String-matching and update through Algebraic Signatures in Scalable Distributed Data Structures
Riad Mokadem, Witold Litwin
Dauphine University, Paris.
Riad.Mokadem@dauphine.fr, Witold.Litwin@dauphine.fr

Abstract:
Scalable Distributed Data Structures (SDDSs) store large scalable files over a distributed RAM of nodes in a grid or a P2P network. The files scale transparently for the applications. The prototype system, designed by CERIA, experiments with this technology for Wintel multicomputers. The application may manipulate data much faster than on local disks.

We present the functions we have put into the prototype we now call SDDS-2004. We improve the searches and updates of records in our SDDS files. An original property of these functions is the use of the algebraic signatures. This technique serves the distributed non-key record search. The search may concern the entire field or a (sub)string. The algebraic properties of the signatures act similarly to hash schemes in [KR87]. In particular, sending a few-byte signature of the searched string alone, suffices for the search. This makes the communication between the SDDS client and server more efficient. It is also more confidential, since the signature in an intercepted message does not disclose the searched string. On the other hand, we use the signatures for the update management. The clients do not need to then to send updates which in fact do not change the stored records. Finally, our signatures help managing the concurrency control. We present our architecture and design choices. Performance measures validate our implementation. It is now available for download in site of CERIA.

1. Introduction
Scalable Distributed Data Structures (SDDSs) store large scalable files over a distributed RAM of nodes in a grid or a P2P network. The files scale transparently for the applications. The prototype system termed SDDS-2004, designed by CERIA experiments with this technology for Wintel multicomputers. Experiments with the SDDS files over 1.8 GHz Dell nodes linked by the 1 Gbs Fast Ethernet at CERIA show the successful key search time for the 100-byte record under 30 μs. That is thus about 300 times faster than the typical disk search time of 10 ms.

It appeared useful, experiments with the version termed SDDS-2003, to have new update and search records operations. It was better to update only the record that has effectively changed with respect to its before image in the SDDS bucket. One has to then somehow compare at the client, the before and after images of record. On the other hand, we wished to provide the non-key searches (parallel scans).

A signature scheme guarantees that a rapidly computed and a few-byte long signature of different data units, e.g., our records or parts, differ at least almost surely. The SHA-1 standard is perhaps the most used [M2a]. An original property of algebraic signatures is that the units may differ for sure, [LS2], [LS3]. The algebraic signature may also be shorter than SHA-1. These signatures were therefore our choice. For the update management, we basically compare the signatures of the before and after images. For the search, we use them in a more complex the way we describe below.

We consider the reader familiar with the theory of the algebraic signatures. In what follows, we present our implementation. We justify it through the experimental performance analysis.

Section 2 overviews our new SDDS scan capabilities. Section 3 overviews the update processing. Section 4 discusses the performance analysis. Section 4 presents the conclusions.

2. SDDS-2004 Scan Capabilities

2.1. System Architecture
Our scans perform the string matching operations. These match the occurrences of a string of symbols over a finite alphabet, termed pattern in a larger string, over the same alphabet, usually a text. In SDDS 2004, the client initiates any scans, on behalf of the application that provides the patterns. The client calculates then the algebraic signature of the pattern. It then sends it out, together with the indication of the search type detailed below to all the servers in parallel. The servers that find the matching records send the records back. The servers evaluate the match against the non-key fields of the records. The client matches the received records against the original pattern to eliminate collisions. It provides the final result to the publication.
The client sends the command using a UDP multicast request. Each server processes the command and acknowledges the execution to the client. Even, if it does not find any relevant records. The acknowledgements include the RP* range of each bucket (we recall that an SDDS client may not be aware of all the existing servers). If the client finds that all the buckets that should have replied, it informs the application about the successful termination. Otherwise, it starts some recovery action.

2.2. Scan Processing
There are two kinds of pattern search scans. We call them complete search and partial search. The complete search matches the records where the pattern constitutes the entire non-key field. A partial search searches for every record with the pattern in it.

The complete search is straightforward. The client calculates the signature of pattern and sends it to the servers. Each server visits every record and compares the received signature with that in the non-ky field in the record. The latter was calculated at last update and stored with the record.

The partial search is more involved. The client starts with the calculation of the pattern signature. It then sends it to servers. For the partial search, the client also may ‘XOR’ the symbols in the pattern. It sends then the value together and the pattern size together with the signature.

The basic pattern matching uses only the algebraic signatures. It compares the received one to the string of the symbols in the record of the size of the pattern. A mismatch shifts the comparison to the right by one symbol at the time and computes the new signature. The process is somehow similar to that of the Karp-Rabin (KR) algorithm, [KR87]. Shifting the search to the right by one symbol does not require to recomputed the whole signature. We therefore compare later our algorithm to one of better known versions of the latter algorithm.

The other implemented method observes that ‘XOR’ calculus is a faster comparison method, though less precise in turn. Each server calculates thus first the moving ‘XOR’ in the record, comparing it to ‘XOR’ received. Only when there is a match, the server calculates the algebraic signature. If these match, then the search succeeds at the server. The ‘XOR’ calculus is well known. Here we discuss the algebraic signatures only.

2.3. Algebraic Signature Calculus
The algebraic signature is a specific power series in a Galois Field GF (2\(^f\)), with \(f=8,16\) typically, [LS3]. Addition and multiplication in a GF are associative, commutative and distributive. We compute the addition \(a + b\) as XOR of the (symbols) representing these elements. We multiply \(a \cdot b\) and \(a, b \neq 1\) using the log/antilog calculus:

\[ a \cdot b = \text{antilog} (\log_a a + \log_a b) \bmod N. \]

Here \(N = 2^f - 1\) and \(a\) is a primitive element. We recall that in a GF, every \(a \neq 0\) is \(a^i\) form some \(i = 0,1,...,f-1\). The log and antilog values are respectively tabulated. The size of each table can be \(N - 1\). To avoid the mod \(N\) calculus, one can also double the log table to the size of \(2^f\) in practice.

We calculate the algebraic signatures as follows. We consider a string \(P\) called generically page as a vector of symbols \(p_1, p_2, ..., p_n\) of the GF used. This is GF(2\(^f\)) for partial search and GF(2\(^{16}\)) otherwise in our case. Each \(p\) is thus one byte or a two-byte word.

Let \(p'\) such that \(p = \log_a p'\). Let \(\alpha = (\alpha_1, \alpha_2, ..., \alpha_n)\) be the vector of different non-zero elements of the GF. We consider here specifically that \(\alpha_1\) is primitive. For \(i = 2, ..., n\), we have \(\alpha_i = \alpha_1^i\). Let it be then for each \(\alpha \in \alpha\):

\[ \text{Sign}_a(P) = \Sigma p^i \alpha^i : i = 1,2,..,n. \]

Then, the \(n\)-symbol algebraic signature of \(P\) is the vector:

\[ \text{Sign}_a(P) = (\text{Sign}_{a_1}(P), \text{Sign}_{a_2}(P), ..., \text{Sign}_{a_n}(P)). \]

The use of \(p'\) in the formulae is purely formal. In practice, we consider the symbols \(p\) directly as logarithms. In other words, we do not calculate \(p'\), but, for each \(\alpha_j, j = 1,..,n\), we directly add \(p_i\) to \(\log_a \alpha_i\). This speeds up the calculation, with respect to the direct application of the signature definition in [LS3].

Likewise, a natural approach to \(\text{Sign}_a(P)\) calculus is to compute \(\text{Sign}_{a_1}(P)\), then \(\text{Sign}_{a_2}(P)\) etc. In our case, it appeared notably faster to calculate the contribution of \(p_1\) to \(\alpha_1\), \(\alpha_n\), followed by this of \(p_2\) to \(\alpha_2\), \(\alpha_n\) etc. This approach was our final choice. The reason is likely the influence of L1 and L2 caches.

The crucial property of the algebraic signatures for our application is that the probability that two objects differ by only few symbols have same signature, i.e. they collide, can be made negligible. In general, that probability is \(2^{-n}\), or even zero, [LS3]. Besides, as long as \(P_1, P_2\) do not differ by more than \(n\) symbols, collision probability is zero, provided the every \(P\) size is under \(2^f - 1\) symbols for GF (2\(^f\)) used. This property is at present unique to algebraic signatures.

For the complete search and update management, we used the 2-symbol GF(2\(^{16}\))
signature. For the partial search, our signatures are basically 1-symbol and over GF(2^8), assuming ASCII code or similar. The collision probability is usually 2^{-32} or 2^{-8} for the partial search. It would be 2^{-16} for Unicode 2-byte symbols, in the latter case.

3. Update in SDDS-2004

3.1. Overall description

An SDDS update operation manipulates the non-key field of a record. Several clients may attempt to read or update concurrently the same record. It is best to let every client read any record without any wait. The subsequent updates should not however override each other. Our approach to this classical constraint is inspired by the optimistic option of the concurrency control of Ms Access. Our particularity is the use of the algebraic signatures, to compare the before and after images, at the clients and servers. The schemes depend on the update type that can be as follows.

3.2. Normal update

After search request, when record is found, the application may update it. A normal update requests from the client a key search operation and eventually receives the record \( R_a \). Application returns to the client \( R_b \) and \( R_e \). Client computes the signature-after, \( S_a \), and the signature-before, \( S_b \) of \( R_b \) and \( R_e \) resp. Then, the client compares \( S_a \) to \( S_b \). We have two cases:

- The client finds \( S_a = S_b \). We have an update without any effective change, i.e., a pseudo-update. Pseudo-updates terminate at the client.
- The client finds \( S_a \neq S_b \). It is an effective update, with change. The client sends \( S_a \) and \( R_e \) to the server. The server compares the signature of the record at the server to \( S_b \). Then:
  - Either both signatures match. No other client has updated the stored data since the client got the record. The server allows for the update and sends the positive ack.
  - Or, the signatures differ. Thus, another client has updated the data concurrently. The new update is not allowed. The client gets a negative ack.

3.3. Blind update

We also allow the blind updates. An update comes then from the application, without any search for the existing before image. The client computes \( S_a \) and sends the key of \( R_e \) to server requesting the signature \( S \) of the stored record. The server computes \( S \) and sends it. The client uses it as \( S_b \) and proceeds as for the normal update. Calculating and sending \( S \) alone as \( S_b \) already avoids the transfer of \( R_e \) to the client. The savings can be substantial, e.g., for a surveillance camera.

4. Performance Analysis

4.1. Overall description

Our clients and servers were 1.8 GHz P4 PCs under Windows 2000 Server. A 1Gbs Ethernet linked the machines. The calculus and insertion of data were for strings of less or more random ASCII characters. Our experiments showed that the calculation of signatures using GF(2^{16}) turned out to be slightly faster than for GF(2^8). In any case, the time to calculate a signature was under 5 usec/ Kb.

4.2. Sub(string) search

Each non-key text had size \( T \) of 24 bytes or 60 bytes. The last record of the bucket had in fact the text of size of 16 bytes only. For the partial search, the pattern had 6 bytes. Table 1 presents the results for various bucket sizes.

<table>
<thead>
<tr>
<th>Bc</th>
<th>CST (ms)</th>
<th>PSTX (ms)</th>
<th>PSTA (ms)</th>
<th>PSTX (ms)</th>
<th>PSTA (ms)</th>
</tr>
</thead>
<tbody>
<tr>
<td>99</td>
<td>4.2</td>
<td>8.80</td>
<td>17.4</td>
<td>17.42</td>
<td></td>
</tr>
<tr>
<td>199</td>
<td>8.5</td>
<td>18.3</td>
<td>34.53</td>
<td>34.8</td>
<td></td>
</tr>
<tr>
<td>990</td>
<td>49</td>
<td>100</td>
<td>182.3</td>
<td>184.1</td>
<td></td>
</tr>
<tr>
<td>7990</td>
<td>439</td>
<td>842</td>
<td>1503</td>
<td>1516</td>
<td></td>
</tr>
</tbody>
</table>

Be: Bucket Capacity (Record), CST: Complete Search Time, PSTX: Partial Search Time(Alg Sign + Xor), PSTA: Partial search time(Algebraic Sign alone)

Table 1: Partial and complete Search Time using Algebraic Signature.

Complete search takes always less time as it should. For the partial search, we experiment with the Xor calculus and with the matching using the algebraic signatures alone. The search time using ‘Xor’ is about always slightly faster. The partial search time increases linearly with the record size. Globally, the search speed was about 2 sec per MB (time to traverse from record to record within the bucket not included, e.g., 0.5sec. for 7990 records).

4.3. Comparison to Karp Rabin:

We search here in records having size of 950 byte ASCII characters. The last record of the bucket has the text of 60 bytes. The pattern to search was of 20 or 40 bytes. Our results are in Table 2.

<table>
<thead>
<tr>
<th>Bc</th>
<th>STP (ms)</th>
<th>ST (Byte)</th>
<th>PST (ms)</th>
<th>PKRT (ms)</th>
</tr>
</thead>
<tbody>
<tr>
<td>100</td>
<td>99</td>
<td>20</td>
<td>625</td>
<td>457</td>
</tr>
<tr>
<td>500</td>
<td>299</td>
<td>20</td>
<td>3552</td>
<td>2380</td>
</tr>
<tr>
<td>100</td>
<td>99</td>
<td>40</td>
<td>492</td>
<td>560</td>
</tr>
<tr>
<td>500</td>
<td>299</td>
<td>40</td>
<td>2806</td>
<td>2987</td>
</tr>
</tbody>
</table>

Be: Bucket Capacity (Record), STP: String to search position, ST: Size of text to search, PST: Partial search time (Alg Sign Scheme), PKRT: Partial search time(KR Scheme)

Table 2: Partial Search Time using karp Rabin Algorithm performance Analysis.
The version we used performs the multiplications by power of 2 using the left shifts. The power is the pattern length. The shift time is about linear with the power. For shorter pattern, of length 20 here, the time to shift is small enough to make KR faster. Beyond, our search becomes faster. The speed-up is about 10% in Table 2, i.e. for a 40-byte pattern.

4.4. Update performance

A normal update took 0.92 msec per 1KB record. A pseudo-update took only 0.28 msec per 1KB record. The saving amounts to about 70% of the normal update time. These times include the time it takes to access the record at the server. Processing times for blind updates are: 0.74 msec for the actual update and 0.22 msec for a pseudo-update. The saving for normal pseudo update is also about 70%. The times include the key search, the update processing, and the transfer of the record signature. For smaller records, of 100B, all times are faster, but the savings for pseudo-updates decrease to about 50%. This is still a sizable gain.

5. Conclusion and Future Work

Our implementation is now integrated in the SDDS-2004 prototype. Our future work addresses further applications of algebraic signatures for SDDSs. On the one hand, we plan to encode the records using so-called cumulative algebraic signatures, [LMS5]. The method provides the incidental disclosure protection and further accelerates pattern matching as well as other types of string search. Our signature based update scheme holds, on the other hand, the promises of interesting possibilities for transactional concurrency control, beyond the mere avoidance of lost updates. It should prove useful in the client-server based database systems in general.

Acknowledgements

This work was partly supported by the research grants from Microsoft Research, and from the European Commission project ICONS project no. IST-2001-32429, as well by the SCU IBM grant 41102-COEN-RSCH-JG-IG09.

References

[CL] Crochemore M, Lecroq.T. Pattern matching and text compression algorithms. Institute Gaspard Monge