In modern data warehousing, data skipping is essential for achieving high query performance. Traditional index structures such as B-trees or hash tables allow precise pruning, but their large storage requirements make them impractical for indexing secondary columns. Therefore, many systems use approximate indexes like min/max sketches (ZoneMaps) or Bloom filters for cost-effective data pruning.

For example, Google PowerDrill skips more than 90% of data on average using such indexes.

We consider a **stripe** as the unit of access, meaning we always scan all values within a stripe. An index allows us to skip over entire stripes but cannot prune within a stripe.

Given, A data item d and a set of stripes S . The task is to identify a subset R that potentially contains d (allowing false positives but no false negatives). The goal is to minimize the number of stripes in R . The **scan rate** is defined as:

$$\text{Scan rate} = \frac{|\text{False Positive Stripes (FP)}|}{|\text{True Negative Stripes (TN)}|} \quad \text{where } |TN| > 0$$

If $|TN| = 0$, i.e., when a value appears in all stripes, we define the scan rate as zero.

Thus, our main objective is:

- Create an effective index structure
- Fetch fewer stripes during I/O
- Ensure no missing true positives

Introduction

Cuckoo index is a secondary index structure that represents the many to-many relationship between keys and data partitions. It leverages the concept of fingerprint and stripe bitmaps to store data in a space efficient manner, further improvising it with the scan rate optimization and block bitmaps.

Our goal is to create an effective index structure that fetches less stripes during I/O for a query, without missing out on stripes that actually contain the required data. Our aim is to minimize scan rate(will be explained further)

Compared to Bloom filters and ZoneMaps, Cuckoo produces correct results for lookups with keys that occur in the data. CI allows to control the ratio of false positive partitions for lookups with non-occurring keys.

Cuckoo Filter

A Cuckoo filter is a data structure that can answer approximate set-membership queries. Like a Cuckoo hash table, a Cuckoo filter consists of m buckets. Figure 2 shows an example of a filter with four buckets. In contrast to a Cuckoo hash table, a Cuckoo filter does not store full keys but small (e.g., 8-bit) fingerprints that are extracted from hashes:

Given a lookup key, the filter determines whether the key might be contained in the set or is definitely not part of the set. Cuckoo filters hash keys using hash functions and then, extracts fingerprints from these hashes.

$$fingerprint(key) = extract(hash(key))$$

Later, it stores these fingerprints along with the stripe bitmaps in buckets (containers that store keys).

Each fingerprint has two possible buckets that it can be placed in: its primary and its secondary bucket. A Cuckoo filter uses partial-key cuckoo hashing to find an alternative bucket i_2 for fingerprints stored in bucket i_1 .

$$i_2 = hash(fingerprint) \oplus i_1$$

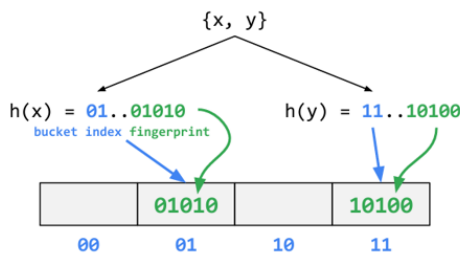


Figure 2: A Cuckoo filter.

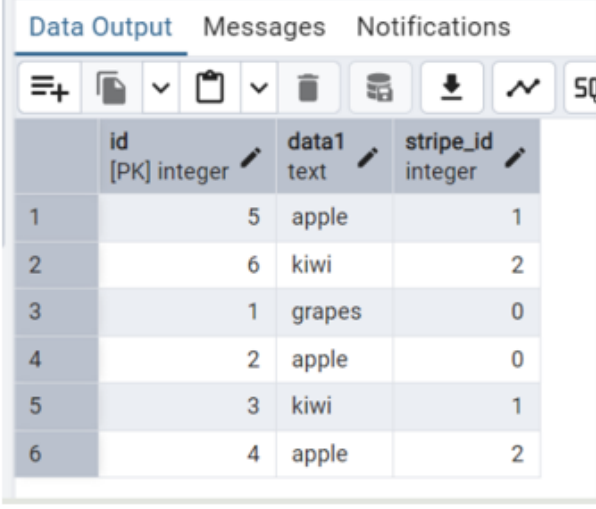


Figure 3: A Cuckoo Index is built for each column in a data file (segment). Each fingerprint in the Cuckoo filter is associated with a stripe bitmap that indicates qualifying stripes. The figure illustrates a query for the country “US”.

CI consists of this Cuckoo filter where each key fingerprint is associated with a fixed-size bitmap indicating qualifying stripes. To query CI, we first probe its Cuckoo filter with a hash of the lookup key. The filter lookup returns the offset of the matching fingerprint stored in the filter (if such a fingerprint exists) which we then use to locate the corresponding bitmap. This works well for low cardinality columns stripe bitmap.

One of our key techniques is to cross-optimize (i.e., holistically optimize) fingerprints and bitmaps to satisfy a given scan rate. The idea is to store fewer fingerprint bits (for false positive reduction) for fingerprints associated with sparse bitmaps, and likewise use more bits for fingerprints associated with dense bitmaps. The rationale behind that is that for sparse bitmaps we need to scan few stripes anyway, thus a false positive that matches with a sparse bitmap is less expensive than a false positive that matches with a dense bitmap.

The following figures show the sample data that I took for experimenting and the bucket table stored in PostgreSQL ,with 2 slots per bucket. The third figure shows the output of the code,which clearly displays which key maps to which bucket and gets inserted in which bucket. If the buckets are full,it displays an error message.



	id [PK] integer	data1 text	stripe_id integer
1	5	apple	1
2	6	kiwi	2
3	1	grapes	0
4	2	apple	0
5	3	kiwi	1
6	4	apple	2

DATA

	bucket_id [PK] integer	bitmap_1 character varying (4)	bitmap_2 character varying (4)	fingerprint_2 character varying (4)	fingerprint_1 character varying (4)
1	2	1110	[null]	[null]	2
2	3	1110	[null]	[null]	3

BUCKET TABLE

```

INSERTING: apple | FP: 0 | Primary: 0 | Secondary: 2
  → Storing apple in Empty Bucket 0
INSERTING: kiwi | FP: 0 | Primary: 0 | Secondary: 2
  → Storing kiwi in Bucket 0, Slot 2
INSERTING: grapes | FP: 2 | Primary: 2 | Secondary: 0
  → Storing grapes in Empty Bucket 2
INSERTING: apple | FP: 0 | Primary: 0 | Secondary: 2
  → Bucket 0 FULL. Trying Secondary 2
  → Storing apple in Bucket 2, Slot 2
INSERTING: kiwi | FP: 0 | Primary: 0 | Secondary: 2
  → Bucket 0 FULL. Trying Secondary 2
X Buckets 0 and 2 are FULL. Skipping kiwi.
INSERTING: apple | FP: 0 | Primary: 0 | Secondary: 2
  → Bucket 0 FULL. Trying Secondary 2
X Buckets 0 and 2 are FULL. Skipping apple.
LOOKING UP KEY: apple | FP: 0 | Primary: 0 | Secondary:

```

OUTPUT

When inserting entries into a Cuckoo filter, there can be fingerprint collisions among keys. That occurs when two keys share the same bucket and have the same fingerprint. This is a problem because then, if we search for one key, other key's fingerprint will be mapped and we would get the stripes containing the other key. So, preventing this is necessary.

Collision Handling

1. Union Bitmaps

This approach solves fingerprint collisions solely on the bitmap side. When we find that a key fingerprint already exists in the filter, we union the bitmaps of the new and the existing fingerprint.

```
if fingerprint_1 == fingerprint:
    print(f" → Fingerprint {fingerprint} exists in Primary {primary_bucket}. Merging Bitma
    new_bitmap = merge_bitmaps(bitmap_1, bitmap_str)
    cur.execute("UPDATE buckets SET bitmap_1 = %s WHERE bucket_id = %s",
                (new_bitmap, primary_bucket))
elif fingerprint_2 == fingerprint:
    print(f" → Fingerprint {fingerprint} exists in Primary {primary_bucket}, Slot 2. Mergi
    new_bitmap = merge_bitmaps(bitmap_2, bitmap_str)
    cur.execute("UPDATE buckets SET bitmap_2 = %s WHERE bucket_id = %s",
                (new_bitmap, primary_bucket))
elif fingerprint_1 is None:
    print(f" → Storing {key} in Bucket {primary_bucket}, Slot 1")
    cur.execute("UPDATE buckets SET fingerprint_1 = %s, bitmap_1 = %s WHERE bucket_id = %s"
                (fingerprint, bitmap_str, primary_bucket))
elif fingerprint_2 is None:
    print(f" → Storing {key} in Bucket {primary_bucket}, Slot 2")
    cur.execute("UPDATE buckets SET fingerprint_2 = %s, bitmap_2 = %s WHERE bucket_id = %s"
                (fingerprint, bitmap_str, primary_bucket))
```

Union Bitmap implementation in my code

While this strategy does not require any extra space, it introduces additional false positive stripes: In the event that a lookup maps to a bitmap that has been unioned with another bitmap, we may introduce false positive stripes. Since we want maximum pruning power, we move on to better methods.

2. Variable Sized Fingerprints (Scan Rate Optimization)

We compute the minimum number of hash bits required to ensure unique fingerprints on a per-bucket basis. We make all fingerprints in a bucket share the same length (number of bits) for space efficiency reasons.

To correctly match a lookup key with its stored fingerprint, we need to carefully

determine the minimum number of fingerprint bits per bucket. In particular, we need to consider all items that have a certain bucket as primary bucket instead of only considering those that are actually stored in that bucket. Otherwise, a lookup for an item that is stored in its secondary bucket might get falsely matched with an item in its primary bucket.

	<u>Key</u>	<u>Hash</u>	<u>Primary Bucket</u>	<u>Secondary Bucket</u>
Bucket 0	a:	00		
	b:	10		
Bucket 1	c:	01		
	a	00..	0	1
	b	10..	0	1
	c	01..	0	1

In the figure, since fingerprint c is present in its secondary bucket, without counting it, the number of fingerprints in bucket 0 will be 2. This will result in the fingerprint size for bucket 0 to be 1, which does not differentiate between all the 3 keys.

Also, we have to choose an optimal number of bits per bucket that maximizes the primary ratio. This is the scan rate. To correctly match a lookup key with its stored fingerprint, we need to carefully determine the minimum number of fingerprint bits per bucket.

Every additional fingerprint bit halves the probability that a random lookup fingerprint matches with the stored fingerprint. Second, assuming the fingerprints match, the associated bitmap can cause a further scan rate reduction. In the best case, the bitmap has only one bit set, limiting the number of false positive stripes to one.

Hence, the formula for the expected scan rate of a fingerprint/bitmap pair is:

$$\text{local scan rate} = \frac{1}{2^{\text{fingerprint bits}}} * \text{bitmap density}$$

Since optimizing the scan rate across all buckets can be expensive, we optimize the scan rate on a per-bucket basis. In particular, we may increase the fingerprint length of

a bucket such that the expected scan rate of that bucket stays below a certain threshold. Essentially, we increase the number of fingerprint bits until the fingerprints are unique. To check for that condition, we compute the actual scan rate of the bucket by averaging the local scan rates of the individual fingerprint/bitmap pairs. We also account for the table density: If a lookup “ends up” in an empty bucket, there is no probability of a false match at all and it will correctly be identified as a negative. Furthermore, we account for the fact that a lookup “checks” up to two buckets: the primary and the secondary bucket of the lookup key. By multiplying by two in this step, we treat both lookups (primary and secondary) as independent random experiments. Thereby, we slightly overestimate the probability of a false match since these lookups are actually not independent. That is, if a lookup matches with a fingerprint in the primary bucket, it cannot also match with a fingerprint in the secondary bucket.

First, I implemented using 100 rows and then 10,000 rows of the `l_orderkey` column of lineitem table (TPC-H).

For the smaller data set,

NUM of BUCKETS = 6

BUCKET SIZE = 5

NUM of STRIPES = 6

TARGET SCAN RATE = 0.1

For this, the optimal fingerprint bit size per bucket was 3 bits. 3 bits was enough to ensure that no two keys mapped to the same bucket had the same fingerprint

While for bigger data set,

NUM BUCKETS = 64

BUCKET SIZE = 50

NUM STRIPES = 32

TARGET SCAN RATE = 0.0000001

For this, the optimal fingerprint bit size per bucket was 17 bits (can be seen in output)

Algorithm 1: Returns the minimum number of fingerprint bits required to avoid fingerprint collisions and to satisfy a given scan rate.

Input: *keys, table, bucket, target_scan_rate*
Output: *num_bits*

// Get bucket density of *table* (ratio of non-empty buckets).
1 *table_density* \leftarrow *GetBucketDensity(table)*

// Get minimum number of hash bits to avoid fingerprint collisions of keys that have *bucket* as primary bucket.
2 *num_bits* \leftarrow *GetCollFreeHashLength(keys, bucket)*

3 **while true do**
4 *false_match_probability* $\leftarrow 1/2^{\text{num_bits}}$
 // Compute scan rate of *bucket* by averaging local scan rates.
5 *sum_scan_rate* $\leftarrow 0.0$
6 **for entry in bucket do**
7 *bitmap_density* \leftarrow
 GetBitmapDensity(entry.bitmap)
8 *sum_scan_rate* +=
 *false_match_probability * bitmap_density*
9 *actual_scan_rate* \leftarrow
 sum_scan_rate / bucket.num_entries()
 // Account for *table_density* (reduces *actual_scan_rate* based on the fact that lookups may end up in an empty bucket).
10 *actual_scan_rate* *= *table_density*
 // Double *actual_scan_rate* to account for the secondary lookup.
11 *actual_scan_rate* *= 2
12 **if actual_scan_rate <= target_scan_rate then**
13 break
14 ++*num_bits*
15 **return num_bits**

Algorithm for Scan rate optimization

```

def estimate_table_density(data, num_buckets):
    """Estimate bucket density for fingerprint sizing."""
    bucket_counts = [0] * num_buckets
    for key, _ in data:
        fingerprint = extract_fingerprint(key, 2) # Returns a binary string
        primary_bucket, secondary_bucket = get_bucket_indices(int(fingerprint, 2)) # Convert to int

        if bucket_counts[primary_bucket] < BUCKET_SIZE:
            bucket_counts[primary_bucket] += 1
        elif bucket_counts[secondary_bucket] < BUCKET_SIZE:
            bucket_counts[secondary_bucket] += 1

    non_empty_buckets = sum(1 for count in bucket_counts if count > 0)
    return non_empty_buckets / num_buckets

def compute_fingerprint_bits(keys, num_buckets, target_scan_rate):
    """Dynamically computes fingerprint bits per bucket."""
    estimated_density = estimate_table_density(keys, num_buckets)
    num_bits = 2 # Start with 2 bits

    while True:
        false_match_probability = 1 / (2 ** num_bits)
        sum_scan_rate = sum(false_match_probability * 0.5 for key, _ in keys) # Approximate bitmap density
        actual_scan_rate = (sum_scan_rate / len(keys)) * estimated_density * 2
        if actual_scan_rate <= target_scan_rate:
            break
        num_bits += 1

    return num_bits

```

Scan rate optimization part of my code

OUTPUT (100 rows of l_orderkey)

```

[✓] Initialized bucket_num_bits: {0: 3, 1: 3, 2: 3, 3: 3, 4: 3, 5: 3}
key:1
num_bits:3
fp:110,bucket:0,stripe_bitmap:1
insert:010000
key:1
num_bits:3
fp:110,bucket:0,stripe_bitmap:2
ex:['0', '1', '0', '0', '0', '0']
mebp:011000
key:1
num_bits:3
fp:110,bucket:0,stripe_bitmap:3
ex:['0', '1', '1', '0', '0', '0']
mebp:011100
key:1
num_bits:3
fp:110,bucket:0,stripe_bitmap:4
ex:['0', '1', '1', '1', '0', '0']
mebp:011110
key:1
num_bits:3
fp:110,bucket:0,stripe_bitmap:5
ex:['0', '1', '1', '1', '1', '0']
mebp:011111

```

```

fp:011,bucket:3,stripe_bitmap:3
ex:['1', '1', '1', '1', '1', '0']
mebp:111110
key:98
num_bits:3
fp:011,bucket:3,stripe_bitmap:4
ex:['1', '1', '1', '1', '1', '0']
mebp:111110
num_bits:3
fp:110,pb:0
[✓] Found in Bucket 0 with fingerprint 110 and bitmap 011111
[?] The key '1' was found **5 times** in the database.

```

Here, as you can see, the code has union the bitmaps of all keys with same fingerprint. The bitmap after every union is also displayed. Also, the code has displayed the bucket in which the key was found and is present in how many stripes(hits)

OUTPUT (10,000 rows of l_orderkey)

```

mebp:00000000000000001000011010000000
key:10304,fp:001001010000111000,bucket:56
key:10304,bp:['0', '0', '0', '0', '0', '0', '0', '0', '0', '0', '0', '0', '0', '0', '0', '0', '1', '0', '0', '0', '0', '0', '1',
'0', '0', '0', '0']
mebp:10000000000000001000011010000000
key:10304,fp:001001010000111000,bucket:56
key:10304,bp:['1', '0', '0', '0', '0', '0', '0', '0', '0', '0', '0', '0', '0', '0', '0', '0', '1', '0', '0', '0', '0', '0', '1',
'0', '0', '0', '0']
mebp:10000000000000001000011010000000
key:10304,fp:001001010000111000,bucket:56
key:10304,bp:['1', '0', '0', '0', '0', '0', '0', '0', '0', '0', '0', '0', '0', '0', '0', '0', '1', '0', '0', '0', '0', '0', '1',
'0', '0', '0', '0']
mebp:10000000000000001000111010000000

mebp:000001000000000100000001100000000
key:20164,fp:01101001011100101,bucket:37
key:20164,bp:['0', '0', '0', '0', '0', '0', '1', '0', '0', '0', '0', '0', '0', '0', '0', '0', '1', '0', '0', '0', '0',
'0', '0', '0', '0']
mebp:100001000000000100000001100000000
key:20164,fp:01101001011100101,bucket:37
key:20164,bp:['1', '0', '0', '0', '0', '0', '1', '0', '0', '0', '0', '0', '0', '0', '0', '0', '1', '0', '0', '0', '0',
'0', '0', '0', '0']
mebp:100001000000000100001001100000000
key:20165,fp:01111100010001101,bucket:13
key:20165,fp:01111100010001101,bucket:13
key:20165,bp:['0', '0', '0', '0', '1', '0', '0', '0', '0', '0', '0', '0', '0', '0', '0', '0', '0', '0', '0', '0', '0',
'0', '0', '0', '0']
mebp:000100000000000000000000000000100
key:20165,fp:01111100010001101,bucket:13
key:20165,bp:['0', '0', '0', '0', '1', '0', '0', '0', '0', '0', '0', '0', '0', '0', '0', '0', '0', '0', '0', '0', '0',
'0', '1', '0', '0']
mebp:000100000100000000000000000000100
Found in Bucket 56 with fingerprint 001001010000111000 (6 hits)

```

Similarly here, too the code has displayed the bucket in which the key was found and is present in how many stripes(hits)

Block Bitmaps

To store variable sized fingerprints, we use the concept of block bitmaps.

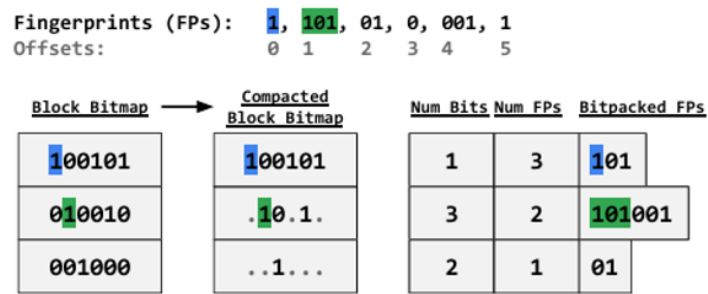


Figure 7: Dense storage of six variable-sized fingerprints in three different blocks. Each block stores fixed-bit-width fingerprints in bitpacked format. Block bitmaps indicate membership of a fingerprint in a block. We *compact* block bitmaps by “leaving out” set bits in subsequent bitmaps. By ordering blocks based on *decreasing cardinality* we increase the effect of this optimization.

Each block stores fingerprints of same size/width, along with the number of fingerprints and compacted block bitmaps. We can identify a fingerprint using its slot number.

To identify the block of a given slot in the Cuckoo table, we maintain one bitmap per block. Each such block bitmap indicates which slots are stored in the corresponding block.

To retrieve a fingerprint at a certain slot index:

- We check the corresponding bit of all block bitmaps until we encounter a set bit.
- We then extract the fingerprint from the corresponding block.

In particular, to compute the offset of the fingerprint in the bitpacked storage:

- We perform a **rank** operation on its block bitmap.

- Multiply that count with the bit width of the block.

Below is the pseudo-algorithm for compacted block bitmaps:

Algorithm 2: Returns the fingerprint stored in the slot with the given ID.

Input: *blocks, block_bitmaps, num_blocks, slot_idx*

Output: *fingerprint*

```

1 for block_idx in 1 to num_blocks do
2   block  $\leftarrow$  blocks[block_idx]
3   block_bitmap  $\leftarrow$  block_bitmaps[block_idx]
4   if GetBit(block_bitmap, slot_idx) then
5     if IsInactive(block) then
6       return NULL
7     idx_in_block  $\leftarrow$ 
8       Rank(block_bitmap, compacted_slot_idx)
9     return GetFingerprint(block, idx_in_block)

    // As we advance to the next block, offset
    // slot ID by the number of slots present in
    // the current block.
slot_idx  $\leftarrow$  slot_idx - Rank(block_bitmap, slot_idx)

```

Code Implementation:

```

def get_fingerprint_from_slot(bit_width, slot_idx):
    blocks = ci_index.get(bit_width, [])
    for block in blocks:
        if slot_idx >= len(block['block_bitmap']):
            slot_idx -= block['block_bitmap'].count("1")
            continue

        if block['block_bitmap'][slot_idx] == "1":
            idx_in_block = rank(block['block_bitmap'], slot_idx)
            return block['fingerprints'][idx_in_block]

        slot_idx -= rank(block['block_bitmap'], slot_idx)

    return None

```

Returns the fingerprint stored in block with given slot idx


```

def rank(bitmap, idx):
    return bitmap[:idx + 1].count("1")

def insert_into_block(fingerprint, stripe_id, bit_width):
    if bit_width not in ci_index:
        ci_index[bit_width] = []

    blocks = ci_index[bit_width]

    if not blocks or len(blocks[-1]['block_bitmap']) >= BLOCK_SIZE:
        blocks.append({
            'fingerprints': [],
            'stripe_bitmaps': [],
            'block_bitmap': []
        })

    block = blocks[-1]

    if fingerprint in block['fingerprints']:
        idx = block['fingerprints'].index(fingerprint)
    else:
        block['fingerprints'].append(fingerprint)
        block['stripe_bitmaps'].append(["0"] * NUM_STRIPES)
        block['block_bitmap'].append("1")
        idx = len(block['fingerprints']) - 1

    block['stripe_bitmaps'][idx][stripe_id] = "1"

```

Implementation of the rank function (that returns the index of the fingerprint in the block) and the insert into bucket function.

```

key:20163,stripe_id:20,bit_width:16
key:20163,stripe_id:20,bit_width:16
key:20164,stripe_id:22,bit_width:16
key:20164,stripe_id:14,bit_width:16
key:20164,stripe_id:5,bit_width:16
key:20164,stripe_id:23,bit_width:16
key:20164,stripe_id:0,bit_width:16
key:20164,stripe_id:19,bit_width:16
key:20165,stripe_id:3,bit_width:16
key:20165,stripe_id:29,bit_width:16
key:20165,stripe_id:9,bit_width:16
☑ All data inserted into CI block structure.
☑ Found in block 0, fingerprint=0100101000111000, stripes=6
☒ CI index persisted to PostgreSQL table `ci_index` (with merging).

```

OUTPUT

Implementation of compacted block bitmap storage was done in Python as part of the CI index table. As seen from the output figure, we are able to extract the fingerprint 10304 from bucket 0 and it is present in 6 stripes. Block bitmaps, thus speed up query processing, despite taking up space.

Rough overview of ci_index table in Python:

```
{
  6: {
    'fingerprints': ['010101', '110011', ...],
    'bitmaps': [
      ['0', '1', '0', ...], # for 010101
      ['0', '0', '1', ...], # for 110011
      ...
    ]
    'block_bitmap': "100...."
  },
  7: {...},
  ...
}
```

The benefit of block bitmaps over bucket is that it is faster for lookups and ensures great results for positive lookups.

In summary, CI filter works better than XOR and Bloom for all sizes. CI strongly benefits from an increasing hit rate since it guarantees exact results for positive lookups. Hence, depending on the workload characteristics, CI can also be beneficial for high cardinality, despite its possibly larger size. Thus, due to its low false positive rate and high hit rate, it is advantageous to all others.