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

Cédric du Mouza
CNAM
Paris, France
dumouza@cnam.fr

Witold Litwin
Univ. of Paris Dauphine
Paris, France
witold.litwin@dauphine.fr

Philippe Rigaux
INRIA Saclay
Orsay, France
philippe.rigaux@inria.fr

Thomas Schwarz
Santa Clara Univ.
Santa Clara, CA, USA
tjschwarz@scu.edu

ABSTRACT

AS-Index is a new index structure for exact string search in disk resident databases. It uses hashing, unlike known alternatives, whether baesd on trees or tries. It typically indexes every n-gram in the database, though non-dense indexing is possible. The hash function uses the algebraic signatures of n-grams. Use of hashing provides for constant index access time for arbitrarily long patterns, unlike other structures whose search cost is at best logarithmic. The storage overhead of AS-Index is basically 500 - 600%, similar to that of alternatives or smaller.

We show the index structure, our use of algebraic signatures and the search algorithm. We present the theoretical and experimental performance analysis. We compare the AS-Index to main alternatives. We conclude that AS-Index is an attractive structure and we indicate directions for future work.

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) [17]. In main memory, matching a pattern P against a string S runs in $O(|S|/|P|)$ at best [4]. Searching very large data sets requires an index, despite the storage overhead and 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 a search on any substring of the records' contents. We call the input to the string the *pattern*.

Currently deployed systems for *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, experimental, and focus typically on specific applications [7, 20]. The literature attributes this to 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 insertions and deletions that make 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 [7] and n-gram inverted index [18, 11]. 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 $O(|P| + \log_B N)$ 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. Its novel attractive property, unique at present to our knowledge, is constant disk search time, independent of the size of the database and of the length of the pattern. This results from its standard database approach for large-scale indexing (namely hashing). AS-Index hash calculus is however specific. As the name suggests, it relies on algebraic signatures, (ASs) [15], whose algebraic properties we can exploit.

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 that String B-Trees. 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. All

ACM CIKM '09, Hong Kong, China

these properties should make AS-Index a practical solution for text indexing.

The paper is organized as follows. Section 2 gives a bird's eye view of the AS-Index basic principles. We recall the theory of algebraic signatures (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. 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, (Fig. 1). Buckets are pointed to by the hash directory. Simplicity and performance of such files attracted countless applications. The main advantage is constant access time, independent of file and pattern size. Constant time is not possible for a tree/trie access method.

Each bucket stores a list of *entries*, each indexing some n-gram in the database. The basic 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, scalabilty and distribution.

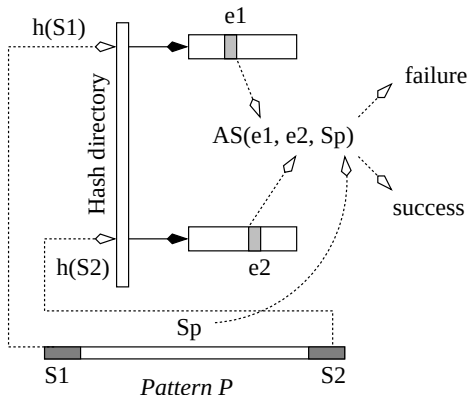

Figure 1: A matching attempt with AS-Index

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. Hashing on S_1 locates the bucket with every entry e_1 indexing an n-gram in the database with the signature of 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. An algebraic calculation $AS(e_1, e_2, S_p)$ determines whether S_p may match the suffix of S as well. 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 the 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.

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

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

Table 1 compares the analytical behavior of AS-Index with those of two competitors (String B-Trees and n-gram index) and summarizes its expected advantages. The size of all structures is linear in the size 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-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, whereas n-gram index cost is linear in $|P|$.

In summary, AS-Index efficiently identifies matches with only two lookups, for any pattern length. This efficiency is achieved through extensive use of properties of algebraic signatures, described in the next section.

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 Unicode. 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 [15]. 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 non-zero elements of the Galois field. It is well known that there always exist primitive elements.

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 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 [15] for more details about definitions and properties of algebraic signatures.

In our examples, we give 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 $s_2 s_1 = \#1234$.

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

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 $l \geq n-1$. With other words:*

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

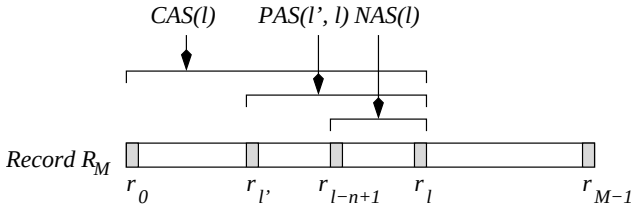

Figure 2: Computing $CAS(l)$, $PAS(l', l)$ and $NAS(l)$ in record R_M

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 i-th symbol of a CAS, as $CAS_m(l)_i$ and proceed similarly for NAS and PAS.

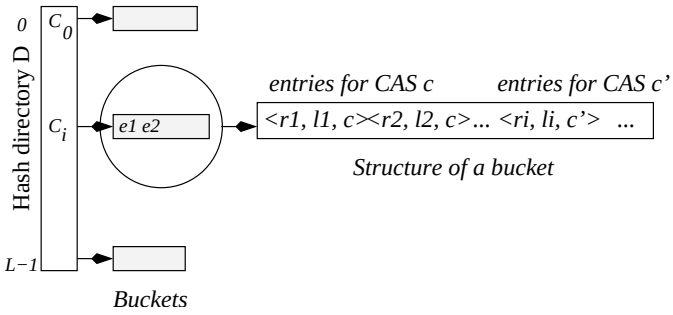

Figure 3: Structure of the AS-index

Properties 3 and 4 let us incrementally calculate next CAS and NAS while building the file, or preprocessing 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 appear. Property 6 finally is fundamental for the match attempt calculus.

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

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

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

For $0 \leq l' < l$:

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

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 symbols from our Galois field. Our search finds all occurrences of a pattern in the non-key field of every 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, and by a different entry. To construct the index, we process all n-grams in the database.

AS-Index is a hash file, denoted $\mathbf{D}[0..L-1]$, with directory length $L = 2^v$ being a power of 2 (Fig. 3). Elements of $\mathbf{D}$ 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 $\mathbf{D}[i]$ 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 cut lower 30 bits.*

Now, consider the record with RID 73 and non-key field 'University Paris Dauphine'. Assume that $NAS_4(\texttt{'Unive'}, 4)$ 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 $P = p_0 \ldots p_{K-1}$ 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 by extracting values that become then entries into AS-Index to locate matches.

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, denoted as S_2;

There are several ways to compute these signatures. For instance, one may compute S_1, then S_p, then extract the m-symbol value of S_2 through 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.

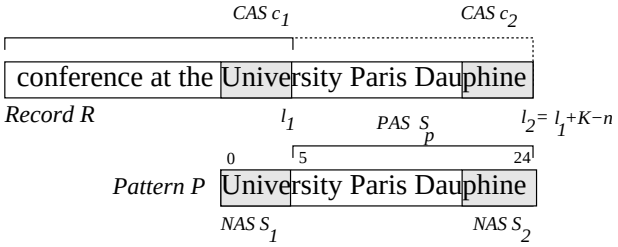

Figure 4: The search algorithm

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 equal the first and last n-grams in pattern P.

We search every pair (G, G') able to characterize a matching string $S = P$. We must have $R' = R$ and $l_2 = (l_1 + K - n) \bmod (2^f - 1)$. The last component in G', c_2 should match the value implied by c_1 and S_p (see Figure 4). Algebraic property 6 implies that c_2 should be

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

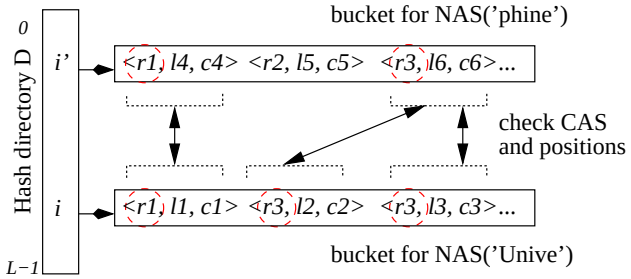

Figure 5: Using AS-Index for a search

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 $P = p_0 \ldots p_{K-1}$, the n-gram size n
Output: the list of records that contain P
begin
// Preprocessing phase
$S_1 := NAS_m(P, n-1)$
$S_2 := NAS_m(P, K-1)$

Nr. Collisions	Taken	Expected
0	16,568,642	16 568 642.3
1	207,274	207272
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 209881 moderately sized English words.

$S_p := PAS_1(P, n, K-1)$
$i := h_L(S_1)$ // 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. Also, it should be RAM-based and therefore typically negligible with respect to the disk search time. We thus do not detail it here.

6. ANALYSIS

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

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. Only the four characters 'A', 'C', 'G', and 'T' appear as characters in such a file. There are only $4^6 = 4K$ 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.

For a different experiment, we chose a list of English words (209881 words - 2.25MB) six or more characters longer taken from a world 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 words with more than five characters (65536 words) and calculated the number of signatures attained by i words as well as this number for a perfect hash function (Table 3). 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 (≥ 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, that we do not present here due to space limitation, at the expense of a proportional increase in search time.

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, $\mathbf{D}[i]$ and $\mathbf{D}[i']$. 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, the average complexity of this phase should be (under our uniformity assumption) $2l$. The worst case is $O(l^2)$. This 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 costs. Altogether, the search cost is

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

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

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.

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 [8] 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 $2l$. In practice it is $l * E$, where E is a couple of visited entries 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 $E = 500ns$ 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 [3]. 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 $n + 1 = 11$ symbols.

7. NON-DENSE INDEXING

While our basic scheme performs well, the storage overhead of about 500% might be too high for some applications. We now describe a variant that addresses this issue.

7.1 Skewed n-gram signature distribution

As previously observed, a skewed distribution can lead to many entries in a single AS-Index line. 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 pattern. The simplest choice is to have the n-grams evenly distributed over the pattern.

The search first determines the shortest line length of all lines indexed by any of the q n-grams. If any of these lines is empty, then we are done: the pattern is not found in the database. If the shortest line happens to be the one indexed by the initial n-gram in the pattern, nothing changes. 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 line indexed by the last n-gram and c_2 the one in the first line, 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 line $\mathbf{D}[h_L(S_a)]$, we search for a matching entry in the line 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.

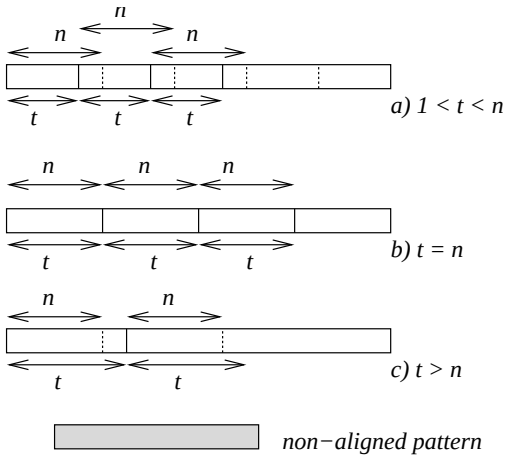

Figure 6: Non-dense AS indexing: (a) overlapping n-grams, (b) tiling, (c) lossy

Our next variant lowers the storage overhead by indexing only *some* instead of *all* n-grams. The disadvantages are higher search costs and a larger minimal size for the patterns that we can search for. 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 $P = p_0p_1, \ldots p_{K-1}$. 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 line 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 sub-patterns 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 P = 'University Paris Dauphine', Figure 6. Let $n = 4$ and $t = 4$. 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 tradeoffs. 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. In the running example of our 100 GB database, the reduced AS-Index size would be about 125 MB, already fitting current mass-produced flash disks.

7.2 Scalable Distributed AS-Index

This variant targets the maintenance of an AS-Index of a growing database that might ultimately reach a very large size. An AS-Index with static length and width will eventually see many and large overflows. We propose to deal with this problem by converting AS-Index into a Scalable Distributed Data Structure (SDDS). Appropriate choices are LH^*_{LH} [14] or its high-availability variants LH^*_{RS} [13]. These schemes target especially the distributed RAM (or flash) for storage as they are orders of magnitude faster than disks. In a nutshell, an SDDS variant of AS-Index, let us call it SDAS-Index, would work as follows.

We create the SDAS-Index as AS-Index at some node (site), called node 0, or LH^*-bucket 0. Hashing of n-grams into lines in SDAS is dynamic using the linear hash addressing. The number of lines grows through splits. These are triggered by inserts that result in an overflow of an AS-Index line. If the number of lines exceeds the capacity of a node, then LH^* will split the content of a node, moving about half to a new node. During this split, every other line remains at the current node and the remaining ones are sent to the new node.

A search in SDAS-Index is essentially the same as in AS-Index, except that access to a line involves messaging. Network latency becomes the dominant part of the search cost if the contents of a node are stored in memory. Also, processing line entries is now done in parallel. If we apply performance results from implementations of LH^* systems, then SDAS-Index searches should complete in a few milli seconds. The precise analysis and experimental confirmation remain to be done.

An SDAS-Index implemented in distributed RAM can easily grow to thousands of nodes, each with a capacity of easily 1GB. This gives a total capacity of about 1TB. Depending on whether we use non-dense indexing or not, we could index a database of 200 GB to 1TB with expected search times in the order of a 0.1 msec to a few milli-seconds. Using disks, we can target PB-scale databases with search times somewhere between 0.1 sec to 1 sec.

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 [7] and an n-gram index, based on inverted lists [24]. The rationale

for String B-Tree, discussed more in what follows, is that it appears attractive for disk-based use and is among most recent proposals. As Section 9 discusses, the inverted list was the common basis for many variants, e.g., n-gram/2L [11]. Notice that all the discussed structures are tree/trie based. Hence, none offers the constant search performance of AS-Index. In other words, their search speed must deteriorate beyond the one of AS-Index for a sufficiently large database.

All structures are coded in C++, and we run the experiment under Linux on a 2.40 GHz duo-core processor with 2GB in main memory and two 80GB disks.

We conduct our experiments on a database of records identified by a unique ID and with content consisting of a sequence of one-byte characters. We treat each character as an element in the Galois field $GF(2^8)$. The design of the database is meant to represent a large spectrum of situations ranging from many small records to large-size protein descriptions. We also consider databases of DNA sequences. All records are stored on disk.

8.1 Settings

We implemented a String B-Tree structure [7], making our best efforts to minimize the storage overhead. Each node contains a compact representation of a Patricia Trie [19]. In our implementation a disk block occupies 4K, and we can store at most 550 entries in each leaf.

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 6-bytes entries. The final step builds the B+-tree which allows to access quickly to a list given an n-gram. Our simple construction in bulk is fast and creates a compact structure.

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 (2 bytes) and an offset. The latter is a 1B integer obtained by taking the actual offset of the n-gram in the record, modulo 255. Using the CAS as a secondary sort key, one places the entries into the required order for insertion into the bucket.

We use three types of datasets with quite distinct characteristics: `alpha`, `dna` and `text`. 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_{full} with all the 256 symbols that can be encoded with $f = 8$ bits. We call the resulting datasets `alpha(26)` and `alpha(full)`. 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[1]. For the types `alpha(full)`, `alpha(26)` and `dna`, we composed datasets ranging from 10MB to 100MB. The last type, `text`, consists of real text records created from ASCII files of large English books. The typical size of a text record is 1-2 MB, and we created a 100MB database of `text` files. The n-gram size is set to 8 for DNA files, and to 4 for the other datasets.

8.2 Space occupancy and build time

[1]http://hgdownload.cse.ucsc.edu/

File	AS-index		n-gram index		Str. B-Tree
5 MB	23.1	(20+3.1)	35	(30+5)	36.9
10 MB	43.1	(40+3.1)	65	(60+5)	73.8
30 MB	123.1	(120+3.1)	185	(180+5)	221.2
60 MB	243.1	(240+3.1)	365	(360+5)	442.5
100 MB	403.1	(400+3.1)	605	(600+5)	737.6

Table 5: Index sizes in MB for `alpha(26)` files

File	AS-index		n-gram index		Str. B-Tree
5 MB	20.4	(20+0.4)	31	(30+1)	36.9
10 MB	40.4	(40+0.4)	61	(60+1)	73.8
30 MB	120.4	(120+0.4)	181	(180+1)	221.2
60 MB	240.4	(240+0.4)	361	(360+1)	442.5
100 MB	400.4	(400+0.4)	601	(600+1)	737.6

Table 6: Index sizes in MB for DNA files

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 size of the B+tree. 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 respectively the index sizes for `alpha`, `DNA` and `text` files.

The sizes of indexes are comparable. The size of AS-Index benefits from its smaller entries (4 bytes), mostly due to the 1B size of the offset. Sophisticated compression techniques [24] would benefit all structures. n-gram index offsets could be limited to 3B or even 2B, which would still allow the size of records to grow to up to 2^{24} or 2^{16} symbols. If the database has a few long records with many occurrences of the same n-gram, then we can save space by storing each rid only once in the list, followed by a possibly compressed list of offsets. If, on the contrary, the database consists of many, relatively small records, compression based on delta-coding can be envisaged. Both implementations use space better. The String B-Tree requires more space with 7B entries in the leaves and 11B entries for internal nodes.

For real datasets, either `dna` or `text` files, the distribution is far from being uniform. Table 8 shows the distribution of the number of entries from two real 100MB databases. The average number of entries is 1,534 for the DNA database, and 372 for the `text` database, with an important variance. In the worst case (`text` files), the largest list has 1,480,008 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.

The building time is proportional to the size of the index. 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 100 MB file is about 1,000 s, and the bulding rate is about 120 KB/s. 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.

8.3 Search time

File	AS-index		n-gram index		Str. B-Tree
100 MB	402.7	(400+2.7)	602.8	(600+2.8)	843.0

Table 7: Index sizes in MB for `text` files

File	Average	Min	Max	Std. dev.
DNA	1,534	1	145,599	1,953
text	372	1	1,480,008	5,543

Table 8: Distribution of entries for the real datasets (100 MB files)

K	10	50	100	200	300	400	500
AS	18.3	15.6	14.9	18.7	15.4	17.8	16.4
ngram	73.0	173.7	321.2	586.9	835.5	1076.6	1367.7
Spd-up	3.99	11.13	21.56	31.39	54.25	60.49	83.4
Str BT	43.6	43.7	43.6	43.9	43.2	43.7	43.5
Spd-up	2.38	2.80	2.93	2.35	2.81	2.46	2.65

Table 9: Search time in ms for 100 MB `alpha(full)` files

We performed extensively pattern searches in our databases. We extracted the patterns from the files to guarantee that at least one result is found. Pattern sizes range from 10 symbols to 500 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 one hundred search operations.

We report the results on search time, in ms, in Tables 9, 10, 11 and 12, for 100MB files.

As expected, the String B-Tree and AS-Index behavior is constant regardless of the length of the pattern, while n-gram index performance degrades linearly with this length. For our 100MB files, the height of the String B-Tree is 3 independently of the alphabet size, for 10^6 indexed substrings (recall that the fanout if 550, 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 a single disk access 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 4-5 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 17 ms for `alpha(full)`, 30 ms for `alpha(26)` files, 30 ms for `dna` files and 18 ms for `text` files (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(full)` 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 compare to 256^4 for `alpha(full)`.

For real data (DNA and `text`), 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` files. Table 8 shows that, on average, the number of entries in a bucket is larger for DNA (1,534 entries) than for `text` (372). The cost of DNA search is accordingly slightly higher ($\approx$ 30 ms, against $\approx$ 20 ms). Recall that our algorithm chooses the smallest bucket for driving the search, which limits the impact of skewed datasets and the variance of search times.

Figure 7 summarizes the results discussed above, and illustrates the linearity of search times for ngram-index and the constant search times for AS-Index and String B-Tree.

K	10	50	100	200	300	400	500
AS	34.7	28.8	31.6	29.2	28.3	32.6	30.4
ngram	69.1	130.7	227.3	418.6	597.2	758.8	908.1
Spd-up	1.99	4.54	7.19	14.34	21.10	23.28	29.87
Str BT	44.1	43.3	44.0	43.6	43.7	43.7	43.6
Spd-up	1.27	1.50	1.39	1.49	1.54	1.34	1.43

Table 10: Search time for 100 MB for `alpha(26)` files

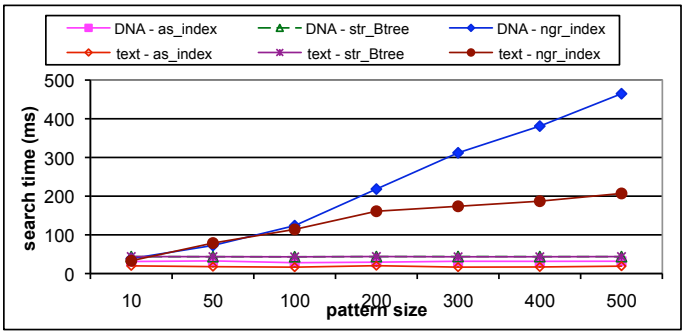

Figure 7: Search result on 100MB files, for varying pattern size

Figure 8 shows the evolution of search times as the size of the database increases from 10 MB to 100 MB. Each curve represents the results for a given dataset, with patterns consisting of 10 to 200 symbols. As before, the AS-Index exhibits an almost constant behavior (appr. 20-30 ms), even for very large patterns (200 symbols) searched in large files (100 MB). The search time with the String B-Tree slightly increases with the size of the file (but remains constant with the pattern size). The tree height remains equals to 3, however the number of nodes increases with the file size. This accounts for a lower probability of a buffer hit during tree traversal. The logarithmic behavior of the String B-Tree is almost blurred here, because of the large node fanout (550). It would appear with larger files. Given a file size greater than 167MB for instance ($i.e.$, corresponding to more than 550^3 strings), its height would increase to 4, with two (2) additional disk accesses on average for a search.

The search time evolves (sub)linearly for n-gram index, both in the size of the pattern, and in the size of the database. The slope is steeper for large files. 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 Figure 8. However the cost remains sublinear (the cost for 100MB is only 3 times higher than the cost for 5MB). This is due to the merge process which stops when the smallest list has been fully scanned, thereby avoiding a complete access to all lists.

Figure 9 and Figure 10 summarize the ratio of search times, giving the speed-up of AS-Index over respectively the n-gram index and the String B-Tree. The `alpha(full)` dataset family is a special case. Because of the uniform distribution that produces a large number of n-grams ranging over all the possible values, finding the

K	10	50	100	200	300	400	500
AS	31.1	32.7	28.0	29.4	31.7	31.5	31.8
ngram	38.7	72.6	123.8	218.3	312.0	381.3	464.7
Spd-up	1.24	2.22	4.42	7.43	9.84	12.1	14.61
Str BT	43.7	43.6	43.4	43.8	44.0	43.8	43.7
Spd-up	1.41	1.33	1.55	1.49	1.39	1.39	1.37

Table 11: Search time for 100 MB `dna` files

K	10	50	100	200	300	400	500
AS	19.9	17.8	15.5	20.3	16.6	17.0	19.1
ngram	33.1	78.8	114.1	161.0	173.8	187.0	207.0
Spdup	1.66	4.43	6.92	7.93	10.47	11.0	10.84
ST-BT	43.9	43.6	43.2	44.0	43.3	43.4	43.8
SpdUp	2.21	2.45	2.79	2.17	2.61	2.55	2.29

Table 12: Search time for 100 MB `text` files

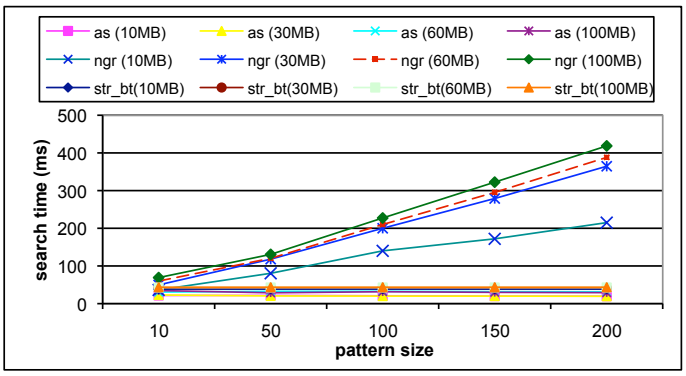

Figure 8: Search result, for varying pattern and file size

reference to a list through a traversal of the B+tree takes time. The number of traversals increases linearly with the pattern size, which explains the large cost with respect to the other datasets. For these datasets our algorithm clocked in as twice as fast than n-gram based search for small patterns (10 symbols) and about 30 times faster for large patterns (with hundreds of symbols). Figure 10 shows a constant gain of AS-Index over String B-Tree, about 1.5 with `DNA` and `alpha(26)` files and about 2.5 for `text` and `alpha(full)` files. This difference is due to the higher selectivity of n-grams for `text` and `alpha(full)` what implies smaller buckets. The String B-Tree is not affected by the selectivity of n-grams, *i.e.*, the alphabet size.

8.4 False positives

Table 13 shows the false-positives (F/P) retrieved when searching for patterns of 50 symbols in a set of files with a total size of 10MB, with respect to the alphabet and the n-gram size. As expected (see Section 6) the number of F/P highly depends on the choice of n. While 4 or 6-grams (resp 2-3 grams) generate a significant number of F/P for DNA (resp. alpha(26)), this number quickly

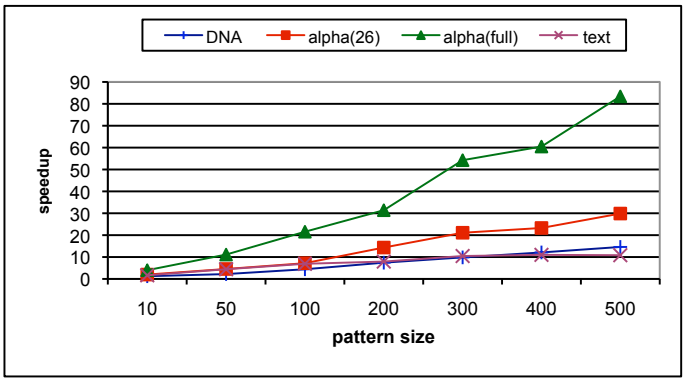

Figure 9: Ratio of search times AS-Index/ngram-index with respect to pattern size

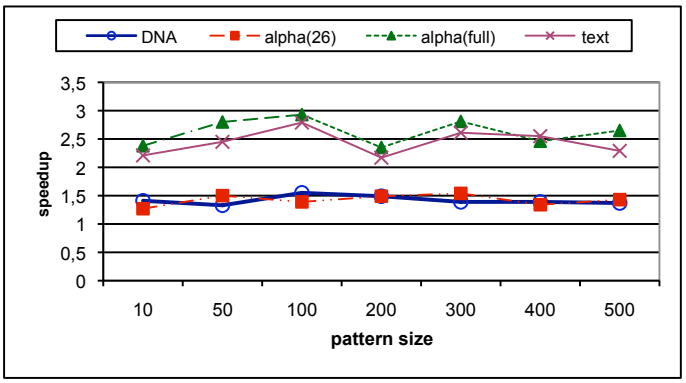

Figure 10: Ratio of search times AS-Index/String B-Tree with respect to pattern size

alphabet	**n**	ratio f/p
DNA	4	0.999
	5	0.986
	6	0.798
	7	0.019
	8	0.001
alpha(26)	2	0.994
	3	0.31
	4	0.001

Table 13: False-positives ratio for various alphabets and ngram size

drops to less than one on a thousand with 8-grams (resp. 4-grams). The behavior for small-size alphabets can be explained by the low number of possible n-grams (*e.g.* 4^4 for DNA with $n = 4$) and thus by the low number of NAS possible signatures (see Section 6). Since AS-Index does not store the n-grams but their signature, the choice of the n has no impact on index size and performance.

9. RELATED WORK

Finding patterns in a large database of sets is a fundamental problem in Computer Science and its applications such as bioinformatics. 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 [9] have received much attention. Recent work by Kurtz [12], Tata, Hankins, and Patel [22] 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 [12]. Related data structures such as Manber's suffix arrays [16], Kärkkäinen's suffix cacti [10], or Anderson and Nilsson [1] suffix tries lower storage overhead at the prize of an increase in search time. Dementiev, Kärkkäinen, Mehnert, and Sanders [5] give methods to make suffix arrays effective and efficient for truly large files

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 en-

gine. Life is much simpler if the database consists of words and we restrict ourselves to word indexes that can be stored much more compactly [24].

Signatures files were proposed in [6] and shown to be inferior to inverted indexing in [25]. Some other attempts for indexing sequences are the ed-tree [21] for DNA files, and the q-gram index [2]. 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 [23]. 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, is reported in [11]. 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.

10. CONCLUSION AND FUTURE WORK

We have presented 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 is a new approach to indexing, which takes advantage of the interpretation of character as symbols in a mathematical structure to develop new computational techniques.

Future works include more complete performance studies including the study of a non-dense indexing variant and a variant that deals with skewed distribution by considering the most selective ngrams of the searched pattern. Scalable and distributed AS-Index constitute other promising research directions that we plan to investigate.

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

11. REFERENCES

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

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

[3] C. Charras, T. Lecroq, and J. D. Pehoushek. A Very Fast String Matching Algorithm for Small Alphabets and Long Patterns. In *Proc. Combinatorial Pattern Matching*, 1998. LNCS 1448.

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

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

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

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

[8] J. Gray and B. Fitzgerald. Flash Disk Opportunity for Server-Applications. In *Intl. Conf. on Innovative Data Systems research (CIDR)*, 2007.

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

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

[11] M. S. Kim, K. Whang, J. G. Lee, and M. J. Lee. 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.

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

[13] W. Litwin, R. Moussa, and T. J. E. Schwarz. LH*$_{RS}$ - a highly-available scalable distributed data structure. *ACM Trans. on Database Systems*, 30(3):769–811, 2005.

[14] W. Litwin, M.-A. Neimat, and D. A. Schneider. LH* - A Scalable, Distributed Data Structure. *ACM Trans. on Database Systems*, 21(4):480–525, 1996.

[15] W. Litwin and T. Schwarz. Algebraic Signatures for Scalable Distributed Data Structures. In *Proc. Intl. Conf. on Data Engineering (ICDE)*, 2004.

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

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

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

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

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

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

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

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

[24] I. Witten, A. Moffat, and T. Bell. *Managing Gigabytes:*

Compressing and Indexing Documents and Images. Morgan-Kaufmann, 1999.

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