AS-Index: A Structure For String Search Using n-grams and Algebraic Signatures. JOURNAL OF COMPUTER SCIENCE AND TECHNOLOGY : 1– Jul. 2014

Camelia Constantin[1], Cédric du Mouza[2], Witold Litwin[3], Philippe Rigaux[2], Thomas Schwarz[4]

[1]LIP6 *laboratory, University Pierre et Marie Curie, Paris, France*
[2]CEDRIC *laboratory, Conservatoire National des Arts et Métiers, Paris, France*
[3]LAMSADE *laboratory, University Paris-Dauphine, Paris, France*
[4]DICC *laboratory, Catolica del Uruguay, Montevideo, Uruguay*

E-mail: camelia.constantin@lip6.fr; dumouza@cnam.fr; witold.litwin@dauphine.fr; philippe.rigaux@cnam.fr; tschwarz@ucu.edu.uy

AS-Index: A Structure For String Search Using n-grams and Algebraic Signatures

Abstract We present the AS-Index, a new index structure for exact string search in disk resident databases. AS-index relies on a classical inverted file structure, its main innovation being a probabilistic search based on the properties of algebraic signatures used both for n-grams hashing and pattern search. Specifically, the properties of our signatures allow to carry out a search by inspecting only two of the posting lists. The algorithm thus enjoys the unique feature of requiring a constant number of disk accesses, independently from both the pattern size and the database size. We conduct extensive experiments on large datasets to evaluate our index behavior. They confirm that it steadily provides a search performance proportional to the two disk accesses necessary to obtain the posting lists. This makes our structure a choice of interest for the class of applications that require very fast lookups in large textual databases.

We describe the index structure, our use of algebraic signatures and the search algorithm. We discuss the operational trade-offs based on the parameters that affect the behavior of our structure, and present the theoretical and experimental performance analysis. We next compare the AS-Index to the state-of-the-art alternatives and show that (i) the construction time matches that of the competitors, due to the similarity of structures, (ii) the search time constantly outperforms the standard approach, thanks to the economical access to data complemented by signature calculations, which is at the core of our search method.

Keywords Full text indexing, Large scale indexing, Algebraic signatures

1 Introduction

Databases increasingly store data of various kinds such as text, DNA records, and images. This data is at least partly unstructured, which creates the need for full text searches (or pattern matching) [1]. In main memory, matching a pattern P against a string S runs in $O(|S|/|P|)$ at best [2]. Searching very large data sets requires a disk-resident index, involving some storage overhead and a possibly long index construction time.

We address the problem of searching arbitrarily long strings in external memory. We assume a database $D = \{R_1, R_2, \cdots, R_n\}$ of records, viewed as strings over an alphabet Σ. The database supports record insertion, deletion and updates, as well as search on any substring, called *pattern*, of the records' contents.

Currently deployed systems for textual documents use almost exclusively inverted files indexing *words* for keyword search. However, our need for full pattern matching in records where the concept of word may not exist rules out this solution. Suffix trees and arrays form a class of indexes for pattern matching. Suffix trees work best when they fit into RAM. Attempts to create versions that work from disks are recent and experimental. They raise difficult technical issues [3, 4] due to a bad locality of reference, a necessarily complex paging scheme, and structural deterioration caused by inserts into the structure. A suffix array stores pointers on a list of suffixes sorted in lexicographic order, and uses binary search for pattern matching. However, a standard database architecture stores records in blocks and supports dynamic inser-

tions and deletions, which makes difficult the maintenance of sequential storage. A suffix array search needs $O(\log_2 N)$ block accesses, where N is the size (number of characters) of D. We are not aware of a generic solution to these difficulties in the literature, and without one, these costs disqualify suffix arrays for our needs.

Two approaches that explicitly address disk-based indexing for full pattern-matching searches are the String B-Tree [3] and n-gram inverted index [5, 6]. The String B-Tree is basically a combination of B^+-Tree and Patricia Tries. The global structure is that of a B^+-Tree, where keys are pointers to suffixes in the database. Each node is organized as a Patricia Trie, which helps guiding the search and insert operations. A String B-Tree finds all occurrences of a pattern P in $O(|P|/B + \log_B N)$ disk accesses, where B is the block size. Another direction for text search needs are indexes inverting n-grams (n consecutive symbols) instead of entire words. They are "disk friendly" in that they rely on fast sequential scans, provide a good locality of reference, and easily adapt to paging and partitioning. However, the search cost is linear, both in the size of the database and the size of the pattern.

In the present paper, we introduce a new data structure for index-based full-text search called *Algebraic Signature Index* (AS-Index). It follows the path of an inverted file based on n-grams, and combines standard database large-scale indexing (namely hashing) with a new hash calculus based on algebraic signatures, (ASs) [7]. *The main originality of our approach is to take advantage of the interpretation of characters as symbols in a mathematical structure. The operations in this structure (a Galois Field) contribute to develop new computational techniques that identify the result items of a search in constant time.* As a result, our index structure enjoys the attractive property, unique at present to our knowledge, of performing records lookup with constant number of disk accesses, independent from both the size of the database and the length of the pattern.

Our experiments show AS-Index to be a very fast solution to pattern searches in a database. AS-Index search only needs two disk lookups when the hash directory fits in main memory. In our experiments, it proved itself to be up to one order of magnitude faster than n-gram indexes and twice faster than String B-Trees. Although the AS-Index is probabilistic by nature and could returns some false-positives, we analytically demonstrate, and confirm by our experiments, that they are very unlikely. The basic variant of AS-Index has a storage overhead of about 5 to 6. A variant of our scheme only indexes selective n-grams and has lower storage overhead at the costs of slower search times and higher rates of false-positives. All these properties make AS-Index a practical solution for text indexing.

This paper extends [8] by proposing a complete theoretical analysis of the expected AS-Index performance and of the probability of false-positive. It also introduces a non-dense indexing variant. This extended version includes a detailed description of our implementations for the different structures, incorporating data compression, and a large revisited experimental section with larger datasets (20 GB versus 100 MB in the conference paper, with support of the data compression), a study of false positives, and evaluation of the non-dense indexing variant.

The paper is organized as follows. Section 2 gives an overview of the AS-Index basic principles. We recall the theory of algebraic signatures in Section 3. We then discuss the AS-Index structure in Section 4 and the search algorithm in Section 5. Section 6 analyses the scheme's behavior, especially collision and false positive probability, as well as performance and Section 7 presents the variants.

Section 8 explains the details of our implementation of String B-Trees, n-gram index and AS-Index and experiments. We review related work in Section 9. Finally, we summarize and give future research directions in Section 10.

2 AS-Index overview

AS-Index is a classical hash file with variable length disk-resident buckets, (see Fig. 1). Buckets are pointed to by the hash directory. Simplicity and performance of such files attracted countless applications. Their main advantage is a constant number of disk accesses, independent of file and pattern size. Constant number of disk accesses is not possible for a tree/trie access method. Each bucket stores a list of *entries*, each entry indexing some n-gram in the database. The basic variant of AS-Index is dense, indexing every n-

gram. The hash function providing the bucket for an entry uses the n-gram value as the hash key. The hash function is particular: it relies on *algebraic signatures* of n-grams, to be described in the next section. In what follows, we only deal with the static AS-Index, but the hash structure may use standard mechanisms for dynamicity, scalability and distribution.

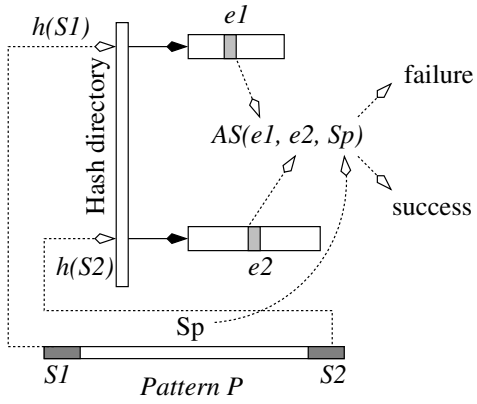

Fig. 1. A matching attempt with AS-Index

In overview, a search for a pattern P proceeds as follows (Fig. 1): First, we preprocess P for three signatures: (i) of the initial n-gram S_1, (ii) of the final n-gram S_2 and (iii) of the suffix S_p of P after S_1 (and including S_2). Hashing on S_1 locates the bucket with every entry e_1 indexing an n-gram in the database with the same signature as S_1. Likewise, hashing on S_2 locates the bucket with every entry e_2 indexing an n-gram with the signature of S_2. We only consider pairs of entries that are in the same record and at the right distance among them. We thus locate any string S matching P on its initial and terminal n-gram, at least by signature.

Finally, an algebraic calculation $AS(e_1, e_2, S_p)$ determines whether S_p matches the suffix of S as well. *This calculation is made possible by the specific properties of the signatures that allow to check some semantics properties (e.g., string matching) in spite of their very compact representation, something which could not be achieved with standard hash functions.* The method is probabilistic in nature, with low chances of false positives. We can avoid even a minute possibility of a false match by a symbol for symbol comparison between pattern and the relevant part of the record.

By limiting disk accesses to the two buckets associated to the first and last n-grams of P, AS-Index search runs independently from P's size. The cost of the search procedure outlined above is reduced to that of reading two buckets. The hash directory itself can often be cached in RAM, or needs at most two additional disk accesses, as we will show. With an appropriate dynamic hashing mechanism that evenly distributes the entries in the structure and scales gracefully, the bucket size is expected to remain uniform enough to let the AS-Index run in constant time, independently of the database size.

Table 1 compares the analytical behavior of AS-Index with those of two competitors (String B-Trees [3] and n-gram index [5, 6]) and summarizes its expected advantages. The size of all structures is linear in the size $|D|$ of the database. The ratio directly depends on the size of index entries. We mention in Table 1 the ratio obtained in our implementation, before any compression. The asymptotic search time in the database size is linear for n-gram index, logarithmic for String B-Trees and constant for AS-Index. Moreover, once the pattern P has been pre-processed, AS-Index runs independently from P's size, whereas n-gram index cost is linear in $|P|$.

	String B-Trees	n-Gram	AS-Index
Constr.	$O(\|D\| \times \log_B \|D\|)$	$O(\|D\|)$	$O(\|D\|)$
Storage (ratio)	$O(\|D\|)$ (∼6-7)	$O(\|D\|)$ (∼6)	$O(\|D\|)$ (∼5-6)
Preproc.	none	$O(\|P\|)$	$O(\|P\|)$
Search	$O(\|P\|/B + \log_B \|D\|)$	$O(\|D\| \times \|P\|)$	$O(1)$

Table 1. Disk-based index structures for searching a pattern P in a database D. B is the block size.

In summary, AS-Index efficiently identifies matches with only two disk lookups, whatever the pattern length. This efficiency is achieved through extensive use of properties of algebraic signatures, described in the next section. *It also relies on the robustness of signatures to skewed distributions, analyzed in Section 6.*

3 Algebraic signatures

We use a Galois field $GF(2^f)$ of size 2^f. The elements of GF are bit strings of length f. Selecting $f = 8$ deals with ASCII records and $f = 16$ with Uni-

code records. We recall that a Galois field is a finite set that supports addition and multiplication. These operations are associative, commutative and distributive, have neutral elements 0 and 1, and there exist additive and multiplicative inverses. In a Galois field $GF(2^f)$, addition and subtraction are implemented as the bitwise XOR. Log/antilog tables provide usually the most practical method for multiplication [7]. We adopt the usual mathematical notations for the operations in what follows. We use a *primitive* element α of $GF(2^f)$. This means that the powers of α enumerate all the nonzero elements of the Galois field. It is well known that there always exist primitive elements.

Symbol	Interpretation
f	Size (in bits) of a GF element in $GF(2^f)$ ($f = 8$ or $f = 16$)
n	Size of n-grams
m	Size of signature vectors, $m \leq n$
M	Size of records
K	Size of patterns, $K > n$
L	Number of lines in the AS-Index
$r_0, r_1, \cdots r_{M-1}$	Record characters/symbols
$s_1, \cdots s_m$	One-symbol signatures
$S_1, \cdots S_m$	One-symbol n-gram signatures

Table 2. Table of the symbols used in the paper

Let $R = r_0 r_1 \cdots r_{M-1}$ be a record with M symbols. We interpret R as a sequence of GF elements. Identifying the character set of the records with a Galois field provides a convenient mathematical context to perform computations on record contents.

Definition 1. *The* m-symbols algebraic signature *(AS) of a record* R *is a vector* $AS_m(R)$ *with* m *coordinates* $(s_1, s_2, \ldots s_m)$ *defined by*

$$\begin{cases} s_1 = r_0 + r_1 \cdot \alpha + r_2 \cdot \alpha^2 \ldots + r_{M-1} \cdot \alpha^{M-1} \\ s_2 = r_0 + r_1 \cdot \alpha^2 + r_2 \cdot \alpha^4 \ldots + r_{M-1} \cdot \alpha^{2(M-1)} \\ \vdots \\ s_m = r_0 + r_1 \cdot \alpha^m + r_2 \cdot \alpha^{2m} \ldots + r_{M-1} \cdot \alpha^{m(M-1)} \end{cases} \quad (1)$$

We refer the reader to [7] for more details about definitions and properties of algebraic signatures.

In our examples, we represent the m-symbol AS of R as the concatenation of the values $s_m, \cdots, s_1$ in hexadecimal notation. For instance if $s_1 = \#34$ and $s_2 = \#12$, then we write the 2-symbol AS as $s_2 s_1 = \#1234$.

We use different partial algebraic signatures of pattern and database records, as we now explain.

Definition 2. *Let* $l \in [0, M-1]$ *be any position (*offset*) in* R*. The* Cumulative Algebraic Signature *(CAS) at* l*,* $CAS_m(R, l)$*, is the algebraic signature of the prefix of* R *ending at* r_l*, i.e.,* $CAS_m(R, l) = AS_m(r_0 \ldots r_l)$.

The *Partial Algebraic Signature (PAS) from* l' *to* l is the value $PAS_m(R, l', l) = AS_m(r_{l'} r_{l'+1} \cdots r_l)$, with $0 \leq l' \leq l$, Finally, we most often use the PAS of substrings of length n, i.e., of n-grams.

Definition 3. *The* n*-gram Algebraic Signature (NAS) of* R *at* l *is* $NAS_m(R, l) = PAS_m(R, l-n+1, l)$*, for* $l \geq n-1$*. In other words:*

$$\begin{aligned} NAS_m(R,l) \;=\; & (r_{l-n+1} + \cdots + r_l \cdot \alpha^{n-1}, \\ & r_{l-n+1} + \ldots + r_l \cdot \alpha^{2(n-1)}, \\ & \vdots, \\ & r_{l-n+1} + \ldots + r_l \cdot \alpha^{m(n-1)}) \end{aligned} \quad (2)$$

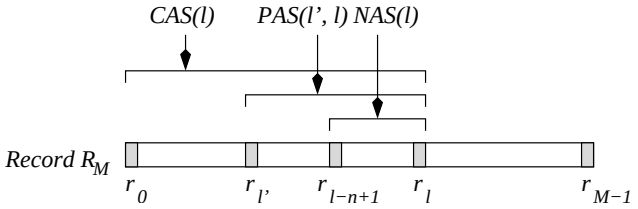

Fig. 2. Computing $CAS(l)$, $PAS(l', l)$ and $NAS(l)$ in record R_M

In all the definitions, we may drop R whenever it is implicit for brevity's sake. Figure 2 shows the respective parts of the record that define the CAS, PAS and NAS at offset l. The following simple properties of algebraic signatures, expressed for coordinate i, $1 \leq i \leq m$, are useful for what follows. We note the i-th symbol of a CAS at l as $CAS_m(l)_i$ and proceed similarly for NAS and PAS.

$$CAS_m(l)_i = CAS_m(l-1)_i + r_l \cdot \alpha^{il} \quad (3)$$

$$NAS_m(l)_i = \frac{NAS_m(l-1)_i - r_{l-n}}{\alpha} + r_l \cdot \alpha^{i(n-1)} \quad (4)$$

$$NAS_m(l)_i = \frac{CAS_m(l)_i - CAS_m(l-n)_i}{\alpha^{i(l-n+1)}} \quad (5)$$

For $0 \leq l' < l$:

$$CAS_m(l)_i = CAS_m(l')_i + \alpha^{i(l'+1)} PAS_m(l'+1, l)_i \quad (6)$$

Properties 3 and 4 let us incrementally calculate next CAS and NAS while scanning an input record or the pattern, instead of recomputing the signature entirely. This speeds up the process considerably. Property 5 also speeds up the pattern preprocessing, as it will be shown in the following. Property 6 finally is fundamental for the match attempt calculus. Table 2 summarizes the symbols used in the paper.

4 Structure

Our database consists of records that are made up of a Record IDentifier (RID) and some *non-key* field. (Extensions to databases with more than one key and/or non-key field are straight-forward.) We assume that the non-key field consists of strings of characters interpreted as symbols from our Galois field. Our search finds all occurrences of a pattern in the non-key field of any record in the database. When we talk about offsets and algebraic signatures, we refer only to the non-key field. If R is such a field and r_i a character (Galois field element) in R, then we call i the *offset*. An n-gram $G = r_{l-n+1} \cdots r_l$ is any sequence of n consecutive symbols in R. By extension, we then call l the *offset* of the n-gram.

An AS-Index consists of *entries*.

Definition 4. *Let G be an n-gram at offset o in R. The* entry *indexing G, denoted $E(G)$, is a triplet (rid, o', c) where rid is the RID of R, c is $CAS_1(R, o)$, and o' is o modulo $2^f - 1$.*

We assure constant size of the entries by taking the remainder. The choice of the modulus is justified by the identity $\chi^{2^f-1} = \chi$ for all Galois field elements χ.

The indexing is "dense", i.e., every n-gram in the database is indexed by a distinct entry. To construct the index, we process all n-grams in the database.

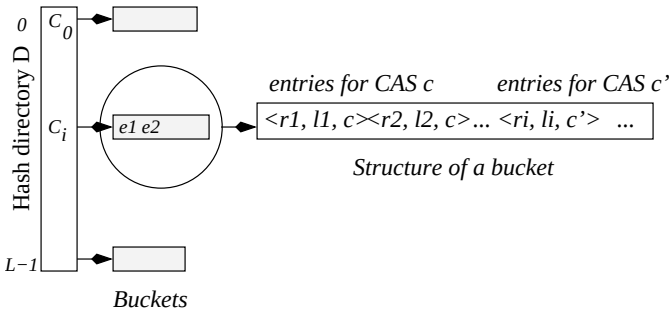

Fig. 3. Structure of the AS-index

AS-Index is a hash file, denoted $\boldsymbol{D}[0..L-1]$, with directory length $L = 2^v$ being a power of 2 (Fig. 3). Elements of $\boldsymbol{D}$ refer to buckets or *lines* of variable length, each containing a list of entries. Lines are of variable length to accommodate a possible uneven distribution of n-gram values. Each $\boldsymbol{D}[i]$ contains the address of the i-th bucket.

All together, the AS-Index line structure is similar to a posting list in an inverted file, except for the presence of the CAS c in each entry and a specific representation of the offset l. Since we use a hash file, lines should have a collision resolution method such as classical separate chaining that uses pointers to an overflow space. Such a technique accomodates moderate growth, but if we need to accomodate large growth, then we need a dynamic hashing method such as linear hashing.

We now describe how to calculate the index i of the line for an entry $E(G) = (rid, o, c)$. We calculate i from the m-symbol NAS $NAS_m(G) = (s_1, \ldots, s_m)$. The coordinates of the NAS are bit strings. By concatenating them, we obtain a bit string $S = s_m s_{m-1} \ldots s_1$ that we interpret as a large, unsigned integer. The index i is:

$$i = h_L(S) = S \mod L$$

Since $L = 2^v$, this amounts to extracting the last v bits of S. It is easy to see that m should be such that $m \leq n$ and $m \geq \lceil v/f \rceil$. The choice of AS-Index parameters m, n and L will be further discussed in Section 6.

Example 1. *Consider a 100 GB database with byte-wide symbols ($f = 8$). For the sake of example, let $L = 2^{30}$, leading to buckets with $\lceil 10^{11}/2^{30} \rceil = 93$ entries on the average. Let $n = 5$ and $m = 4$. To calculate line index i of n-gram G we thus concatenate $s_4..s_1$ of $NAS_4(G)$. Then we keep the last 30 bits.*

Now, consider the record with RID 73 and non-key field 'University Paris Dauphine'. Assume that NAS_4('Unive', 4) is #3456789a. Since this is the first 5-gram, $CAS_1(73, 4)$ has the same value as the first component, i.e., is #9a. For subsequent 5-grams, the first coordinate of the NAS and the CAS usually differ. The entry is $E = (73, 4, \#9a)$. Its line index is $\#3456789a \bmod 2^{30} = \#3456789a$ (we remove the leading 2 bits).

5 Pattern Search

Let $P = p_0 \dots p_{K-1}$ be the pattern to match. AS-Index search delivers the RID of every record in the database that contains P. The search algorithm first prepocesses P, then retrieves (possible) matching thanks to As-index.

5.1 Preprocessing

The preprocessing phase computes three signatures from the pattern P:
(a) the m-symbols AS of the 1st n-gram in P, called S_1;
(b) the 1-symbol PAS of the suffix of P following the 1st n-gram, $S_p = PAS_1(P, n, K-1)$.
(c) the m-symbols AS of the last n-gram in P, called S_2;

Observe we use a m-symbols AS for the n-grams to obtain a large range of hash values that fit the directory length, while we compute a 1-symbol PAS since for reducing storage costs we store 1-symbol CAS values in entries. There are several ways to compute these signatures. For instance, one may compute S_1, then S_p, then calculate the m-symbol value of S_2 thanks to property (5).

Figure 4 shows the parts of the pattern that determine the signatures S_1, S_2 and S_p on our running example. Recall that $n = 5$ and $m = 4$. We preprocess the pattern P ='University Paris Dauphine' and obtain
(a) $S_1 = NAS_4(P, 4)$ (for 5-gram 'Unive');
(b) $S_2 = NAS_4(P, 24)$ (for 5-gram 'phine') and
(c) $S_p = PAS_1(P, 5, 24)$.
This information is used to find the occurrences of the pattern in the database.

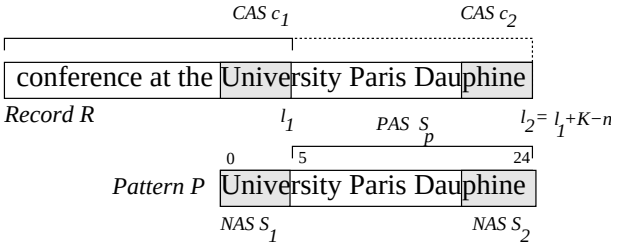

Fig. 4. The search algorithm

5.2 Processing

Let $i = h_L(S_1)$ and $i' = h_L(S_2)$. Every entry (R, l_1, c_1) in bucket $D[i]$ indexes an n-gram G in a record R whose NAS S_G is such that $h_L(S_G) = h_L(S_1)$. Likewise, every entry (R', l_2, c_2) in bucket $D[i']$ indexes an n-gram G' such that $h_L(G') = h_L(S_2)$. Up to possible *collisions*, G and G' respectively correspond to the first and last n-grams in P.

We only consider pairs (G, G') that possibly characterize a matching string $S = P$. This involves the additional constraints $R' = R$ and $l_2 = (l_1 + K - n) \bmod (2^f - 1)$. The last component in G', c_2, has to match the value computed from c_1 and S_p thanks to property 6, namely:

$$c_2 = c_1 + \alpha^{l_1+1} \cdot S_p. \tag{7}$$

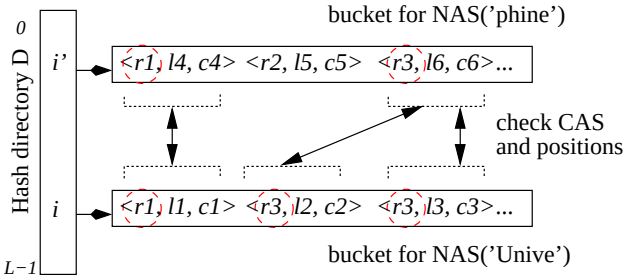

Fig. 5. Using AS-Index for a search

Figure 5 illustrates the process. By hashing on S_1 = 'Unive', we retrieve the bucket $D[i]$ which contains, among others, the entries indexing all occurrences of 'Unive' in the database. Similarly, we retrieve the bucket $D[i']$ which contains entries for all the occurrences of the n-gram 'phine'. A pair of entries $[e_i(r_1, l_1, c_1), e'_i(r_1, l_4, c_4)]$ in $(D[i], D[i'])$ represents a substring s of r_1 that begins with 'Unive' and ends with 'phine'. Checking whether s matches

P involves two tests: (i) we compute $c_1 + \alpha^{l_1+1}.S_p$ and compare it with c_4 to check whether the signatures of the middle parts match, and (ii) we verify as discussed whether l_1 matches l_2 given K, i.e., S and P have the same length. If both tests are succesful, we report a probable match. The next attempt will consider (r_3, l_2, c_2) in $D[i]$ and (r_3, l_6, c_6) in $D[i']$. Note that (r_2, l_5, c_5) in $D[i']$ is skipped because there is no possible match on r_2 in $D[i]$. The pseudo-code of the algorithm is given below.

Algorithm AS-SEARCH
Input: a pattern $P = p_0 \dots p_{K-1}$, the n-gram size n
Output: the list of records that contain P
begin
  // Preprocessing phase
  $S_1 := NAS_m(P, n-1)$
  $S_2 := NAS_m(P, K-1)$
  $S_p := PAS_1(P, n, K-1)$
  $i := h_L(S_1)$ // i is the first line index
  $i' := h_L(S_2)$ // i' is the second line index
  // Processing phase
  for each entry $E(R, l_1, c_1)$ in $D[i]$
    $c_2 = c_1 + \alpha^{l_1+1} \cdot S_p$
    $l_2 = (l_1 + K - n)\ mod\ (2^f - 1)$
    if (there exists an entry $E'(R, l_2, c_2)$ in $D[i']$) **then**
      Report success for R
    endif
  endfor
end

The selectivity of the process relies on its ability to manipulate three distinct signatures, S_1, S_2 and S_p. Therefore the pattern length must be at least $n + 1$.

5.3 Collision Handling

As hashing in general, our method is subject to collisions delivering false positives. To eliminate any collisions, it is necessary to post-process AS-Search by attempting to actually find P in every R identified as a match. This requires a symbol by symbol comparison between P and its presumed match location. It will appear however that AS-Search should typically have a negligible probability of a collision. Hence post-processing may be left to presumably rare applications needing full assurance. It is also RAM-based and therefore typically negligible with respect to the disk search time.

6 Analysis

We now present a theoretical analysis of the expected AS-Index performance.

6.1 Hash uniformity

Algebraic signature values tend to have a more uniform distribution than the distribution of character values, due to the multiplications by powers of α in their calculations. However, the total number of strings or n-grams in a dataset gives an upper bound for the number of algebraic signatures calculated from them. Biological databases often store DNA strings in an ASCII file, encoded with only four characters 'A', 'C', 'G', and 'T'. There are only $4^6 \approx 4K$ different 6-grams in the file and the number of different NAS of these signatures cannot exceed this value. Increasing the number of coordinates in the NAS beyond 5 symbols is not going to achieve better uniformity. This caution applies only to files using a small alphabet.

Nb Collisions	Taken	Expected
0	16,568,642	16,568,642.3
1	207,274	207,272
2	1,293	1,296.47
3	7	5.40624
> 3	0	0.0169079

Table 3. Actual and expected number of collisions using a 3B signature on a dictionary of 209,881 moderately sized English words.

Let us consider the more classical setting and natural language encoding. We chose a list of English words (209,881 words - 2.25MB) of six or more characters, taken from a word list used to perform a dictionary attack on a password file (by an administrator trying to weed out weak passwords). We calculated the 3B three component signature of all possible words with more than five characters (65,536 words) and calculated the number of words that correspond to each signature as well as this number for a perfect hash function (Table 3). So, *e.g.* we see that only 1,297 signa-

tures reveal a collision for 2 words, and only 7 are signatures for 3 words. The χ^2 value of 0.479166 shows very close agreement. When repeating the test using 2B two component signatures, we obtained $\chi^2 = 0.21$. A much smaller set had $\chi^2 = 4.79742$, but there were less signatures attained by a high number ($\geq$ 5) of words. These results give experimental verification for the "flatness" of signatures.

6.2 Storage and Performance

Index construction time. The properties (2) - (6) of algebraic signatures allow us to calculate all entries with a linear sweep of all records. We need to keep a pointer to the symbol just beyond the current n-gram and to the first symbol of the current n-gram. Using equations (3) and (4), we can then calculate the NAS of the next n-gram from the old one and update the running CAS of the record. Since creating the entry for an n-gram and inserting it into the index take constant time, building the index takes time linear to the size of the database.

Storage costs. The storage complexity of AS-Index is $O(N)$ for N indexed n-grams. The actual size of a RID should be 3-4 bytes since 3 bytes already allow a database with 16M records. The actual storage per entry should be about 5-6 bytes, which results in a storage overhead of about $(5-6)N$. We can lower this storage overhead, *e.g.* to 125%, by non-dense indexing at the expense of a proportional increase in search time, see Section 7.

Example 2. *We still consider a 100 GB database with 8b symbols. Assuming an average record size of 100 symbols, we have 1G records and our record identifier needs to be 4B long. We previously set the size of the CAS to 1B. With the 1B offset into the record, the entry is 6B. AS-Index should use about 6 times more space than the original database.*

Now assume records of 10KB each. The record identifiers can be 3B long. This gives a total entry size of 5B or a storage factor of 5.

Each element of the hash directory stores a bucket address with at least $\lceil \log_2(L) \rceil$ bits. In the case of our large 100GB database, with $L = 2^{30}$, choosing 4 bytes for the address leads to the required storage of 4.1GB, smaller than the current data servers standard capacity. In most cases, D is expected to fit in main memory.

Pattern preprocessing. To preprocess the pattern, we need to calculate a PAS and two NAS. We calculate both in a similar manner as above and obtain preprocessing times linear in the size of the pattern. Since the result depends on all symbols in the pattern, we cannot do better.

Search speed. We assume that entries are close to uniformly distributed. The search algorithm picks up two cells in the hash directory, in order to obtain the number of entries and the bucket references of, respectively, $\boldsymbol{D}[i]$ and $\boldsymbol{D}[i']$. Then the buckets themselves must be read. Each of these accesses may incur a random disk access, hence a (constant) cost of at most four disk reads. If the hash directory resides in main memory, the cost reduces to loading the two buckets.

The main memory cost is an in-RAM join of the two buckets. With l denoting the expected line length, and assuming the lines are ordered on document id and then on position in the document, the average complexity of this phase is (under our uniformity assumption) $2l$ while the worst case is $O(l^2)$. Observe that the worst case is highly unlikely as all the entries on both lines should fit into the same record. The optional collision resolution (symbol-to-symbol) test adds $O(|P|)$. This test is typically performed in RAM and makes only a negligible contribution to the otherwise constant cost Altogether, the search cost is

$$S = C_{hash} + C_{buck} + C_{ram} + C_{post} \qquad (8)$$

where C_{hash} represents the Hash Directory access cost, C_{buck} the bucket access cost, C_{ram} the RAM processing cost and C_{post} the post-processing.

Device	Access time
Processor speed	2 - 3 Gz per core
RAM speed	100ns
Flash disk access	0.4 - 0.5 ms
Magnetic disk	5 - 7 ms
Disk transfert rate	300 MB/s.

Table 4. Hardware characteristics

We now evaluate the actual search time that may result from the above complexity figures. We take as

basis the characteristics of the current popular hardware shown in Table 4 (see also [9] for a recent analysis).

C_{hash} is the cost of fetching 2 elements in the hash directory. The transfer overhead is negligible, and the (worst case) cost is therefore in the range $[10, 14]$ ms for magnetic disks. The bucket access cost C_{buck} is similar to C_{hash} regarding random disk accesses, but we fetch far more bytes per access. We need to transfer $2l$ entries. Table 4 suggests that we can transfer up to 300 KB per ms (Flash transfer rate are similar.) Since the size of an entry is typically 5-6 bytes, each search loads $10 - 12l$. It follows that for $l = 1K$, the line transfer cost is negligible. For $l = 16K$, 160 to 200 KB must be transferred. The cost is about 0.5 to 0.7 ms. This is still negligible with respect to disk accesses for AS-Index on magnetic disk, but not on solid one. In the latter case, the transfer cost is equivalent to an additional disk access.

The basic formula (average cost) for C_{ram}, the in-RAM join of the two lines, is $2l * E$ where E is a visited entry processing cost. In detail, we have 2 RAM accesses to Rids. In this test is successful, we need 2 additional accesses to CASs, 2 to offsets o and 1 access to the log table for algebraic computations. A conservative evaluation of $E = 250ns$ seems fair. The cost of the in-RAM join can thus be estimated as $100\mu s$ for $l = 200$, $500\mu s$ for $l = 1,000$, and $8ms$ for $l = 16K$. The first cost is negligible whatever the storage media; next one is so for disk, but not for flash, the latter is not for either.

Finally, the postprocessing cost P can be estimated based on a unit cost of $100ns$ per symbol. Even for a 1,000 symbols long pattern, the postprocessing costs $100\mu s$, which remains negligible for both magnetic disk or flash memories. Its importance also depends on the number of matches, of course.

6.3 Choice of file parameters

The previous analysis leads to the following conclusions regarding the choice of AS-Index parameters. As a general rule of thumb, one must choose parameter L so as to maintain the hash directory D in RAM. Caching D in RAM, whenever possible, saves two disk access on four. Setting a limit on L may lead to increase the average line length l, but our analysis shows that this remains beneficial even when l reaches hundreds or even thousands of entries. Under our assumptions, l should reach $16K$ to add the equivalent of 1 random access to the search cost, at which point one may consider enlarging the hash directory beyond the RAM limits.

Large values for l may also be beneficial with respect to other factors. First, larger lines accommodate a larger database for a fixed n. Second, we may choose a smaller n, with a smaller minimal size of $n + 1$ for patterns.

Let us illustrate this latter impact. We continue with our running example of a 100 GB database with byte-wide symbols ($f = 8$), L is 2^{30}, and an average load of $l = \lceil 10^{11}/2^{30} \rceil = 93$ entries per line. This implies the choice of m, which must be such that $2^{mf} \geq L$, i.e., $m \geq (\log_2 L)/f$. In our example, the line index i needs to consist of at least 30 bits. Correspondingly, NAS_m needs to be at least that long. Since each coordinate of NAS_m consists of 8b, the value for m needs to be at least 4. Each NAS then contains at least 32 bits.

The n-grams used need to contain at least m symbols. Otherwise, the range of n-gram values is smaller than L and certain lines will not contain any entries. If the n-grams are reasonably close to uniformly distributed, the range of values is 256^n and we can pick $n = m$. Still referring to our example with $L = 2^{30}$, we can choose $n = 4$.

However, the actual character set used is most often smaller than 256, or only a fraction of the characters appear frequently. This requires a larger n, since the range of possible n-grams must contain at least L values. Let v be the number of values we expect per symbol. In a simple ASCII text, the number of printable character codes is $v = 96$. DNA encoding represents an extreme case with $v = 4$. The n-gram size must be such that $v^n \geq L$. With $v = 96$ (simple ASCII text) and $L = 2^{30}$, n must be set to 5, the smallest value such that $96^n \geq L$, to generate all required NAS values. These parameter values were actually used for Example 1.

Consider now the case of a DNA database where only 4 of the possible 256 ASCII characters appear in records. We need to set $n = 15$ in order to obtain the 2^{30} possible signatures. $n + 1$ is the minimal pattern length we allow to search for. Such limit should not be

nevertheless a practical constraint for a search over a small alphabet. The need there is rather for long patterns [10]. If nevertheless it was a concern, one may choose a smaller n at the price of fewer, hence longer, lines. For instance choosing $n = 10$ and $L = 2^{20}$ for our DNA database results in the average of $95K$ entries in each line. The minimal pattern size decreases by five, i.e., to $n + 1 = \tilde{}11$ symbols.

6.4 Selectivity

We now evaluate how AS-Index performs with respect to searching random patterns in random files. If our scheme diagnoses a match at a given offset within a given record, we call this a *diagnosed match*. We now calculate how often we should have diagnosed matches and later deduce from this value the chance for false positives in searches for patterns that exist in the database. We first assume that the non-key fields for the records are smaller than $2^f - 1$. The conditions for a diagnosed match are then:
(1) The m-symbol NAS of the first n-gram in the pattern matches the NAS of the first n-gram in the substring.
(2) The m-symbol NAS of the last n-gram in the pattern matches the NAS of the last n-gram in the substring.
(3) The 1-symbol algebraic signatures of pattern and substring match.

We further assume that the length of the pattern is greater than $2n$. As functions, NAS and PAS are linear over our Galois field $GF(2^f)$ and – a fortiori – over the trivial Galois field $\{0, 1\}$. We can therefore express the diagnosed matching of a substring to a given, fixed pattern P as an inhomogeneous system of linear equations in the bits of the substring as unknown. If $\boldsymbol{x}$ is the bit pattern written as a vector over $GF(2) = \{0, 1\}$ and $\boldsymbol{p}$ similarly a vector representing the pattern, then $\boldsymbol{x}$ constituting a diagnosed match is equivalent to the validity of

$$\boldsymbol{M} \cdot \boldsymbol{x} = \boldsymbol{M} \cdot \boldsymbol{p} \tag{9}$$

In this equation, the matrix $\boldsymbol{M}$ has the form

$$\boldsymbol{M} = \begin{pmatrix} \boldsymbol{C} & 0 & 0 \\ 0 & 0 & \boldsymbol{C} \\ 0 & \boldsymbol{A_1} & \boldsymbol{A_2} \end{pmatrix} \tag{10}$$

Here, $\boldsymbol{C}$ is a matrix with nf columns and mf rows that corresponds to taking the m-dimensional NAS (a string of size mf) of the n symbol (= nf) bits n-gram. Recall that we choose $m \geq n$. The algebraic properties of signatures imply that the rank of $\boldsymbol{C}$ is $\min(mf, nf)$. Similarly, matrix $(\boldsymbol{0}, \boldsymbol{A_1}, \boldsymbol{A_2})$ encapsulates the calculation of the algebraic signature between the two n-grams. (Implementations will differ in *how* we ascertain (3) without affecting our argument here.) Its rank is f. The event that a substring at a random offset within a random record consisting of random symbols is a diagnosed match for any fixed pattern is the event that a random vector $\boldsymbol{x}$ satisfies equation (9). The rank of $\boldsymbol{M}$ is $(2n+1)f$. The number of possible values for $\boldsymbol{M} \cdot \boldsymbol{x}$ is $2^{(2n+1)f}$. The probability that it is equal to the *given* value $\boldsymbol{M} \cdot \boldsymbol{p}$ is hence $2^{-(2n+1)f}$.

Example 3. *We continue with our previous example. Recall that we chose $n = 5$ for our 100 GB database with $f = 8$. Hence, the chance of a random diagnosed match is 2^{-88}. Even if we throw away all but the trailing 34 bits, we obtain a chance of a random diagnosed match of 2^{-76}. Since the database contains less than 2^{37} ($\approx 100G$) substrings the probability of a random diagnosed match is 2^{-51}.*

We can also estimate the chance of a false positive assuming that we are looking for an existing pattern. Assume that the pattern occurs only once. If the probability of a random diagnosed match is p, then the probability that a diagnosed match is a true positive is $1/(1 + p)$.

We now deal with the possibility of additional false diagnosed matches introduced by our storing only the offset of an n-gram modulo $2^f - 1$. If on average the record size is larger or equal to $K(2^f - 1)$ with integer K, then we have K more chances of a diagnosed match for test (1) and K more chances of a diagnosed match for test (2). Test (3) is not affected. We therefore have to multiply our probability with K^2 to obtain $p = K^2 2^{-(2n+1)f}$ as an upper bound of the probability of a random diagnosed match.

7 AS-index variants

While our basic scheme performs well, a skewed distribution of the n-grams leads to very large AS-index buckets or oppositely to very small ones, with an obvious impact on the search performances regard-

ing the search uses the former or the latter. On the other hand, the storage overhead of about 500% might be too high for some applications. We now describe two variants that address these two issues.

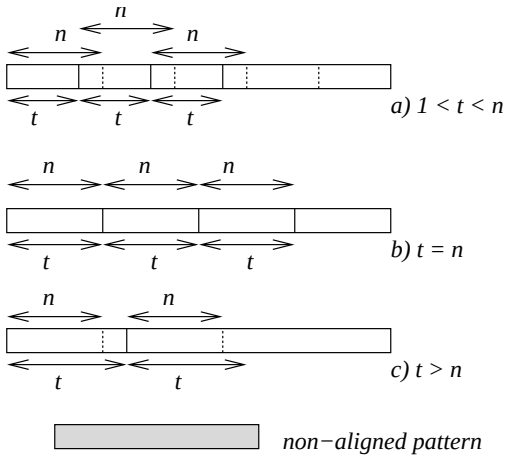

Fig. 6. Non-dense AS indexing: (a) overlapping n-grams, (b) tiling, (c) lossy

7.1 Skewed n-gram signature distribution

As previously observed, a skewed distribution can lead to many entries in a single AS-Index bucket. We now modify our search procedure as follows. In our pre-processing phase, we choose q n-grams, $q \geq 2$, among them the starting and final n-gram of the searched pattern. The simplest choice is to have the n-grams evenly distributed over the pattern.

The search first determines the shortest bucket length of all buckets indexed by any of the q n-grams. If any of these buckets is empty, then we are done: the pattern is not found in the database. If the shortest bucket happens to be the one indexed by the initial n-gram in the pattern, our processing remains the one of the basic scheme. If it is the one indexed by the final n-gram in the pattern, the calculation of the PAS still uses formula (7) unchanged because subtraction and addition are the same in the Galois field. Now c_1 is the CAS in the bucket indexed by the last n-gram and c_2 the one in the first bucket, while l does not change.

Otherwise, let a be the offset of the n-gram with minimal count of entries and S_a its NAS. For each entry in bucket $\boldsymbol{D}[h_L(S_a)]$, we search for a matching entry in the bucket of the first or of the last n-gram in the pattern, depending on which one has the smaller count. This amounts to matching the part of the pattern between the selected n-gram and either the beginning or the end of the pattern. If this succeeds, we continue to use our calculus to match the other part of the pattern. Since we eliminate most mismatches in the first step, AS-Index will now come more quickly to a decision on average.

7.2 Non dense indexing

Our next variant lowers the storage overhead by indexing only *some* instead of *all* n-grams. This results in significant gain for the storage, but implies higher search costs, a larger minimal size for the patterns that we can search for and a higher ratio of false-positives. Figure 6 shows the idea. Starting with the first n-gram in a record, we only index the n-grams that are $t > 1$ symbols apart, where parameter t is the *indexing rate*. We thus index only n-grams starting at the offsets $0, t, 2t, \cdots$. The size of the index is now reduced by a factor of about $1/t$.

We can distinguish three cases, *lossless* indexing if $n = t$, *lossy* indexing if $n < t$, and *overlapping* indexing if $n > t$ (Figure 6). In all cases, trailing characters of the pattern might not contribute to any indexed NAS.

The search procedure is the same in all three cases. It needs to be modified from the base procedure because the occurrence of a pattern in a string might not match the tiling of the records by the indexed n-grams. Assume that the pattern is $P = p_0 p_1, \ldots p_{K-1}$. We define substrings $P_i, i = 0, 1, \ldots t-1$ of the pattern as $P_i = p_i, p_{i+1}, \ldots p_{i+lt+n-1}$ with $l = \lfloor (K-n)/t \rfloor$. Thus, P_0 is the substring of the pattern that begins with the first n-gram in P, ends with the last n-gram in P starting at offset $lt-1$, and contains all symbols of P in between. P_1 starts at the second character and finishes with the last n-gram starting a multiple of t characters after the first one, etc. Our search procedure now first

tries to match P_0 in the database. More precisely, we process the bucket indexed by the first n-gram. For an occurrence, the last n-gram in the substring matching the sub-pattern is also in AS-Index and our matching will succeed. We then successively search for subpatterns P_1, P_2, ... P_{t-1}. This is guaranteed to find any occurrence of the pattern in the database. Since we actually match various subpatterns, a diagnosed match is not based on all symbols in the pattern and we might need to verify that the pattern actually occurred.

Consider the search for our pattern P = 'University Paris Dauphine', in Figure 6. Let $n = 4$ and $t = 4$ (lossless indexing). Suppose the use of a nondense tiling AS-Index (Figure 6.b). The search begins, as with the dense index, by attempting to match S_1 = 'Univ'. The last n-gram S_2 for the dense index would be S_2 = 'hine'. Now, S_2 = 'phin', in subpattern P_0. The matching n-gram 'hine' in the visited record cannot be indexed if 'Univ' is. Provided the match of P_0 succeeds, we read the record and attempt to match 'e' to the symbol following P_0 in the record (provided it exists). Next, we attempt matching with P_1, using thus $S1$ = 'nive' and S_2 = 'hine'. If this attempt succeeds, we attempt to mach 'U' as above. The matching attempts continue with P_2 having S_1 = 'ive' and S_2 = 'auph'. The completion requires finding 'Un' and 'ine' in the record before and after the P_2 match in the record. The final round attempts to match P_3 with S_1 = 'vers' and S_2 = 'uphi' followed by direct testing of 'Uni' and of 'ne'. The final result is the union of all the successful matches.

Compared with dense indexing, the storage space of this (tiling) AS-Index is reduced by factor of four, e.g., falls to (only) 125% of the database size. Likewise, this is at least four times less than any alternative method we discussed. In contrast, we need four times more disk accesses, i.e., 16 or 8 at best usually. Finally, the minimal indexed pattern is in turn $n = 8$ symbols, instead of five. Clearly, many applications may gladly accept the discussed trade-offs. In particular, since smaller AS-Index may then fit onto a flash disk. As this one is about ten times faster than a magnetic one, all together the AS-indexing gets actually about 2.5 times faster.

8 Experimental evaluation

We describe in this section our experimental setting and results. We implemented our AS structure as well as a String B-Tree [3] and an n-gram index, based on inverted lists [11]. The rationale for String B-Tree, further discussed in what follows, is that it appears attractive for disk-based use and is among most recent proposals.

8.1 Implementation of n-gram index

We build the n-gram index in two steps. First, we scan all record contents in order to extract all n-grams with their position. For each n-gram, we obtain a triple *<ngram, rid, offset>*, which we insert in a temporary file. The second step sorts all triples and creates lists of 8-bytes entries (4B for the rid, 4B for the offset). The final step builds the B+-tree which allows to quickly access to a list given an n-gram. Our simple construction in bulk is fast and creates a compact structure, illustrated on Figure 7.

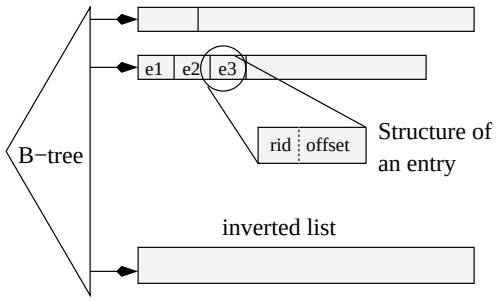

Fig. 7. Structure of the n-gram index

We search for a pattern P with n-gram index in the following manner. First, we tile P into n-grams $(g_1, g_2, \cdots, g_k)$. Let L_i denote the list that stores the positions of n-gram g_i. We then merge these lists, stopping whenever the k entries refer to the same record. We then compare the offsets for these records. A match is found if for a tuple of offsets $(l_1, l_2, \cdots, l_k)$ the relationships $l_1 = l_2 - n = l_3 - 2n = \cdots = l_k - (k-1)n$ hold. The merge continues until the smallest list has been completely examined.

Assume we are searching the pattern 'University Paris Dauphine', with $n = 4$. The tiling n-grams are {'Univ', 'ersi', 'ty P', 'aris', ' Dau', 'phin', 'hine'}. Note that the

last n-gram partially covers the previous one. This is necessary to ensure a full representation of the pattern content with fixed-size n-grams. The merge combines seven lists, one for each 4-gram. The search cost with the n-gram index is linear, both in the size of the database and in the size of the pattern.

8.2 Implementation of String B-tree

We implemented a String B-Tree structure [3], making our best efforts to minimize the storage overhead. Each node contains a compact representation of a Patricia Trie [12] as a list of *entries* $< e_0, e_1, \cdots, e_n >$. Each entry e_i refers to a string s_i (suffix in the String B-Tree terminology) in the database, and is represented by a triple $< lcp_i, lnc_i, addr_i >$. Here, lcp_i is the length of the longest common prefix between s_i and s_{i-1} (2 bytes), lnc_i is $s_i[lcp_i + 1]$ (1 byte) and $addr_i$ is the disk address of s_i (8 bytes). The size of a leaf entry is 11 bytes. The size of an internal node entry, which also stores the sub-node address, is 19 bytes.

We construct a String B-Tree statically by first sorting the list of suffixes in lexicographic order. From this sorted list, the B-Tree is built bottom-up. In our implementation a disk block occupies 4K, and we can store at most 300 entries in each leaf.

We search for a pattern P with a standard top-down traversal of the String B-Tree. At each node N, the search procedure finds an entry e_i. The string s_i associated to e_i is then loaded from the database, and the relevant child of N is determined (see [12] for details.) Unlike traditional B-Trees, two (2) look-ups are necessary at each level: one for the B-Tree node, and one for loading s_i.

8.3 Implementation of the AS-Index

Our implementation of AS-Index is static as well. First, the n-grams are collected. For each n-gram at offset l in a record R the hash key $k = h_L(NAS_m(R, l))$ and the CAS $c = CAS_1(R, l)$ are computed. The quadruplet $< k, c, rid, l >$ is inserted in a temporary file. Second, the temporary file is sorted on the hash key k. This groups together entries which must be inserted into the same bucket. An entry consists of a CAS (1 byte), a record id (4 bytes) and an offset (4 bytes). Using the CAS as a secondary sort key, one places the entries into the required order for insertion into the bucket.

This implements the basic AS-Index structure faithfully according to our description in Section 4. In order to speed up the in-memory join of the two buckets, our implementation proceeds as follows. Let i be the index of the smallest bucket, and i' the index of the greatest one. We first load and scan the bucket $D[i']$ and create an index of the CAS present in the bucket. The index is a simple array of size 2^f (at most) that gives, for each possible CAS value c, the offset in $D[i']$ of the sequence of entries with c (the offset is -1 if c does not appear). The array allows us to access the entries that correspond to a given c in constant time. The search algorithm checks the size of the targeted buckets, and chooses the smallest one for driving the search.

The search then proceeds by loading and scanning the two buckets $\boldsymbol{D}[i]$ and $\boldsymbol{D}[i']$. For each entry $< r, l, c_1 >$, we compute the CAS c_2 at position $l + |P| - n$ using equation (7), look in $D[i']$ for entries with CAS c_2, using the array on CASs, as explained above, and search for a matching entry.

8.4 Data compression

For AS-Index and n-gram index, the entries which are in the memory buckets are compressed before being written to the disk. We use a variable length encoding for each entry, which are unsigned integers (4-byte representation), into a succession of bytes in a binary memory buffer that is further dumped to the disk. For each byte of the encoding, we used 7 bits of data and 1 bit of overhead. This overhead is 0 for all bytes of the encoded integer, except for the final byte, for which it is set to 1.

Example 4. *Consider for instance a key with value* 2509. *Stored as a long integer, it requires 4 bytes:* 00000000 00000000 000001001 11001101. *Using variable length encoding we can store this key using only two bytes:* **0**0010011 **1**1001101. *Observe that the first bit of the first byte, in bold, is set to* 0 *which means that the following byte must be considered for the current key-value, while the first bit of the second byte is set to* 1 *which means that the key-value ends with this byte.*

Moreover, when possible, we reduce the storage space by only encoding deltas instead of the whole in-

teger, since our entries are kept ordered by document id and by entry offset. More precisely, for each entry in as-index we store its cas (which is one byte-long and cannot be further reduced), and information on its document id and offset. Both id and offset benefit from delta encoding. This data compression achieves a 50% space saving for a negligible decoding time extra-cost ($\sim 1\ ms$) during search execution.

8.5 Settings

We use four types of datasets with quite distinct characteristics: `alpha`, `dna`, `text` and `wikipedia`. The `alpha(`Σ`)` type consists of synthetic ASCII records, with uniform distribution, ranging over an alphabet Σ which is a subset of the extended ASCII characters. We consider two alphabets: Σ_{26}, with only 26 characters, and Σ_{256} with all the 256 symbols that can be encoded with $f = 8$ bits. We call the resulting datasets `alpha(26)` and `alpha(256)`. For these types, we composed datasets ranging from 20MB to 20GB, each one compound by 20 files. They allow us to compare the behavior of our structure to the theoretical analysis in Section 6.

The second type, `dna`, consists of real DNA records extracted from the UCSC database. This dataset is composed by 97 files whose size covers a large range, from dozens of KB to hundreds of MB, or even for 2 files several GB. Its total size is 9.9GB. The type `text` consists of real text records created from ASCII files of large English books extracted from the Gutenberg digital library. The typical size of a text record is 0.5-2 MB, and size distribution is almost uniform and within that range. We created a 16.6GB database of 29,539 `text` files. Finally the last type, `wikipedia`, consists of a dump of free encyclopedia Wikipedia. This dump appends all XML records from Wikipedia into a single XML record whose size is 24GB. To limit collisions for the n-gram signature due to the alphabet-size, we set the n-gram size to 8 for DNA files, 6 for `wikipedia` and to 4 for the other datasets.

File size	Dir. (MB)	AS-index size(MB) (ρ)	n-gram index size(MB) (ρ)	Str. B-Tree size(MB) (ρ)
20 MB	63.6	163 (8.14)	143 (7.14)	153 (7.64)
200 MB	64.0	1,092 (5.46)	892 (4.46)	1,532 (7.66)
2 GB	64.0	9,582 (4.79)	7,540 (3.77)	15,264 (7.63)
20 GB	64.0	94,535 (4.73)	74,034 (3.70)	153,009 (7.65)

Table 5. Index sizes (and ratio) for `alpha(256)` files

File size	Dir. (MB)	AS-index size(MB) (ρ)	n-gram index size(MB) (ρ)	Str. B-Tree size(MB) (ρ)
20 MB	7.0	100 (5.00)	80 (4.00)	148 (7.37)
200 MB	7.0	818 (4.09)	618 (3.09)	1,473 (7.37)
2 GB	7.0	8,170 (4.08)	6,120 (3.06)	14,761 (7.38)
20 GB	7.0	81,503 (4.07)	61,003 (3.05)	147,334 (7.37)

Table 6. Index sizes (and ratio) for `alpha(26)` files

Dataset	File (GB)	Dir. (MB)	AS-index size(MB) (ρ)	n-gram index size(MB) (ρ)	Str. B-Tree size(MB) (ρ)
`adn`	9.9	64.0	35,857 (3.62)	25,814 (2.61)	73,022 (7.37)
`text`	16.6	36.7	50,028 (2.94)	33,008 (1.94)	139,938 (8.43)
`wiki`	24	64.0	83,298 (3.47)	54,495 (2.27)	204,482 (8.52)

Table 7. Index sizes in MB for DNA, text and wikipedia datasets

8.6 Space occupancy and build time

The size of the AS-index is the sum of the size of cells and the directory which stores the number of entries for each bucket. The size of the n-gram index is the sum of the size of inverted lists and the directory. The size of the String B-Tree is the size of the B+tree where each node is a serialized Patricia Trie. Tables 5, 6, and 7 give, for the three indexes, respectively the index sizes (and its ratio w.r.t. dataset size) for `alpha(256)`, `alpha(26)` and our three real dataset. We also report the size of the directory.

The storage efficiency of these indexes heavily depends on the distribution of n-grams. If the distribution is uniform, the number of n-gram values can become very high, and this severely impacts the storage efficiency. The `alpha(256)` dataset shows this behav-

http://hgdownload.cse.ucsc.edu/
http://www.gutenberg.org/
http://dumps.wikimedia.org/

ior (Table 5). With $n = 4$, we obtain $256^4 = 4.3\ 10^9$ distinct n-grams. For the 20MB and 200MB datasets, each list in the n-gram index consists of only one entry for a given file (i.e., it is very unlikely to find twice a same n-gram in the same file), which explains the important size of the indexes, 7.14 and 4.46 times the dataset size respectively. When the size of the (uniformly generated) dataset reaches the possible number of distinct n-grams, all lists contain at least an entry for a given file and the growth of the index becomes linear in the size of the dataset (see the index whose size is 3.7 times the size of the dataset for both 2GB and 20GB datasets). The same remark holds for the AS-index. We observe that the AS-index requires around 25% additional space than n-gram index. This corresponds to the extra-information needed for the structure, namely the CAS, that is not affected by compression (so one additional byte for GF^8). A naïve implementation (without clustering entries from a list with the same document id and without compression) would exhibit an n-gram (resp. as) index 8 (resp. 9) times larger than the dataset. Finally we observe a constant size for the directory. Indeed the number of lists is bound by the number of possible n-grams, the number of m-symbol NAS possible and the L factor (see Section 4). Here m is set to 3 which means 3 signatures of one byte concatenated, *i.e.* 2^{24} possible index lists. Moreover we have $256^4 = 4.3\ 10^9$ distinct possible n-grams and L is set to 2^{22}. So this latter limits the number of lists. Since the hash directory needs for each list 16 bytes to store a pointer and the list size (required for tha data management on disk), the maximal directory size is $2^{22} \times 16 = 64MB$. Thus, as expected for our different dataset and uniform distribution, all lists are non-empty and we observe a 64 MB directory. The String B-Tree requires more space (regarding the dataset between 7.3 or 8.5 times the data indexed) with 11B entries in the leaves and 19B entries for internal nodes. Moreover String B-Tree can not benefit from compression techniques which leads to the more important storage overheads.

We continue to consider a uniform distribution of characters, but now use a smaller alphabet Σ_{26} (Table 6). The number of possible 4-grams decreases to $26^4 = 456,976$. m-symbol NAS size and L are unchanged. Consequently the number of lists is now bound by the number of n-grams possible which leads to a $456,976 \times 16 = 7MB$ directory. The size for both index also decreases. The reason is that in all lists we find for all documents several entries, due to the limited number of possible n-grams. As a consequence the compression technique is particularly efficient, especially for n-gram index.

File	Avg.	Min	Max	Std. dev.
`dna`	2,508	1	242,951,376	2,375.6
`text`	7,422	1	712,735,151	214,988.3
`wiki`	6,050	22	303,119,079	5,556.5
alpha(26)	388	276	463	52.8
alpha(256)	5,111	4,497	5,717	115.0

Table 8. Distribution of entries for the real and synthetic datasets

For real datasets, either `dna`, `text` or `wikipedia`, the distribution is far from being uniform. Table 8 shows the distribution of the number of entries from these real databases. The average number of entries is 2,508 for the DNA database, 7,422 and 6,050 for respectively the `text` and `wikipedia` database, with an important variance. In the worst case (`text` files), the largest list has 712,735,151 entries. This fully justify our choice of storing the number of entries in the directory, and of using this information to scan the smallest list during a search operation.

However, the indexes for the three real datasets exhibit some differences. For `adn`, the size of indexes is smaller than the one for uniform datasets: *e.g.* 3.62 for AS-index versus 4.73 (resp. 4.07) for `alpha(256)` (resp. `alpha(26)`). Here the compression technique fully benefits from the number of distinct n-grams, since for our 8-grams only $4^8 = 65,536$ values are possible. Consequently lists are larger, with several occurrences inside each file leading to a better compression. For `text` dataset, there exist potentially $256^4 = 2^{32}$ distinct 4-grams. In fact all the `ASCII` symbols are not used within this collection and the number of distinct symbols is close to 128, so 2^{28} possible n-grams. However, most of these n-grams do not correspond to an existing n-gram in the language of choice (*e.g.* `qmgw`, `uaio`, etc). Consequently, the real number of buckets with entries is not that large. More-

over, Table 8 shows that there is a large discrepancy in the bucket size due to the well-known *Zipf*-distribution of words (and character-sequences) in common languages [13]. So, the number of occurrences for frequent words is several orders of magniture larger than other words. These two phenomena result in a high compression rate of these large lists and the an n-gram (resp. AS) index size only twice (resp. three times) the size of the dataset. Finally the XML syntax of the `wikipedia` dataset (with tags, parameter names and values, etc) and the presence of dozens of hundreds of author's name allow more possible n-gram than `text`. This explains poorer compression ratios.

Table 9 exhibits the impact of different parameters, namely n-gram size, Galois Field size and L value, on the AS-index size for the wikipedia dataset. Using GF^{16} produces sensibly larger index: 96.6GB (resp. 104.8GB) versus 69.8GB (resp. 83.3GB) for GF^8 and 4-grams (resp. 6-grams). The difference is mainly due to the CAS stored for each entry that requires now 2 bytes instead of 1. Since our dataset is 24GB large, and the CAS stored in each entry is not subject to any compression mechanism, the index size is expected to increase by 24GB (one byte per entry). The index size is however not exactly enlarged by 24GB (+27GB for 4-gram and +22GB for 6-gram) since modifying signatures also impact entries distribution and consequently compression.

n-gram	GF size	Hash	Dir.	AS-index (ρ)	
4	GF^8	2^{22}	42.9	69,850.6	(2.90)
4	GF^{16}	2^{22}	42.9	96,610.2	(4.03)
6	GF^8	2^{22}	64.0	83,298.3	(3.47)
6	GF^{16}	2^{22}	64.0	104,808.5	(4.36)
6	GF^8	2^{24}	256.0	84,150.4	(3.51)

Table 9. AS-index sizes in MB for wikipedia datasets and different settings

For a same GF size, both directory and index size increase with the size of the n-gram: *e.g.* for GF^8 the directory size is $42.9MB$ for 4-grams and $64MB$ for 6-grams, and meanwhile the index size increases from $69.8GB$ to $83.3GB$. Same result holds for GF^{16}. Indeed 6-grams provide more combinations (up to 256^6) than 4-grams, so more distinct signatures. With 6-grams the directory reaches $64MB$, which is the maximal size for the directory ($L = 2^{22}$, thus maximal size is $2^{22} \times 16B = 64MB$). As a consequence, lists are larger with 4-grams than with 6-grams. This allows better compression rate and a smaller size for the index. Finally we see that the value of the L parameter only impacts the directory size and not the index size. With $L = 2^{24}$ the maximal size for the directory, when all buckets are not empty, is $2^{24} \times 16B = 256MB$.

	AS-index		n-gram index		Str. B-Tree	
	time	speed	time	speed	time	speed
File size	s	KB/s	s	KB/s	s	KB/s
20 MB	31	661	31	661	41	484
200 MB	178	1,150	170	1,205	341	587
2 GB	1,572	1,334	1,681	1,248	3,817	524
20 GB	27,721	756	17,022	1,232	36,832	543

Table 10. Building time in ms for `alpha(26)` files

The building time varies with the size of the index but remains in a range of $600KB - 1,400KB$ per second. Our structures are built in bulk after sorting all n-gram entries in a temporary file. This leads to comparable performances. On our machine, the building time for a 20 GB file is about 28,000 s, and the bulding rate is about 756 KB/s. Index construction also includes compression computation both for temporary files, and then for the final index. A comparison of dynamic builds remains for future work. For AS-Index, this would reduce mostly to the standard technique of maintaining a dynamic hash file. The String B-Tree exhibits the highest building times, due to a larger storage overhead (so more time to write the index on disk). However, its indexing speed in KB/s is higher since there is no compression computation.

8.7 Search time

For search experiments, we extracted the patterns from the files to guarantee that at least one result is found. Pattern sizes range from 25 symbols to 200 symbols. To avoid initialization costs and side effects such as CPU or memory contention from other OS processes, we performed each search repeatedly until the search times stabilized. We report the average search time over a run of 500 search operations.

We report the results on search time, in ms, in Tables 11, 12, 13, 14 and 15 (with 20GB for the two last datasets). As expected, the String B-Tree and the AS-Index behavior is constant regardless of the length of the pattern, while n-gram index performance degrades linearly with this length.

K	25	50	75	100	200
AS	103	108	96	101	96
ngram	190	298	386	466	774
Spd-up	1.84	2.76	4.02	4.61	8.04
Str BT	159	170	143	153	147
Spd-up	1.54	1.57	1.49	1.51	1.53

Table 11. Search time in ms for `dna` files

K	25	50	75	100	200
AS	667	659	677	643	652
ngram	2,293	3,876	5,973	7,507	16,102
Spd-up	3.44	5.88	8.82	11.67	24.70
Str BT	974	942	995	971	958
Spd-up	1.46	1.43	1.47	1.51	1.47

Table 12. Search time for `text` files

K	25	50	75	100	200
AS	175	188	172	188	181
ngram	716	1,383	2,639	3,480	5,930
Spd-up	4.09	7.36	15.34	18.51	32.76
Str BT	291	320	296	314	304
Spd-up	1.66	1.70	1.72	1.67	1.68

Table 13. Search time for `wikipedia` files

K	25	50	75	100	200
AS	66	68	68	68	68
ngram	186	314	467	615	1,195
Spd-up	2.82	4.62	6.87	9.04	17,57
Str BT	150	147	147	152	151
Spd-up	2.27	2.16	2.16	2.23	2.22

Table 14. Search time for `alpha(26)` files

K	25	50	75	100	200
AS	36	40	38	38	37
ngram	97	163	202	259	505
Spd-up	2.69	4.08	5.32	6.82	13.65
Str BT	88	91	90	89	92
Spd-up	2.44	2.28	2.37	2.34	2.49

Table 15. Search time for `alpha(256)` files

For our 20GB files, the height of the String B-Tree is 5 independently of the alphabet size, for 20.10^9 indexed substrings (recall that the fanout is 300, and that our bulk insertion creates full nodes). The root is always in the cache, as well as a significant part of the level below the root, depending on the indexed file size. The String B-Tree traversal is generally reduced to $(5-2) = 3$ disk accesses for loading a leaf node. In addition, each lookup in a node requires an additional random disk access to the database in order to fetch the full string. This leads to a final cost of 8-9 physical disk accesses. The search time with String B-Tree is independent of the size of the pattern and of the size of the alphabet. Searching with the AS-Index takes about 38 ms for `alpha(256)`, 68 ms for `alpha(26)` files, 100 ms for `dna` files, 660 ms for `text` files and 180 for `wikipedia`(real data). This is consistent with the analytical cost discussed in Section 6. The `alpha` datasets are uniformly generated, and this results in an almost constant number of entries per bucket. Accordingly, the search is done in few operations. The difference between `alpha(256)` and `alpha(26)` is explained by the size of the buckets which are larger for `alpha(26)` since the n-gram

values range over the set of 26^4 possibilities compared to 256^4 for `alpha(256)`.

For real data (`dna`, `text` and `wikipedia`), buckets are likely to be larger, either because the alphabet is so small that the set of existing n-gram values is bounded and cannot fully benefit from the hash function (see Subsection 6.1 for a discussion), or because of non uniformity. The former case corresponds to the DNA, the latter to our real `text` and `wikipedia` files. Table 8 shows that, on average, the number of entries in a bucket is larger for `dna` (2,508 entries) compare to `alpha(256)` (5 entries on average). The cost of DNA search is accordingly higher ($\approx$ 100 ms, against $\approx$ 38 ms). The impact of the standard deviation also explains the higher searching costs for `text` and `wikipedia` files. Indeed for these datasets the deviation is over 200,000 so two orders of magnitude higher than for `dna` and three orders than for `alpha(26)` and `alpha(256)`. Recall however that our algorithm chooses the smallest bucket for driving the search, which limits the impact of skewed datasets and the variance of search times. This explains theat the searching time for `text` and `wikipedia` files is one order greater than `alpha(256)` while the alphabet size is 256 for all these datasets. Finally searching in `wikipedia` is faster than in `text` due to the existence of larger buckets which leads to longer searches. The ratio of search times, giving the speed-up of AS-Index over respectively the n-gram index and the String B-Tree also reported in Tables 11 to 15 summarizes the benefit of our proposal.

File size	AS	n-gram	(Spd-up)	Str. BT	(Spd-up)
20 MB	24	56	(2.33)	41	(1.71)
200 MB	31	72	(2.32)	62	(2.00)
2 GB	34	87	(2.56)	67	(1.97)
20 GB	36	97	(2.69)	88	(2.44)

Table 16. Index comparison for searching in varying file size for `alpha(256)`

Table 16 shows that the search time with the String B-Tree increases with the size of the file for `alpha(256)`. The theoretical logarithmic behavior of the String B-Tree is almost blurred here, because of the large node fanout (300). For the 20 MB file, its height is only 3, while it reaches 4 for the 200 MB and 2 GB files, and even 5 for the 20 GB file. An increase by one of the height corresponds to two (2) additional disk accesses on average for a search. Even for a constant tree height of 4, the String B-Tree performs a search faster with the smallest file (62 for the 200 MB file versus 67 for the 2 GB file). Indeed the number of nodes increases with the file size. This accounts for a lower probability of a buffer hit during tree traversal and explains the search time increase w.r.t. the file size.

The search time evolves (sub)linearly for n-gram index, both in the size of the pattern, and in the size of the database. This is explained by the necessity to scan a number of inverted lists which is proportional to the size of the pattern. In addition, larger files imply larger lists, hence the behavior illustrated by Table 16. However the cost remains sublinear (the cost for 20 GB is only 3 times higher than the cost for 20 MB). This is due to (i) the merge process which stops when the smallest list has been fully scanned, thereby avoiding a complete access to all lists, and (ii) the data compression that is more efficient for large datasets since more entries with the same signature are found within a given document.

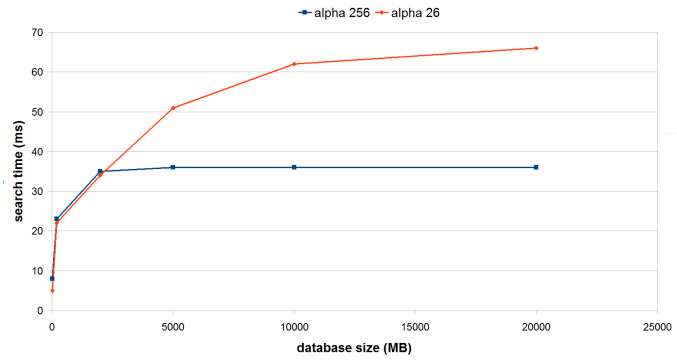

Fig. 8. Search time with AS-index for varying file size

Oppositely, AS-Index exhibits an almost constant behavior (appr. 25-35 ms), even when searching in large files (20 GB). This observation is confirmed by Figure 8 which shows the evolution of search times as the size of the database increases from 20 MB to 20 GB. Each curve represents the results for a given alphabet, with patterns consisting of 25 symbols. Search time becomes constant for large databases. Indeed a search corresponds to two disk accesses plus the signatures computation and comparisons for all entries. The

difference between `alpha(256)` and `alpha(26)` is explained by the size of the buckets due to different n-gram values ranges (26^4 possibilities for `alpha(26)` vs 256^4 for `alpha(256)`, so potentially 8^4 more entries for `alpha(26)`, leading to potentially $(8^4)^2$ times more comparisons). Small databases benefits from caching and a reduced number of computations and comparisons.

8.8 Non-dense indexing and false-positives

We study in this Section the non-dense index proposed in Section 7. Our experiments reveal, as expected, that the gain is significant for the space occupancy. Tables 17, 18, 19, 20, 21 report a gain for space occupency around 48% when saving one n-gram on two whatever the alphabet is. When skipping 2 or 3 symbols the memory gain is respectively around 64% and 73%. Oppositely the matching time increases with the size of the skip for most of the alphabets. There are 2 phenomena in competition: for t symbols skipped when indexing, we perform $2t$ searches when matching instead of 2 for the basic version (*i.e.* $t = 1$); however as observed the size of the index, and consequently of the buckets, also decreases with t which reduces the processing time since less combinations have to be tested. Thus for synthetic datasets (Table 17 and 18) the matching time increases but for $t = 2$ it is not twice the time required for the basic version of the AS-index. Same observation holds for real datasets. For `wikipedia` files (Table 21), the matching time overhead diminishes with i from $t = i$ to $t = i+1$. For `adn` dataset (Table 19), the matching time even decreases from $t = 2$ to $t = 3$ and to $t = 4$, since the index could almost be totally cached.

Index	size	matching time	Spd-up	f/p
ngram	59.5 GB	181 ms	-	-
AS t=1	79.5 GB	66 ms	2.74	0.1%
AS t=2	40.9 GB	68 ms	2.68	0.2%
AS t=3	28.1 GB	75 ms	2.41	0.4%
AS t=4	21.6 GB	84 ms	2.15	0.5%

Table 17. Non-dense index characteristics for `alpha(26)` files

Index	size	matchig time	Spd-up	f/p
ngram	72.2 GB	97 ms	-	-
AS t=1	92.2 GB	36 ms	2.69	0.0%
AS t=2	48.0 GB	48 ms	2.02	0.0%
AS t=3	32.5 GB	64 ms	1.52	0.0%
AS t=4	24.6 GB	65 ms	1.49	0.0%

Table 18. Non-dense index characteristics for `alpha(256)` files

Index	size	matchig time	Spd-up	f/p
ngram	25.2 GB	190 ms	-	-
AS t=1	35.0 GB	103 ms	1.84	0.2%
AS t=2	18.3 GB	145 ms	1.31	14.8%
AS t=3	12.4 GB	127 ms	1.50	25.5%
AS t=4	9.4 GB	119 ms	1.60	43.5%

Table 19. Non-dense index characteristics for `adn` files

Index	size	matchig time	Spd-up	f/p
ngram	25.2 GB	2,293 ms	-	-
AS t=1	35.0 GB	667 ms	3.44	0.2%
AS t=2	18.3 GB	838 ms	2.74	9.6%
AS t=3	12.4 GB	873 ms	2.63	21.2%
AS t=4	9.4 GB	949 ms	2.42	29.9%

Table 20. Non-dense index characteristics for `text` files

Index	size	matchig time	Spd-up	f/p
ngram	53.2 GB	716 ms	-	-
AS t=1	81.3 GB	175 ms	4.09	4.1%
AS t=2	43.0 GB	329 ms	2.18	20.7%
AS t=3	29.6 GB	390 ms	1.86	33.8%
AS t=4	22.7 GB	442 ms	1.62	42.6%

Table 21. Non-dense index characteristics for `wikipedia` files

Non-dense indexing offers consequently an important space saving for a moderate increase of the matching time. However it is expected to produce more false-positives. For synthetic datasets with a uniform distribution, the false-postives are uncommon: less than 0.05% for `alpha(256)` and less than 0.5% for `alpha(26)`. For real datasets the number of false-positives is limited for the basic AS-index version for (*i.e.*, $t = 2$): 0.2% for `adn` and `text` files. `wikipedia` dataset provides more false-positives, due to the tag-nature of its content. Indeed when searching a pattern that starts or ends with a tag, the bucket that corresponds to the n-gram in this tag is very large and does not filter out as expected. These results are coherent with our analysis in Section 6.4. Non-dense indexing lead to a significant number of false-positives: around 15% for $t = 2$ and more than 40% for $t = 4$ (see Table 19, Table 20 and Table 21). Indeed a higher t value leads to a higher number of matching attempts, so a more important probability to retrieve false-positives. For synthetic datasets (see Table 17 and Table 18) the uniform distribution guarantees a low false-positives rate even for the tiling indexing (here for 4-grams it means $t = 4$). For real data, among t consecutive n-grams we have a non-negligible probability which increases with t, to have a frequent n-gram, so a higher probability of retrieving false-positives. For `text` the probability to retrieve a large bucket is lower since there exists less large buckets with this distribution (but existing ones are larger than with other distributions). Consequently this dataset produces less false-positives than other real datasets on average.

9 Related work

Finding patterns in a large database of sets is a fundamental problem in Computer Science and its applications such as bioinformatics. [14] presents a comparison of tree-based and hash-based solutions for n-gram indexing. The theoretically best algorithms and data structures allow linear construction of the index in the database, have low storage overhead, and allow searches that are processed in time linear on the size of the pattern. Among the many algorithms, those based on suffix trees [15] have received much attention. Recent work by Kurtz [16], Tata, Hankins, and Patel [17] among others tries to make the theoretically optimal behavior of suffix trees practical. A great part of the problem is caused by the blow-up of the index size over the database size, typically ten to twenty times [16]. Related data structures such as Manber's suffix arrays [18], Kärkkäinen's suffix cacti [19], or Anderson and Nilsson [20] suffix tries lower storage overhead at the prize of an increase in search time. Dementiev, Kärkkäinen, Mehnert, and Sanders [21] give methods to make suffix arrays effective and efficient for truly large files . A survey of full-text substring indexes in external memory is presented in [22].

Suffix arrays and suffix trees are static indexes, designed to index a single file content. If we create such an index for every record, then search times will depend on the size of the database. If we however create the index for a collection of records – as we obviously should – then deleting and inserting records becomes very difficult, and it is unclear how we can adapt the binary search of suffix arrays to the indirection mechanism used by the storage engine. Life is much simpler if the database consists of words and we restrict ourselves to word indexes that can be stored much more compactly [11].

Signatures files were proposed in [23] and shown to be inferior to inverted indexing in [24]. Some other attempts for indexing sequences are the ed-tree [25] for DNA files, and the q-gram index [26]. Both focus on the specific problem of *homology search* in genomic databases.

Our method is predominantly based on previous work on n-gram based inverted file indexing. The technique has been advocated for string search in larger, hence naturally disk based, partly or totally unstructured files or databases (full-text, hypertext, protein, DNA). In bioinformatics, CAFE prototype uses $n = 3$ for protein and $n = 9$ for DNA string search, and is reported several times faster than previous systems [27]. All these systems used the basic n-gram index for many GB disk-resident datasets.

The latest attempt of using n-grams for a large, (hence diskbased) database, are reported in [6, 28]. In [6], like us, it improves storage overhead and, especially, search time, over the basic n-gram scheme. The n-Gram/2L uses a "normalized" representation with two indexes: (i) one n-gram index on the subsequences of size m indexing the n-grams found in each subsequences, and (ii) one n- gram-index indexing the

subsequences found in the files. The two indexes are smaller than the original index and though a search needs to use both indexes, it can use less look-up. If AS-Index saved storage for larger alphabets it appears to be slightly less efficient for small ones compared to n-Gram/2L However like n-gram index this structure offers a search proportional to the database size and to the query size oppositely to our constant time claim. [28] presents a system named Maguro for an efficient search in very large (Web) collection of texts. It indexes any atoms (n-grams, words or tuples) through a distributed structure. Moreover it exploits the long tail distribution of the atoms in Web document thanks to a two-level hash-structure: popular atoms being stored in DRAM while less popular are stored on HDD. Observe that our AS-Index could also benefit from this two-level hash-structure (and conversly AS-index could be used to improve Maguro's performance).

10 Conclusion and Future work

We present a novel approach to string search in databases, based on Algebraic Signatures and algebraic computations. The contribution of our paper is a simple and fast search algorithm which finds a pattern of arbitrary length in a database of arbitrary size in constant time. We showed through analysis and experiments that our technique outperforms other disk-based approaches. To our knowledge, our work constitutes an original approach to indexing, which takes advantage of the interpretation of character as symbols in a mathematical structure to develop new computational techniques.

Scalable and distributed AS-Index constitute a promising research directions that we plan to investigate.

Acknowledgments. This work has been partially funded by the Advanced European Research Council grant Webdam.

References

[1] Margaritis G, Anastasiadis S V. SeFS: Unleashing the Power of Full-text Search on File Systems. In *Usenix Conf. on File and Storage Technology*, 2007, pp. 12–12.

[2] Crochemore M, Lecroq M. *Pattern Matching and Text Compression Algorithms*. CRC Press Inc, 2004.

[3] Ferragina P, Grossi R. The String B-tree: A New Data Structure for String Search in External Memory and Its Applications. *J. ACM*, 1999, 46(2):236–280.

[4] Phoophakdee B, Zaki M J. Genome-scale disk-based suffix tree indexing. In *Proc. Intl. Conf. on Management of Data (SIGMOD)*, 2007, pp. 833–844.

[5] Miller E, Shen D, Liu J, Nicholas C. Performance and Scalability of a Large-Scale n-gram Based Information Retrieval System. *Journal of Digital Information*, 2000, 1(5).

[6] Kim M S, Whang K, Lee J G, Lee M J. n-Gram/2L: A Space and Time Efficient Two-level n-Gram Inverted Index Structure. In *Proc. Intl. Conf. on Very Large Databases (VLDB)*, 2005, pp. 325–336.

[7] Litwin W, Schwarz T. Algebraic Signatures for Scalable Distributed Data Structures. In *Proc. Intl. Conf. on Data Engineering (ICDE)*, 2004, pp. 412–423.

[8] Mouza C, Litwin W, Rigaux P, Schwarz T J E. AS-Index: a Structure for String Search using n-Grams and Algebraic Signatures. In *Proc. Intl. Conf. on Information and Knowledge Management (CIKM)*, 2009, pp. 295–304.

[9] Gray J, Fitzgerald B. Flash Disk Opportunity for Server Applications. *ACM Queue*, 2008, 6(4):18–23.

[10] Charras C, Lecroq T, Pehoushek J D. A Very Fast String Matching Algorithm for Small Alphabets and Long Patterns. In *Proc. Intl. Symp. on Combinatorial Pattern Matching (CPM)*, 1998, pp. 55–64.

[11] Witten I, Moffat A, Bell T. *Managing Gigabytes: Compressing and Indexing Documents and Images*. Morgan-Kaufmann, 1999.

[12] Na J C, Park K. Simple Implementation of String B-tree. In *Proc. String Processing and Information Retrieval (SPIRE)*, 2004, pp. 214–215.

[13] Baeza-Yates R, Ribeiro-Neto B, editors. *Modern Information Retrieval*. Addison-Wesley, 1999.

[14] Robenek D, Platos J, Snásel V. Efficient Inmemory Data Structures for n-grams Indexing. In *Proc. Intl Work. on DAtabases, TExts, Specifications and Objects (DATESO)*, 2013, pp. 48–58.

[15] Gusfield D. *Algorithms on Strings, Trees, and Sequences: Computer Science and Computational Biology*. Cambridge University Press, 1997.

[16] Kurtz S. Reducing the Space Requirement of Suffix Trees. *Software - Practice and Experience*, 1999, 29(13):1149–1171.

[17] Tata S, Hankins R, Patel J. Practical Suffix Tree Construction. In *Proc. Intl. Conf. on Very Large Databases (VLDB)*, 2004, pp. 36–48.

[18] Manber U, Myers E W. Sufix Arrays: A New Method for On-Line String Searches. *SIAM Journal on Computing*, 1993, 22(5):935–948.

[19] Kärkkäinen J. Suffix Cactus: A Cross between Suffix Treee and Suffix Array. In *Proc. Intl.Symp. on Combinatorial Pattern Matching (CPM)*, 1995, pp. 191–204.

[20] A Andersson S N. Efficient Implementation of Suffix Trees. *Software– Practice and Experience*, 1995, 25(2):129–141.

[21] Dementiev R, Kärkkäinen J, Mehnert J, Sanders P. Better External Memory Suffix Array Construction. *ACM Journal of Experimental Algorithmics*, 2008, 12:1–24.

[22] Barsky M, Stege U, Thomo A. *Full-Text (Substring) Indexes in External Memory*. Morgan & Claypool Publishers, 2011.

[23] Faloutsos C. Signature Files. In *Information Retrieval: Data Structures & Algorithms*, pp. 44–65. 1992.

[24] Zobel J, Moffat A, Ramamohanarao K. Inverted Files Versus Signature Files for Text Indexing. *ACM Trans. on Database Systems (TODS)*, 1998, 23(4):453–490.

[25] Tan Z, Cao X, Ooi B C, Tung A K H. The ed-Tree: An Index for Large DNA Sequence Databases. In *Proc. Intl. Conf. on Scientific and Statistical Databases (SSDBM)*, 2003, pp. 151–160.

[26] Cao X, Li S C, Tung A K H. Indexing dna sequences using q-grams. In *Proc. Database Systems for Advanced Applications (DASFAA)*, 2005, pp. 4–16.

[27] Williams H, Zobel J. Indexing and Retrieval for Genomic Databases. *IEEE Transactions on Knowledge and Data Engineering (TKDE)*, 2002, 14(1):63–78.

[28] Risvik K M, Chilimbi T, Tan H, Kalyanaraman K, Anderson C. Maguro, a System for Indexing and Searching over Very Large Text Collections. In *Proc. ACM Intl Conf. on Web Search and Data Mining (WSDM)*, 2013, pp. 727–736.