Pre-computed Algebraic Signatures for Fast String Search, Protection Against Incidental Viewing and Corruption of Data in an SDDS

W. Litwin, R. Mokadem & Th. Schwarz

Abstract

We propose to encode the records of a Scalable Distributed Data Structure (SDDS) using pre-computed algebraic signatures. The *partly* pre-computed algebraic signature of a string encodes each symbol into its contribution to the algebraic signature of the string. The *cumulative* pre-computed algebraic signature encodes each symbol with the signature of the string prefix ending with the symbol. The encoding/decoding according to either scheme occurs at the SDDS clients. For both schemes, and each operation, the overhead is of linear time complexity $O(n)$. It is however slightly higher for the cumulative signature. The schemes protect the SDDS data against incidental viewing by an unauthorized server's administrator. One may use them also to detect and localize the silent corruption. These features should be of interest for P2P and grid computing.

Both schemes provide also fast string search (match) directly on encoded data at the SDDS servers. They appear an alternative to known Karp-Rabin type schemes in our context of a search in a file or a database. Both accelerate the string search with respect to the fast already use of the algebraic signatures on the original data. Moreover, both appear typically the fastest in the context, among any string search algorithms we are aware of. The cumulative signature provides the fastest searches. For the string of l symbols in a field of n symbols, the complexity is almost $O(1)$ for prefix search, and $O(n - l)$ for the string search. The string manipulation capabilities of our schemes should be by themselves of interest to applications.

1 Introduction

At present, a record in a Scalable Distributed Data Structure (SDDS) such as LH* or RP*, is implicitly or explicitly assumed to contain the original values of user/application data, [LNS96], [LNS94]. This is the case for the free SDDS-2004 system [C04]. The SDDS servers store records in buckets in distributed RAM. This greatly enhances the access performance compared to disk buckets. Typically, as in SDDS-2004, SDDS clients and servers are P2P PCs. The key search or insert speed acceleration reaches three hundred times, [DL00], [LMS04], and thus provides an experimental backup for Jim Gray's old conjecture about the advantages of distributed RAM presented at UC Berkeley in 1992 and following the calculations in [G88].

The SDDS data in a PC RAM are at risk of incidental exposure to the server's owner/administrator even during simple maintenance operations such as a storage dump. Both SDDS user and PC administrator might be unhappy with this exposure and would like to limit it especially in regards to the non-key data. (We recall that an SDDS record consists of a key and a non-key field. The former is often a meaningless object ID.)

A PC provided as the SDDS server should typically support concurrently other applications at the discretion of its owner. A malfunction due to execution of any of them may lead to an incidental corruption of SDDS data. User and server administrator may both welcome a method detecting such a corruption. Possibly, indicating also the memory area where the corruption occurred with a practical precision, e.g., 256 B at most.

Most often, the servers should be a part of an organization. There is then usually a business-friendly relationship between the administrators and the user. The protection can be simple and inexpensive because it can assume no malicious action to break it or trust social safeguards. In "real life", confidential letters are circulated in a simple envelope with a stamp "confidential" on it, despite the relative ease to use steam to open the envelope with only a small risk of detection. The threat of a jail term suffices to keep even the easily tempted from opening interesting mail in a normal office setting, and thus saves organizations the costs of armored protection for the mail.

Encoding against incidental viewing and corruption should preserve all user capabilities. These include in SDDS-2004 the key search and the non-key scan by string searches. The latter functions use the algebraic signatures [LS04]. The approach presents advantages with respect to more traditional non-signature based search methods. It may be about independent of the searched string length. The SDDS-2004 client sends indeed only a signature few bytes long to the server. Presently we use 4 byte signatures for strings that are up to 64 KB long.

The communication load is much less. As an important side effect, a network snooper has a harder time to make sense of captured traffic.

Presently, popular encoding schemes including the standard ones under Windows do not meet these requirements well. They either limit the capabilities or require systematic encryption at the server. The latter notably slows record access. Recent work attempts to adapt popular string search algorithm to work on the encoded data appeared therefore recently for selected compression schemes, e.g., [NT04]. At the expense of heavier encoding/decoding, due to the compression, one may have interesting performance for selected types of strings and operations upon them. We propose two schemes that fit our needs better. Both use the pre-computed algebraic signatures [LS04]. The first scheme uses the *partial* pre-computing. It encodes each symbol in a data string to store by the value of its contribution to the signature of the string. The other scheme uses the *cumulative* pre-computing. It replaces each symbol by the signature of the prefix up to it. The SDDS client performs the encoding and the decoding which are both very fast. The server gets the encoded data only.

The algebraic signatures allow for fast string search using an algorithm similar to Karp-Rabin [KR87]. One compares the signature of the searched string with that of the currently examined (sub)string in the data field. The SDDS-2004 prototype uses the scheme for parallel/distributed non-key scans. These search for a string in the non-key field of the SDDS record (we recall that an SDDS record basically contains the key and the non-key field). The pre-computed signature preserves this capability over encoded data. In other words, the scan performs without decoding any data at any server. As one could wish for best data protection. Surprisingly, the string search becomes even notably faster, especially for cumulative pre-computing.

The complexity of the search of a prefix of l symbols in a field of n symbols becomes independent of l (and of n) for the latter scheme. It is O (1) for practical purpose, which is the search time to examine the last symbol of the prefix. Likewise, the complexity of the longest common prefix search should usually be O (1), independent of the expected common prefix length. The time complexity of the string search for the cumulative signatures is almost O (n –l). All these numbers, as well as the only slightly higher ones for partial pre-computing, make these algorithms faster in our context than any other we are aware of, [CL03]. These aspects of our proposal should by themselves be of major interest for many applications. Perhaps, even if the other capabilities they offer are not of the concern. String search is, e.g., the basis of many Web-based applications, that search extensively for tagged data strings in HTML formatted records.

The method also allows for key encoding. For the LH* hash based access the key does not need to be decoded at the server at all. For range partitioning, e.g., in an RP* file, decoding a single character on the fly may suffice. This is the case for character strings and for numerical key represented by its digits. The latter conversion of numerical keys occurs, if needed, at the client during the encoding. The decoding is necessary to progress through the internal indexes.

With some effort, a “professional” hacker could break our encoding. This is the equivalent to the case of bad employee steaming open confidential correspondence. In turn, our method is inexpensive, as the paper envelopes are. It does not require any storage overhead. The encoding/decoding computations are fast, of complexity of O (n). The cost per symbol is about two table lookups, an integer addition, and an XOR for the cumulative signature calculus. In turn, as already said, the cumulative pre-computing offers faster prefix and string search. The reason for the latter is avoiding one XOR per visited symbol.

The key search, insert or update performance at the server are not affected by the encoding of the non-key data. The non-key data at the servers are not decoded for any of the above operations. The key symbols decoding for in-bucket search in a range-partitioned SDDS may slow the discussed operations slightly. Perhaps surprisingly, the overall speed may nevertheless increase. The in-leaf search for the entire key may indeed become sufficiently faster.

To protect the record against corruption the client inserts additional data into it. These are signatures of areas or fields in the record. The client matches them against the values computed on the fly during the decoding. The failure indicates the corruption in the signed area (field).

This function induces a small overhead. It is for instance less than 0.5 % for areas of 256B. Smaller areas lead to proportionally higher overhead. The computational overhead during encoding and decoding is negligible. The reason it that the signature calculus needed largely reuses that performed during the encoding or decoding anyway.

Below, we describe the aspects of our method protecting against the data viewing. We recall the theory of the algebraic signatures and define the encoding, and the decoding algorithms for both the partly pre-computed and the cumulative pre-computed signatures. Next, we address the string search issues. We detail the full non-key match, the prefix match and the string match operations for our schemes. We show faster performance of the cumulative signatures. Afterwards, we present the corruption protection scheme. Next, we discuss the related

work and show that in our context of the file or database string search, our schemes, especially the cumulative one, seems the fastest known. We finally conclude the presentation and discuss the future goals. These include interesting perspective for popular database operations, including basic updates.

2 Protection Against Incidental Viewing

2.1 *Algebraic Signatures*

In SDDS-2004 we use algebraic signatures for non-key data search in particular. We recall that a SDDS-2004 record consists of the (primary) key field and of the non-key field. We apply the signatures to *full match* (all non-key data) and to *partial match* search (of a substring of the non-key data). The partial match may be a *prefix* match search, where the string to find is the prefix of the non-key field. It can alternatively be simply the (general) *string match* search. The string can be anywhere in the field. For the full match, the client sends the signature of the string only. For the partial match, it also sends the string length in symbols. In both cases, the messages are typically shorter than if the client sent the string. Also, the message snooping does not inform about what should be searched. This is an appreciable feature these days.

We now recall the algebraic signature calculus. We consider the field as a string of symbols of f bytes each. Basically $f = 8$, e.g. for an ASCII string, or $f = 16$ otherwise, e.g., for an Unicode string. The symbols are elements of Galois Field GF (2^f). Let p_i be the i^{th} symbol in the field P of l symbols, $i = 1,2\ldots l$; $l < 2^f$. Let α be an element in GF (2^f). Below α is a primitive element, i.e., every non-zero element in GF (2^f) is some power of α. The 1-symbol *algebraic signature* is defined by the formula:

(1) $$\mathrm{sig}_\alpha(P) = \sum_{i=1}^{l} p_i \alpha^i$$

A k-symbol signature, $1 < k \leq 2^f - 1$; is a vector $\boldsymbol{\alpha}$ using as base k different $\alpha_1 \ldots \alpha_k$. A particularly wise choice is:

$$(\alpha, \alpha^2, \ldots, \alpha^k).$$

The k-symbol signature detects for sure any difference of at most k symbols between the signed fields of length $2^f - 1$ symbols at most. This property is at present unique to the algebraic signatures, to our best knowledge. For a larger difference between the fields, the error (collision, false drop…) probability can be as low as about 2^{-kf}. This, assuming the uniform distribution of the symbol values, i.e. where every possible symbol has the same probability 2^{-f} of being used.

Unlike other signature schemes, the algebraic signature has algebraic properties. Some are the basis for the sure detection of collision cited above. Some appear potentially promising in practice for other use as well. We base in particular what follows on some of those, recalling them when need occurs. Other properties remain to be explored or even determined.

For the addition and subtraction in GF, we use XORing. For the multiplication, we apply the α-based log/antilog tables. We recall that our antilog table has double size, of $2^{2f} - 2$ elements. This avoids the modulo $2^f - 1$ calculus. If a field has more than $2^f - 1$ symbols, then we consider it as divided into *pages* (areas) of at most that size. We calculate a *compound* (vector) signature of the successive pages as the field signature.

2.2 *Encoding*

Let p'_i be $p_i\alpha^j$. The *partly pre-computed* algebraic signature of string P is string P' with p'_i for every p^i. We may refer to it below also in short as *partial* signatures and to the related calculus below as to *partial* pre-computing or encoding. Let furthermore P_i denotes prefix of P up to p_i. The *cumulative* pre-computed algebraic signature of P is string P'' with $p''_i = \mathrm{sig}_\alpha(P_i)$. We qualify these signatures in short as well as the related pre-computing and encoding as *cumulative* in consequence. Observe that $p''_1 = p'_1$. The rationale for the name is the property (notice that '+' is in GF):

$$p''_{i+1} = p''_i + p'_{i+1}.$$

This is one of the algebraic properties of our signatures we referred to above. It is of basic importance here. It avoids indeed entire calculus of $\mathrm{sig}_\alpha(P_i)$ for each i, of length proportional to i. That would make the scheme much less practical at best.

The client requested to encode string P produces P' or P''. The choice depends on the application. To calculate the partial encoding, the client performs for each p_i:

$$p'_i = antilog\ (log\ p_i + i).$$

This costs (only) two table lookups and an integer addition. Our encoding into the partly pre-computed signature is thus fast.

To pre-compute the cumulative signature, the client calculates:

$$p''_i = p''_{i-1} \text{ XOR } p'_i$$

We recall that XORing is the operational calculus of the GF addition. This adds up to one XOR. It slows the encoding, that remains however fast. For both encoding schemes the time complexity for *n*-symbol is clearly *O* (*n*).

The client always encodes in this way the non-key field of a record. It encodes also the key upon request from the application, presumably considering the key meaningful. For long fields, the encoding assumes that each *P* is a page of length $l \leq 2^f - 1$.

If the key is not encoded, the client sends out the encoded record as usual for an SDDS. Otherwise, if the key is hashed, then the bucket address results from the encoded key. If the file is range partitioned, e.g., for any RP* scheme supported by SDDS-2004, then the bucket address is given by the original key value. Otherwise, the range partitioning would get lost.

Notice from the signature definition that the encoding algorithm works with our antilog table only till $i = 255$. If the field to encode is longer, one has to set any larger i to $i := i \bmod 255$. This in practice the use of 255-byte long pages.

2.3 *Decoding*

The server returns any searched record encoded. As we will show, it never decodes the non-key field. The key field may get decoded a symbol at the time for the range partitioning, as we explain later. Both options guarantee that even if the server crashes, no portion of the post-mortem dump shows a decoded record or even any portion of it.

Let consider first that the returned field is *P'*, i.e., it contains a partly pre-computed signature. As we have for every *i*: $p'_i = p_i \alpha^i$ hence to decode *P'* the client has to calculate the multiplicative inverse of each α^i, i.e. α^{-i}. Any non zero symbol in a GF has an inverse. Given the encoding formula, it is easy to see that the following calculus decodes then each p'_i:

$$p_i = antilog\ (log\ p'_i - i).$$

Two table lookups and an integer subtraction suffice thus. The decoding or a partly pre-computed signature is hence fast. It is as fast as the encoding in practice.

Decoding a cumulative pre-computed signature requires the determination of each p'_i. We have:

$$p'_i = p''_i - p''_{i-1}.$$

The subtraction of encoded values is here in GF. This translates in practice to the following calculus preceding the decoding of every p'_I:

$$p'_i = p''_i \text{ XOR } p''_{i-1}.$$

The decoding calculus remains thus fast, but slows down by one additional XORing, as for the encoding into the cumulative signature. In both cases, the time complexity is again linear in *n*, i.e., is *O* (*n*).

Example 1

Scrabble players insert interesting records into the anagram file. Among others, the record with non-key field[1]: *P* = 'MALADES_LAMSADE_GERARD' (quotes are not part of the field). Consider $\alpha = 2$ that is a primitive element in GF (2^8). The encoding at the client into the partly pre-computed signature produces field:

$$P' = (p'_1, p'_2 \ldots p'_{15}) = (2\text{M}, 2^2\text{A} \ldots 2^{15}\text{E}).$$

Notice that powers and multiplications are in GF. The encoding of 1st "M" is $p'_1 = antilog\ (log\ \text{M} + 1)$. That of the 2nd one is $antilog\ p'_{10} = antilog\ (log\ \text{M} + 10)$. These are different values in our GF.

The encoding using the cumulative pre-computed signature produces *P''* :

$$P'' = (p''_1, p''_2 \ldots p''_{15}) = (2\text{M}, p''_1 + 2^2\text{A} \ldots p''_{14} + 2^{15}\text{E}).$$

[1] For French speakers: an anagram with no ties to reality.

Again, powers, multiplications, and additions are in GF.

The decoding of a partly pre-computed signature calculates:

p_1 = *antilog* (*log* M + 1 - 1) = M

p_{10} = *antilog* (*log* M + 10 - 10) = M.

The 1-symbol algebraic signature of our field in GF (2^8) is:

sig_2 (*P*) = 2M + 2^2A…+ 2^{15}E.

Again, all the operations are here in GF. Notice that $sig_2(P) = p''_{15}$. Next, if the character 'S' changes to '_' in the field, then the modification changes $sig_2(P)$ for sure. If a typing error inverts 'ES' to 'SE', the error is detected with probability of $1 – 1/2^8 = 0{,}996$. To increase this probability to 1, one has to use 2-symbol algebraic signature.

2.2 *Non-key Search*

We first deal with the non-key scans. For the partial match, we consider below only the search for the strings not longer than $2^f – 1$ symbols. Next, we address the key search.

2.2.1 Full match

A full (non-key) match is the search for every record with entire non-key field equal to the string provided by the application. In SDDS-2004, to deal with fast full match search, the client pre-calculates the *k*-symbol signature and adds it to the non-key field during the insert or update. When the application requests the full match search, the client calculates the signature *s* of the string provided and sends it to every server. Each server delivers the records whose stored signature is *s*. The approach avoids the dynamic calculus of the signature during the search and seems more advantageous for various reasons. In our case in particular, there is no need to look up the encoded data at all.

Currently, SDDS-2004 uses for this purpose GF(2^{16}). The page size is 64K symbols ore 128K bytes. This is above the current record size in the system, limited to 64K bytes anyhow. The current value of *k* is *k* = 2. The probability of the collision is about 2^{-32}, provided the uniform distribution of stored symbols over all the 2^{16} possible values. This choice of *k* seems sufficient in practice. The client may nevertheless double-check whether the result is not a false positive.

The GF (2^{16}) could correspond to the encoding of Unicode symbols. The encoding would produce then the 1st symbol of the stored signature as well. The other $k – 1$ symbols have to be calculated in addition. The complexity is $O(kn)$ for the *n*-symbol field. There is a way to optimize this process during the encoding with respect to the independent calculus for each symbol of the signature. However, the time complexity remains the same. Note that for the cumulative encoding there is no need to store the 1-symbol signature separately. It is already there as p''_n. In other words, the insert should store only the other $k – 1$ symbols of the pre-calculated signature.

Instead of the *k*-symbol algebraic signature, one may store the *k*–symbol suffix S_k of the cumulative signature, $k > 1$. We call it in short the (cumulative) *k*-symbol *suffix* signature. The suffix signature calculates faster, in $O(n)$ time, regardless of *k* value. It is also easy to prove that the collision probability remains in practice almost that of the *k*-symbol signature, i.e., possibly as low as 2^{-kn}. The price to pay is that even if two records differ by at most *k* symbols, the probability of collision becomes above zero. Details of the practical trade-off remain to be studied.

It is possible to store, or calculate the *k*-symbol signature $s_1…s_k$ formed in also other ways from the encoded field or string. For instance, one may sample the cumulative signature in the binary way. That is one picks up the last symbol of the field (string) as s_1, then the 1st symbol as s_2, then the middle one as the s_3, then one in the middle of the interval below the middle symbol as s_4 etc. This strategy may lower the probability of the collision if the suffix symbols are dependent, and one believes the heuristics that symbols further apart are more likely to be independent. A random sampling over the cumulatively encoded field may be yet another strategy. We do not have compelling practical argument in the favor of any of these strategies at present. We consider therefore below the suffix signatures only, as they are the fastest to use.

A similar situation occurs if one inserts only the 1-symbol GF(2^8) signature, for ASCII etc. code, as in Example 1. This speeds up the calculus at the client. In contrast the collision probability obviously increases. It is at best about 2^{-8} here. One may therefore more easily set here for the *k*-byte signature or suffix signature. If the field has $m > 1$ pages, hence more than 255 characters, the client should insert basically the compound *m*-symbol signature or suffix signature. For a search, the client should send the compound (suffix) signature as well.

If the record does not store separately the signature of its non-key field, the full match search boils down to the prefix match discussed in next section. This is the case for the cumulative encoding if the full search uses the suffix signatures only.

The server calculates the signature one symbol at the time. Under the usual assumptions of the mutual independence of the symbols, each new symbol reduces the collision probability by the factor of 2^{-f}. The complexity becomes at least $O(1+256^{-1}+\ldots 256^{-k})$ that is $O(1)$ in practice. The client may double-check that there is no false positive. The overhead is negligible as long as the search returns relatively only a few records. The optimality of k in practice remains nevertheless to be studied.

2.2.2 Prefix match

The application searches here for every record with the non-key field l-symbol prefix matching in every symbol a given prefix. The basic strategy is that the client calculates and sends the 1-symbol signature s of the prefix to find and l. For ASCII the signature is in GF(2^8), for Unicode in GF(2^{16}). For each record in the bucket, if P_l is the prefix of the non-key field, the server calculates $sig_\alpha(P_l)$. If the records are not encoded, as at present in SDDS-2004, then the calculus is according to (1), in full. For the partly pre-computed signature, only the XORing of p'_i's remains ; i =1,2…l. The time complexity is $O(l)$ in both cases, but the latter calculus is obviously faster. For the cumulative pre-computing the server only needs to access p''_l. The complexity becomes $O(1)$. It makes the scheme several times faster then the previous ones for $l > 1$. The collision probability in every case is practice of 1/256 for ASCII or of 1/64K for Unicode. The client may double-check that there is no false positive.

The search may more generally use k-symbol signature. The complexity for the cumulative encoding, obviously the best to use here, becomes as for the full search about $O(1)$ in practice. The optimality of k in practice remains again to be studied.

Notice that the use of algebraic signatures for prefix search at the server has some advantage over any popular approach using simple XORing of non-encoded symbols of the prefix. In the signature, the order of symbols matters. The simple XORing ignores it. Thus, e.g., our signature based XORing is k-likely to distinguish between "Lamsade" and "Malades". Simple XORing will surely mess this up. Finally, with respect our goal of protection against incidental viewing, observe that both prefix search methods above do not decode the stored data at any time.

2.2.3 String match

The basic approach is that the client sends the 1-symbol (suffix) signature s of the string to match, its length l in symbols, and α. Again, the symbols are 8-bit long of GF (2^8) for ASCII, (or EBCDIC etc.) or 16-bit long of GF (2^{16}) for Unicode. For the partial pre-computing, the server proceeds as follows for each record. For simplicity, we consider that there is only one page per non-key field, unless we state otherwise.

1. It calculates the (1-symbol) algebraic signature s_l of the prefix of length l of P'. It does it by XORing p'_i till p'_l. If $s = s_l$, then it selects the record.

2. Otherwise, let s_c be the *current* signature that is the signature of the substring being dealt with. Initially, $s_c = s_l$. Let s' be s initially, and $i = 1$. The server performs:

(2.1) $s_c := s_c \text{ XOR } p'_i \text{ XOR } p'_{l+i}$

(2.2) $s' :=$ antilog (log s' + 1).

3. If $s' \neq s_c$, i.e., there is no match, then $i := i + 1$. The server loops on steps (2) and (3) until the last p'_i in P'. If no match occurs till the end, the server visits next record in the bucket, if any.

Step (2.1) subtracts the 1st symbol of the current signature, and adds the next one. SDDS-2004 performs the similar operation on P. That one however implies the respective multiplications by powers α^i and α^{l+i}. The partial pre-computing avoids them.

Step (2.2) calculates $s'\alpha$. Notice that clever implementation simply accesses the next entry in *antilog* table with respect to the last used one. The computed value is the new signature s' of the searched string at any offset $i + 1$ in the field with respect to the signature s' at offset i, where $s' = s$ initially. For the string search in P, the calculus is the same. Both steps together avoid the re-calculus of the entire s'. Every scheme of string match using the algebraic signature under consideration is in footprints of the Karp-Rabin proposal, [KR83].

Finally, the client or the application may again double-check whether the result is not a false positive. The collision probability remains that of the prefix search above. As above also, the search may use the k-symbol suffix signature matching at the server for the string found, to arbitrarily decrease this probability. If so the client

should of course send *k*-symbol signature as well. If the client finds a collision anyhow, it continues by itself the search in the received field. It may find that the record was relevant anyway.

With respect to the search in *P*, as in SDDS-2004 at present, Step (2.1) is strictly faster while the other operations above are the same. Hence the entire string search using the partial pre-computing is strictly faster. Notice that the complexity of both cases is again linear, i.e., in practice $O(n)$. Notice that it is independent of l. Finally, like the prefix search, the string search algorithm does not decode the stored data at any time. It also respects our goal of best protection against incidental viewing.

The cumulative pre-computing on its turn leads to the following modification of the outlined scheme. Step 1 simply tests whether $s = p''_l$. No XORing thus which makes this step strictly faster. Step (2.1) becomes:

(2.3) $s_c := p''_i \text{ XOR } p''_{l+i}$.

The step computes s_c by direct subtraction of the signature of prefix P_i from that of P_{l+i}. The calculus is faster than of (2.1) by avoidance of one XOR. Further steps are the same for both schemes. String search using the cumulative pre-computing is thus strictly faster than for the partial one. It is the fastest in our class. The complexity becomes $O(n - l)$. Notice again that the calculus uses the encoded symbols only.

If the field has several pages, the calculus becomes slightly more complex, when the search for a string shorter than a page overlaps two pages. It is easy to see for the cumulative signature especially, that moving the test to next position requires one more XOR. This concerns the last symbol of the page where the string starts, e.g., $p"_{255}$ for GF (2^8). The calculus reconstructs the signature that the string would have if it were stored at the examined offset in a page longer than 255 symbols in our case. If the searched string is longer than a page, the search should be amended so to search for the substrings. The analysis is easy so we do not follow up on here.

Example 2

Consider the search for any record with substring 'LAMSADE'. The client calculates the signature *s* and sends the couple (*s*, 7). Each server scans accordingly every record in its bucket. For every visited record, the search starts with the calculus of the prefix signature s_7. For the record encoding *P* = 'MALADES_LAMSADE', the calculus will yield sig_α (MALADES). For *P'*, the calculus will perform the 7-term XORing. For *P''*, it will simply find sig_α (MALADES) in p''_7.

Consider that $s \neq s_7$ for our *P*. The case is very likely. As next step, the server dealing with partial pre-calculus performs the calculus according to Steps 1 and 2 above:

(i) $s_c := s_7$ $s' := s$

(ii) $s_c := s_c$ XOR 'M' XOR '_'

(iii) $s' := \text{antilog}(\log s' + 1)$

The server dealing with the cumulative pre-encoding performs instead of (ii) :

(iv) $s_c := p''_1$ XOR $p''_8 = \text{sig}_\alpha$ (M) XOR sig_α (MALADES_)

After step (ii) or (iv) the server produces the signature s_c of 'ALADES_'. For each encoding, Step (iii) calculates the value $s' = s\alpha$. Same calculus occurs in fact when there is no encoding. The computed value is the contribution of 'LAMSADE' to the signature of the prefix of the searched field, *P*, *P'* or *P'*, if it were at the place of 'ALADES_' in the field. Most likely, we have $s' \neq s_c$. The server then continues to loop over steps (2.1) or (2.3) and (2.2). It does it, until it finds 'LAMSADE', unless there is a collision earlier. In the latter case the client determines that it is a collision and finishes the successful search for 'LAMSADE' in the received record.

2.2.4 Longest Prefix Match

For this search, the application provides a string *S* and requests every record whose non-key field *P* shares with *S* the longest prefix with respect to all the records in the file. It also requests the calculated prefix.

The basic method for both encodings is the sequential match over the successive symbols of the visited record. It is similar to the traditional search over non-encoded strings. Again, for simplicity we limit the analysis to searches of a 1-page field only. Then, first, the client encodes *S* to string $S'' = s''_1 s''_2 \ldots s''_l$ that it sends to every bucket. Next for each record in the bucket, the server matches $p"_i$ and $s"_i$ in the order $i = 0,1\ldots$ as long as $p"_i = s"_i$. When there is no more match, the server checks the prefix found against the longest known from the previous records, kept in some *match list L*. If the new prefix length is the same, it adds it to *L*, as well as the record key. If it longer, both data replace the existing one in *L*. If it is shorter finally, the server does not modify

L, and visits the next record, if there is still any. Otherwise, the bucket sends to the client the lengths of prefixes in *L*, even if *L* is empty. The client polls the servers with the longest prefix found.

The sequential search provides zero probability of collision for both encodings. For the cumulative one, it results from the central property of the algebraic signature that the change of up to *k* symbols is detected for sure for *k*-symbol signature. In this case the property plays for $k = 1$. The complexity is $O(\lceil m \rceil)$, where *m* is the expected common prefix length. As long as $m \approx 1$, the method works just fine.

However there are many application where $m >> 1$. As motivating examples, assume that the non-key field are URLs. *P* could start with 'http://www.' in every record. The speed of the sequential match would deteriorate about 10 times. Likewise, consider the relational table Hotel (Class, City, Zipcode,…) of hotels in the reach of Paris metro. The hotel records are most likely to have the prefix '**__;Paris_____;7501' that is 30-symbol long, assuming that City field is, e.g., 20-symbol long to suffice for the suburbs like Boulogne-Billancourt. Etc.

The cumulative encoding avoids the related deterioration through the algorithm that follows. One rationale is that the algorithm uses the binary search to determine the common string between *S* and the visited *P*. Such search is logarithmic in the size of the prefix, hence much faster for a longer one than the sequential match. Another rationale is that the algorithm avoids the sequential match for the prefix of the new visited record. It matches instead only the symbols *p"* and *s"* at the offset of the last symbol of the prefix already in *L*. Besides, the algorithm should perform in practice for $m \approx 1$ about as well as the basic one. We now present this algorithm. It starts similarly, i.e., the client encodes *S* to string $S'' = s''_1 s''_2 \ldots s''_l$ and sends *S"* to every bucket. Each bucket proceeds then as follows.

1) First prefix match. Find 1st record R_1 with $p''_1 = s''_1$. If the search is not successful, then exit. Otherwise, starting with $i = 1$, match p''^i_2 to s''^i_2, where $i = 1,2\ldots$ Continue as long as the match is successful or *i* becomes such that p''^i_2 would be beyond $min(p''_l, p''_n)$, the latter being the last symbol in the field. In that case match instead p''_n and s''_n. As long as the match is successful, set *current prefix* P''_c to $P''_c = p''_1 \ldots p''_i$.

Consider now that for some $i = u$ where $i \geq 2$ and $2^i \leq n$ the match is unsuccessful. For simplicity, consider now only that $min(l, n)$ is power of 2, the other case being easy to generalize to. Find p''_j where $j = (2^{u-1} + 2^{u-2})$. In other words, match symbol p''_j in the middle of the interval between $p''^{2^{u-1}}_2$ and $p''^{2^u}_2$. If p''_j matches successfully, extend P''_c with the string up to p''_j and not yet in P''_c. Then, continue the binary search within the upper half of the interval, i.e., interval $(p''_j, p''^{2^u}_2)$. Otherwise, continue within the lower half interval $(p''^{2^{u-1}}_2, p''_j)$. Each time, there is the match, extend P''_c accordingly. This until all the binary search has no more p'' to visit. Note R_1 (by its key) in the match list *L*.

2) Remaining prefixes match. For each not yet visited record, let it be R_2 proceed as follows. Let r''_c be the last symbol in current P''_c. If p''_c in R_2 does not match r''_c, then visit next record and set it as R_2 unless there is no more such records. Otherwise, perform the binary search in P_2 as described in Step (1), and as if p''_c was p''_1 in P_2 (but multiplied by α^c, more strictly speaking). However, if a non-empty matching string results from, then add it to current P''_c and replace R_1 with R_2 in *L*. Otherwise, (in the case of $p''_c = s''_c$ only), add R_2 to *L* (*ex equo* match).

3) Each bucket sends to the client the lengths of prefixes in *L*, even if *L* is empty. The client polls the servers with the longest prefix found.

4) The client may double check the result for the collision. It does it, basically by sequentially testing for each record the located string. It returns to the client only the records that pass the test. If the returned set reduces however to no records for bucket *m*, the client alerts bucket *m* for a new search. This one has to search for a shorter string, but at least as long as the longest string accepted till now. Bucket *m* could miss such a string during the 1st search in the bucket. One of the jumps calculating the extension could indeed be incorrectly accepted because of the collision, removing the actual longest prefix from consideration. The extension of the string already found by the steps above should however not be binary anymore. It should be basically sequential instead, one symbol after another. This mode makes the probability of a collision strictly zero. Other strategies with low probability, but possibly faster, obviously exist for specific cases. Hence one can attempted them as well. In any case, each bucket *m* returns the result to the client. The client repeats Step 3.

Discussion. Step 2 may only extend P''_c. Its correctness and efficiency result crucially from the property that every p'' is the algebraic signature of the prefix of *P* up to P''. As for the prefix search in Section 2.2.2, testing p'' alone typically suffices with high probability. Likewise, the use of the binary search in Step 1, obviously results from this property in fact as well.

Step 3 is optimistic in the sense it neglects the possibility that *R* contains a collision. If experiments confirm, it may happen in practice, one can trivially amend Step 3 so to check records in *R* accordingly. It may then be useful to store in *R* some next longest (but shorter than the chosen one) selections as well.

Performance of the binary search is known to be $O(\log_2 m)$, where m is the expected length of the common string examined here, i.e., $m \leq min(l, n)$. If m = 1K the search speed is thus $O(10)$. For a visited record whose prefix matches only a part of that already in *L*, or does not match it at all, the visit has the complexity of $O(1)$. This is due to Step 2 as discussed above. Hence, the time complexity per record should be between $O(1)$ and $O(\log_2 m)$. It is close to $O(1)$ for a larger bucket capacity *b*. Larger common prefix length plays otherwise, but to much lesser extend, given the logarithmic incidence. If most records have common prefix, a quite important speed up with respect to the sequential match results from. For our motivating examples, assuming, e.g, $b > 1000$, the speed-up could reach respectively almost 10 and 30 times. If there is a longer common prefix, e.g. 1K long, perhaps for fingerprinting, the comparative speed up could reach 1000 times. Hence, the algorithm should not only be usually at least as fast as the sequential search, but also should provide a much more stable search time performance.

Example 3

In the anagram file there are also the records with non-key fields (quotes are not the part of the field):

'CAIRE CERIA WITOLD'

'CAIRE CRAIE CERIA CHRISTOPHE'

'CAIRE CRAIE ICAR CERIA GERARD'

The application searches for all the records best matching the string *S* = 'CAIRE CRAIE ICAR CERIA CARIE', e.g., to find out whether anyone got all these anagrams. Assume that 1st record above is in bucket 1 and are the only records there together with 'MALADES…'. The two other records are the only in bucket 2, stored in the above order. The client sends to both buckets *S"*. In bucket 1, there is no match of $s"_1$ against $p"_1$(M) hence the rest of this record is skipped and *L* remains empty. Matching against next record leads successively to the tests of $p"_1$(C), $p"_2$(A), $p"_4$(R), and of $p"_8$(E). The last test most likely fails. Next tests is $p"_6$( ), it succeeds hence the next is $p"_7$(C). There is no more symbols to visit, hence the record enters *L* with prefix 'CAIRE C' of length 7. There is no more records to visit, hence the bucket sends its reply to the client.

In bucket 2, similar search probes first $p"_1$(C), $p"_2$(A), $p"_4$(R), $p"_8$(R), and $p"_{16}$(I) that most likely fails. It backtracks to $p"_{12}$( ) that succeeds. Next it probes hence $p"_{14}$(E), typically fails, backing thus to $p"_{13}$(C) and failing typically as well. It thus puts into *L* the key of the record, the prefix ''CAIRE CRAIE ' and its length of 12. It then moves to next record. Here it starts through Step 2, checking directly $p"_{12}$( ). Since it matches, it continues successfully with $p"_{11+2=13}$(I), then $p"_{11+4=15}$(A), and $p"_{11+8=19}$(E). Here, it cannot go to $p"_{11+16}$ as it is beyond $l = 23$ in *S*. It obviously resets the binary search in the remaining interval of 4 symbols and finds out that the record matches the entire *S*. It amends *L* accordingly. The bucket sends then its reply to the client.

Comparing both replies, the client finds out that it should poll only bucket 2. It does so, decodes the prefix and the record found and delivers both to the application.

2.2.5 Largest Common String

Here the goal is to find record *R* with the longest (sub)string of string *S* sent by the client. We only consider the cumulative encoding. One algorithm for this search is to iterate on the prefix search for each visited record R_v, shifting the search by one symbol to the right after each iteration. Namely, while positioned at a given substring in *S* we shift the search in R_v any time the current search reaches the end of R_v. When we visit in this way in R_v the rightmost possible string of the largest length *l* already determined, we shift the offset of the string to search in *S* to the right by one symbol and we restart the search from p_L in R_v. This, provided of course that *l* is smaller than the length of *S*. Whenever we find in R_v a string longer than *l* we increase *l* accordingly and we store the string and R_v key for backtracking in the case of collision. The server returns the record found, with the offset of 1st symbol of the string found and *l*. The client double-checks whether there is no collision. It proceeds similarly to the collision test for the longest prefix match in Step 4 above.

The actual complexity for R_v is somewhere between $O((n - l)(m - l))$ and $O(nm)$ considering that *n,m* are the lengths of *R* and *S* respectively. The lower bound is optimistic, assuming we reach *l* early. The upper bound is the worst, assuming we find nothing. The more precise calculus assuming the slow increase of *l* during the search seems hard and we leave it for further study. If *l* is large with respect to *n* and *m*, then the savings with respect to the basic method are substantial. We recall that that one has obviously the $O(nm)$ complexity.

It is perhaps worth to remind here that since we are in an SDDS, the formulae for the overall search complexity become $O(\lambda b (n - l) (m - l))$ and $O(\lambda bnm)$, regardless of the scale of the file (its number of record N). Here λ denotes the average bucket load factor and b is bucket capacity. In a traditional non distributed data structure, this complexity would deteriorate linearly with N. In a non-scalable parallel/ distributed data structure with k buckets it would be in the order of N/k. The difference can obviously be in the orders of magnitude in favor of an SDDS. The reminder applies of course accordingly to the other operations.

To illustrate the algorithm with an example, consider the search for S = 'CERIA CHRISTOPHE' in the file in Example 3. The search through 1st record will end up with string 'CERIA '. Once this string is identified in this record, further search in it would only examine the symbols encoding the last character encoding a string of length $l = 6$. The search in 2nd record will start at 6th symbol. It will end up after the shifts by identifying the entire string S. Hence there is no further shifts in S. The value of l will increase to 16. The search through 3rd record will start at p_{16}. It will end up without success, unless a collision occurs. The server will return 2nd record with $l = 16$, and the offset of 'C' starting the string found.

2.3 *Key Search*

There is no point to encode a meaningless key. Hence the encoding concerns only a meaningful key. If the file is hash partitioned, as for the LH* schemes, then the client hashes the encoded key value and sends the search (or insert r update) query accordingly. If the file is range partitioned, then the client uses for the addressing the actual key. It determines the adequate server (correct or not, in which case some forwarding may occur). It sends then to the server the encoded key and α. The encoding is in the symbol mode. That is, a numeric key is converted into encoded symbols, right aligned and padded with zeros at the left.

There is a large number of SDDS range partitioning schemes. We deal here basically with the RP* schemes as the model, since they are supported by SDDS-2004. Each bucket has its interval $I = (c_{min}, c_{max}]$. Key c is in the bucket only if it is in I. The interior of the bucket is structured into 4 KB pages, [DL00]. A B-tree type memory index contains pairs (maximal key, page pointer) to direct the in-bucket addressing. Key c enters the page whose maximal key is the smallest larger or equal to c. The record with c is finally searched by the traversal of the in-page sequential link between the records, maintained in the increasing key order.

The server receiving an encoded key c starts with the test whether $c \in I$. For this purpose, it decodes c one symbol at the time, starting at the leftmost one. The decoding continues only as long as the test is positive (see Example 3 below). Only one server will finally have $c \in I$. Decoding next symbol erases the previous one from the server's memory. As for the present RP* schemes, the negative test triggers either nothing (RP*$_n$) or a forwarding (RP*$_c$ and RP*$_s$). The positive result continues with the traversal of the in-bucket index. This calculus follows the same principle, of comparison using a single decoded symbol at the time. In other words, the server acts here as if it performed the trie-like search through the index [L81]. The correct page can obviously be localized that way. Finally, the in-page link traversal follows the same principle.

In this way, at any time only a single symbol remains disclosed at the server during the key search. This is in practice typically meaningless. Even if the server crashes while the search is in progress, no post-mortem dump discloses more from the bucket actual (decoded) content.

Once the traversal located the leaf of a B-tree, to find the record with the matching key, one basically compares the symbols of the searched key to the stored ones. Most often records are of variable length with no additional in-leaf indexes, hence the visits to the stored records are sequential. One typically applies then to every visited key the sequential algorithm or Boyer-Moore; or Knuth's algorithm etc. Before entering any such algorithm, one compares the lengths for variable length key fields. The encoding, especially the cumulative one, should often greatly accelerate this search. Indeed, it boils down to the prefix search above. It suffices thus typically to compare two symbols per record, or perhaps four if keys are of variable length. Only to confirm/infirm the match one has to examine as many symbols as traditionally without any of our encodings. Notice however that in our case, one may use the encoded symbols only. The function can be systematic at the server. Alternatively, one may optimistically deport it to the client. It is then involved at the server only when the client came back with the collision report.

Example 4

Consider now that 'MALADES' is the encoded primary key c. Next, that the root in the page B-tree of bucket with interval I = [IL, MEMO] has the entries:

…LAC, MA, MAL, MASSE…

For simplicity, we consider these entries decoded. The decoding of encoded 'M' will lead to this bucket only, perhaps with some forwarding for RP*$_C$ and RP*$_S$. Next 'M' leads towards first page entry with 'M' as 1st

symbol, hence ‘MA’. Symbol ‘M’ can now be discarded from the memory. Next the decoding of ‘A’ will not move a further entry. Both ‘MA’ and ‘MAL’ share indeed this symbol at 2nd position. Now, ‘A’ can be discarded. The comparison of decoded ‘L’ will move the pointer to ‘MAL’ entry. This is the final choice, as next entry has ‘S’ at the same position. Hence ‘MASSE’ is the smallest entry strictly greater than our *c*. If the tree had next index level, similar process continues using further digits.

Once in the leaf, one has to locate the ‘MALADES’ key. One may do it by individual symbol comparison, e.g., the basic one: first ‘M’, then ‘A’ if ‘M’ matched etc. If keys are stored non-encoded, and the key signature is not pre-computed and in any way, the signature based search for a non-encoded key does not seem practical. Otherwise, with the partial encoding, one may replace the character comparison by XORing. It is not sure at present that any speed-up may result from in practice. In the case of the cumulative encoding, if lengths match, one visits p''_7 and compares it to *s*, until it finds $p''_7 = \mathrm{sig}_\alpha$ (MALADES). Only in this case, the server or the client perhaps examine the key found traditionally, to eliminate collisions.

2.4 *Performance*

Functionally, with respect to the key and string search scan, both encodings offer to the application the same capabilities as the storage of the decoded data. With respect to the performance overhead, the processing of the encoding, decoding and of a search is low. In detail, we measured the time for encoding data to be under 0.045ms/KB. Decoding time will be under 0.042ms/KB. Compared to the insertion time for a record of size 1KB, the difference is about 14% and 15% for decoding and encoding respectively. Basically, a few table lookups and of XORs per symbol. The fact that the non-key string searches cost less than for non-encoded data is an especially attractive and perhaps surprising property. Finally, both encoding do not introduce any storage overhead.

With respect to mutual comparison of the partial versus cumulative encoding, the cumulative encoding has the processing overhead of one XOR per symbol of the field. If one should store anyhow the signature of the field in the GF used for encoding, then this calculus occurs anyway. The decoding of the cumulative signature costs the similar overhead. This time unavailable. For optimal performance of an application, one has to weight thus this overhead towards the benefit of faster prefix and string search.

From the security standpoint, the stored data remain encoded entirely all the time or with the transient exception of a single symbol of the key at the time. The method fulfils thus its design goal. An occasional hacker may have quite hard time to break the encoding. In contrast, the method has no pretension to resist a professional attack. One may disassembly the server program, use frequency tables, test systematically all primitive elements in the GF…

We experimented with inserting the string to search in the end of record. In another experiment, we inserted the record having the character to search in the last position of the data bucket. Searching a prefix situated in the first record of size 100Bytes takes 0.37 ms. Since a key based lookup of the record takes 0.3 ms, this result is very interesting. Time necessary to found parts of 10 bytes is about 0.6ms. These 10 bytes are placed in end of 100Byte record having size of 100B. It makes about 80 comparisons of pre-computed signatures.

Record position	Record Size	Size of text to search	Offset of text to search	Time of search (ms)
1	20	5	13	0.44
1	100	20	70	0.68
1	100	20	80	0.689
100	250	15	80	451
100	250	30	80	437
200	250	15	80	884

String Search Time

Record position	Size of records	Size of prefix to search	Time of search (ms)
1	100	20	0.369
100	250	20	37.8
100	250	35	37.7
200	250	20	71.3
300	250	20	120.53
500	250	20	197.5

Prefix Search Time

Using cumulative signatures (on 100B records) saves about 30%. We also compared the pre-computed algebraic signature search with an implementation of Karp and Rabin’s algorithm. In this experiment, we used a byte-wise XOR. This saved 3% compared to our pre-computed signature.

Size of case	Size of inserted data	Size of last record	Size of data to search	Offset in last record	Time of Algebraic Signature Search	Time of Karp Rabin Search	Time of search Cumulative Signature

100	100	25	15	5	205	151	147
200	100	25	15	5	368	275	268
500	100	25	15	5	1123	725	702
1000	100	25	15	5	2254	1580	1526

We note that the worst-case time complexity of the Karp-Rabin algorithm is quadratic (as it is for the brute force algorithm) but that its expected running time is O(m+n). Several linear time algorithms have been proposed for the problem of pattern matching. The Knuth, Morris, Pratt and Boyer and Moore algorithms require, for fast implementation, several registers to store a table of pointer O(n). Using algebraic signatures requires, like Karp-Rabin, a constant number of registers and needs a substring of length n of the text in main memory.

While our signature differs from that of Karp Rabin and its successors, it also evaluates quickly all substrings of a given length in a larger string. The large number of substrings of a given length in an SDDS file virtually guaranties collisions, but these false positives are not dangerous since each server evaluates the strings returned by the search and verifies them byte per byte.

For the longest prefix match, we made different experiments putting the longest prefix at the beginning, the middle and the end of the bucket. Pre-computed algebraic signatures turned out to be the fastest method.

Record position	Size of data inserted	Size of last record	Size of prefix to search	Offset prefix in last record	Time of search (ms)
1	50	50	25	1	0.48
1/100	250	50	25	1	380
49/100	250	50	25	1	423
99/100	250	50	25	1	453

3 Protection Against Incidental Corruption

As we already stated, the goal here is to detect the silent record corruption and identify with required precision the area where it occurred. This corruption could especially concern the non-key data, being normally unobserved by the SDDS client and server. Corruption of the key or of other internal parts of the record or of the bucket should generate a functional anomaly, important enough to be detected by the SDDS software. Ultimately, the bucket may be declared as unavailable and recovered, e.g. as in LH*$_{RS}$ scheme. Notice that the silent corruption of non-key data is particularly preoccupying in the latter case, as it may corrupt the recovered records in other buckets. This is true for any parity calculus recovery scheme.

A simple approach could be to test the record content against the algebraic signature inserted by SDDS-2004 (Section 2.2.1). This approach identifies as the corrupted area the area up to 2^{16} bytes. For practical reasons, we target smaller area sizes. At most about that of the size of a disk sector. Such granularity may be precious for the administrator in charge to fix the problem. Our basic choice is thus 255 B that is also the page area for GF (2^8). If the field is smaller, then the area size is that of the field.

To reach the goal, the client inserts into the field the *checksums*. Each checksum is the *n*-symbol signature of one area. For the partial encoding, we basically consider $n = 1$ only. It is likely enough in practice. The client computes the checksums during the encoding process by XORing the successive encoded symbols, up to the size of the area for each checksum. For 255 B area and GF (2^8), the checksum is particularly simple: it is just p''_{255}. The calculus for smaller area sizes for GF (2^8) is trivial. The situation is similar for GF (2^{16}).

The above calculus obviously does not apply to the cumulative encoding. Besides, the corruption there is more dangerous. That of any single p''_i is likely to lead to erroneous decoding of both: p_i and p_{i+1}. To detect and localize it one solution is to compute as the checksum the 2-symbol signature $s_1 s_2$ per area, where $s_1 = p''_n$ for an *n*-byte area. Only s_2 calculus and storage is then the corresponding overhead. Both encodings lead thus to the same storage overhead. They also provide the same collision probability. The processing induced by the cumulative one is however longer.

Each approach protects for sure against any single symbol corruption in the field. Including the checksum itself. It provides for the detection of a larger corruption with the already discussed collision probability. If one feels it too low, one solution is the checksums constituted from longer signatures. We recall that *n*-symbol signature detects then any *n*-symbol corruption in the area for sure, and the probability of non-detection would be $1/2^{nf}$. Better insurance costs however additional higher processing and storage premium.

When the record comes back from the server, the client recalculates the checksums during the decoding. The process is about no cost for the partial encoding. It requires the re-calculus of s_2 for the cumulative one. If every result matches, the record is likely to be OK with the error likelihood already discussed. If the match is negative, the client alerts the server, identifying the area.

Our protection against the record corruption is in this way very cheap. For the partial encoding, it costs nothing if the signature of the field should be computed anyhow. Otherwise, it costs only a couple of XORing. For the cumulative encoding it is always more involved, but remains obviously very fast. The storage overhead is basically of 1/255. In particular, for both 8-byte and 16-byte symbols, the processing is cheaper than the alternative test of the 2-symbol signature inserted automatically by SDDS-2004 we spoke about above (albeit perhaps less precise with respect to the collisions).

4 Related work

At this stage we are not aware of any specific work sharing our assumption of incidental protection. Any trivial approach we are aware of does not match our string search performance. For instance, this is the case of a table-based arbitrary substitution of characters at the client.

With respect the string matching, a number of algorithms exist. The domain was among most investigated, [CL03]. The Karp-Rabin type schemes share with ours the principle of the incremental calculus of the new value to match when the search moves to next symbol. As our schemes, through the searched string's signature calculus, several algorithms propose some pre-computing of the string to search, [C97]. Algorithms exist also to search in the compressed text, e.g., [N04] and older articles this reference refers to. Performance of those naturally seems to vary more strongly, depending on the compression efficiency. Generally, the detailed comparative performance analysis to the existing algorithms, even only best known, remains to be done. However, given the complexity and the speed of our algorithms, *they seem usually faster for the same operation than other known in our context.* Especially, the cumulative encoding, with its $O(1)$ prefix search complexity and the $O(n - l)$ string search one. All other major algorithms we know have higher complexity, of $O(l)$ and of $O(n)$ respectively, at best [CL03.

It is perhaps useful to recall our application context more in depth in the light of this enticing conclusion. One may indeed consider that the cost analysis of the existing string search algorithms implicitly considers that the application gets the retrieved record with its original content. Next, one may observe that the analysis also implicitly considered an individual record search, sitting in some buffer. If one subscribes to that point of view, one needs to add up the encoding and decoding costs to per record overall cost for fair comparison. Our conclusion does not apply anymore.

We recall therefore that our context is that of a non-key search in a large file or database. The searches are typically predominant over the inserts, e.g., by an order of magnitude at least. The typical result of the selection is the delivery of only a small fraction of the visited records, e.g., 5 %, at best. Our $O(n)$ encoding/decoding costs are cheap compared to the search costs. The predominance of searches wipes out the incidence of the encoding in per record visit cost. The high selectivity size makes negligible the incidence of the decoding cost. Our attractive complexity figures and overall conclusions fully apply.

The protection against corruption constitutes a form of server's data *scrabbing* [S&al04]. Ours occurs at the client that piggybacks it to a record access. There is no processing overhead for the server. The downside is that one detects the corruption of less used data respectively later. Finding the cause becomes then problematic. To alleviate the problem, one may add the scrubbing at the server, of all its data, at regular intervals. See [S&al04] for related trade-offs.

5 Conclusion

We have presented a method for protection of data stored in an SDDS against incidental viewing and corruption at the server. The main idea is the pre-calculus of the symbols summed up to form a 1-symbol algebraic signature. The method appears attractive. It does not imply any storage overhead. It creates only small processing overhead. Next, it notably speeds up the string search, the prefix searches especially, if one uses the cumulative encoding. The latter is in our context less complex and likely to be faster than any other known algorithm. Finally, the method may also locate the incidental data corruption at no or little processing cost, and negligible storage cost. In particular, the string search speed may be of importance *per se*, e.g., for text files, and Web applications, even if the stored data protection virtues of our method are of secondary concern.

Further work at our side will concern the implementation under SDDS-2004 and the experimental performance evaluation. One should investigate also other string manipulation functions. These can be other interesting searches, e.g., for the given or the largest common suffix. The former is obviously quite similar to the prefix

search. For the cumulative encoding the main difference is only the additional multiplication by α^{n-l} and the backward binary search (towards the beginning of the record). In the wake, one should investigate the algorithms for string updates decreasing the amount of transfers among the clients and servers, and of writes at the servers. This could make the SDDS updates sizably faster. A file or database update often concerns indeed only a few symbols. Finding efficiently the largest common prefix and suffix in the before and update images, could let the client to transfer only a small part of the record effectively changed. Likewise, the server could have much less to write back to the storage.

Consider for instance the record of a person containing a photo as prefix, then some traditional ASCII and numerical personal data, and finally some biometric ID. Both, photo and biometric ID are rarely updated. But, usually they require orders of magnitude more symbols than the traditional personal data. The cumulative encoding could help to efficiently filter such data out of the updates, without knowing anything about the related higher level semantics.

The speed of string matches of the cumulative signature seems to open an interesting perspective for the popular Join, Group by, Rollup and Cube database operations. Finally, one could investigate the port of our encoding ideas to the Karp-Rabin original hash function as well.

Acknowledgements

We thank Jim Gray for comments on early ideas and the support for this work through MS grant. We are also grateful to Lionel Delafosse for comments on the related work.

References

[CL03] Crochemore, M. and Lecroq, T., Pattern Matching and Text Compression Algorithms, in *The Computer Science and Engineering Handbook*, A.B. Tucker, Jr, ed., CRC Press, Boca Raton, 2003, Chapter 8.

[DL00] Diene, A., W., Litwin, W. Performance Measurements of RP* : A Scalable Distributed Data Structure For Range Partitioning. 2000 Intl. Conf. on Information Society in the 21st Century: Emerging Techn. and New Challenges. Aizu City, Japan, 2000.

[G88] Gray, J. The Cost of Messages. 7th CAM Symposium on Principles of Distributed Computing, ACM Press. Aug. 1988, 1-7.

[KR87] Karp, R. M., Rabin, M. O. Efficient randomized pattern-matching algorithms. IBM Journal of Research and Development, Vol. 31, No. 2, March 1987.

[L81] Trie Hashing. ACM-SIGMOD 1981 Intl. Conf. On Management of Data. Ann Arbor, Etats-Unis (May, 1981), 19-29.

[LMS04] LITWIN, W., MOUSSA, R., SCHWARZ, T., LH*$_{RS}$: A Highly Available Distributed Data Storage System. Intl. Conf. On Management of Very Large Databases, VLDB-04, Toronto 2004.

[LNS94] Litwin, W., Neimat, M-A, Schneider, D. RP*: A Family of Order-Preserving Scalable Distributed Data Structures. VLDB-94.

[LNS96] Litwin, W., Neimat, M-A, Schneider, D. LH* - A Scalable Distributed Data Structure. ACM Transactions on Data Base Systems. December 1996.

[LS04] Litwin, W., Schwarz, Th. Algebraic Signatures for Scalable Distributed Data Structures. IEE Intl. Conf. On Data Eng., ICDE-04, 2004.

[C04] SDDS-2004 Prototype. http://ceria.dauphine.fr/.

[NT04] Navarro, G.,Tarhio J. LZgrep: A Boyer-Moore String Matching Tool for Ziv-Lempel Compressed Text. Software Practice and Experience (SPE), to app.

[S&al04] Schwarz, T., Xin, Q., Miller, E., Long, D., Hospodor, A., Ng, S.: Disk Scrubbing in Large Archival Storage Systems. 12th IEEE International Symposium on Modeling, Analysis and Simulation of Computer and Telecommunications Systems (MASCOTS) 2004.