Fast Pseudoalignment Queries on Compressed Colored de Bruijn Graphs

Alessio Campanelli
Ca' Foscari University of Venice, Italy

Giulio Ermanno Pibiri
Ca' Foscari University of Venice, Italy
ISTI-CNR, Pisa, Italy

Rob Patro
University of Maryland, College Park, MD, USA

Abstract

Motivation. Indexes for the colored de Bruijn graph (c-dBG) play a crucial role in computational biology by facilitating complex tasks such as read mapping and assembly. These indexes map k-mers (substrings of length k) appearing in a large collection of reference strings to the set of identifiers of the strings where they appear. These sets, colloquially referred to as *color sets*, tend to occupy large quantities of memory, especially for large pangenomes. Our previous work thus focused on leveraging the repetitiveness of the color sets to improve the space effectiveness of the resulting index. As a matter of fact, repetition-aware indexes can be up to one order of magnitude smaller on large pangenomes compared to indexes that do not exploit such repetitiveness. Such improved space effectiveness, on the other hand, imposes an overhead at query time when performing tasks such as pseudoalignment that require the collection and processing of multiple related color sets.

Methods. In this paper, we show how to avoid this overhead. We devise novel query algorithms tailored for the specific repetition-aware representations adopted by the Fulgor index, a state-of-the-art c-dBG index, to significantly improve its pseudoalignment efficiency and without consuming additional space.

Results. Our results indicate that with increasing redundancy in the pangenomes, the compression factor provided by the Fulgor index increases, while the relative query time actually reduces. For example, while the space of the Fulgor index improves by 2.5× with repetition-aware compression and its query time improves by 1.6× on a collection of 5,000 Salmonella Enterica genomes, these factors become $(6.1\times, 2.8\times)$ and $(11.2\times, 3.2\times)$ for 50,000 and 150,000 genomes respectively. For an even larger collection of 300,000 genomes, we obtained an index that is 22.3× smaller and 2.2× faster.

2012 ACM Subject Classification Applied computing → Bioinformatics

Keywords and phrases Colored de Bruijn graphs, Pseudoalignment, Repetition-aware compression

Digital Object Identifier 10.4230/LIPIcs.WABI.2025.6

Supplementary Material *Software (Source Code)*: `https://github.com/jermp/fulgor` [30]
archived at `swh:1:dir:c9c71b24bc997397673f39f0cb7905660f09ff88`

Funding *Giulio Ermanno Pibiri*: Partially supported by DAIS – Ca' Foscari University of Venice within the IRIDE program.
Rob Patro: This work is supported by the NIH under grant award numbers R01HG009937 to R.P. Also, this project has been made possible in part by grants DAF2024-342821, DAF2022-252586 from the Chan Zuckerberg Initiative DAF, an advised fund of Silicon Valley Community Foundation. Rob Patro is a co-founder of Ocean Genomics inc.

1 Introduction

Consider a collection of strings, $\mathcal{R} = \{R_1, \ldots, R_N\}$, over the DNA alphabet $\{\mathsf{A}, \mathsf{C}, \mathsf{G}, \mathsf{T}\}$. A k-mer of R_c is a substring of length k of R_c and, for ease of notation, we consider each R_c as the set of its distinct k-mers. The *order-k colored* de Bruijn graph (c-dBG) of $\mathcal{R}$ is a directed graph where: (i) nodes are the distinct k-mers of the strings in $\mathcal{R}$; (ii) there is an edge (x, y) if the last $k-1$ characters of x are equal to the first $k-1$ characters of y; (iii) each k-mer x is annotated with its *color set* – the set of the (identifiers of the) strings where x appears. Formally, we define the color set of the k-mer x as $\textsc{ColorSet}(x) = \{c : x \in R_c\}$. In equivalent terms, we say that a k-mer x "has colors" $c \in \textsc{ColorSet}(x)$. We then regard an *index* for the c-dBG as the exact map $x \to \textsc{ColorSet}(x)$.

Intuitively, a c-dBG index arises as the composition of a k-mer dictionary that stores all the distinct k-mers of $\mathcal{R}$ and a collection of compressed color sets [20, 2, 13, 8]. Each color set is actually sorted to allow good compression and fast decoding. This indexing framework is particularly powerful for handling large-scale genomic data, where the identification of k-mers across diverse reference genomes is essential [19, 25, 34, 26, 21, 24].

However, the color sets in c-dBG indexes tend to occupy large quantities of memory [13, 31, 8, 2, 5], especially for large pangenomes. This poses a severe limitation on the size of the collections that can be indexed in internal memory. Our previous research efforts focused on addressing this challenge by leveraging the repetitiveness inherent in the color sets. Specifically, we developed the Fulgor index [13, 31, 8] – a compressed c-dBG index that identifies repetitive patterns across color sets and encodes them only once. By adopting this *repetition-aware* paradigm for compression, Fulgor can achieve reductions in memory utilization of up to one order of magnitude when applied to large pangenomes (e.g., including hundreds of thousands of genomes), compared to other indexing approaches that do not remove such redundancy. Fulgor currently offers the best overall trade-off for in-memory indexing of c-dBGs, being both highly space-efficient and the fastest to query [13, 31, 8].

In this paper, we make Fulgor even faster. In fact, while the repetition-aware compressed representations permit the indexing of larger datasets, they also impose some overhead during query execution, particularly when performing *pseudoalignment* [7] – a query modality tailored for c-dBGs that, essentially, performs intersections or unions of color sets. A user, therefore, has to trade off query efficiency for improved space. For example, our fastest variant of Fulgor can index 150,000 Salmonella Enterica genomes in 71 GB of RAM and execute 6.6×10^6 pseudoalignment queries in 37 minutes. This is the fastest query time reported in the literature for this benchmark (using 16 processing threads and under the *full-intersection* query modality; see Section 2.1). The smallest Fulgor variant, instead, indexes the same collection in just 6.3 GB (11× less space) but also takes 1 hour and 50 minutes to process the same query workload (3× slower). We show that this does not have to be the case.

Contributions. We study the problem of accelerating pseudoalignment queries directly over the compressed, repetition-aware representations developed for the Fulgor index. More precisely, we devise new query algorithms that exploit the repetitiveness of the color sets to actually perform *less* work, for both intersection and union of color sets. Our experimental results across large pangenomes show that such repetition-aware pseudoalignment queries can be generally performed $2-3\times$ faster. To make a concrete example, the smallest Fulgor index mentioned above now processes the same workload (6.6×10^6 pseudoalignment queries) in 34 minutes, rather than the previous 1 hour and 50 minutes, and without consuming more space than 6.3 GB of RAM.

2 Background and notation

In this section, we give necessary background information and fix the notation. More precisely, we first give a general introduction to pseudoalignment and define the two query algorithms we focus on. We then review the Fulgor index [13, 31, 8], with particular attention to its compressed representations for color sets because our goal is to support pseudoalignment directly over them, without decompression.

2.1 Pseudoalignment

Pseudoalignment is a specific algorithm for computing an approximate form of sequence mapping originally introduced by Bray et al. [7]. While *exact* alignment methods [22, 23, 16, 33, 3, 15, 14] report the positions (if any) where a query sequence Q appear in the references of $\mathcal{R}$ and the edits required to transform it into the matched substring of the reference, pseudoalignment only returns the identifiers of the references that *might* contain Q. Intuitively, Q is likely to be present in reference R_c if Q and R_c have *many* k-mers in common [35]. This consideration makes c-dBG indexes particularly well suited for pseudoalignment because, as they implement the map $x \to \textsc{ColorSet}(x)$ and color sets are represented explicitly in some compressed form, it is efficient to retrieve all the color sets of the k-mers of Q to combine them into a result set $\mathcal{X} \subseteq \{1, \ldots, N\}$. The two commonly used algorithms to compute $\mathcal{X}$ are *full-intersection* and *threshold-union*. These modalities are those supported by all recent c-dBG indexes [2, 13, 8, 5, 20].

In the following, since we are only interested in the k-mers of a query sequence Q that have a non-empty color set (sometimes, called "positive" k-mers), we refer to Q as the set of such k-mers with a little abuse of notation.

Full-intersection. The color c belongs to the result $\mathcal{X}$ if R_c contains *all* the k-mers of Q, that is $\mathcal{X} = \bigcap_{x \in Q} \textsc{ColorSet}(x)$.

Threshold-union. Given a parameter $\tau \in (0, 1]$, the color c belongs to the result $\mathcal{X}$ if R_c contains a least $T = \lfloor \tau \cdot |Q| \rfloor$ of the k-mers of Q. Let $U = U(Q)$ be the multi-set union of the color sets of all k-mers of Q, that is $U = U(Q) = \biguplus_{x \in Q} \textsc{ColorSet}(x)$. where $\biguplus$ is the multi-set union operator. We indicate with $\mu_c(U)$ the multiplicity of color c in U – the *score* of c. The result is then $\mathcal{X} = \{c \in U : \mu_c(U) \geq T\}$.

A customary value for τ is 0.8, as to require that at least 80% of the k-mers of Q are present in the references whose identifiers are in $\mathcal{X}$. This is the value we are going to use for our experimental analysis (Section 5).

Note that the result of threshold-union is equal to that of full-intersection for $\tau = 1$, that is, threshold-union is a relaxation of full-intersection. Intuitively, since threshold-union is less strict than full-intersection for $\tau < 1$, we trade off precision, allowing some sequencing errors and variations, for an increased mapping rate.

2.2 State of the art on indexing c-dBGs: the Fulgor index

Fulgor [13, 31, 8] is a c-dBG index based, at its core, on two different data structures: the SSHash [28, 29] k-mer dictionary, and a compressed inverted index [32] to represent the color sets. More precisely, Fulgor considers a *compacted* c-dBG where non-branching paths of nodes having the same color set are collapsed into strings called *unitigs*, exploiting the fact

that consecutive k-mers are very likely to have the same color set. In other words, we have $\textsc{ColorSet}(x) = \textsc{ColorSet}(u)$ for all k-mers x appearing in the unitig u. It is therefore profitable to store color sets for unitigs only, rather than for every distinct k-mer. Based on this observation, Fulgor implements the map $x \to \textsc{ColorSet}(x)$ as the composition of two maps: $x \xrightarrow{(1)} u(x) \xrightarrow{(2)} \textsc{ColorSet}(u(x))$, where $u(x)$ is the unitig comprising k-mer x. The first map is represented with the SSHash dictionary that explicitly stores the collection of unitigs of the compacted c-dBG. For the second map, instead, Fulgor exploits the property of SSHash of being *order-preserving*: the unitigs can be stored in *any order* without impacting the correctness or the efficiency of the dictionary. Fulgor thus sorts the unitigs in SSHash by their color sets, so that unitigs having the same color set are placed consecutively. This order makes it computationally efficient, both in time and space, to implement the second map, i.e., the map from $u(x)$ to its color set identifier i, assuming that $\textsc{ColorSet}(u(x)) = C_i$. Lastly, the compressed representation of the color set C_i is accessed from an inverted index, representing the collection of all the distinct color sets.

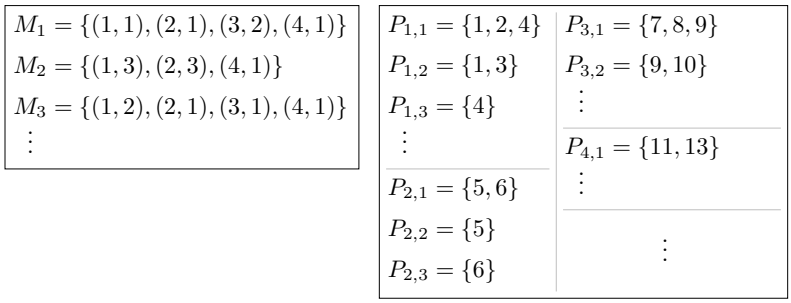
Queries. Answering a pseudoalignment query for a string Q comprises a three-step process over the Fulgor index.

1. *Lookup.* The set $\mathcal{U}(Q) = \{u(x) : x \in Q\}$ is computed using SSHash. Note that $|\mathcal{U}(Q)| \leq |Q|$, since consecutive k-mers are likely to be part of the same unitig and all unitigs include at least one k-mer.
2. *Deduplication.* Different unitigs can have the same color set. It is therefore useful to deduplicate the color sets of the unitigs. This is done by computing the set $\mathcal{C}(Q) = \{i : u \in \mathcal{U}(Q) \wedge \textsc{ColorSet}(u) = C_i\}$. Similarly to the previous step, we have that $|\mathcal{C}(Q)| \leq |\mathcal{U}(Q)|$, further reducing the number of color sets to process in the next step.
3. *Color set processing.* Apply either the full-intersection or the threshold-union algorithm on the color sets $\{C_i : i \in \mathcal{C}(Q)\}$ to compute the result $\mathcal{X}$.

In this paper, we focus entirely on the third step. This is the most resource-intensive step when performing pseudoalignment over large collections (see also Figure 7 at page 20) and for queries that have a non-trivial result set (e.g., non-empty and with many colors). Our problem reduces to that of computing the intersection and the (threshold-)union of color sets over some specific compressed representations that we review below. For the rest of the paper, we therefore assume to work with a set of $z > 1$ color sets which, without loss of generality, we indicate with $C_1, \ldots, C_z$.

Compressed representations. Fulgor supports three main representations for the color sets, each having different tradeoffs between space effectiveness and (until the current work) query efficiency [8]. These representations can be combined to achieve even further space reductions.

- *Density-aware representation (DA).* Each color set C_i is compressed using one of three different encodings based on its density, i.e., the ratio $|C_i|/N$. The set C_i is considered *sparse* when $|C_i|/N < 1/4$. In this case, C_i is coded as a stream of gaps between consecutive integers, where each gap is compressed using Elias' δ code [12]. The set C_i is considered *dense*, instead, when $1/4 \leq |C_i|/N < 3/4$. In this case, it is coded as a bitvector of N bits, say $B[1..N]$, where $B[c] = 1$ if $c \in C_i$, and $B[c] = 0$ otherwise. Lastly, when $|C_i|/N \geq 3/4$ the set C_i is *very dense* and it is more economical to represent the complement set $\overline{C}_i = \{1, \ldots, N\} \setminus C_i$, which can be encoded as a sparse set.
  The density thresholds (1/4 and 3/4) are optimal for the three encodings used, allowing to represent every color set using at most N bits.



$C_1 = \{1,2,4|5,6|9,10|11,13\}$; $C_2 = \{4|6|11,13\}$; $C_3 = \{1,3|5,6|7,8,9|11,13\}$

Figure 1 An example of the MP representation. Meta color sets are drawn on the left; on the right, some partial color sets (vertical dots indicate that there are other partial color sets in the groups). At the bottom, the color sets C_1, C_2, and C_3 are obtained by concatenating the partial color sets as indicated by the meta color sets. Vertical bars highlight the start of a new partial color.

- *Repetition-aware representation: Differential and Representative color sets (DR).* This representation leverages the inherent similarity of the color sets by partitioning them into groups $\{\mathcal{N}_1, \ldots, \mathcal{N}_r\}$, where each group $\mathcal{N}_j$ is a collection of "similar" color set, e.g., sharing many colors. Thus, for each $\mathcal{N}_j$, a *representative* color set A_j is computed by finding the most repetitive colors between the sets in the group. Then, each color set $C_i \in \mathcal{N}_j$ is represented as a *differential* color set $D_i = C_i \Delta A_j$, where Δ is the symmetric set difference between two sets, defined as $X \Delta Y \coloneqq (X \cup Y) \backslash (X \cap Y)$. To decode a color set, it is sufficient to compute $C_i = D_i \Delta A_j$.
  The rationale behind this representation is that, since the color sets in each group $\mathcal{N}_j$ are similar, we expect to have $|D_i| \ll |C_i|$.
- *Repetition-aware representation: Meta and Partial color sets (MP).* This variant exploits the fact that genomes from the same species are very similar, thereby inducing very similar color sets. Hence, the set of N strings $\mathcal{R}$ is partitioned into groups $\{\mathcal{N}_1, \ldots, \mathcal{N}_r\}$ of similar strings. Since each group comprehends $|\mathcal{N}_j|$ strings, a set of *partial* color sets $\{P_{j,1}, P_{j,2}, \ldots\}$ is derived. It follows that the original color set C_i can be expressed as a sequence M_i of pointers to partial color sets, each of them belonging to a different group. Each of these pointers, called *meta* colors, is a pair (j, p), indicating that the p-th partial color set $P_{j,p}$ from group $\mathcal{N}_j$ is used. Importantly, the pointers in each meta color set M_i are sorted on the first component, i.e., the set $\{j : (j,p) \in M_i\}$ is sorted.
  This results in a two-layer representation where the first layer is composed by the meta color sets M_i and the second layer by the partial color sets that they point to. Refer to Figure 1 for a schematic illustration.

2.3 Set notation and iteration

Recall that the sets we deal with in this paper (the color sets of a c-dBG) are *ordered* sets of integers. Throughout the paper, we assume that we can perform iteration and search queries over an ordered set in a *stateful* manner, hence providing an "iterator-like" abstract interface for a set. More precisely, this interface exposes the following primitives for a set X:

- X.Value() returns the value currently pointed to by the iterator.
- X.Next() updates the value pointed to by the iterator to the one immediately after and returns such value.

- X.NextGEQ(ℓ) updates the value pointed to by the iterator to the smallest value of X that is greater-than or equal-to ℓ and returns the corresponding value. Note that if one performs a sequence of queries for $\ell_0 < \ell_1 < \cdots < \ell_m$, the logical pointer of the iterator never moves backwards. (This is the query pattern of interest that we are going to issue in our pseudocodes.)

If there is no element after the one currently pointed to by the iterator or if ℓ is larger than the largest element in X, both Next and NextGEQ return the sentinel value ∞.

When discussing asymptotic complexities, we assume that Value and Next take $\mathcal{O}(1)$, and that NextGEQ takes $\mathcal{O}(s)$ to skip to s positions ahead, even when the set X is compressed, as it is in our implementations in practice. (Note that, while X.NextGEQ(ℓ) can be executed in time $\mathcal{O}(\log |X|)$ by binary searching ℓ, in practice it is more cache-friendly, and therefore faster, to just scan forward, especially for short skip lengths s.)

Now, let M be a meta color set, that is, a set of pairs (j, p) as described in Section 2.2. We define the primitive M.First() that returns the set of the first components from each pair, that is the set $\{j : (j, p) \in M\}$. As already observed, this is also a sorted set. Lastly, we define M.PartialSet(j) that moves the iterator to the smallest pair (t, p) such that $t \geq j$ and returns $P_{t,p}$. Formally: $t = M$.First().NextGEQ(j) if $t \neq \infty$, otherwise (when $t = \infty$, that is, j is larger than the largest group identifier in M), it returns $\varnothing$.

▶ **Example 1.** Let $M = \{(1, 9), (5, 2), (6, 33), (11, 3)\}$ be a meta color set. Then, $F = M$.First() $= \{1, 5, 6, 11\}$, regardless of the element currently pointed to by the iterator. Assuming the iterator is pointing to the first pair $(1, 9)$, then M.PartialSet(3) returns the partial color set $P_{5,2}$ because F.NextGEQ(3) $= 5$. Similarly, M.PartialSet(8) returns $P_{11,3}$. We instead have M.PartialSet(j) $= \varnothing$ for any $j > 11$.

3 Full-intersection

As described in Section 2.1, performing a full-intersection pseudoalignment consists of computing the intersection of z color sets, $C_1, \ldots, C_z$. Fulgor supports the intersection of color sets using the few primitives introduced in Section 2.3 and as shown in Algorithm 1 (this algorithm is also called "adaptive" in the literature [10, 11, 4]). In addition, we assume the color sets are sorted by size in non-decreasing order. While this order does not affect the complexity of the algorithm, which is $\mathcal{O}\left(\sum_{i=1}^{z} |C_i|\right)$, it is beneficial in practice: the elements of the smallest set help in filtering out elements that are not part of the intersection.

In the following, we show how to improve over this approach by exploiting the specific compressed representations reviewed in Section 2.2.

DA representation. This representation compresses a color set using one of three different strategies, each tailored to the density of the set. Among the three density classes – sparse, dense, and very dense – the last one presents the most significant opportunities for efficiency gains. Remember that very dense color sets are compressed by storing their complement set, denoted with $\overline{C}$, which is sparse by definition. So, rather than iterating through a large set, we can directly leverage the complement set for faster processing: given z very dense sets, represented as $\overline{C}_1, \ldots, \overline{C}_z$, we can use de Morgan's laws to compute the intersection of their original sets, that is $\bigcap_{i=1}^{z} C_i = \overline{\left(\bigcup_{i=1}^{z} \overline{C}_i\right)}$. In other words, the intersection of z sets is equivalent to the complement of the union of the complement sets. The refined intersection algorithm is therefore:

1. Compute the union of the complements of all very dense color sets.
2. Compute the intersection of all other color sets.
3. Return the intersection between the complement of the set computed at step 1. and the set computed at step 2.

Algorithm 1 Intersection algorithm for color sets $\{C_1, \ldots, C_z\}$.

```
1:  function INTERSECT({C_1, ..., C_z})
2:      X = ∅
3:      c = C_1.VALUE(); i = 2
4:      while c < ∞ do
5:          for i ≤ z; i = i + 1 do
6:              v = C_i.NEXTGEQ(c)
7:              if v ≠ c then
8:                  c = v, i = 1
9:                  break
10:         if i = z + 1 then
11:             X = X ∪ {c}
12:             c = C_1.NEXT(); i = 2
13:     X
```

DR representation. We recall that in this variant the color sets are stored as the symmetric differences $D_i = C_i \Delta A_j$ between the original color set C_i and the representative set A_j of the group $\mathcal{N}_j$, such that $C_i \in \mathcal{N}_j$ and $|D_i| \ll |A_j|$. Note that simply using Algorithm 1 over this representation is inefficient. In fact, to decode C_i, it is necessary to read its whole differential set D_i and representative set A_j, for which we have that $|D_i| + |A_j| \geq |C_i|$. Furthermore, Algorithm 1 does not exploit the fact that multiple color sets can be part of the same group $\mathcal{N}_j$, and thus have the same representative set A_j. Consequently, each A_j could be processed redundantly, leading to a significant performance bottleneck.

Consider the case where all color sets belong to the same group $\mathcal{N}_j$. For ease of notation, throughout the rest of this section, we will simply write A and $\mathcal{N}$, instead of A_j and $\mathcal{N}_j$. Then, the time complexity of the intersection is $\mathcal{O}\left(z|A| + \sum_{i=1}^{z}(|D_i| + |C_i|)\right)$. This complexity is not ideal because we would like to scan each D_i only rather than D_i *plus* C_i, and A only once rather than z times. To do so, we give the following lemma (the proof can be found in Section A).

▶ **Lemma 2** (DR intersection). *Given z sets, $C_1, \ldots, C_z$, represented differentially as $D_i = C_i \Delta A$ with respect to the representative set A, for $i = 1, \ldots, z$, their intersection can be computed as*

$$\bigcap_{i=1}^{z} C_i = \left(A \cap \overline{D}_1 \cap \cdots \cap \overline{D}_z\right) \cup \left(\overline{A} \cap D_1 \cap \cdots \cap D_z\right).$$

In other words, Lemma 2 says that a color is part of the intersection if it appears only in the representative set or, exclusively, in all differential sets. Note that this alternative formulation does not involve the original C_i but only their symmetric differences D_i and the representative set A.

The intersection of z differential color sets, belonging to the same group, can be implemented as in Algorithm 2. The algorithm has a time complexity of $\Theta\left(N + |A| + \sum_{i=1}^{z} |D_i|\right)$, since we have to read only once the representative and differential sets, plus the array $counts[1..N]$. For $c = 1, \ldots, N$, we define $counts[c] := |\{1 \leq i \leq z : c \in D_i\}|$, i.e., the number of differential sets that include c. We use this array to compute the intersection between the A and every $\overline{D}_i$, and the intersection between $\overline{A}$ and every D_i, as follows. For any $c = 1, \ldots, N$, if $c \in A$ and $counts[c] = 0$ (c appears in all $\overline{D}_i$), or $c \notin A$ and $counts[c] = z$ (c appears in all D_i), then c is part of the result. Note that while one can simply rely on Algorithm 1 to intersect

Algorithm 2 Intersection algorithm for differentially-coded color sets $\{D_1, \ldots, D_z\}$. These sets all belong to the same group, hence are relative to the same representative set A.

```
1:  function D-Intersect({D_1, ..., D_z}, A)
2:      X = ∅
3:      counts[1..N] = [0..0]
4:      for all i ∈ [1..z] do
5:          c = D_i.Value()
6:          while c < ∞ do
7:              counts[c] = counts[c] + 1
8:              c = D_i.Next()
9:      for all c ∈ [1..N] do
10:         v = A.NextGEQ(c)
11:         if (counts[c] = 0 ∧ c = v) ∨ (counts[c] = z ∧ c ≠ v) then
12:             X = X ∪ {c}
13:     X
```

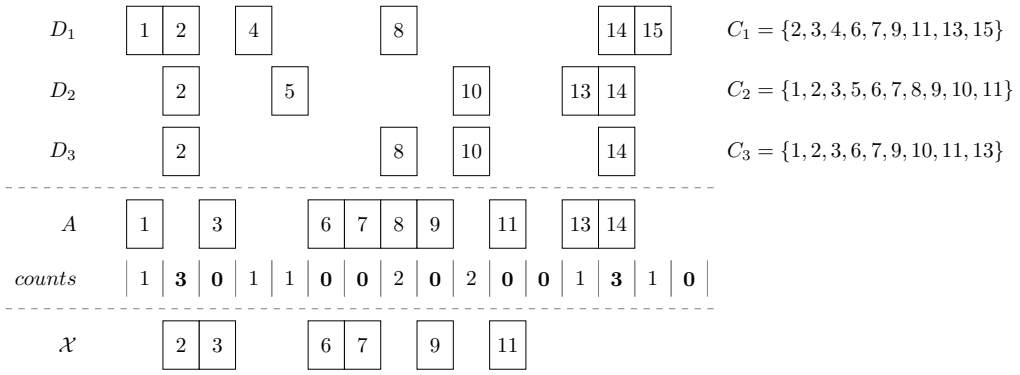


Figure 2 Full-intersection for $z = 3$ color sets stored using DR representation, with $N = 16$ colors. All color sets belong to the same group. The values forming the differential color sets (D_1, D_2, D_3) and the representative set (A) are vertically aligned to better visualize the intersection. To the right of each differential set D_i, the original color set C_i is reported so that it is easy to see that $\mathcal{X} = C_1 \cap C_2 \cap C_3$. The values 0s and 3s in the array *counts* are highlighted in bold font as they are the candidates to be inserted in the intersection.

the color sets without the additional $\Theta(N)$ cost, this algorithm should be executed twice: once for each operand in the union of Lemma 2. Given that $|A| + |\overline{A}| = |D_i| + |\overline{D}_i| = N$, this results in a time complexity of $\mathcal{O}(zN)$, which is worse than that of Algorithm 2, especially considering that $|D_i| \ll |A| \leq N$.

It is also important to consider that accessing the sets sequentially to build the *counts* array leads to better cache locality and further improves performance in practice.

▶ **Example 3.** Figure 2 shows how the full-intersection is performed on $z = 3$ color sets stored using the DR representation. Assume that the number of colors is $N = 16$, and all color sets belong to the same group. (The group may obviously contain other color sets, more than the z we consider.)

We start by counting the number of occurrences of each differential color $c \in D_i$, for $i = 1, 2, 3$, and storing that value inside the array *counts*. Then, the intersection step follows: if $counts[c] = 0$ and $c \in A$, or $counts[c] = 3$ and $c \notin A$, then $c \in \mathcal{X}$. Consequently, all colors whose count is not 0 or 3 – $\{1, 4, 5, 8, 10, 13, 15\}$ in Figure 2 – are surely not part of the intersection. The colors 3, 6, 7, 9, and 11 are all part of the representative set, but do not

Algorithm 3 Intersection algorithm for meta color sets $\{M_1, \ldots, M_z\}$ that, as explained in Section 2.2, are pointers to partial color sets.

```
1:  function M-Intersect({M_1, ..., M_z})
2:      X = ∅
3:      G = {M_i.First( ) : i ∈ [1..z]}                ▷ Set of group identifiers
4:      F = Intersect(G)
5:      for all j ∈ F do
6:          P_j = ∅                                    ▷ Collection of partial color sets from group j
7:          for all i ∈ [1..z] do
8:              P_j = P_j ∪ {M_i.PartialSet(j)}
9:          Deduplicate(P_j)
10:         X_j = Intersect(P_j)
11:         X = X ∪ X_j
12:     X
```

appear in any differential set ($counts[c] = 0$), therefore, they are part of the intersection. On the contrary, 2 is in all differential sets ($counts[2] = 3$) but is missing from the representative, making the second case for the inclusion true, and thus it appears in the intersection. Lastly, while also color 14 is present in all differential sets ($counts[14] = 3$), it is present in the representative set as well, so it is *not* part of the intersection.

In the general case, when not all color sets belong to the same group, we first apply Algorithm 2 to every group and then apply Algorithm 1 to the intermediate results.

MP representation. This representation organizes color sets in two layers: the first layer is composed of meta colors, which are pointers to groups of partial color sets that form the second layer. Meta colors pairs of integers (j, p), with j being the group identifier, and p the identifier of the partial color set inside the group.

Since groups form a partition, we can perform the intersection independently on each group and then join the results. Consequently, if group j does not appear in a meta color set, it can be discarded directly, as this will lead to an empty group intersection. This means we can devise a two-layer intersection algorithm, where the first pass filters out groups that do not appear in all meta color sets, and the second one performs the intersection on the partial colors within each group. Moreover, instead of computing the intersection of *all* partial color sets, we can compute the intersection between the *distinct* partial color sets.

These ideas lead to Algorithm 3. If $\mathcal{F}$ represents the first-layer intersection (i.e., among the group identifiers, only), its time complexity is $\mathcal{O}\left(\sum_{i=1}^{z} |M_i| + \sum_{(j,p)\in M_i \,:\, j\in\mathcal{F}} |P_{j,p}|\right)$. Although this complexity is higher than that of Intersect (Algorithm 1) in the worst case – due to the additional need to intersect meta color sets alongside the partial color sets that constitute the original color sets – it is always beneficial in practice on repetitive collections as we are going to see in our experimental analysis in Section 5.

▶ **Example 4.** Figure 3 shows all possible scenarios occurring when intersecting color sets stored using the MP representation. After identifying the common groups, we obtain the filtered set $\mathcal{F} = \{1, 2, 4\}$, since some meta color sets are missing meta colors from groups 3, 5, and 6. Then we process partial color sets. For group 1, all meta colors are different, so a full-intersection must be computed between all partial sets $P_{j,p}$ with $j = 1$. In contrast, group 2 includes repeated meta colors: both $(2, 1)$ and $(2, 3)$ appear twice. After the deduplication step, the intersection has to be performed only on the two distinct partial sets, cutting the

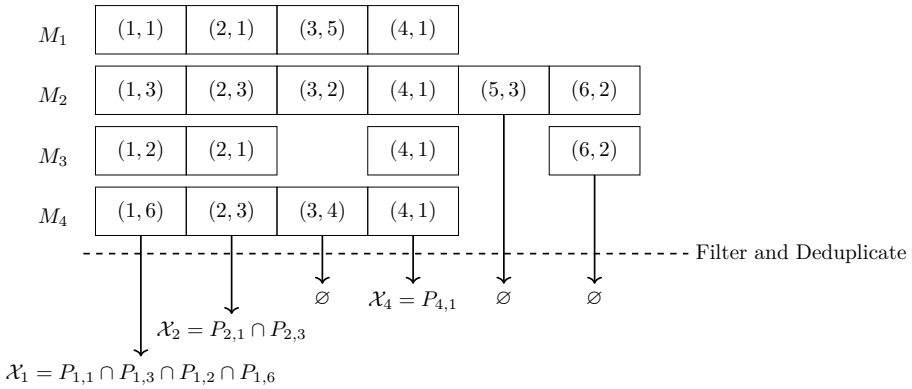


Figure 3 Full-intersection for four color sets stored using the MP representation, assuming 6 groups. Meta colors are vertically aligned based on their group identifier, i.e., the first value of each pair. The dashed line represents the filtering (intersection of the groups) and meta color deduplication steps. For each group, a vertical arrow points to the operation performed to compute the final result. Groups ignored after the filtering step point to an empty set ∅.

computation by half compared to the previous approach, which would otherwise process four sets rather than two. Finally, group 4 presents an ideal case: all meta color sets include $(4, 1)$, thus no intersection is needed and $P_{4,1}$ can be directly decoded and included in the result.

4 Threshold-union

As explained in Section 2.1, the threshold-union is a relaxation of the full-intersection strategy that includes in the result all colors c such that their score $\mu_c \geq T$. To be able to determine such score for each color, the first two steps of the pseudoalignment pipeline – Lookup and Deduplication – must also account for the frequency of k-mers associated with the same color set. We define the score γ_i of the color set C_i as $\gamma_i := |\{x \in Q : \textsc{ColorSet}(x) = C_i\}|$, i.e., the number of k-mers of Q having color set C_i. It is straightforward to see that μ_c can be expressed as $\mu_c = \sum_{i\,:\,c \in C_i}^{z} \gamma_i$. Algorithm 4 uses these concepts to implement the threshold-union strategy. The algorithm has a time complexity of $\Theta\left(z \left|\bigcup_{i=1}^{z} C_i\right|\right)$ since it has to compute the score of each color present in at least one color set.

As done for full-intersection, we now present faster algorithms for threshold-union tailored for our compressed representations.

DA representation. The main drawback of Algorithm 4 is that, besides not being cache-friendly, it has to perform a membership query to every color set for each color encountered. In this way, all color sets are queried the same number of times, with potentially many negative results in the case of short color sets.

A better approach, which we call *counting union*, is to scan the color sets to build the score of each color incrementally. To do so, the scores are maintained in an array $scores[1..N]$, so that $scores[c] = \mu_c$ at the end of the scan. Then, it is sufficient to scan this array and return all colors whose score is $\geq T$. This algorithm has a time complexity of $\Theta\left(N + \sum_{i=1}^{z} |C_i|\right)$. It is worth noting that this complexity, despite being better then $\Theta\left(z \left|\bigcup_{i=1}^{z} C_i\right|\right)$ in the worst case (all sets are disjoint), may still be worse when z is small and the sets C_i share many integers (i.e., their union is small). Nevertheless, our experiments in Section 5 show that this new algorithm is more efficient overall than Algorithm 4, especially thanks to the improved cache locality, as color sets are scanned linearly Yet, we can improve even more as follows.

Algorithm 4 Union algorithm for color sets $\{C_1, \ldots, C_z\}$ having scores $\{\gamma_1, \ldots, \gamma_z\}$ respectively, and a minimum score threshold T.

```
1:  function UNION({C_1, ..., C_z}, {γ_1, ..., γ_z}, T)
2:      X = ∅
3:      c = min{C_1.VALUE(), ..., C_z.VALUE()}
4:      while c < ∞ do
5:          min = ∞
6:          μ_c = 0
7:          for all i ∈ [1..z] do
8:              v = C_i.VALUE()
9:              if v = c then
10:                 μ_c = μ_c + γ_i
11:                 v = C_i.NEXT()
12:             if v < min then min = v
13:         if μ_c ≥ T then X = X ∪ {c}
14:         c = min
15:     X
```

As explained in Section 2.2, color sets with a density higher than 3/4 are stored using their complement set. Therefore, reading the complement set is at least 3× faster than reading the set directly. We can exploit this fact by noting that we are not interested in the actual score of a color, but in the *difference between the score and the threshold*. In fact, either adding γ_i to the score of every color $c \in C_i$, or subtracting γ_i from the threshold T and from μ_c for all $c \in \overline{C}_i$, yields the same result. We call this technique *dynamic thresholding*, since we adjust the threshold in the presence of very dense sets, making T a dynamic value instead of fixed. The following lemma formalizes this technique, and Algorithm 5 shows how to use it for threshold-union, together with the counting-union technique described above. (In the statement of the lemma and its proof in Section A, "v.d." stands for "very dense".)

▶ **Lemma 5** (Dynamic Thresholding). *For any $c \in [1..N]$, $\mu_c \geq T$ if and only if*

$$\sum_{\substack{i:\ c \in C_i \wedge \\ C_i \text{ not v.d.}}}^{z} \gamma_i - \sum_{\substack{i:\ c \in \overline{C}_i \wedge \\ C_i \text{ v.d.}}}^{z} \gamma_i \geq T - \sum_{i:\ C_i \text{ v.d.}}^{z} \gamma_i.$$

This lemma implies that we can directly process the *complement* sets of the very dense color sets, which are sparse by definition, thus avoiding iterating over very large sets.

▶ **Example 6.** Let us consider an example of dynamic thresholding. Let $N = 10$ be the number of references, $T = 8$ the threshold, and $z = 4$, with $C_1 = \{1, 7, 10\}$, $C_2 = \{2, 3, 7, 9\}$, $C_3 = \{1, 2, 3, 4, 5, 6, 9, 10\}$, $C_4 = \{1, 3, 4, 5, 6, 7, 8, 10\}$, and $\gamma_1 = 3$, $\gamma_2 = 2$, $\gamma_3 = 2$, $\gamma_4 = 4$. Note that, since C_3 and C_4 are very dense, they are stored as $\overline{C}_3 = \{7, 8\}$ and $\overline{C}_4 = \{2, 9\}$.

The following table reports, for each row, the score of each color after processing the color set in the first column. The second column shows the score γ_i of the color set, while the third column shows the current threshold, in this case, constant. Bold numbers represent the elements the algorithm updated, while the underlined ones in the last row are the ones for which $\mu_c \geq T$.

	γ_i	T	μ_1	μ_2	μ_3	μ_4	μ_5	μ_6	μ_7	μ_8	μ_9	μ_{10}
C_1	3	8	**3**	0	0	0	0	0	**3**	0	0	**3**
C_2	2	8	3	**2**	**2**	0	0	0	**5**	0	**2**	3
C_3	2	8	**5**	**4**	**4**	**2**	**2**	**2**	5	0	**4**	**5**
C_4	4	8	**9**	4	**8**	**6**	**6**	**6**	**9**	**4**	4	**9**

Algorithm 5 Union algorithm with counting-union and dynamic-thresholding (see Lemma 5) for color sets $\{C_1, \ldots, C_z\}$ having scores $\{\gamma_1, \ldots, \gamma_z\}$ respectively, and a minimum score threshold T.

```
1:  function CountingUnionDynamicThresholding({C_1,...,C_z}, {γ_1,...,γ_z}, T)
2:      X = ∅
3:      scores[1..N] = [0..0]
4:      for all i ∈ [1..z] do
5:          if IsVeryDense(C_i) then
6:              T = T − γ_i
7:              c = C̄_i.Value()
8:              while c < ∞ do
9:                  scores[c] = scores[c] − γ_i
10:                 c = C̄_i.Next()
11:         else
12:             c = C_i.Value()
13:             while c < ∞ do
14:                 scores[c] = scores[c] + γ_i
15:                 c = C_i.Next()
16:     for all c ∈ [1..N] do
17:         if scores[c] ≥ T then X = X ∪ {c}
18:     X
```

In total, the static threshold algorithm iterates over $3 + 4 + 8 + 8 = 23$ colors (total number of bold values in the table). Now, we compare the table above with the following one, showing the scores at each step of the dynamic threshold algorithm.

	γ_i	T	μ_1	μ_2	μ_3	μ_4	μ_5	μ_6	μ_7	μ_8	μ_9	μ_{10}
C_1	3	8	**3**	0	0	0	0	0	**3**	0	0	**3**
C_2	2	8	3	**2**	**2**	0	0	0	**5**	0	**2**	3
$\overline{C}_3$	2	**6**	3	2	2	0	0	0	**3**	**-2**	2	3
$\overline{C}_4$	4	**2**	3	**-2**	2	0	0	0	3	-2	**-2**	3

It is easy to see that the number of updated scores – and therefore colors considered – is much smaller, as there are only $3 + 4 + 2 + 2 = 11$ of them, less than half compared to the previous method.

DR representation. As similarly done for intersection, consider the case where the z differential color sets belong to the same group and hence are relative to the same representative set A. Let the score of the group be $\gamma = \sum_{i=1}^{z} \gamma_i$. To perform the union, in this case, we leverage a variant of Lemma 2, as formalized in the following lemma (and proven in Section A).

▶ **Lemma 7** (DR union). *Given z sets, $C_1, \ldots, C_z$, represented differentially as $D_i = C_i \Delta A$ with respect to the representative set A, for $i = 1, \ldots, z$, their threshold-union can be computed by including all $c \in [1..N]$ such that $\mu_c \geq T$ where*

$$\mu_c = \begin{cases} \mu'_c & \text{if } c \notin A \\ \gamma - \mu'_c & \text{if } c \in A \end{cases}$$

with $\mu'_c = \sum_{i:c \in D_i}^{z} \gamma_i$ being the sum of the scores of the color sets C_i for which $c \in D_i$.

In this way, it is sufficient to read the differential and representative sets once, plus the array of scores to determine the colors c for which $\mu_c \geq T$, obtaining a total time complexity of $\Theta\left(N + \sum_{i=1}^{z} |D_i| + \sum_{j=1}^{r} |A_j|\right)$, with r being the number of groups in the index.

MP representation. We can adapt the two-layer intersection algorithm described in Section 3 to also work for the threshold-union strategy, by considering the scores of the colors. In the first layer, we compute the score of each group and filter out all groups whose score does not reach the threshold, as it is impossible for their partial sets to gain a score $\geq T$. Then, in the second layer, for each group that "survives" the initial filtering, we deduplicate its meta colors before performing the union of the distinct partial color sets.

The efficiency of this process, unlike the ones described previously, critically depends on the value of T (remember that this value is controlled via a parameter $0 < \tau \leq 1$): if T is small, fewer groups are filtered away, increasing the computational load on the second layer. If $\mathcal{F}_T$ indicates the set of groups whose score is at least T, then the complexity of the algorithm is $\mathcal{O}\left(N + \sum_{i=1}^{z} |M_i| + \sum_{(j,p)\in M_i \,:\, j\in\mathcal{F}_T} |P_{j,p}|\right)$.

5 Experimental results

In this section, we measure the practical impact of the proposed algorithms when performing pseudoalignment over large pangenomes. An important remark is in order. Prior published work [13, 31, 8] demonstrated the higher efficiency of all Fulgor variants when compared to other indices, such as Themisto [2], MetaGraph [20], and COBS [5]. Consequently, we avoid comparing against those indices again here, but we encourage interested readers to consult [8] for more details[1].

All experiments were conducted on a machine equipped with an Intel Xeon Platinum 8276L CPU (clocked at 2.20GHz), 500GB of RAM, and running Linux kernel version 4.15.0. Our software is written in C++ and available under a MIT license at `https://github.com/jermp/fulgor`. The software was compiled with `gcc` 11.1.0, for the experiments we here discuss.

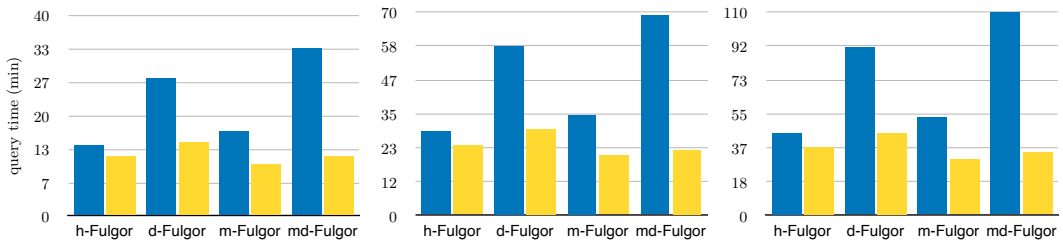
Methodology. For all experiments, we use a k-mer length of $k = 31$, which is common throughout prior work. For the threshold-union queries, we set the threshold to the customary value $\tau = 0.8$. Reported times are relative to a second run of each experiment, where the indexes are loaded from the disk cache, benefiting mostly the larger indexes than the smaller ones. To avoid recording I/O overhead, the pseudoalignment output of all the experiments was written to `/dev/null`.

To better measure the impact of our algorithms, we tested them under a *high-hit query workload*, i.e., with a mapping rate (percentage of mapped reads over the total number of queried reads) of about 90% or more, as it is more informative of the efficiency of the color set processing step. A low-hit workload, on the other hand, stresses the speed of negative k-mer lookups on the dictionary, and is therefore not relevant for the scope of this work. Figure 7 in Section B shows how query time is distributed among the different steps of pseudoalignment: as evident, processing the color sets is the most resource-intensive step for

[1] To provide a reference point, we give an example for a collection of 100,000 Salmonella Enterica genomes. Fulgor performs 6.6×10^6 full-intersection pseudoalignment queries in 29 minutes (43 GB of RAM; see also Table 1), whereas Themisto takes 1 hour and 22 minutes (76 GB), MetaGraph takes almost 10 hours (4.8 GB), and COBS takes 9 hours (> 500 GB). Our smallest variant of Fulgor, md-Fulgor (4.6 GB; see next), is as small as MetaGraph in this benchmark.

Table 1 Total query time (formatted as h:mm:ss) and memory used in GB (max. RSS) for **full-intersection** as reported by `/usr/bin/time`, using 16 processing threads.

Dataset	Mapping rate	h-Fulgor			d-Fulgor			m-Fulgor			md-Fulgor		
		before	after	GB	before	after	GB	before	after	GB	before	after	GB
EC	98.99	2:12	**2:10**	1.67	4:52	**2:29**	0.78	3:08	**1:32**	0.73	6:07	**1:41**	0.57
SE-5K	89.49	1:14	**1:10**	0.80	1:54	**1:44**	0.41	1:25	**1:09**	0.37	2:10	**1:21**	0.32
SE-10K	89.71	2:29	**2:20**	2.06	4:14	**2:54**	0.90	2:56	**2:07**	0.77	4:55	**2:30**	0.65
SE-50K	91.25	14:05	**12:00**	18.24	27:25	**14:50**	5.82	17:00	**10:10**	3.64	33:25	**11:50**	2.95
SE-100K	91.41	29:00	**24:00**	42.40	58:10	**29:50**	11.58	34:40	**20:30**	6.08	1:09:00	**22:50**	4.63
SE-150K	91.52	44:30	**37:00**	70.55	1:31:00	**44:20**	18.55	53:00	**30:20**	8.29	1:50:00	**33:50**	6.29
GB	92.91	**1:10**	**1:10**	36.01	**1:00**	1:26	28.25	1:09	**1:02**	29.79	1:03	**1:02**	26.88



(a) SE-50K **(b)** SE-100K **(c)** SE-150K

Figure 4 Graphical comparison between previous (blue bars) and current full-intersection (yellow bars) query times, as reported in Table 1.

these high-hit workload queries (with the exception of GB, where processing the color sets is not the most time-consuming step because the sets are very short on average; see below). Although not reported herein, similar results were observed for the threshold-union.

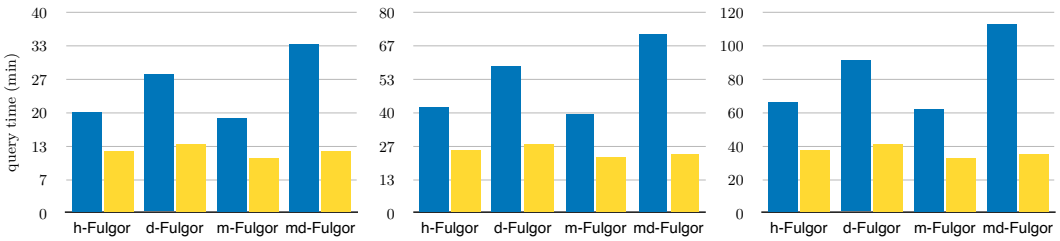
For the rest of the paper, we use the following naming convention to distinguish between the different representations used: h-Fulgor employs the DA representation, while d-Fulgor and m-Fulgor employ repetition-aware compression schemes, the DR and MP representations, respectively. The md-Fulgor variant instead – our most succinct index to date – combines the two repetition-aware representations: it first partitions the color sets using the MP approach, then applies the DR scheme on the partial color sets of each group.

Datasets. The evaluations were performed using the following datasets, as also used in prior work [13, 8]: 3,682 Escherichia Coli (EC) genomes from NCBI [1]; different collections of Salmonella Enterica (SE), ranging from 5,000 to 150,000 genomes from the "661K" collection [6]; 30,691 genomes assembled from human gut-bacteria samples (GB), from [17]. More details on these pangenomes can be found in [8, Table 1]. We remark that the GB dataset is much more diverse than the others: it features very short color sets (44 integers on average) but contains an order of magnitude more unitigs than the SE and EC datasets.

The queried reads consist of all FASTQ records in the first read file of the following nucleotide runs: `SRR1928200` for EC, `SRR801268` for SE, and `ERR321482` for GB. These files contain around $6.5 - 7$ million reads each. The lengths of each read in the FASTQ file are: 100 bases (70 k-mers) for EC, 53 bases (23 k-mers) for SE, and 90 bases (60 k-mers) for GB.

Table 2 Total query time (formatted as h:mm:ss) and memory used in GB (max. RSS) for **threshold-union** with $\tau = 0.8$ as reported by `/usr/bin/time`, using 16 processing threads.

Dataset	Mapping rate	h-Fulgor			d-Fulgor			m-Fulgor			md-Fulgor		
		before	after	GB	before	after	GB	before	after	GB	before	after	GB
EC	99.71	3:34	**1:55**	1.67	5:07	**2:06**	0.78	3:32	**1:37**	0.73	6:15	**1:39**	0.57
SE-5K	89.60	1:30	**1:08**	0.79	1:53	**1:21**	0.41	1:31	**1:14**	0.37	2:14	**1:19**	0.32
SE-10K	89.87	3:12	**2:15**	2.06	4:15	**2:40**	0.91	3:09	**2:14**	0.77	5:03	**2:30**	0.65
SE-50K	91.56	19:57	**12:08**	18.25	27:23	**13:42**	5.84	18:58	**10:48**	3.64	33:32	**12:12**	2.96
SE-100K	91.79	42:07	**25:07**	42.19	58:16	**27:26**	11.61	39:32	**22:09**	6.11	1:11:10	**23:33**	4.66
SE-150K	91.93	1:06:11	**37:32**	70.20	1:31:16	**41:05**	18.54	1:02:02	**32:16**	8.33	1:52:19	**35:15**	6.33
GB	94.66	2:31	**1:12**	36.03	1:50	**1:28**	28.27	2:04	**1:09**	29.81	2:07	**1:12**	26.90



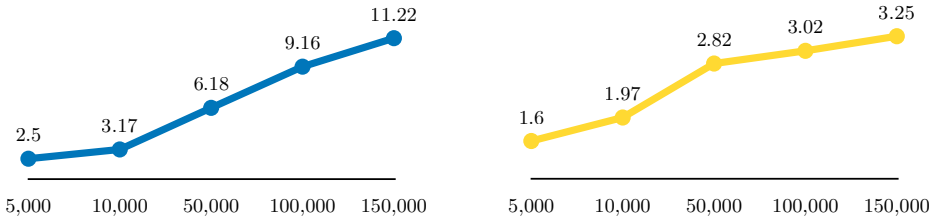
(a) SE-50K **(b)** SE-100K **(c)** SE-150K

Figure 5 Graphical comparison between previous (blue bars) and current threshold-union (yellow bars) query times, as reported in Table 2.

Experimental results. Table 1 reports the results of the full-intersection algorithm, while Figure 4 provides a graphical comparison between the most significant improvements observed. Similarly, Table 2 and Figure 5 report the results of the threshold-union algorithm. In both tables, for each Fulgor variant, we show the query time *before* applying any optimization and *after* using the optimizations described in the previous sections, together with the RAM used while performing the queries. Importantly, the RAM usage does not change when using the new algorithms, so we report it once.

The clear result of these experiments is that all indexes are now substantially faster, by $2 - 3\times$ *on average.* More precisely:

- h-Fulgor, while being the fastest (but also largest) variant without optimizations, is now even faster thanks to the density-aware optimization, with speedup factors of $1.2\times$ for full-intersection, and $1.8\times$ for threshold-union.
- d-Fulgor is twice as fast for both pseudoalignment algorithms after the optimizations. As a result, its performance is now comparable to that of h-Fulgor – while it remains generally 20% slower, it also consumes $3\times$ less space.
- The tailored m-Fulgor intersection algorithm leads to a consistent 20% improvement over h-Fulgor. This result is remarkable considering it comes with a large reduction in memory usage, e.g., $8\times$ less space on the largest SE-150K collection. Table 3 from Section C motivates why this is the case: the number of meta colors is nearly three orders of magnitude smaller than the number of colors. Furthermore, by applying filtering and deduplication of the meta color sets, the number of colors intersected is reduced by $36 - 46\%$, significantly reducing the time spent in intersecting the partial color sets.



(a) Space improvement. **(b)** Query time improvement.

Figure 6 Space and query time improvement factors thanks to repetition-aware compression, on the various SE collections and for the full-intersection query mode. Specifically, plot (a) shows the ratio between the space of h-Fulgor (not repetition-aware) and md-Fulgor (repetition-aware). Likewise, plot (b) shows the ratio between the query time of md-Fulgor before (not repetition-aware) and after (repetition-aware) the optimizations.

- md-Fulgor, which combines the benefits of both the DR and MP representations, is the representation that improves the most with the optimizations, reaching speedups of up to 3.2× for both algorithms.

6 Conclusions and future work

We have introduced novel algorithms to process the compressed color sets of c-dBGs, tailored to the repetition-aware representations used in the Fulgor index (although these techniques can be applied to any index using the same color set representations). Our experiments demonstrate that these algorithms significantly reduce query times with no additional space usage. Notably, the md-Fulgor representation can now match (or even improve over) the performance of the fastest Fulgor variant, while using one order of magnitude less memory.

An interesting trend emerges when examining how performance scales with dataset size. As shown in Figure 6, space effectiveness and query efficiency *increase* with the number of genomes in the dataset. In particular, looking at Figure 6a, the compression ratio between the original h-Fulgor and md-Fulgor goes from 2.5× on the smallest SE dataset, to 11.22× for the largest dataset. Similarly, we can see in Figure 6b that md-Fulgor achieves a 1.6× speedup on the smallest dataset over the non-repetition-aware algorithm, which grows to a 3.25× speedup on the largest dataset. We are therefore led to think that these factors can grow more for larger collections (see Section D for further experiments on larger collections). These considerations open several directions for future work.

We plan to investigate batch query strategies, which could further improve query throughput because similar queries in a batch have similar results. As writing the output of pseudoalignment is itself a very time-consuming task, future versions of Fulgor will explore ways to write and compress the output efficiently. Lastly, a key goal is to build a Fulgor index over massive databases such as RefSeq [27], AllTheBacteria [18], and Logan [9], for which an external-memory based approach to compress and search the color sets is foreseen.

References

1 Jarno N. Alanko. 3682 E. Coli assemblies from NCBI, 2022. URL: `https://zenodo.org/records/6577997`.

2 Jarno N Alanko, Jaakko Vuohtoniemi, Tommi Mäklin, and Simon J Puglisi. Themisto: a scalable colored k-mer index for sensitive pseudoalignment against hundreds of thousands of bacterial genomes. *Bioinformatics*, 39(Supplement_1):i260–i269, 2023.

3 Almodaresi, Fatemeh and Zakeri, Mohsen and Patro, Rob. PuffAligner: a fast, efficient and accurate aligner based on the Pufferfish index. *Bioinformatics*, 37(22):4048–4055, June 2021. doi:10.1093/BIOINFORMATICS/BTAB408.

4 Jérémy Barbay, Alejandro López-Ortiz, and Tyler Lu. Faster adaptive set intersections for text searching. *Experimental Algorithms*, page 146, 2006.

5 Timo Bingmann, Phelim Bradley, Florian Gauger, and Zamin Iqbal. Cobs: a compact bit-sliced signature index. In *International Symposium on String Processing and Information Retrieval*, pages 285–303. Springer, 2019. doi:10.1007/978-3-030-32686-9_21.

6 Grace A. Blackwell, Martin Hunt, Kerri M. Malone, Leandro Lima, Gal Horesh, Blaise T. F. Alako, Nicholas R. Thomson, and Zamin Iqbal. Exploring bacterial diversity via a curated and searchable snapshot of archived DNA sequences. *PLOS Biology*, 19(11):1–16, November 2021. URL: http://ftp.ebi.ac.uk/pub/databases/ENA2018-bacteria-661k.

7 Nicolas L Bray, Harold Pimentel, Páll Melsted, and Lior Pachter. Near-optimal probabilistic RNA-seq quantification. *Nature biotechnology*, 34(5):525–527, 2016.

8 Alessio Campanelli, Giulio Ermanno Pibiri, Jason Fan, and Rob Patro. Where the patterns are: repetition-aware compression for colored de Bruijn graphs. *Journal of Computational Biology*, 31(10):1022–1044, 2024. doi:10.1089/CMB.2024.0714.

9 Rayan Chikhi, Brice Raffestin, Anton Korobeynikov, Robert Edgar, and Artem Babaian. Logan: planetary-scale genome assembly surveys life's diversity. *bioRxiv*, pages 2024–07, 2024.

10 J Shane Culpepper and Alistair Moffat. Efficient set intersection for inverted indexing. *ACM Transactions on Information Systems (TOIS)*, 29(1):1–25, 2010. doi:10.1145/1877766.1877767.

11 Erik D Demaine, Alejandro López-Ortiz, and J Ian Munro. Adaptive set intersections, unions, and differences. In *Proceedings of the eleventh annual ACM-SIAM symposium on Discrete algorithms*, pages 743–752, 2000. URL: http://dl.acm.org/citation.cfm?id=338219.338634.

12 Peter Elias. Universal codeword sets and representations of the integers. *IEEE Transactions on Information Theory*, 21(2):194–203, 1975. doi:10.1109/TIT.1975.1055349.

13 Jason Fan, Jamshed Khan, Noor Pratap Singh, Giulio Ermanno Pibiri, and Rob Patro. Fulgor: a fast and compact k-mer index for large-scale matching and color queries. *Algorithms for Molecular Biology*, 19(1):3, 2024. doi:10.1186/S13015-024-00251-9.

14 Ragnar Groot Koerkamp. A*PA2: Up to 19× Faster Exact Global Alignment. In *24th International Workshop on Algorithms in Bioinformatics (WABI 2024)*, volume 312, pages 17:1–17:25, 2024.

15 Ragnar Groot Koerkamp and Pesho Ivanov. Exact global alignment using A* with chaining seed heuristic and match pruning. *Bioinformatics*, 40(3):btae032, 2024.

16 Mahdi Heydari, Giles Miclotte, Yves Van de Peer, and Jan Fostier. Browniealigner: accurate alignment of illumina sequencing data to de bruijn graphs. *BMC bioinformatics*, 19:1–10, 2018. doi:10.1186/S12859-018-2319-7.

17 Pranvera Hiseni, Knut Rudi, Robert C Wilson, Finn Terje Hegge, and Lars Snipen. HumGut: a comprehensive human gut prokaryotic genomes collection filtered by metagenome data. *Microbiome*, 9(1):1–12, 2021. URL: https://arken.nmbu.no/~larssn/humgut/index.htm.

18 Martin Hunt, Leandro Lima, Daniel Anderson, Jane Hawkey, Wei Shen, John Lees, and Zamin Iqbal. AllTheBacteria – all bacterial genomes assembled, available and searchable. *bioRxiv*, pages 2024–03, 2024. URL: https://allthebacteria.org.

19 Zamin Iqbal, Mario Caccamo, Isaac Turner, Paul Flicek, and Gil McVean. De novo assembly and genotyping of variants using colored de bruijn graphs. *Nature genetics*, 44(2):226–232, 2012.

20 Mikhail Karasikov, Harun Mustafa, Amir Joudaki, Sara Javadzadeh-no, Gunnar Rätsch, and André Kahles. Sparse Binary Relation Representations for Genome Graph Annotation. *Journal of Computational Biology*, 27(4):626–639, April 2020. doi:10.1089/CMB.2019.0324.

21 John A Lees, T Tien Mai, Marco Galardini, Nicole E Wheeler, Samuel T Horsfield, Julian Parkhill, and Jukka Corander. Improved prediction of bacterial genotype-phenotype associations using interpretable pangenome-spanning regressions. *MBio*, 11(4):10–1128, 2020.

22 Antoine Limasset, Bastien Cazaux, Eric Rivals, and Pierre Peterlongo. Read mapping on de bruijn graphs. *BMC bioinformatics*, 17:1–12, 2016.

23 Bo Liu, Hongzhe Guo, Michael Brudno, and Yadong Wang. debga: read alignment with de bruijn graph-based seed and extension. *Bioinformatics*, 32(21):3224–3232, 2016. doi:10.1093/BIOINFORMATICS/BTW371.

24 Nina Luhmann, Guillaume Holley, and Mark Achtman. BlastFrost: fast querying of 100, 000s of bacterial genomes in bifrost graphs. *Genome Biology*, 22(1), January 2021.

25 Buwani Manuweera, Joann Mudge, Indika Kahanda, Brendan Mumey, Thiruvarangan Ramaraj, and Alan Cleary. Pangenome-Wide Association Studies with Frequented Regions. In *Proceedings of the 10th ACM International Conference on Bioinformatics, Computational Biology and Health Informatics*. ACM, September 2019. doi:10.1145/3307339.3343478.

26 Ilia Minkin and Paul Medvedev. Scalable multiple whole-genome alignment and locally collinear block construction with SibeliaZ. *Nature Communications*, 11(1), December 2020.

27 Nuala A O'Leary, Mathew W Wright, J Rodney Brister, Stacy Ciufo, Diana Haddad, Rich McVeigh, Bhanu Rajput, Barbara Robbertse, Brian Smith-White, Danso Ako-Adjei, et al. Reference sequence (refseq) database at ncbi: current status, taxonomic expansion, and functional annotation. *Nucleic acids research*, 44(D1):D733–D745, 2016.

28 Giulio Ermanno Pibiri. Sparse and skew hashing of k-mers. *Bioinformatics*, 38(Supplement_1):i185–i194, 2022. doi:10.1093/BIOINFORMATICS/BTAC245.

29 Giulio Ermanno Pibiri. On weighted k-mer dictionaries. *Algorithms for Molecular Biology*, 18(3), 2023. doi:10.1186/S13015-023-00226-2.

30 Giulio Ermanno Pibiri and Alessio Campanelli. Fulgor. Software, version 4.0.0., swhId: swh:1:dir:c9c71b24bc997397673f39f0cb7905660f09ff88 (visited on 2025-08-01). URL: https://github.com/jermp/fulgor, doi:10.4230/artifacts.24311.

31 Giulio Ermanno Pibiri, Jason Fan, and Rob Patro. Meta-colored compacted de bruijn graphs. In *International Conference on Research in Computational Molecular Biology*, pages 131–146. Springer, 2024. doi:10.1007/978-1-0716-3989-4_9.

32 Giulio Ermanno Pibiri and Rossano Venturini. Techniques for inverted index compression. *ACM Computing Surveys (CSUR)*, 53(6):1–36, 2020. doi:10.1145/3415148.

33 Mikko Rautiainen and Tobias Marschall. Graphaligner: rapid and versatile sequence-to-graph alignment. *Genome biology*, 21(1):253, 2020.

34 Giorgos Skoufos, Fatemeh Almodaresi, Mohsen Zakeri, Joseph N. Paulson, Rob Patro, Artemis G. Hatzigeorgiou, and Ioannis S. Vlachos. AGAMEMNON: an accurate meta-Genomics and MEtatranscriptoMics quaNtificatiON analysis suite. *Genome Biology*, 23(1), January 2022.

35 Esko Ukkonen. Approximate string-matching with q-grams and maximal matches. *Theoretical computer science*, 92(1):191–211, 1992. doi:10.1016/0304-3975(92)90143-4.

A Omitted proofs from Sections 3 and 4

▶ **Lemma 2** (DR intersection). *Given z sets, $C_1, \ldots, C_z$, represented differentially as $D_i = C_i \Delta A$ with respect to the representative set A, for $i = 1, \ldots, z$, their intersection can be computed as*

$$\bigcap_{i=1}^{z} C_i = \left(A \cap \overline{D}_1 \cap \cdots \cap \overline{D}_z\right) \cup \left(\overline{A} \cap D_1 \cap \cdots \cap D_z\right).$$

Proof. The symmetric difference between two sets A and D can be expressed as

$$A \Delta D = (A \cup D) \backslash (A \cap D) = (A \backslash D) \cup (D \backslash A) = (A \cap \overline{D}) \cup (\overline{A} \cap D). \tag{1}$$

Using the formulation of Equation 1, we can rewrite the intersection of 2 differentially stored color sets, belonging to the same group, as

$$\begin{aligned} C_1 \cap C_2 &= (A \Delta D_1) \cap (A \Delta D_2) \\ &= \big((A \cap \overline{D}_1) \cup (\overline{A} \cap D_1)\big) \cap \big((A \cap \overline{D}_2) \cup (\overline{A} \cap D_2)\big) \end{aligned}$$

and distributing the intersection over the union, we obtain

$$\begin{aligned} &= (A \cap \overline{D}_1 \cap A \cap \overline{D}_2) \cup (A \cap \overline{D}_1 \cap \overline{A} \cap D_2) \cup \\ &\quad (\overline{A} \cap D_1 \cap A \cap \overline{D}_2) \cup (\overline{A} \cap D_1 \cap \overline{A} \cap D_2) \\ &= (A \cap \overline{D}_1 \cap \overline{D}_2) \cup (\overline{A} \cap D_1 \cap D_2). \end{aligned}$$

The same argument can be extended to z sets and the lemma follows. ◀

▶ **Lemma 5** (Dynamic Thresholding). *For any $c \in [1..N]$, $\mu_c \geq T$ if and only if*

$$\sum_{\substack{i:\ c \in C_i \wedge \\ C_i \text{ not v.d.}}}^{z} \gamma_i - \sum_{\substack{i:\ c \in \overline{C}_i \wedge \\ C_i \text{ v.d.}}}^{z} \gamma_i \geq T - \sum_{i:\ C_i \text{ v.d.}}^{z} \gamma_i.$$

Proof. In the following, we prove $\implies$ only, given that $\impliedby$ follows a symmetric argument. We start by rewriting μ_c, by grouping the terms based on their density as follows

$$\mu_c = \sum_{i:\ c \in C_i}^{z} \gamma_i = \sum_{\substack{i:\ c \in C_i \wedge \\ C_i \text{ not v.d.}}}^{z} \gamma_i + \sum_{\substack{i:\ c \in C_i \wedge \\ C_i \text{ v.d.}}}^{z} \gamma_i \ . \tag{2}$$

Since $\mu_c \geq T \iff \mu_c - v \geq T - v$ for any given value v, by setting $v = \sum_{i:\ C_i \text{ v.d.}}^{z} \gamma_i$ and writing μ_c as in Equation 2 we get

$$\sum_{\substack{i:\ c \in C_i \wedge \\ C_i \text{ not v.d.}}}^{z} \gamma_i + \sum_{\substack{i:\ c \in C_i \wedge \\ C_i \text{ v.d.}}}^{z} \gamma_i - \sum_{i:\ C_i \text{ v.d.}}^{z} \gamma_i \geq T - \sum_{i:\ C_i \text{ v.d.}}^{z} \gamma_i \ . \tag{3}$$

Given that the inclusion of a color c in a set also creates a partition, we can write the sum of the scores of all dense colors as

$$\sum_{i:\ C_i \text{ v.d.}}^{z} \gamma_i = \sum_{\substack{i:\ c \in C_i \wedge \\ C_i \text{ v.d.}}}^{z} \gamma_i + \sum_{\substack{i:\ c \notin C_i \wedge \\ C_i \text{ v.d.}}}^{z} \gamma_i = \sum_{\substack{i:\ c \in C_i \wedge \\ C_i \text{ v.d.}}}^{z} \gamma_i + \sum_{\substack{i:\ c \in \overline{C}_i \wedge \\ C_i \text{ v.d.}}}^{z} \gamma_i \ . \tag{4}$$

Substituting Equation 4 in Equation 3, the lemma follows. ◀

▶ **Lemma 7** (DR union). *Given z sets, $C_1, \ldots, C_z$, represented differentially as $D_i = C_i \Delta A$ with respect to the representative set A, for $i = 1, \ldots, z$, their threshold-union can be computed by including all $c \in [1..N]$ such that $\mu_c \geq T$ where*

$$\mu_c = \begin{cases} \mu'_c & \text{if } c \notin A \\ \gamma - \mu'_c & \text{if } c \in A \end{cases}$$

with $\mu'_c = \sum_{i:c \in D_i}^{z} \gamma_i$ being the sum of the scores of the color sets C_i for which $c \in D_i$.

Proof. Consider the first case, $c \notin A$. Since c does not belong to the representative set, if $c \in D_i$, then $c \in C_i$. Consequently, γ_i contributes to the score μ_c. Now, consider the second case, $c \in A$. For it to be in a color set C_i, it must hold that $c \notin D_i$. Therefore, μ_c is the sum of the scores γ_i of all differential sets in which c does not appear. Since the inclusion of c forms a partition on the color sets, $\gamma = \sum_{i:c \in D_i}^{z} \gamma_i + \sum_{i:c \notin D_i}^{z} \gamma_i = \mu'_c + \mu_c$, thus $\mu_c = \gamma - \mu'_c$. ◀

B Query time breakdowns

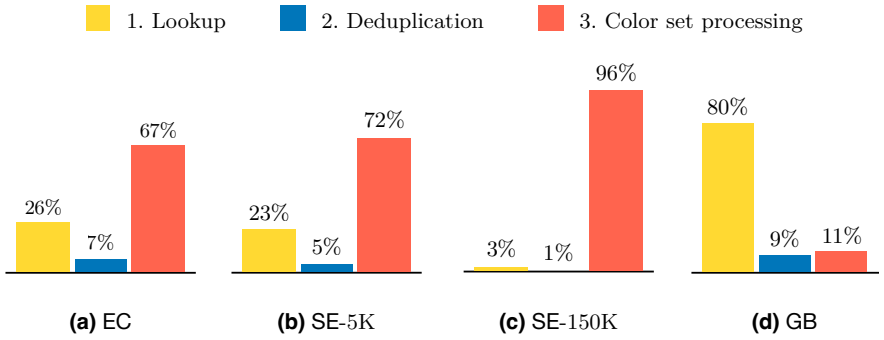


Figure 7 Pseudoalignment query time breakdown for some datasets, under full-intersection.

C Additional experimental data

Table 3 Total number of colors and meta colors after each step of Algorithm 3 (at page 9). On the left, the number of meta colors at the start of the algorithm (Total), after intersecting the partitions (Filtered), and after the deduplication (Dedup.). On the right, the number of colors at the start of the algorithm (Total) and intersected with the INTERSECT algorithm (Intersected). Percentages show the ratio between the step and the total.

	Meta Colors ($\times 10^6$)				Colors ($\times 10^9$)		
	Total	Filtered		Dedup.	Total	Intersected	
SE-5K	287.30	176.87	(62%)	82.84 (29%)	103.88	66.49	(64%)
SE-10K	633.60	298.69	(47%)	117.32 (19%)	261.24	138.76	(53%)
SE-50K	4,552.91	2,059.82	(45%)	472.71 (10%)	2,015.63	1,129.44	(56%)
SE-100K	6,945.72	2,921.84	(42%)	589.45 (8%)	4,488.86	2,884.84	(64%)
SE-150K	10,141.07	3,800.93	(37%)	711.68 (7%)	7,192.40	4,390.90	(61%)

D Results on the "All-The-Bacteria" collection

The following experiments were conducted inside a Docker container running Linux kernel version 6.8.0, on a machine equipped with an AMD Ryzen Threadripper PRO 7985WX CPU (128 cores, each clocked at 5.4GHz) and 269GB of RAM.

The evaluations were performed on different collections of Salmonella Enterica (SE), ranging from 50,000 to 300,000 genomes from the "AllTheBacteria" collection [18].

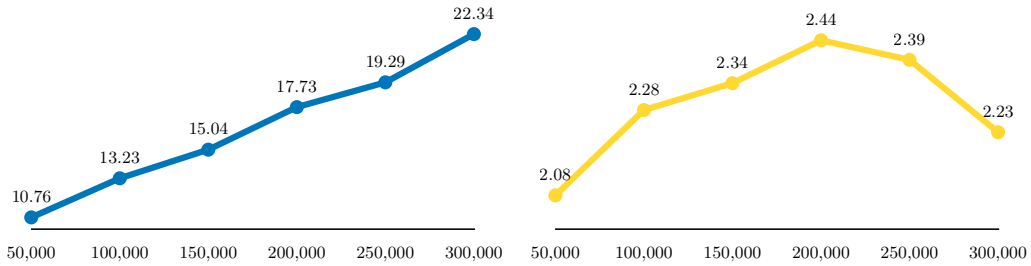
These experiments confirm the superiority of repetition-aware pseudoalignment algorithms against traditional ones. Focusing our attention on md-Fulgor, our new methods improve query times by more than 2× on average, while using more than an order of magnitude less memory. As shown in Tables 4 and 5, for SE-300K – our largest tested collection to date – query times are 2.23× faster, and require 22.23× less memory. Moreover, as highlighted by Figure 8, space compression ratios keep increasing with even bigger datasets, confirming the scalability of our indexes.

Table 4 Total query time (formatted as h:mm:ss) and memory usd in GB (max. RSS) for **full-intersection** as reported by `/usr/bin/time`, using 16 processing threads.

Dataset	Mapping rate	h-Fulgor			d-Fulgor			m-Fulgor			md-Fulgor		
		before	after	GB	before	after	GB	before	after	GB	before	after	GB
SE-50K	88.94%	9:14	**6:43**	20.29	16:06	**9:13**	6.02	9:22	**6:47**	2.62	16:25	**7:54**	1.89
SE-100K	89.41%	19:12	**13:38**	48.85	34:20	**18:28**	13.31	19:43	**13:14**	5.34	35:38	**15:39**	3.69
SE-150K	89.57%	29:00	**20:37**	78.93	52:02	**27:22**	20.57	29:32	**19:04**	7.21	55:29	**23:43**	5.25
SE-200K	90.81%	49:58	**35:34**	111.17	1:14:15	**37:54**	27.94	41:30	**27:04**	8.81	1:17:52	**31:55**	6.27
SE-250K	90.93%	1:07:00	**52:06**	144.98	1:33:36	**48:50**	35.31	54:11	**34:09**	11.07	1:38:58	**41:21**	7.52
SE-300K	90.99%	1:23:33	**1:04:53**	180.68	1:52:03	**58:26**	43.49	1:03:15	**42:45**	12.98	1:59:36	**53:44**	8.09

Table 5 Total query time (formatted as h:mm:ss) and memory usd in GB (max. RSS) for **threshold-union** with $\tau = 0.8$ as reported by `/usr/bin/time`, using 16 processing threads.

Dataset	Mapping rate	h-Fulgor			d-Fulgor			m-Fulgor			md-Fulgor		
		before	after	GB	before	after	GB	before	after	GB	before	after	GB
SE-50K	89.27%	18:04	**8:03**	20.31	19:00	**8:19**	6.02	11:29	**7:13**	2.62	18:31	**7:24**	1.91
SE-100K	89.85%	39:54	**16:25**	48.88	41:47	**16:38**	13.32	25:21	**14:27**	5.37	40:04	**15:07**	3.73
SE-150K	90.04%	1:01:08	**24:53**	79.05	1:04:40	**25:02**	20.61	38:36	**20:38**	7.25	1:02:23	**22:18**	5.20
SE-200K	91.32%	1:35:16	**43:46**	111.15	1:29:52	**35:14**	27.98	53:02	**29:27**	8.88	1:26:58	**31:31**	6.31
SE-250K	91.45%	2:05:26	**59:38**	145.00	1:56:29	**44:59**	35.27	1:06:23	**36:45**	11.07	1:53:14	**39:38**	7.47
SE-300K	91.54%	2:33:55	**1:14:46**	180.34	2:20:13	**53:47**	43.12	1:21:56	**44:21**	12.64	2:17:09	**49:47**	7.85



(a) Space improvement. **(b)** Query time improvement.

Figure 8 Space and query time improvement factors thanks to repetition-aware compression, on the various SE collections and for the full-intersection query mode. Specifically, plot (a) shows the ratio between the space of h-Fulgor (not repetition-aware) and md-Fulgor (repetition-aware). Likewise, plot (b) shows the ratio between the query time of md-Fulgor before (not repetition-aware) and after (repetition-aware) the optimizations.