Cuckoo Index: A Lightweight Secondary Index Structure

Andreas Kipf[1*], Damian Chromejko[2], Alexander Hall[3*],
Peter Boncz[4], David G. Andersen[5*]

[1]MIT CSAIL [2]Google [3]RelationalAI [4]CWI [5]CMU

kipf@mit.edu dchromejko@google.com alex.hall@relational.ai boncz@cwi.nl dga@cs.cmu.edu

ABSTRACT

In modern data warehousing, data skipping is essential for high query performance. While index structures such as B-trees or hash tables allow for precise pruning, their large storage requirements make them impractical for indexing secondary columns. Therefore, many systems rely on approximate indexes such as 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.

In this paper, we introduce Cuckoo Index (CI), an approximate secondary index structure that represents the many-to-many relationship between keys and data partitions in a highly space-efficient way. At its core, CI associates variable-sized fingerprints in a Cuckoo filter with compressed bitmaps indicating qualifying partitions. With our approach, we target equality predicates in a read-only (immutable) setting and optimize for space efficiency under the premise of practical build and lookup performance.

In contrast to per-partition (Bloom) filters, CI 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. Our experiments with real-world and synthetic data show that CI consumes significantly less space than per-partition filters for the same pruning power for low-to-medium cardinality columns. For high cardinality columns, CI is on par with its baselines.

PVLDB Reference Format:
Andreas Kipf, Damian Chromejko, Alexander Hall, Peter Boncz, and David G. Andersen. Cuckoo Index: A Lightweight Secondary Index Structure. *PVLDB*, 13(13): 3559-3572, 2020.
DOI: https://doi.org/10.14778/3424573.3424577

1. INTRODUCTION

Many multi-level or sharded storage systems use sketches such as Bloom filters [7] to reduce the number of accesses needed to find data items of interest. Examples include the LSM-tree LevelDB [1] and PowerDrill [21]. In each of these cases, searching for a data item d may have to look at many different "stripes" (chunks of data) where d could be stored, and the goal is to reduce the number of stripes that must be touched. Bloom filters and related approximate set-membership data structures are a good match for this problem, as they combine efficient space use with one-sided error appropriate for the problem at hand: The filter may occasionally require accessing a stripe that does not contain the data, but will always return a "true positive" result if the data is indeed there.

While most systems maintain one Bloom filter per stripe and may test each of them at query time, SlimDB [37] (an LSM-tree) suggests replacing per-level Bloom filters with a single Cuckoo filter [15] that maps each key fingerprint to the most recent level containing that key. Using this approach, SlimDB avoids lookups in multiple such filters. While this is a valid strategy for LSM-trees where data items (i.e., keys) are only associated with a single level, it does not handle the general case where data items may occur in multiple stripes.

In this paper, we extend this idea to the case of indexing secondary columns by associating each key indexed in a Cuckoo filter [15] with a bitmap indicating qualifying stripes (i.e., stripes containing that data item). We term this new data structure Cuckoo Index (CI). In contrast to a Cuckoo filter that uses a fixed number of fingerprint bits per key, CI stores *variable-sized* fingerprints to avoid *key shadowing* [39]. A fingerprint stored in CI uniquely identifies its key. With this design, CI provides correct results for lookups with keys that occur in the data (positive lookups). As with all schemes that do not store full keys, there can be false positives for lookups with non-occurring keys (negative lookups). To mitigate the impact of those, CI allows to fine-tune the expected number of false positive stripes.

We store the variable-sized fingerprints in a space-efficient block-based storage. Each block stores (bitpacked) fingerprints with a fixed number of bits. Additional bitmaps indicate the membership of a certain fingerprint in a block. Further, CI maximizes *primary bucket assignments* at construction time, which reduces space consumption.

For low-to-medium cardinality columns, CI is significantly smaller than per-stripe filters. The rationale behind the memory savings is simple: for secondary columns such as "country" each data item such as "US" will only be indexed once in CI while previously "US" would have been stored in multiple per-stripe filters. The extra space required by the bitmaps is insignificant compared to the space savings by only indexing each data item once.

[*]Work done while at Google.

Proceedings of the VLDB Endowment, Vol. 13, No. 13
ISSN 2150-8097.
DOI: https://doi.org/10.14778/3424573.3424577

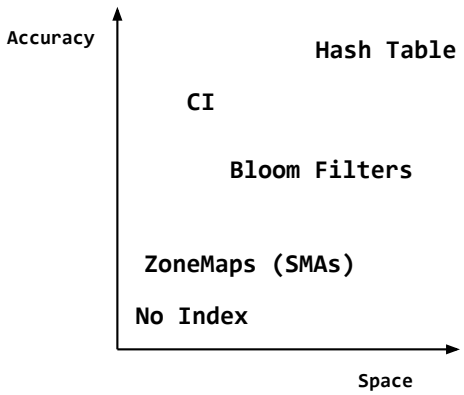


Figure 1: Cuckoo Index (CI) strikes a good balance between space and pruning power.

In our work, we assume an analytical storage system such as Mesa [20] or PowerDrill [21] that partitions columns into stripes of up to 65 K rows. Further, we assume a *secondary indexing* setting in which we cannot control the data placement (sort order). Data may be sorted by the primary key, by arrival time, or not at all, and we will show the effect of different sort orders on filter sizes and pruning power. Our focus is on improving the space efficiency (i.e., the trade-off between space and pruning power) of the filter. We assume that the costs for retrieving a false positive stripe from storage and scanning it in memory dominate performance. We further restrict the scope of this work to the "write once-read many" case common in both warehousing and log-structured systems, where data is only written once into an immutable state. Finally, we target equality predicates (e.g., country *equals* "US") which are very common in our use cases.

We evaluate CI with real-world (IMDb [30] and DMV [2]) and synthetic data and show that in many cases it consumes less space for the same pruning power than existing approaches. For example, for a medium cardinality column (with 6.34% of values being unique) and 8 K rows per stripe, CI consumes 23.7% less space than the best baseline, per-stripe Xor filters [19].

In summary, CI creates a new Pareto-optimal solution to the data skipping problem, striking a good balance between space efficiency and pruning power (cf. Figure 1).

Contributions. Our contributions include:

- A new lightweight secondary index structure with large space savings for low-to-medium cardinality columns and on-par space consumption for high cardinalities compared to per-stripe filters.
- A new heuristic-based construction of Cuckoo hash tables that maximizes the ratio of items assigned to their primary bucket.
- A comparison of this heuristic against an optimal solution that maximizes primary bucket assignments by solving a maximum weight matching problem.
- An unaligned bitmap compression scheme that is more space efficient than Roaring [8] in many cases while still allowing for partial decompression.
- An open-source implementation of our approach and a benchmarking framework for (approximate) secondary indexes. Both artifacts are available on GitHub[1].
- An evaluation of our approach using this framework on real-world and synthetic datasets.

2. BACKGROUND

In this section, we describe the storage layout used in this work, provide a problem definition, and give some background on Cuckoo filters.

Storage Layout. On a high level, we assume a PAX-style storage layout [5] in which a table is horizontally partitioned into segments (essentially files stored in distributed storage), which are organized in a columnar fashion (i.e., same attributes are grouped together) for better cache locality and compression. Such a data layout is typical for cloud databases such as Snowflake [10].

In our setting, a segment typically consists of 1-5 M rows. Each column in a segment is further divided into logical "stripes", consisting of up to 2^{16} (65,536) rows each. We consider a stripe the unit of access, meaning we always scan all values within a stripe. In other words, an index would allow us to skip over stripes, but not further reduce scan ranges within a stripe. Throughout this paper, our focus is on indexing a single segment (file). We further assume that the file API allows us to retrieve individual stripes from storage without reading the entire file. Such an API is supported by cloud storage systems such as Amazon S3.

Problem Definition. We define S as the set of stripes that represents a column within a segment. Given a data item d and a set of stripes S, the task is identify a subset $R \subseteq S$ of stripes that *potentially* contain d (i.e., false positives are allowed), without missing out on stripes that *do* contain d (i.e., no false negatives). The goal is to make R contain as few stripes as possible. In the best case, $R = C$ with $C \subseteq S$ being the stripes that actually contain d (true positives).

We measure the quality of an index using the **scan rate** metric. We define the scan rate as the ratio of false positive stripes $FP \subseteq R$ to true negative stripes $TN \subseteq S$ ($|TN| > 0$):

$$\textbf{scan rate} = \frac{|FP|}{|TN|} \tag{1}$$

In other words, the scan rate is the ratio of true negative stripes that is misclassified as positive. The idea behind defining scan rate like this is to align it with the false positive rate of per-stripe filters, assuming that lookups in different filters are independent of each other. When $|TN| = 0$, i.e., when a given value is contained all stripes, we set the scan rate to zero.

Each false positive stripe $fp \in FP$ comes with a penalty, most notably an unnecessary I/O. For queries that only select a few stripes (i.e., are very selective), an effective index can save a lot of unnecessary work, potentially leading to large speedups in overall query performance.

For practical purposes, we want the index structure to be significantly smaller than the compressed column. Thus, the overarching goal is to strike a good balance between space consumption and pruning power. Lookups in this structure should still be practical, i.e., significantly faster than a disk I/O which we assume to be in the order of milliseconds.

[1] `https://github.com/google/cuckoo-index/`

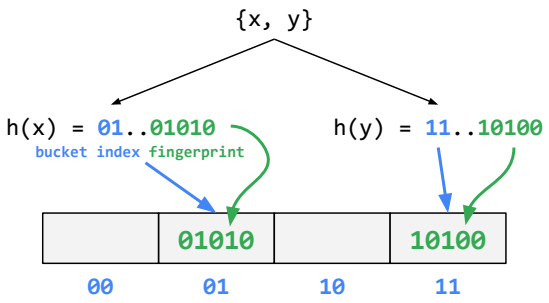


Figure 2: A Cuckoo filter.

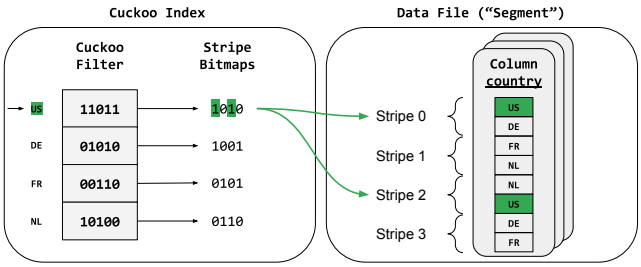


Figure 3: A Cuckoo Index is built for each column in a data file (segment). Each fingerprint in the Cuckoo filter is associated with a stripe bitmap that indicates qualifying stripes. The figure illustrates a query for the country "US".

Cuckoo Filter. Similar to a Bloom filter [7], a Cuckoo filter [15] is a data structure that can answer approximate set-membership queries: Given a lookup key, the filter determines whether the key *might be* contained in the set or is *definitely not* part of the set. Like a Cuckoo hash table [36], 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:

$$fingerprint(key) = extract(hash(key)) \quad (2)$$

For example, the least significant bits of the hash value can be used as fingerprint bits.

Similar to keys in a Cuckoo hash table, each fingerprint has two possible buckets that it can be placed in: its primary and its secondary bucket. If a bucket is full, the insertion algorithm "kicks" a random item to its alternate bucket and inserts the to-be-inserted item into the freed spot. There is a (configurable) maximum number of kicks that are allowed to occur during an insert after which construction fails. In such cases, a filter needs to be rebuilt from scratch with a more conservative configuration (i.e., more buckets).

Compared to a Cuckoo hash table that uses two independent hash functions on full keys, a Cuckoo filter uses *partial-key cuckoo hashing* [15] to find an alternative bucket i_2 for fingerprints stored in bucket i_1:

$$i_2 = hash(fingerprint) \oplus i_1 \quad (3)$$

However, the restriction of not being able to use two independent hash functions only applies in an online setting, i.e., already inserted items are no longer accessible. In this work, we are targeting the offline setting where all items are available at once, hence we are free to use independent hash functions on the full keys.

A Cuckoo filter typically reserves space for multiple items per bucket. This parameter is called bucket size b (or number of slots). With $b = 1$, the maximum load factor of the filter (ratio of occupied slots) is 50% and increases to 84% and 95% with $b = 2$ and $b = 4$, respectively [15].

3. CUCKOO INDEX

In this section, we first provide an overview over the design of Cuckoo Index (CI) before describing several optimizations to reduce space consumption and increase pruning power. We note that all design decisions are made with practical build and probe performance in mind.

3.1 Overview

On a high level, CI consists of a Cuckoo filter where each key fingerprint is associated with a fixed-size bitmap indicating qualifying stripes as shown in Figure 3. In contrast to traditional per-stripe filters, in this design there is only a *single* index per column and segment. 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. Note that there is a one-to-one mapping between fingerprints and bitmaps.

For low cardinality columns, such as "country", having only one such filter per column has the advantage that every unique value will only be indexed once. For example, assume that "US" is contained in every single stripe. With the per-stripe filter design, "US" would occupy around 10 bits per stripe (assuming a 1% target false positive rate). With say 20 stripes, it would require 200 bits. With CI's design, space consumption would be reduced to 10 bits for storing "US" in the filter and another 20 bits for the corresponding bitmap and thus save 85% in memory. For high cardinality columns, this design may consume more memory than the per-stripe filter design. However, this effect can be mitigated by cross-optimizing fingerprints and bitmaps as we will show later. Besides the potentially large space savings, another advantage of this design is that lookups only need to probe a single filter.

On the other hand, this design also introduces a few challenges. 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. In a regular Cuckoo filter, we would simply *not* insert the redundant fingerprint. In CI, however, each fingerprint has an associated value (i.e., a bitmap). Thus, we need a strategy to resolve such collisions. We discuss different options and their trade-offs in Section 3.2.

Another challenge is the representation of bitmaps. The most straightforward approach would be to allocate a separate array of uncompressed bitmaps that has the same size (i.e., number of entries) as the array that represents the Cuckoo filter, enabling a simple offset access from one into the other (cf. Figure 3). There are many possible optimizations to this design, such as bitmap compression or adding a level of indirection and only maintaining distinct bitmaps. We discuss our bitmap encoding scheme in Section 3.6.

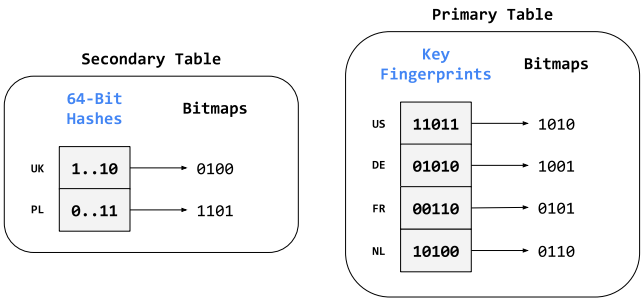


Figure 4: Storing 64-bit hashes of colliding keys in a secondary table to ensure a collision-free primary table that only stores short key fingerprints.

One of our key techniques is to *cross-optimize* (i.e., holistically optimize) fingerprints and bitmaps to satisfy a given scan rate (cf. Section 2). 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 anyways, thus a false positive that matches with a sparse bitmap is less expensive than a false positive that matches with a dense bitmap.

We choose a Cuckoo over a Bloom filter for the following reasons: (i) a Cuckoo filter has a lower space consumption than a Bloom filter for false positive rates under 3% [15], and (ii) in a Cuckoo filter, it is straightforward to associate values (bitmaps) with keys (fingerprints). We note that the recently proposed Xor filter [19] was not yet available at the design phase of this work. However, we expect it to have similar drawbacks as Invertible Bloom Lookup Table [18] (cf. Section 5) when associating it with bitmaps, including the possibility of unsuccessful lookups.

3.2 Collision Handling

We consider three different strategies to resolve fingerprint collisions. While `Union Bitmaps` introduces false positives, `Secondary Table` and `Variable-Sized Fingerprints` guarantee correct results for positive lookups. That is, for a lookup key that exists in the data, the latter two strategies always return the correct set of stripes, while `Union Bitmaps` may return false positives.

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. Such a conflict resolution would not be possible if we would associate each fingerprint with a regular value such as an SSTable ID in an LSM-tree [37] (i.e., one cannot *union* IDs without requiring extra space). 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. In the worst case, we may access $S-1$ false positive stripes with S being the total number of stripes (which is the case when a bitmap with one set bit is combined with a bitmap with S set bits). Since we aim for maximum pruning power, we now study strategies that do not introduce false positive stripes.

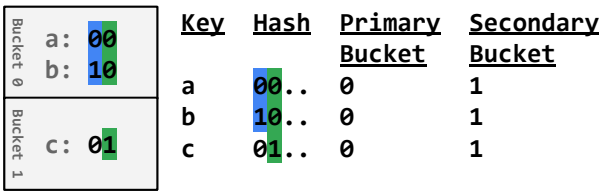


Figure 5: Three keys indexed in a CI with two slots per bucket (bitmaps omitted from figure). All keys have Bucket 0 as their primary bucket. Only c is placed in its secondary bucket, a and b are stored in their primary bucket. While the blue bit would be sufficient to differentiate between a and b, we need to store an extra bit (marked in green). Without this bit, a lookup for c would falsely get matched with a (a lookup always first checks the primary bucket). Hence, to determine the minimum number of fingerprint bits for Bucket 0, we need to consider all keys that have this bucket as primary bucket.

Colliding Entries in Secondary Table. The multi-level Cuckoo filter in SlimDB [37] addresses the problem of colliding fingerprints by introducing a secondary table (a hash table storing full keys) to store conflicting entries. Lookups first search the secondary table and only consult the primary table (the Cuckoo filter) if the key was not found in the secondary table. This approach not only affects lookup performance, but also may require a significant amount of extra space for the secondary table. For example, assuming a key length of 128 bits and a collision probability of 1%, the secondary table adds an overhead of 1.28 bits per key. This storage overhead of course increases with longer keys such as strings.

To bound the required space overhead, we could only store 64-bit hashes of keys in the secondary table (cf. Figure 4). Since our filter is intended as an immutable ("write once-read many") data structure, we know all keys at build time and can therefore ensure that their hashes are unique (i.e., there are no 64-bit hash collisions). In the unlikely event of a hash collision, we use a different hash function. This optimization allows us to effectively use this technique with string keys. The increased lookup cost, however, remains.

Variable-Sized Fingerprints. This design avoids the overhead of a secondary table by storing collision-free, variable-sized fingerprints in the primary table. 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 as shown in Figure 5. 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. To mitigate the size impact of this effect, it is important to maximize the ratio of items that are stored in their primary

buckets since those *do not* influence the fingerprint lengths of other buckets (as opposed to items stored in their secondary buckets). In Section 3.4 we will introduce two key-to-bucket-assignment algorithms that yield the optimal and an almost optimal *primary ratio*, respectively.

3.3 Scan Rate Optimization

As mentioned earlier, we use the scan rate metric (cf. Section 2) to measure the quality of our index, which is the ratio of false positive to true negative stripes. While our variable-length fingerprint design yields correct results for positive lookups (a guaranteed scan rate of 0%), it may return false positive stripes for lookups with non-existing keys.

CI therefore takes the target scan rate (for the entire index) as a parameter. 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.

Considering a single fingerprint/bitmap pair stored in a certain bucket, there are two components that determine the scan rate of this particular entry. First, 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:

$$\textbf{local scan rate} = \frac{1}{2^{\textit{fingerprint bits}}} * \textbf{bitmap density} \quad (4)$$

with **bitmap density** being defined as the ratio of set bits.

For high cardinality columns, we observe sparse bitmaps (depending on the data distribution among stripes) and thus we tend to require fewer fingerprint bits in these cases.

Algorithm 1 shows the complete process of determining the number of fingerprint bits for a certain bucket such that we avoid fingerprint collisions and satisfy a given target scan rate (for the entire index). We first compute the bucket density of the table (i.e., the ratio of non-empty buckets):

$$\textbf{table density} = \frac{\textit{num non-empty buckets}}{\textit{num buckets}} \quad (5)$$

Then we determine the minimum number of hash bits to avoid fingerprint collisions (cf. Section 3.2). Essentially, we increase the number of fingerprint bits until the fingerprints are unique. We now increase the number of hash bits until we satisfy the given target scan rate (cf. Lines 3 to 14). 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 (cf. Lines 5 to 9). 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.

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 ← GetBucketDensity(table)
   // Get minimum number of hash bits to avoid
      fingerprint collisions of keys that have
      bucket as primary bucket.
2  num_bits ← GetCollFreeHashLength(keys, bucket)
3  while true do
4      false_match_probability ← 1/2^num_bits
       // Compute scan rate of bucket by averaging
          local scan rates.
5      sum_scan_rate ← 0.0
6      for entry in bucket do
7          bitmap_density ←
             GetBitmapDensity(entry.bitmap)
8          sum_scan_rate +=
             false_match_probability * bitmap_density
9      actual_scan_rate ←
         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
```

3.4 Assigning Keys to Buckets

We follow two goals when assigning keys to buckets. First, and most importantly, we want to find a placement where all keys have a bucket. To ensure that this is possible, we need to size the Cuckoo table accordingly. That is, we need to allocate a sufficient number of buckets to accommodate the key set considering maximum load factors (that depend on the number of slots per bucket, cf. Section 2). Second, we aim to *maximize* the **primary ratio**:

$$\textbf{primary ratio} = \frac{\textit{num keys in primary buckets}}{\textit{num keys}} \quad (6)$$

We refer to items that reside in their primary or secondary buckets as primary or secondary items, respectively. With a high primary ratio, we mitigate the impact of secondary items influencing the fingerprint length of their primary buckets (cf. Section 3.2).

We have experimented with three different algorithms. First, we use the well-known kicking procedure that is also used by the Cuckoo filter. The only difference is that we do not use partial-key Cuckoo hashing and instead use two independent hash functions, assuming we have access to all

Table 1: Performance of different algorithms for key-to-bucket assignment.

Algorithm	AVG primary %	AVG matching time
Kicking	64%	192 ns
Matching	77%	101 μs
Biased Kicking	76%	1.2 μs

keys at build time. Second, we use a *matching algorithm* that finds the *optimal* placement of keys that maximizes the primary ratio (cf. Section 3.4.1). This approach is similar to prior work on using matching in a Cuckoo hashing context by Dietzfelbinger et al. [11] that aims to speed up lookups rather than decreasing footprint size. Lastly, in Section 3.4.2 we introduce a *biased* kicking algorithm that achieves almost optimal primary ratios in short time. We now describe the matching and the biased kicking algorithms.

3.4.1 Matching Algorithm

The kicking algorithm might produce a key-to-bucket assignment that assigns many keys to their secondary bucket or even fail to find an assignment altogether. To mitigate that, we model the problem as a minimum-cost unbalanced linear assignment problem. In order to maximize the ratio of primary items, we assign a cost of 1 to a matching between a key and its primary bucket and a cost of 2 to a matching with its secondary bucket. The problem can be solved using an algorithm proposed by Goldberg and Kennedy ([16], with an optimization from [17]) in $O(n\sqrt{n}\log(n))$, where n is the total number of keys and buckets.

3.4.2 Biased Kicking Algorithm

To avoid the high runtime of the matching-based approach, we modify the Cuckoo kicking algorithm to optimize the primary ratio. In particular, we consider both the primary and the secondary bucket of the current in-flight item to find a secondary item. We leave a small chance (an experimentally obtained likelihood) to kick primary items to resolve cycles. We set the maximum number of allowed kicks to a sufficiently high value (50 K kicks) to mitigate the impact of build failures in practice. Using this algorithm, we achieve primary ratios that are within 1% of the optimum found by the matching algorithm.

Table 1 and Figure 6 show the impact of using different assignment algorithms. We ran the experiments with 1 M unique values and 1, 2, 4, and 8 slots per bucket, using a single thread on a machine equipped with an Intel Xeon Skylake CPU (3.70 GHz).

3.5 Storing Variable-Sized Fingerprints

To store variable-sized fingerprints in a compact manner, we have devised a custom storage layout. Fingerprints can have from 0 (including[2]) up to 64 bits, even though we rarely see cases with more than 20 bits for real-world columns (assuming a 1% target scan rate). Figure 7 shows an example with six fingerprints and three different bit widths.

Fixed-Bit-Width Blocks. For each bit width, we allocate one *block* that stores fingerprints of a fixed bit width (in the example, Block 0 stores all 1-bit fingerprints). Each block

[2]That is the case when there is only a single key that has a bucket as primary bucket.

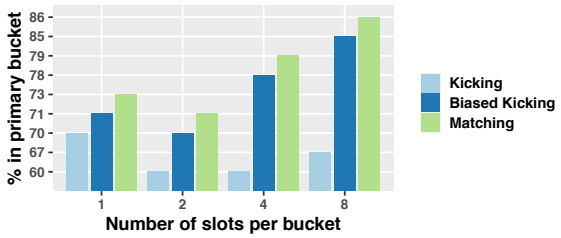


Figure 6: Performance of different algorithms for key-to-bucket assignment, broken down by the number of slots per bucket. Note the zoomed in y-axis.

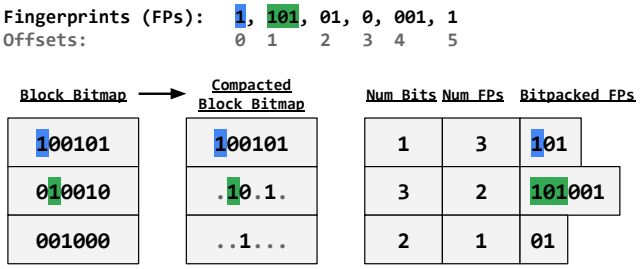


Figure 7: Dense storage of six variable-sized fingerprints in three different blocks. Each block stores fixed-bit-width fingerprints in bitpacked format. Block bitmaps indicate membership of a fingerprint in a block. We *compact* block bitmaps by "leaving out" set bits in subsequent bitmaps. By ordering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

stores the bit width and the number of fingerprints (both 32-bit integers), followed by the bitpacked fingerprints.

Block Bitmaps. 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. In addition, we maintain a bitmap that indicates empty slots. When encoding the bitmaps, we do not differentiate between this bitmap and the actual block bitmaps. The only difference is that this bitmap does not have a 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 and multiply that count with the bit width of the block. To accelerate `rank` calls, we maintain a rank lookup table that stores precomputed ranks (represented as `uint32_t`) for 512-bit blocks, which causes a space overhead of 6.25% [44].

Compacting Block Bitmaps. Based on the insight that block bitmaps are pairwise disjoint, we "leave out" *set* bits in subsequent bitmaps to save space (as shown in the middle of Figure 7). Using `rank` calls we can still reconstruct the bit offset of a slot at lookup time.

The complete fingerprint lookup algorithm (with compacted block bitmaps) is shown in Algorithm 2.

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 ← blocks[block_idx]
3     block_bitmap ← block_bitmaps[block_idx]
4     if GetBit(block_bitmap, slot_idx) then
5         if IsInactive(block) then
6             return NULL
7         idx_in_block ←
            Rank(block_bitmap, compacted_slot_idx)
8         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.
9     slot_idx ← slot_idx − Rank(block_bitmap, slot_idx)
```

3.6 Bitmap Encoding

In this section, we describe our approach to representing bitmaps in CI. We store one bitmap per slot in the Cuckoo table allowing for an offset access between the two.

Bitmap compression is essential to reduce the footprint of CI on disk and in memory. We use a custom bitwise (unaligned) run-length encoding (dense) scheme that can save significant space over word- or byte-aligned approaches such as WAH [41] or Roaring [8]. Our dense scheme consists of two vectors. The first stores 8-bit "run lengths" that consist of one bit to differentiate between a run and a literal, followed by 7 bits to denote the run length or the literal length. The second vector stores the compressed bit sequences. For sparse bitmaps that exceed an empirically obtained density threshold, we use bitpacked position lists (sparse scheme). That is, we encode the positions of the set bits with as few bits as possible.

We compress all bitmaps of a single CI at once: That is, we encode all bitmaps (back-to-back) in a single bitmap, which we call *global bitmap*, to achieve better compression. However, to access individual bitmaps, we require fast random access, without decompressing the entire global bitmap.

To allow for partial decompression, we maintain another vector that stores $\sqrt{\#runs}$ many *skip offsets*. One such offset entry contains the number of bits covered by the previous $\sqrt{\#runs}$ run lengths and the corresponding number of bits in the compressed bit sequence (recall that we store the run lengths and the compressed bits in two separate vectors). Thus, this structure effectively allows us to skip over chunks of run length entries. This approach is similar to fence pointers proposed by UpBit [6] with two differences: First, skip offsets contain relative bit counts whereas fence pointers contain absolute counts. Second, the size of our structure is relative to the size of the compressed bitmap while the size of fence pointers is relative to the size of the uncompressed bitmap. Hence, fence pointers favor lookup performance ($O(1)$ vs. $O(\sqrt{\#runs})$) whereas our structure favors space consumption.

For on-disk storage, we additionally apply Zstandard compression (zstd) [4] which further reduces the size and allows for efficient encoding and decoding.

Table 2: Cardinalities of selected columns.

Dataset	Column	#rows	Cardinality
IMDb	`country_code`	3,538,744	218
IMDb	`company_name`	3,538,744	224,385
IMDb	`title`	3,538,744	1,483,626
DMV	`color`	11,429,681	224
DMV	`city`	11,429,681	31,349

4. EVALUATION

We evaluate our approach with IMDb, DMV, and synthetic data and compare against ZoneMaps and per-stripe filters. In the following, we first describe the experimental setup, before we study the space consumption of the individual indexes. We also show the impact of different workloads (varying ratios of positive/negative lookups) and sort orders. We further experiment with synthetic data and report build and lookup times. Finally, we integrate CI into PostgreSQL and show its impact on query performance. We run all experiments on a machine with an Intel Xeon Gold 6230 CPU and 256 GiB of RAM.

Baselines. We compare our approach to ZoneMaps and to per-stripe approximate set-membership data structures. For the latter, we choose a space-optimized Bloom filter as well as the recently announced Xor filter [19] as representatives.

A ZoneMap maintains the minimum and the maximum value of each stripe. Its space consumption is constant and independent from the concrete data distribution. A ZoneMap is most effective when data is unique and sorted. A per-stripe filter is a collection of filters, and as the name suggests, one per stripe. By default, we construct each per-stripe filter with 10 bits per element, which yields a false positive probability of around 1% and 0.3% for Bloom and Xor, respectively. Thus, for a negative lookup, we expect 1% or 0.3% of stripes being misclassified as positive (assuming independence). As a Bloom filter implementation, we use LevelDB's built-in Bloom filter, which is a classical space-optimized Bloom filter. In the following, we refer to this approach as *Bloom*. For the Xor filter, we use its open-source implementation [3] and refer to it as *Xor*.

Datasets. We have denormalized the IMDb dataset [30] for our experiments. The resulting denormalized table has 3.5 M rows and 14 columns. DMV [2] contains vehicle, snowmobile, and boat registrations in New York State and consists of 11.9 M rows and 19 columns (we have excluded excluded the unique `vin` column, which is a primary index candidate). To simplify the experimental setup, we dictionary-encode strings as dense integers in an order-preserving way.

We focus our evaluation on the following representative columns: For IMDb, we choose `country_code`, `company_name`, and `title`, and for DMV, we choose `color` and `city`. Table 2 shows their cardinalities. The IMDb column `title` has the highest ratio of unique values (41.9%). We use the datasets in their *original* tuple order unless noted otherwise. In addition to this original order, we will study the cases when data is randomly shuffled or sorted.

Stripe Sizes. We load these datasets with four different stripe sizes, ranging from 2^{13} (8,192) to 2^{16} (65,536) rows per stripe (cf. Table 3 for the number of stripes).

Table 3: Number of stripes.

Rows per stripe	2^{13}	2^{14}	2^{15}	2^{16}
IMDb	431	215	107	53
DMV	1,395	697	348	174

This parameter affects both the space consumption and the pruning power of the different indexes. For example, CI's per-fingerprint bitmaps have as many bits as there are stripes. Thus, with fewer rows per stripe (and thus more stripes), the number of bits increases and CI may require more space. Similarly, and in particular for low cardinality columns, per-stripe filters may consume more space with fewer rows per stripe. Frequent values that occur in all stripes, are redundantly stored in every single per-stripe filter. On the other hand, a lower number of rows per stripe also means higher precision and has the advantage of less overhead in case of a false positive.

4.1 Space Consumption

We evaluate the sizes of the individual filters on our representative columns for different stripe configurations. We instantiate both Bloom and Xor with 10 bits per element which yields false positive stripe ratios (scan rates) of around 1% and 0.3%, respectively. We use CI in its most space-efficient configuration with one slot per bucket and a load factor of 49%. Recall that empty slots in CI are encoded using a single bit. We further use the biased kicking approach to maximize the primary ratio of items, which minimizes index size (cf. Section 3.4.2). To control the impact of lookups with non-occurring keys, we instantiate CI with two different scan rates: 0.1% (CI/0.1) and 1% (CI/1). We leave out ZoneMap from this experiment since its space consumption solely depends on the number of stripes.

In addition to the in-memory sizes, we report the sizes of the filters when compressed with Zstandard (zstd) [4]. Compressed filter sizes are interesting for data warehousing use cases where filters are stored alongside the actual (compressed) data on blob storage such as Google Cloud Storage. Compression may also reduce filter load times (I/O). We use zstd in its fastest mode (zstd=1) which still allows for decent compression ratios in our experience.

Figure 8a shows the filter sizes for the `country_code` IMDb column. The dashed line in the plot indicates 2% of the zstd-compressed column size. This column contains 218 unique values, representing 0.01% of the entire column. With more rows per stripe, the space consumption decreases for all approaches. Notably, CI consumes less space than Bloom and Xor for all stripe sizes. With zstd compression, the sizes of Xor and CI are further reduced. For example, with 2^{13} rows per stripe, Xor and CI/1 are 1.25× and 1.13× smaller when zstd-compressed, respectively. The reason that Xor compresses so well is its sparse table layout. Likewise, CI's bitmaps have additional compression potential. Bloom, on the other hand, does not benefit from compression. This is expected due to its already high entropy. When zstd-compressed, CI/1 consumes 8.90× (2^{13} rows per stripe) and 6.64× (2^{16}) less space than Xor, which achieves a comparable scan rate. The filter sizes for the medium cardinality column `company_name` are shown in Figure 8b. With zstd-compression, CI/1 is again smaller than Xor in all cases.

Table 4: Total size in MiB of compressed filters for IMDb (14 columns) and DMV (19 columns).

Dataset	CI/0.1	CI/1	Bloom	Xor
IMDb (2^{13})	5.39	**5.12**	6.63	6.45
IMDb (2^{16})	3.90	**3.26**	4.10	3.99
DMV (2^{13})	1.66	**1.63**	6.11	7.45
DMV (2^{16})	0.64	**0.60**	1.70	1.85

For example, for 2^{13} rows per stripe, CI/1 and Xor consume 1,121 KiB and 1,470 KiB, respectively. For the high cardinality column `title` the results look different (cf. Figure 8c). Here, CI/1 consumes more space than Xor. However, we argue that CI is still favorable over Xor in such cases, since it does not produce any false positives for lookups with occurring keys. A similar picture presents itself for the low and high cardinality columns in the DMV dataset (cf. Figure 9).

Table 4 shows the overall space consumption of the zstd-compressed filters over *all* IMDb/DMV columns. For both datasets, CI/1 consumes significantly less space than its competitors. Since DMV has lower cardinalities than IMDb (0.05% vs. 3.74% unique values across all columns), CI's advantage is higher on DMV.

In summary, CI is competitive with Bloom and Xor in terms of size for high cardinality columns, and is more space-efficient for low-to-medium cardinality columns. Also, we observe that CI's scan rate configuration does not have a large impact on its size.

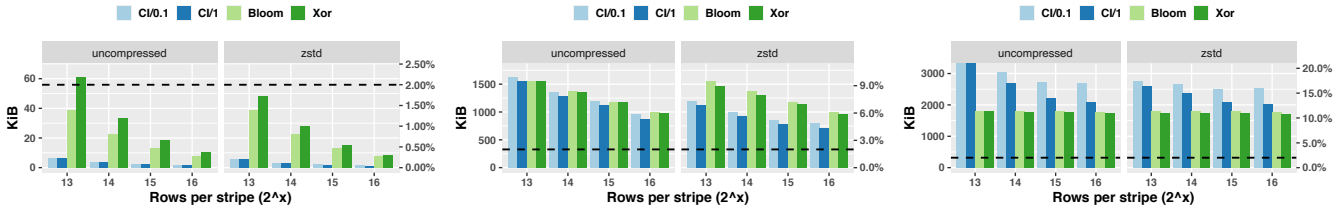
4.2 Mixed Workloads

We have shown that CI can be much smaller than per-stripe filters for a fixed scan rate (assuming only negative lookups). However, its real benefit only becomes clear when considering workloads that contain a mix of positive and negative lookups.

We therefore now we vary the *hit rate* of lookups. For example, a hit rate of 10% means that 10% of the lookups are with keys that occur in at least one stripe. We draw these keys uniformly from the set of column values and generate non-existing keys for negative lookups. We again exclude ZoneMap here, since it yields almost no pruning for unsorted data (cf. Section 4.3).

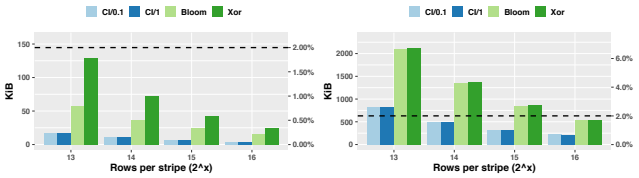
Figure 10 shows the results for the representative IMDb columns and 2^{13} rows per stripe. We omit DMV from this experiment since the results look similar to IMDb. Bloom and Xor are largely unaffected by the increase in positive lookups and achieve scan rates close to their configured false positive probabilities of around 1% and 0.3%, respectively. Notably, Xor uses approximately the same amount of memory than Bloom, and is thus to be preferred over Bloom in this setting. 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 columns such as `title`, despite its possibly larger size.

This experiment also shows that our defensive (per-bucket) scan rate estimation (cf. Section 3.3) *underestimates* the scan rate. For example, with a target scan rate of 1%, CI actually achieves a scan rate of less than 0.7% in all cases. We leave improvements to this estimation for future work.



(a) `country_code` (0.01% unique) (b) `company_name` (6.34% unique) (c) `title` (41.9% unique)

Figure 8: Uncompressed and zstd compressed filter sizes for IMDb columns. The right y-axis shows the size relative to the compressed column size. The dashed line is at 2% of the compressed column size.



(a) `color` (<0.00% unique) (b) `city` (0.27% unique)

Figure 9: Uncompressed filter sizes for DMV.

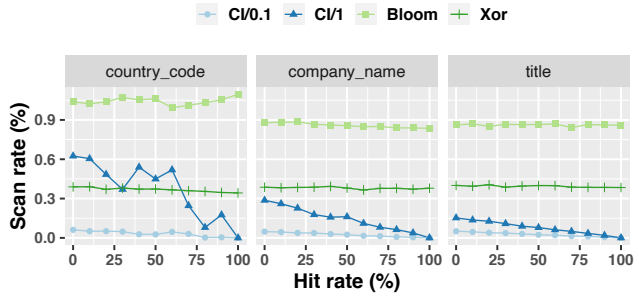


Figure 10: Scan rate with an increasing hit rate (ratio of positive lookups) and 2^{13} rows per stripe.

Table 5: DMV `city` column with different sort orders and 2^{13} rows per stripe.

	CI/1	Bloom	Xor	ZoneMap
Size (KiB, zstd)				
Original	552	2,093	1,986	1.66
Shuffled	556	2,360	2,239	1.42
Sorted	80	41	61	1.85
Scan rate (%)				
Original	0.00	0.89	0.39	99.9
Shuffled	0.00	0.87	0.38	99.9
Sorted	0.00	0.60	0.41	0.00

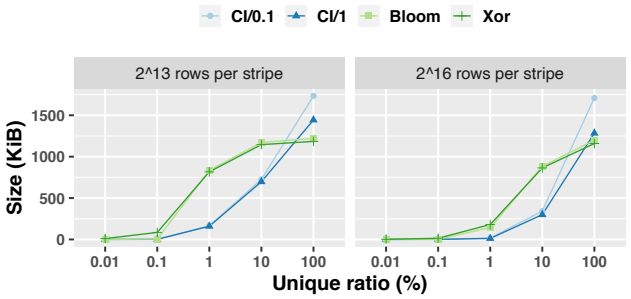


Figure 11: Compressed filter sizes for uniform data with 1 M rows and increasing cardinality.

4.3 Different Sort Orders

Now we study the impact of sort orders on index size and precision. We use the medium-cardinality column `city` (DMV dataset) and 2^{13} rows per stripe for this experiment. We note that the effect of different sort orders is similar for other columns. Table 5 shows the results. The size of the different (zstd-compressed) indexes stays about the same when shuffling the data. This is expected since cities are not particularly clustered in the original data. ZoneMap clearly consumes the least amount of space. When sorting the data, the space consumption of CI/1, Bloom, and Xor decreases. ZoneMap's size increases since there is now a higher number of different min/max values to encode. The reason that Bloom/Xor benefit from sorting is simple: with sorted data, there is less redundancy of unique values across stripes. That is, a single unique value is indexed in fewer Bloom/Xor filters. CI, on the other hand, indexes every unique value only once in any case (independent of the sort order). However, it is well known that sorting can have a large impact on bitmap compression [31], as we have confirmed with this experiment. While the scan rates (only positive lookups) of CI/1 and Xor are rather unaffected by the sort order, Bloom becomes more effective with clustered data. ZoneMap even achieves near-optimal accuracy.

4.4 Uniform Data

Next, we experiment with uniform data and vary the cardinality and the number of rows. We first fix the number of rows to 1 M and increase the cardinality. Figure 11 shows the effect of increasing cardinality on the compressed filter sizes. Up to 10% unique values CI consumes significantly less space than Bloom and Xor. The space savings are more pronounced for fewer rows per stripe. While CI consumes more space than Bloom and Xor with 100% unique values (i.e., a unique column), it may still be preferable in that setting due to its higher accuracy under positive and mixed workloads (cf. Section 4.2). We now fix the cardinality to 10 K and increase the number of rows (and implicitly the number of stripes). As can be seen in Figure 12, filter sizes increase with more rows (stripes).

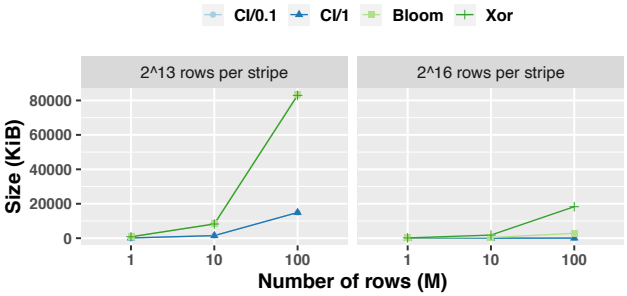


Figure 12: Compressed filter sizes for uniform data with 10 K unique values and an increasing number of rows.

Table 6: Build times in milliseconds with 1 M rows and 10 K unique values.

Rows per stripe	CI/1	Bloom	Xor
2^{13}	76.3	89.6	41.8
2^{16}	45.0	54.9	13.9

The size increase for both Bloom/Xor and CI can be explained by the increased redundancy of unique values among stripes. Put differently, a unique value will be present in more stripes. Since Bloom and Xor maintain one filter per stripe, each unique value is indexed in more filters. In contrast, CI indexes each unique value only once, independent of the number of stripes. However, the number of stripes affects CI's bitmaps, which maintain one bit per stripe. The reason that CI is less affected by the row count is that the added bitmap bits cause less space overhead than the redundant indexing in Bloom/Xor. CI's bitmap compression also mitigates the effect of additional bits to some degree. Likewise, both CI and Bloom/Xor are affected by the number of rows per stripe, with Bloom/Xor being more affected than CI. The reason is again the redundancy of unique values among stripes. Fewer rows per stripe mean more stripes and thus there is more redundancy, which is less of an issue for CI than for Bloom/Xor.

4.5 Build and Lookup Performance

We now study the build and lookup performance of CI compared to Bloom and Xor. We omit CI/0.1 here since it has very similar build and lookup times as CI/1. While the lookup code is partially optimized (with a rank lookup table for each block bitmap), the build code is unoptimized.

Build Performance. Table 6 shows the build times of the different filters with 1 M rows and 10 K unique values. All filters are affected by the number of rows per stripe. For Bloom and Xor the reason is that they need to create more (albeit possibly smaller) filters with more stripes. For CI, in contrast, the reason is the larger bitmaps that need to be compressed. In a breakdown analysis of CI's build time with 2^{13} rows per stripe we found that most time is spent on scanning the data and collecting unique values with their corresponding bitmaps (23.6%), finding the minimum required fingerprint length per slot to ensure no collisions and to satisfy the given scan rate (32.5%), and compressing the global bitmap (34.7%). Against our own intuition, we did not find distributing values among buckets (kicking) to be expensive. We now vary the cardinality. With 2^{13} rows per stripe, the build times in milliseconds of CI (Bloom) with 1 K (0.1%) and 1 M (100%) unique values are 16.6 (47.6) and 5,448 (104), respectively.

Table 7: Lookup latency in nanoseconds with 1 M rows and 10 K unique values.

Lookups	Rows per stripe	CI/1	Bloom	Xor
Positive	2^{13}	13,458	7,815	1,715
Positive	2^{16}	5,564	964	108
Negative	2^{13}	769	6,942	814
Negative	2^{16}	610	836	96.3

Table 8: Size and lookup latency of our compression scheme compared to Roaring (R).

Column	Ours-KiB	R-KiB	Ours-ns	R-ns
`country_code`	5.77	16.0	3,810	3,396
`company_name`	1,425	2,495	2,589	3,704
`title`	3,157	2,966	4,879	3,464

Lookup Performance. Table 7 shows the lookup latency of CI/1 compared to Bloom and Xor. For positive lookups, CI has a significant lookup overhead over Bloom/Xor. Under negative lookups, CI shows much better numbers. Only in the event of a false positive match in the Cuckoo table, it has to extract a bitmap from the compressed global bitmap (cf. Section 3.6). With our current bitmap encoding scheme, the cost of this operation is linear in the size of the compressed bitmap. With more read-optimized schemes such as Roaring, one could trade size for lookup latency (cf. Section 4.6). Note that we use a rank lookup table to speed up lookups in the block bitmaps that indicate the membership of a fingerprint in a block (cf. Section 3.5). Without this optimization, negative lookups would be significantly slower. Bloom and Xor, on the other hand, are less affected by the lookup type. As expected, both per-stripe filters show a linear increase in lookup latency with fewer rows per stripe. CI, on the other hand, is rather unaffected by the number of stripes. With 2^{13} rows per stripe and positive lookups, 93.0% of CI's lookup time is spent on extracting the bitmap that indicates qualifying stripes. Under negative lookups, the most expensive operation is querying the Cuckoo table with 73.5%. Now we again vary the cardinality. With 2^{13} rows per stripe, the latencies in ms for positive lookups of CI (Bloom) with 1 K (0.1%) and 1 M (100%) unique values are 5,673 (7,608) and 3,006 (7,442), respectively.

4.6 Bitmap Compression

We now compare the size and decompression speed of our bitmap encoding scheme (cf. Section 3.6) with Roaring. We use the IMDb columns `country_code`, `company_name`, and `title` with 2^{13} rows per stripe for this experiment. The specific task is to encode the global bitmap and to extract sub-bitmaps. Recall that each sub-bitmap indicates the qualifying stripes of a given key fingerprint stored in the Cuckoo table. We construct a CI/1 index, query with random keys

Table 9: Index sizes in MiB for TPC-H columns (CI is zstd-compressed).

Index	o_custkey	l_orderkey	l_partkey	l_suppkey
btree	322	1,286	1,287	1,287
hash	476	1,973	1,897	2,379
CI (2^{13})	15.6	49.5	68.5	35.2
CI (2^{16})	10.2	31.4	46.7	11.0

from the set of keys, and measure the time taken to extract the sub-bitmap associated with the fingerprint. For Roaring, we intersect the global bitmap with a query bitmap with the relevant bits set. Note that storing individual Roaring bitmaps (as opposed to encoding them back-to-back) would increase its total size by up to 4× (presumably due to the header that is stored with every bitmap). Hence, encoding individual bitmaps as Roaring is not an option. Table 8 shows the results. For country_code and company_name, our compression scheme consumes the least space whereas Roaring is more space-efficient for the sparse title bitmaps. In terms of extracting sub-bitmaps, our encoding scheme is competitive with Roaring.

4.7 Impact on Query Performance

Finally, we integrate CI/1 and Bloom into PostgreSQL (PG) version 12.3. Specifically, we partition lineitem and orders by orderdate with 2^{13} and 2^{16} rows per stripe (partition). We build the indexes on all columns that are subject to (join) predicates. At query time, we probe the respective index and pass the returned partition IDs to PostgreSQL's partition pruner.

We set the size of Bloom to be similar to the size of CI (when zstd-compressed). Xor's implementation does not allow to set the number of bits per element, hence we exclude Xor from this experiment. Since PG does not parallelize queries with partitioning pruning, we disable parallelism to ensure a fair comparison against several baselines, including a full scan and PG's built-in btree and hash indexes. We run three different queries on TPC-H with SF10:

```
Q1 select sum(l_extendedprice*(1-l_discount))
     revenue from lineitem, orders
     where l_orderkey=o_orderkey
     and o_custkey=36901 and l_returnflag='R';

Q2 select sum(l_extendedprice*(1-l_discount))
     revenue from lineitem where l_partkey=155190;

Q3 select sum(l_extendedprice*(1-l_discount))
     revenue from lineitem where l_suppkey=7706;
```

Table 9 shows the index sizes in MiB and Table 10 shows the query latencies in milliseconds. CI is significantly smaller than PG's built-in indexes, which index individual tuples rather than stripes. As expected, CI's size decreases with more rows per stripe.

In terms of query latency, PG's hash index achieves the best performance, followed by btree. CI and Bloom cover the middle ground between the full-fledged indexes and a full scan. Compared to Bloom and with 2^{13} rows per stripe, CI returns on average 2.10× fewer partitions and achieves a 2.45× speedup in query performance.

Table 10: Query latencies in milliseconds.

Index	Q1	Q2	Q3
No index	6,628	4,116	4,236
btree	0.48	0.25	0.79
hash	0.37	0.17	0.68
CI (2^{13})	50.4	34.8	894
Bloom (2^{13})	95.6	110	2,045
CI (2^{16})	456	306	5,284
Bloom (2^{16})	602	887	7,405

5. RELATED WORK

There have been many proposals to secondary indexing striking different balances between space and pruning power.

Small Materialized Aggregates. A popular approach is to maintain Small Materialized Aggregates (SMAs) [33] (e.g., min and max values) per stripe. SMAs (or ZoneMaps) consume little space and are most effective when data is (partially) sorted. Besides sorting, outliers can impact the precision of ZoneMaps. Positional SMAs (PSMAs) [28] extend SMAs with a compact lookup table that maps bytes of values to scan ranges. In contrast to both SMA variants, CI is not impacted by outliers and does not require the data to be (partially) sorted to be effective. Although, data clustering can decrease its size as we have shown.

Approximate Set Membership Structures. Another well-known approach is to maintain one set-membership filter (e.g., a Bloom filter [7, 29]) *per stripe*. With about 10 bits per unique value, Bloom filters produce around 1% false positives. While per-stripe filters are a good strategy for high cardinality columns, their size consumption is far from optimal for low cardinality columns. In addition, to identify all qualifying stripes for a given lookup key, we need to probe *all* per-stripe filters, which becomes increasingly expensive with higher resolutions (more stripes). Other set membership filters such as the Cuckoo filter [15] or the recently proposed Xor filter [19, 12] improve space efficiency but the problem remains: with many *redundant* values across stripes, their space consumption is sub-optimal. Also, and most notably, per-stripe filters may return false positive *stripes* under positive lookups. This is in contrast to CI, which is 100% correct under positive lookups.

Approximate Lookup Tables. Approximate lookup tables allow for associating keys with values with a low probability of false mappings. The most popular example is the Bloomier filter [9] and its follow-up Invertible Bloom Lookup Table (IBLT) [18]. In contrast to our work, Bloomier aims to improve lookup performance. It consists of multiple Bloom filters and requires all filters to use the same number of bits, which leads to sub-optimal space utilization under skew. IBLT allows to list its contents with some probability. A key issue with IBLT is that a lookup may return "not found" for a key. Since we cannot tolerate false negatives, IBLT is not applicable to our use case.

Closely related to our work, Ren et al. [37] propose to maintain a single higher-level Cuckoo filter instead of many "per-LSM-tree-node" filters in an LSM-tree. Each key in the filter maps to the ID of the most recent LSM-tree node containing that key. In contrast, we are interested in all of the

data partitions (stripes) that contain a given key. To resolve fingerprint collisions in the filter, Ren et al. store colliding entries in an external table. Our variable-sized fingerprints offer additional size benefits over that approach.

Cuckoo Hashing Extensions. Our idea of maximizing primary bucket assignments by solving a maximum weight matching problem (cf. Section 3.4) is not entirely new. In fact, it is well known that matching algorithms on bipartite graphs (e.g., the Hopcroft-Karp algorithm [23]) can be used to assign values to buckets in the setting of Cuckoo hashing. The thesis work by Michael Rink [38] provides a good overview in that respect. Furthermore, Dietzfelbinger et al. [11] show that matching algorithms can be used to maximize the ratio of items that end up in their primary location (in the context of Cuckoo hash tables that are divided into *pages*). In contrast to this work, we are interested in reducing the average fingerprint length rather than minimizing page accesses. Reducing the likelihood that we need to access an item's secondary bucket on lookup is rather a positive side effect for us. The same work [11] also shows that a biased insertion procedure that prefers primary locations in case of full buckets achieves almost the same result in short time. Similar to this work, we have developed a biased *kicking* algorithm that prefers kicking items that reside in their secondary locations to maximize primary bucket assignments. To the best of our knowledge, our work is the first to apply such an algorithm in the context of Cuckoo filters (cf. Section 3.4.2).

Minimal Perfect Hashing. Minimal perfect hashing is applicable to the problem studied in this work. The idea is to compute a minimal perfect hash function that maps n keys to the dense integers $[0, n-1]$. In our context, these integers would be offsets into an array of bitmaps indicating qualifying stripes. Esposito et al. [14] have shown that such a hash function only requires storing 1.56 bits per key in practice. While this sounds attractive, constructing such a function remains computationally expensive. Nevertheless, using a minimal perfect hash function such as RecSplit [14] is an attractive alternative to the "Cuckoo part" of our index (which can be seen as a variant of perfect hashing) to further reduce its footprint. The challenge of efficiently storing variable-sized fingerprints and bitmaps, however, remains.

Data Skipping Techniques. In general, there is plenty of research on (secondary) indexing and data skipping. Most related is bitmap indexing which maintains a bitmap for each unique column value indicating qualifying *tuples* [8, 35, 43]. There are two straightforward extensions to classical bitmap indexing. One is to make bitmaps more coarse grained which is similar to our setting. Another is to combine the bitmaps of multiple values, typically values within a certain range (*binning*). Both extensions trade precision for reduced footprint.

Column Imprints [40] and Column Sketches [22] both allow to accelerate in-memory scans by maintaining lossy index structures in the form of bit vectors or fixed-width codes. As opposed to our work, both techniques aim to reduce the cost of in-memory scans and not the number of disk accesses. Specifically, Column Imprints exploit the idea that real data naturally exhibits very localized correlations. Taking them to a much larger granularity than cache lines would make them similar to ZoneMaps.

Recently, machine learning (ML) found its way into indexing. While learned indexes [26, 24, 32, 25] may consume significantly less space than traditional indexes such as B-trees, that observation mostly applies to primary indexing where the base data is already sorted. In a secondary indexing setting such as ours, additional permutation vectors have to be maintained which will likely account for most of the space consumption of the learned index. Follow-up work on learned multi-dimensional indexes [34, 13] features a small footprint and allows for filtering on secondary columns but—in contrast to our approach—requires full control over the sort order of the data. Furthermore, learned indexes have not been adapted to the *approximate* indexing use case as studied in this work. Other ML-based work by Wu et al. [42] allows to construct succinct secondary indexes by exploiting column correlations. However, in contrast to our work, it requires access to a primary index.

6. CONCLUSIONS

We have introduced Cuckoo Index, a lightweight secondary index structure that is targeted at "write once-read many" environments. We have extended Cuckoo filters with variable-sized fingerprints to avoid key shadowing. With this design, our approach allows for correct results for positive lookups and further allows for tuning the *scan rate* for negative lookups. Our novel cross-optimization of fingerprints and bitmaps elegantly makes use of the fact that for sparse bitmaps one needs to store fewer fingerprint bits to achieve a desired scan rate. We have described a heuristic that achieves an almost optimal key-to-bucket assignment which minimizes index size. Using real-world and synthetic data, we have demonstrated that our approach consumes less space than per-stripe filters for low-to-medium cardinality columns that represent the majority of secondary columns in our experience. While we have focused on space efficiency, lookup performance can matter, particularly when the cost for a false positive is relatively low. The cost of a lookup in CI is dominated by `rank` calls on block bitmaps and decompressing stripe bitmaps. We have accelerated rank calls using a rank lookup table. Further, we have shown that our bitwise bitmap encoding compresses better than Roaring [8] while also allowing for partial decompression.

While we have designed CI for immutable environments, using an update-friendly bitmap encoding such as Roaring or Tree-Encoded Bitmaps [27] CI could support updates and deletions since both operations only require updating the bitmap part of the index. To support inserts, we would also need to update the Cuckoo table. One strategy is to overprovision the Cuckoo table, i.e., allocate more slots than required. Another is to allocate an additional k bits per fingerprint, which would allow for doubling the size of the Cuckoo table k times.

Currently, our index is limited to equality predicates. To support range predicates, one may consider replacing the Cuckoo table with a range filter such as SuRF [44]. Variable-sized key suffixes in SuRF would allow for correct results under positive lookups. However, since key suffixes follow a real rather than a uniform (hash) distribution, the conflict probability of two keys may be higher.

In future work, we plan to extend our approach to indexing multiple columns at once which opens up interesting opportunities such as exploiting column correlations.

7. REFERENCES

[1] LevelDB. https://github.com/google/leveldb/.

[2] Vehicle, snowmobile, and boat registrations. https://catalog.data.gov/dataset/vehicle-snowmobile-and-boat-registrations.

[3] Xor Filter. https://github.com/FastFilter/xor_singleheader/.

[4] Zstandard Compression. https://facebook.github.io/zstd/.

[5] A. Ailamaki, D. J. DeWitt, M. D. Hill, and M. Skounakis. Weaving relations for cache performance. In *VLDB 2001, Proceedings of 27th International Conference on Very Large Data Bases, September 11-14, 2001, Roma, Italy*, pages 169–180, 2001.

[6] M. Athanassoulis, Z. Yan, and S. Idreos. UpBit: Scalable in-memory updatable bitmap indexing. In *Proceedings of the 2016 International Conference on Management of Data, SIGMOD Conference 2016, San Francisco, CA, USA, June 26 - July 01, 2016*, pages 1319–1332, 2016.

[7] B. H. Bloom. Space/time trade-offs in hash coding with allowable errors. *Commun. ACM*, 13(7):422–426, 1970.

[8] S. Chambi, D. Lemire, O. Kaser, and R. Godin. Better bitmap performance with Roaring bitmaps. *Softw., Pract. Exper.*, 46(5):709–719, 2016.

[9] B. Chazelle, J. Kilian, R. Rubinfeld, and A. Tal. The Bloomier Filter: An efficient data structure for static support lookup tables. In *SODA*, pages 30–39, 2004.

[10] B. Dageville, T. Cruanes, M. Zukowski, V. Antonov, A. Avanes, J. Bock, J. Claybaugh, D. Engovatov, M. Hentschel, J. Huang, A. W. Lee, A. Motivala, A. Q. Munir, S. Pelley, P. Povinec, G. Rahn, S. Triantafyllis, and P. Unterbrunner. The Snowflake elastic data warehouse. In *Proceedings of the 2016 International Conference on Management of Data, SIGMOD Conference 2016, San Francisco, CA, USA, June 26 - July 01, 2016*, pages 215–226, 2016.

[11] M. Dietzfelbinger, M. Mitzenmacher, and M. Rink. Cuckoo hashing with pages. In *Algorithms - ESA 2011 - 19th Annual European Symposium, Saarbrücken, Germany, September 5-9, 2011. Proceedings*, pages 615–627, 2011.

[12] M. Dietzfelbinger and R. Pagh. Succinct data structures for retrieval and approximate membership (extended abstract). In *Automata, Languages and Programming, 35th International Colloquium, ICALP 2008, Reykjavik, Iceland, July 7-11, 2008, Proceedings, Part I: Tack A: Algorithms, Automata, Complexity, and Games*, pages 385–396, 2008.

[13] J. Ding, V. Nathan, M. Alizadeh, and T. Kraska. Tsunami: A learned multi-dimensional index for correlated data and skewed workloads. *CoRR*, abs/2006.13282, 2020.

[14] E. Esposito, T. M. Graf, and S. Vigna. RecSplit: Minimal perfect hashing via recursive splitting. In *Proceedings of the Symposium on Algorithm Engineering and Experiments, ALENEX 2020, Salt Lake City, UT, USA, January 5-6, 2020*, pages 175–185, 2020.

[15] B. Fan, D. G. Andersen, M. Kaminsky, and M. Mitzenmacher. Cuckoo Filter: Practically better than Bloom. In *CoNEXT*, pages 75–88, 2014.

[16] A. V. Goldberg and R. Kennedy. An efficient cost scaling algorithm for the assignment problem. *Math. Program.*, 71:153–177, 1995.

[17] A. V. Goldberg and R. Kennedy. Global price updates help. *SIAM J. Discrete Math.*, 10(4):551–572, 1997.

[18] M. T. Goodrich and M. Mitzenmacher. Invertible Bloom Lookup Tables. *CoRR*, abs/1101.2245, 2011.

[19] T. M. Graf and D. Lemire. Xor Filters: Faster and smaller than Bloom and Cuckoo Filters. *CoRR*, abs/1912.08258, 2019.

[20] A. Gupta, F. Yang, J. Govig, A. Kirsch, K. Chan, K. Lai, S. Wu, S. G. Dhoot, A. R. Kumar, A. Agiwal, S. Bhansali, M. Hong, J. Cameron, M. Siddiqi, D. Jones, J. Shute, A. Gubarev, S. Venkataraman, and D. Agrawal. Mesa: Geo-replicated, near real-time, scalable data warehousing. *PVLDB*, 7(12):1259–1270, 2014.

[21] A. Hall, O. Bachmann, R. Büssow, S. Ganceanu, and M. Nunkesser. Processing a trillion cells per mouse click. *PVLDB*, 5(11):1436–1446, 2012.

[22] B. Hentschel, M. S. Kester, and S. Idreos. Column Sketches: A scan accelerator for rapid and robust predicate evaluation. In *Proceedings of the 2018 International Conference on Management of Data, SIGMOD Conference 2018, Houston, TX, USA, June 10-15, 2018*, pages 857–872, 2018.

[23] J. E. Hopcroft and R. M. Karp. An $n^{5/2}$ algorithm for maximum matchings in bipartite graphs. *SIAM J. Comput.*, 2(4):225–231, 1973.

[24] A. Kipf, R. Marcus, A. van Renen, M. Stoian, A. Kemper, T. Kraska, and T. Neumann. SOSD: A benchmark for learned indexes. *NeurIPS Workshop on Machine Learning for Systems*, 2019.

[25] A. Kipf, R. Marcus, A. van Renen, M. Stoian, A. Kemper, T. Kraska, and T. Neumann. RadixSpline: a single-pass learned index. In *Proceedings of the Third International Workshop on Exploiting Artificial Intelligence Techniques for Data Management, aiDM@SIGMOD 2020, Portland, Oregon, USA, June 19, 2020*, pages 5:1–5:5, 2020.

[26] T. Kraska, A. Beutel, E. H. Chi, J. Dean, and N. Polyzotis. The case for learned index structures. In *SIGMOD*, pages 489–504, 2018.

[27] H. Lang, A. Beischl, V. Leis, P. A. Boncz, T. Neumann, and A. Kemper. Tree-Encoded Bitmaps. In *Proceedings of the 2020 International Conference on Management of Data, SIGMOD Conference 2020, online conference [Portland, OR, USA], June 14-19, 2020*, pages 937–967, 2020.

[28] H. Lang, T. Mühlbauer, F. Funke, P. A. Boncz, T. Neumann, and A. Kemper. Data Blocks: Hybrid OLTP and OLAP on compressed storage using both vectorization and compilation. In *Proceedings of the 2016 International Conference on Management of Data, SIGMOD Conference 2016, San Francisco, CA, USA, June 26 - July 01, 2016*, pages 311–326, 2016.

[29] H. Lang, T. Neumann, A. Kemper, and P. A. Boncz. Performance-optimal filtering: Bloom overtakes Cuckoo at high-throughput. *PVLDB*, 12(5):502–515, 2019.

[30] V. Leis, A. Gubichev, A. Mirchev, P. Boncz, A. Kemper, and T. Neumann. How good are query optimizers, really? *PVLDB*, 9(3), 2015.

[31] D. Lemire, O. Kaser, and K. Aouiche. Sorting improves word-aligned bitmap indexes. *Data Knowl. Eng.*, 69(1):3–28, 2010.

[32] R. Marcus, A. Kipf, A. van Renen, M. Stoian, S. Misra, A. Kemper, T. Neumann, and T. Kraska. Benchmarking learned indexes. *CoRR*, abs/2006.12804, 2020.

[33] G. Moerkotte. Small Materialized Aggregates: A light weight index structure for data warehousing. In *VLDB*, pages 476–487, 1998.

[34] V. Nathan, J. Ding, M. Alizadeh, and T. Kraska. Learning multi-dimensional indexes. In *Proceedings of the 2020 International Conference on Management of Data, SIGMOD Conference 2020, online conference [Portland, OR, USA], June 14-19, 2020*, pages 985–1000, 2020.

[35] P. E. O'Neil and D. Quass. Improved query performance with variant indexes. In *SIGMOD 1997, Proceedings ACM SIGMOD International Conference on Management of Data, May 13-15, 1997, Tucson, Arizona, USA*, pages 38–49, 1997.

[36] R. Pagh and F. F. Rodler. Cuckoo hashing. *J. Algorithms*, 51(2):122–144, 2004.

[37] K. Ren, Q. Zheng, J. Arulraj, and G. Gibson. SlimDB: A space-efficient key-value storage engine for semi-sorted data. *PVLDB*, 10(13):2037–2048, 2017.

[38] M. Rink. *Thresholds for Matchings in Random Bipartite Graphs with Applications to Hashing-Based Data Structures*. PhD thesis, Technische Universität Ilmenau, Germany, 2015.

[39] S. Shi, C. Qian, and M. Wang. Re-designing compact-structure based forwarding for programmable networks. In *27th IEEE International Conference on Network Protocols, ICNP 2019, Chicago, IL, USA, October 8-10, 2019*, pages 1–11, 2019.

[40] L. Sidirourgos and M. L. Kersten. Column Imprints: a secondary index structure. In *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2013, New York, NY, USA, June 22-27, 2013*, pages 893–904, 2013.

[41] K. Wu, E. J. Otoo, and A. Shoshani. Optimizing bitmap indices with efficient compression. *ACM Trans. Database Syst.*, 31(1):1–38, 2006.

[42] Y. Wu, J. Yu, Y. Tian, R. Sidle, and R. Barber. Designing succinct secondary indexing mechanism by exploiting column correlations. In *Proceedings of the 2019 International Conference on Management of Data, SIGMOD Conference 2019, Amsterdam, The Netherlands, June 30 - July 5, 2019*, pages 1223–1240, 2019.

[43] E. T. Zacharatou, F. Tauheed, T. Heinis, and A. Ailamaki. RUBIK: efficient threshold queries on massive time series. In *Proceedings of the 27th International Conference on Scientific and Statistical Database Management, SSDBM '15, La Jolla, CA, USA, June 29 - July 1, 2015*, pages 18:1–18:12, 2015.

[44] H. Zhang, H. Lim, V. Leis, D. G. Andersen, M. Kaminsky, K. Keeton, and A. Pavlo. SuRF: Practical range query filtering with fast succinct tries. In *Proceedings of the 2018 International Conference on Management of Data, SIGMOD Conference 2018, Houston, TX, USA, June 10-15, 2018*, pages 323–336, 2018.