Abstract:

A Scalable Distributed Data Structure (SDDS) are class of data structures proposed to store a large scalable file over a distributed RAM. The file scales up transparently for the application over the nodes of a local network of PCs. The prototype system termed SDDS 2004, 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 added to SDDS-2003 to search and update text stored in records of our bucket structure. We work on the use of signatures for the distributed non-key record search, including the partial (string) search. The algebraic properties of the signatures play there to some extent similarly to properties of hash schemes in [KR87]. For search part of text, client doesn’t need to send all text to servers. It concerns only a signature of this text. There are many other string matching algorithms. The most efficient are probably Boyer-Moore and Karp Rabin algorithms. We interest also to update this record. In servers, several clients may attempt to read or update concurrently the same SDDS record. It is best to let very client read any record without any wait. The subsequent updates should not however override each and our algebraic signatures are useful in this context. We experiment for this purpose with the algebraic signatures. We present our architecture and design choices. Performance measures validate our implementation. It is now available for download in site of CERIA, as part of new version of our prototype termed SDDS-2004.

1. Introduction

In our prototype system termed SDDS-2000, each server node keeps the SDDS records in the RAM storage area termed bucket. It appeared useful to have new insertion, update, search or delete records in our file. For the efficiency of this process, it was clearly best to update only data that has changed in the bucket in opposite of the automatically update. The usual way to detect the record modification is to have the before and after image of record. Also, we can be able to search part of text in the non-key portion of a record. This search can take less time.

To add the search and update services in this way to SDDS-2003 turned out to be impossible in practice. The service was to be added to the existing complex code, built over years by different designers. It appeared a daunting task to properly identify all the pieces of the code that write at some point to the bucket. These should be updated to write the dirty bits as well. An approach not modifying the existing code, but only adding a new one was necessary.

We recall that a signature calculus scheme guarantees that a rapidly computed and a few-byte long signature computed for a modified data unit, e.g., our page, differ from the original signature of the unit for sure or at least almost surely. We may compute then the record signature at the search only and compare it to signature of text to search or its signature in our record’s structure. We then update the record if the signatures differ. We can perform the whole search service by an additional software module. Without any change to those already working. The same principle is apply to the update system.

Many signature schemes are known. The SHA-1 standard is perhaps the most used [11]. For our purpose, the algebraic signatures...
appeared nevertheless more practical [LS2], [LS3]. To implement these in practice required various design choices, e.g., over the page size and signature length. In what follows, we present our implementation and report on our final choices. We justify them through the experimental performance analysis. We consider the reader familiar with the theory of the algebraic signatures [LS2]. We used also algebraic signatures for search complete and partial text in data bucket. We also update only data have been changed because of the using the algebraic signatures.

The application manipulates data much faster than on local disks. 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 $30 \mu s$. That is thus about 300 times faster than the typical disk search time of 10 ms.

Section 2 overview our scheme. and the search service with comparison to other systems Section 3 overview our update system. Section 4 discusses the experimental performance analysis validating our design. Section 4 presents the conclusions and directions for further work.

2. The SDDS-2004 Search Scheme

2.1. System Architecture

String matching consists in finding one, or more generally, all the occurrences of a pattern in a text. The pattern and the text are both strings built over a finite alphabet (finite set of symbols). We denote pattern by $x=x[0...M-1]$. Its length is equal to $M$. The text is denoted by $y=y[0...N-1]$. Its length is equal to $N$.

We use algebraic signature for search of text store in records compose the data bucket. The client can search text locate in servers. Client can’t need to send all text (N symbols) to servers. Signatures are just calculated in client, sent to servers and then compare to signatures calculate in the servers.

For each insertion of records to data bucket in servers, signature of record is inserted in byte precede the non key data.

In Sdds interface, we can search by key of records or text. For the next case, we have two kind of search: complete search and partial search.

2.2. Text search

In SDDS-2004, we offer to the application tree new commands. These are the two-search command looks for all records to search string containing somewhere in non-key field. There are two possibilities: Complete search and partial search. Partial search command serves to search part of record and the complete search-to-search string in different records.

The SDDS client sends only length and signature of string to search to all the servers. The SDDS servers receive the signature and then calculate individually the signatures of all the substrings of the correct length in their collection of records. They send back all records with such a string and result of search to the client.

The SDDS client sends the search command for a given file $F$ using a UDP multicast request to all the servers. Those that carry a bucket of $F$ process the commands and acknowledge the execution to the client. The acknowledgements include the RP* range of each bucket (we recall that an RP* scheme range partitions the file). The client unions all the ranges to find out whether all the servers of $F$ that should reply did it (we recall that an SDDS client may not be aware of all the existing servers). This is
the case only if the union reaches the whole key space of $F$. In this case, the client informs the application of the successful termination. Otherwise, messages are resent to the missing servers after a timeout and if some still do not reply, a server recovery action may start.

### 2.3. Complete search

In case of complete search, client calculate signature of text to search. This signature is sent to server. He covers all records and compare signature received from client with signatures saved in $Sg$. In case of equality, server sent the key of record where the equality found.

### 2.4. Partial search

Client calculate signature of text to search then send this signature and size of text to search to servers. In partial search, client also calculates the Xor of text to search and send it to servers.

As the Xor calculation is the fast method to comparison, server calculates first Xor of record, compare with the Xor he received. In case of equality, server calculate the algebraic signature and only in case of equality, the search is succeed. In other case (not equality of Xor), we don’t need to calculate algebraic signature. We cover all the records by calculate the Xor and eventually signature of next partition with size has been received. The coverage concerns the other records in case of failure of partial search.

### 2.5. Algebraic Signature Calculus

The algebraic signature is a specific power series in a Galois Field (GF), [LS3]. A Galois field (GF) is finite field. Addition and multiplication in a GF are associative, commutative and distributive. We will now show our calculus of such signatures. We recall that a GF ($\mathbb{N}$) is an algebraic structure with $N$ elements, including elements 0 and 1 with usual properties with respect to the addition, subtraction, multiplication and division in the GF. GF ($2^f$) with $f=8, 16…$ are especially useful. In this case the addition $a+b$ is usually computed as XOR of the bytes or words (symbols) representing the elements. The multiplication $a*b$, or $ab$ as usual in short, is often implemented as the calculus:

$$ab = \text{antilog} \left( (\log_a a + \log_a b) \mod N \right)$$

where $\alpha$ is a primitive element and $a, b \neq 0$ and $a, b \neq 1$. We recall that in a GF, every $a \neq 0$ is $\alpha^i$ for some $i = 0, 1…f-1$. The log and antilog values are then tabulated in tables. The size of each table can be of $N–1$ symbol. To avoid the mod $N$ calculus, one can also double the log table to the size of $2N$ in practice. These calculus methods for the addition and multiplication were or final choice for our implementation. We have also chosen to use GF ($2^{16}$).

We use the algebraic signatures for our purpose as follows. Let $P$ be a page. We consider $P$ as a vector (1-d array) of symbols $p_1, p_2, …, p_n$ of the GF used, GF($2^{16}$) in our case. Technically, each $p$ is one byte or a two-byte word (our case) of $P$. Let $p'$ denote the element of GF used 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 and for $i = 2..n$, we have:

$$\alpha_i = \alpha_i^{-1}.$$  
Let it be for each $\alpha \in \alpha$ :

$$\text{Sign}_\alpha(P) = \sum p_i \alpha_i : i = 1, 2…n$$

Then, the ($n$-symbol) signature of $P$ is the vector denoted $\text{Sign}_\alpha$ :

$$\text{Sign}_\alpha(P) = (\text{Sign}_{\alpha_1}(P), \text{Sign}_{\alpha_2}(P), …, \text{Sign}_{\alpha_n}(P)).$$

We calculate $\text{Sign}_\alpha(P)$ using the log/antilog multiplication calculus. The use of $p'$ in the formulae is purely formal. In practice, we consider $P$ symbols $p$ directly as logarithms. In other words, we do not calculate $p'$, but, for each $\alpha_i$, $j = 1..n$, we directly add $p_j$ to $\log_{\alpha_i} \alpha^j$. This speeds up the calculus, with respect to the direct application of the signature definition in [LS3].

3
Likewise, a natural approach to $\text{Sign}_\alpha(P)$ calculus is to compute $\text{Sign}_{\alpha_1}(P)$, then $\text{Sign}_{\alpha_2}(P)$ etc.

In our case, it appeared notably faster to calculate the contribution of $p_1$ to $\alpha_{1\ldots\alpha_n}$, followed by this of $p_2$ to $\alpha_{1\ldots\alpha_n}^2$ etc. This approach was our final choice. The reason is likely the influence of L1 and L2 caches.

The crucial propriety of the algebraic signature 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 or even zero, [LS3]. Larger $n$ is, smaller is the collision probability is zero, provided the page size under $2^f - 1$ symbols for GF ($2^f$) used. This property is at present unique to algebraic signatures.

For our purpose, the use of 2-symbol (4-byte) signatures sufficed. All things considered with respect to the page granularity, we have also set up our system for data area pages of 16 Kbytes. For the index, pages of 256 Bytes sufficed. As long as $P_1, P_2$ do not differ by more than $n$ symbols, then collision probability is zero, provided the page size under $2^n - 1$ symbols for GF ($2^n$) used. This property is at present unique to algebraic signatures.

For our purpose, the use of 2-symbol (4-byte) signatures sufficed. All things considered with respect to the page granularity, we have also set up our system for data area pages of 16 Kbytes. For the index, pages of 256 Bytes sufficed. As long as the RAM page $P_1$ and its previous disk image $P_2$ differ thus at most 2 symbols, the collision probability is zero. If the application may make them differ arbitrarily, this probability is $2^{-32}$. In general, that probability is $2^{-n}$.

2.6. Karp-Rabin algorithm

Algorithm align first the left ends of the pattern and the text, then compare the characters of the text aligned with the characters of the pattern and after a whole match of the pattern or after a mismatch they shift the pattern to the right. It repeats the same procedure again until the right end of pattern goes beyond the right end of the text.

For a word $a$ of length $m$ and $q$ the biggest integer (32 bytes), let sign ($a$) be defined as follows:

$$\text{sign}(a_0, a_{n-1}) = (2^{m-1}a_0 + 2^{m-2}a_1 + \ldots a_{n-1}) \mod q.$$ 

During the search for the pattern $x$, it is enough to compare sign ($x$) with sign ($y_{i\ldots i+m-1}$) for $0 \leq i \leq n-m$. If an equally is found, it is still necessary to check the equality $x=y_{i\ldots i+m-1}$ symbol by symbol.

Also, It is so easy to calculate consecutive n-gram. Then

$$\text{Sign}_{\alpha}(a_0, a_1, \ldots, a_n, a_{n+1}) = \sum a_i \alpha^n = \text{Sign}_{\alpha}(a_0, a_1, \ldots, a_n) \alpha + a_{n+1} \quad \text{(i=0\ldots n+1)}$$

Also we can used another algorithm using prime number.

3. Update in SDDS-2004

3.1. Overall description

An SDDS update operation manipulates the non-key part of record. Typically, the application nevertheless requests the update from the data management system that typically executes it. Furthermore, on the server side, several clients may attempt to read or update concurrently the same SDDS record $R$. 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 freely inspired by the optimistic option of the concurrency control of Ms Access. Also, application requests a record update, but the new and the old record are in fact identical. We used principle of image of the before and the image after to detect this problem and avoid unnecessary traffic. The signatures for SDDS updates are useful in this context. Our images are algebraic signatures.

If transactions follows the two-step model, we can prevent dirty reads by calculating the signatures of the read set between reading and just before committing the writes. Then, we predict two functionalities: normal and blind update. For the normal operation, we launch request of update after confirmation by client that the record to update exist in SDDS bucket. In the blind update, we search record and update without certainly that it exist.

Since concurrence of our environment, we assure there are not lost of update after
concurrently one (equivalent to level 1 in SQL2 or level of control of concurrence in Ms Access).

Let view an example when the use of our approach is useful. Let \( R_b \) denote the before-image of record \( R \) and \( S_b \) its signature. The before image is the content of record \( R \), subject to the update by a client. The result of the update of \( R \) is the after image that we call \( R_a \) and its signature \( S_a \). The update is normal if \( R_a \) depends on \( R_b \), e.g., \( \text{Salary} := \text{salary} + 0.1 \times \text{sales} \). The update is blind if \( R_a \) is set independently of \( R_b \), e.g., if we request \( \text{Salary} := 1000 \) or if a house surveillance camera updates the stored image. The application needs \( R_b \) for a normal update but not always for a blind one. In both cases, it is often not aware whether the actual result is effectively \( R_a = R_b \).

### 3.2. Principle of normal update

After search request, when record is find, we have via interface of application the choice to throw (launch) operation of update. When application launch operation of normal update, the new data is sent to client which calculate signature-after of this data. It send key of record to server. That one calculates search the record and calculate signature-before of data and sent it to client. Then, the client compares this signature with signature-after he was calculated. We have two cases:

- Signature-Before is equal to signature-after. In this case, we are in update without change. Client sent message to inform application.

- Signature-before differ from signature-after. We are in case of update with change. Client send signature-after to server. The server calculates a new signature-server of data present and compare to signature-before received from client.

We have two cases:
- Signature-server is equal to signature-before. In this case, data was not changed per another client. Update is allowed.
- Signature-server differs from signature-before. In this case, another client changed data. Update is not allowed.

### 3.3. Principle of blind update

We also ran experiments on a modified SDDS2000 implementation that uses signatures to distinguish between updates that in fact change the record and those that do not. The later is a “pseudo-update”. We did this for blind updates. Application send request of update directly. It send key of record to update to the client. The client calculates the signature-after and then send key to server. At that level, server search record concerned and when existed, calculate signature-before and send it to client. Client compares it with signature-after.

We have two cases:

- Signature-after is equal to signature-before. In this case, there is not change.
- Signature-before differs from signature-after. In this case, there are changes. Client send signature-after to server. Server calculate signature and compare it to signature-before.

Two case exist:
- Signature-server is equal to signature-before. In this case, another client does not change data.
- Signature-server differs from signature-before. In this case, another client changes data. Change is not authorised.

### 4. Experimental Performance Analysis

#### 4.1. Overall description

In our search and update application, we usually calculate and send signatures of
records. We can tune the signature calculation as:

```c
GFElement signature (GFElement *page, int pageLength)
{
    GFElement returnValue=0;
    For (int I=0; I< pageLength;I++) {
        If (page [I]!=2f-1)
            ReturnValue+=antilog [I+page [I]];
    }
    return returnValue;
}
```

The signature schema in our case makes sense only if it saves time with respect to the straight disk write of the entire bucket. The algebraic signatures themselves are new and the scheme makes sense only if it is more efficient in our case than a known one. Regardless of the signature scheme used, in our case our total time results from that (i) to calculate and compare all the signatures and from that (ii) to write all the modified pages. First, this time is interesting by itself. Logically it should decrease when there are more server, the interesting question being how.

Next, our scheme makes sense if we typically need less time for the file write than for the straight bucket(s) write. Our fastest case is when we only calculate the signatures, i.e., it appears that no page was changed. Our worst case is when we perform this calculus and write everything. The intermediate case can be, e.g., assuming that a small part of the bucket has changed, like 5 % of the file. The analytical calculus of such times for various bucket sizes, given CPU and disk speeds, the number of servers for the file etc. appears unfeasible in practice. We have therefore performed the experimental analysis we present now.

Our server, client, application and name server were four 1.8 GHz P4 PCs under Windows 2000 Server. A 1Gbs Ethernet linked all the machines.

In details, we measured the time to calculate a signature to be under 5 usec/Kb of data on 1.8 GHz pentium4 machine.

### 4.2. Text search experimental performance analysis

With parallel insertion, we insert part of record to bucket. Each text inserted has size of 24 byte and 60 in next experience. Then, we insert at the last record of this bucket a text with size of 16 bytes. After this, we search the partial or complete text. The text we search has size of 6 bytes. Complete search take less time since search concern signature store in structure of records.

<table>
<thead>
<tr>
<th>Bucket capacity (Record)</th>
<th>String to search position</th>
<th>complete search time 24bytes</th>
<th>Partial search time (first Xor + sign if equality) 24 bytes</th>
<th>Partial search time (Sign alone)</th>
<th>Partial search time (first xor+sign if equality) 60bytes</th>
<th>Partial search time (sign alone 60bytes)</th>
</tr>
</thead>
<tbody>
<tr>
<td>100</td>
<td>99</td>
<td>4.26</td>
<td>8.80</td>
<td>9.15</td>
<td>17.4</td>
<td>17.42</td>
</tr>
<tr>
<td>200</td>
<td>199</td>
<td>8.58</td>
<td>18.32</td>
<td>19.005</td>
<td>34.37</td>
<td>34.8</td>
</tr>
<tr>
<td>300</td>
<td>299</td>
<td>13.4</td>
<td>27.2</td>
<td>28</td>
<td>52.4</td>
<td>52.49</td>
</tr>
<tr>
<td>400</td>
<td>399</td>
<td>18.4</td>
<td>36.9</td>
<td>37.6</td>
<td>71.2</td>
<td>70.05</td>
</tr>
<tr>
<td>500</td>
<td>499</td>
<td>23.9</td>
<td>47.92</td>
<td>49.05</td>
<td>90.37</td>
<td>89.34</td>
</tr>
<tr>
<td>1000</td>
<td>990</td>
<td>49.1</td>
<td>100.8</td>
<td>101.1</td>
<td>182.3</td>
<td>184.189</td>
</tr>
<tr>
<td>1500</td>
<td>1490</td>
<td>78.8</td>
<td>159.6</td>
<td>158.7</td>
<td>268.5</td>
<td>267.1</td>
</tr>
<tr>
<td>2000</td>
<td>1990</td>
<td>105</td>
<td>203.7</td>
<td>204.6</td>
<td>370.3</td>
<td>367.6</td>
</tr>
<tr>
<td>4000</td>
<td>3990</td>
<td>212</td>
<td>411.3</td>
<td>412.1</td>
<td>751.2</td>
<td>750.2</td>
</tr>
<tr>
<td>8000</td>
<td>7990</td>
<td>439</td>
<td>842.5</td>
<td>844.3</td>
<td>1503.6</td>
<td>1501.1</td>
</tr>
</tbody>
</table>

Table 1 Partial and complete Search time using algebraic signature.

In partial search, we experiments:
- Using first the Xor calculation and then, in case of equality, the algebraic signature.
- Using algebraic signature alone.

We remarks that the search time with only use of signature is near to the same time using Xor in first. This is viable by experiments large bucket of data.

When size of the text to search is more less than size of record, time necessary to search partial text is more important because of the multiple search in the same record. This is also valid when the text is placed in the end of record.

The searches stretched over all 8000 records with a 60B non-key field. We manipulated the bucket so that the third-last record contained the 3B string for which we were
searching. The total search time was 1.516 sec, but traversing the bucket already took 0.5 sec. Without this time, we search at a speed of 2 sec per MB.

**Comparison to Karp Rabin search text:**

To evaluate the comparative interest for us of the algebraic signatures with respect to another search scheme, we have also experimented with the probably best-known algorithm, which is the Karp-Rabin algorithm [KR87].

Text inserted in each record has size of 950 byte bytes ASCII characters. Then, we insert at the last record of this bucket a text with size of 30 bytes. After this, we search the partial or complete text. The text we search has size of 20 bytes and the good text is send to the last record with size of 50 bytes (insert in 5th position in this record). Our results are as following:

We tested our signature scheme for the search in the non-key portion of a record. Since we use symbols of length 2 bytes $GF(2^{16})$, and since our records consists of 950 bytes, our code has to take care of an alignment problem, that arises when e.g the second, third, and fourth byte of record make up the string for which we are searching.

<table>
<thead>
<tr>
<th>Bucket capacity (Record)</th>
<th>String to search position</th>
<th>Partial search time (first Xor + sign if equality) 950 bytes</th>
<th>Partial search (Karp Rabin)</th>
</tr>
</thead>
<tbody>
<tr>
<td>100</td>
<td>99</td>
<td>625</td>
<td>457</td>
</tr>
<tr>
<td>200</td>
<td>199</td>
<td>1325</td>
<td>953</td>
</tr>
<tr>
<td>500</td>
<td>299</td>
<td>3252</td>
<td>2380</td>
</tr>
<tr>
<td>1000</td>
<td>995</td>
<td>7157</td>
<td>4895</td>
</tr>
</tbody>
</table>

**Table 2** Partial Search time using karp Rabin algorithm performance analysis.

Because of linear calculation time of Karp Rabin algorithm, the results confirm that it is faster. It used just 1 multiplication and 1 addition but it is not the case for symbol exceeds 32 bytes.

As we notice, it is more efficient for small search (text under 32Bytes). The gain being more less for larger ones where our signatures have not limit of size of text to search. We note also that the worst-case time complexity of the Karp-Rabin algorithm is quadratic (as it is for the brute force algorithm) but its expected running time is $O(m+n)$. Several linear time algorithms have been proposed for the problem of pattern matching. Knuth, Morris, Pratt and Boyer and Moore algorithms require, for fast implementation, several registers to store a table of pointer $O(n)$. Technical of algebraic signatures, like Karp and Rabin algorithm, require 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 follow-ups, it also has the property to evaluate 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 the client evaluates the strings returned by the servers. Since time necessary to calculate our signature is under 5 usec/KB of data in a 1.8 GHZ Pentium4 machine and, also, we manipulate symbols of length 2B ($GF(2^{16})$), most of calculation time is spend on memory transfers and very little on Galois field arithmetic in particular for search concerning for exemple 8000 records with 60B non-key field. Searching 3B contained in last record took 1.516 sec (speed of 2 sec per MB). We compare with Karp Rabin. This took 1.504 sec.

**4.3. Update experimental performance analysis**

It would be a good thing to use algebraic signatures in order to distinguish between updates that in fact change the record and those do not. We can name the latter “pseudo-update”.

We test our signature for blind updates, which- as we recall change the value of the record absolutely- and for blind updates, which set the new value of the record based on the old value.
A normal update in SDDS2004 took 0.84 msec per 1KB record, but without change in this record, it took only 0.27 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.

Processing times for blind updates are 0.82 msec for true update and 0.25 msec for an update without change in our record. The saving for normal pseudo update is also 70%. The times include the key search, the update processing, and the transfer of the record signature. For records more less (100B), times are faster but the savings for pseudo-updates are about 50% in this case.

5. Conclusion

We have added the SDDS string matching capability to our prototype SDDS-2000 system. Our implementation is now integrated in the SDDS-2004 prototype. The use of signatures allowed us to add this function without the modification of the existing code. Our signature based scheme for updating records at the SDDS client should prove its advantages in client-server based database systems in general. It holds the promise of interesting possibilities for transactional concurrency control, beyond the mere avoidance of lost updates.

The signature calculus that our approach required could be made fast enough to remain under 10% of the disk write time. This is a negligible performance penalty in practice. All together, our approach should be of interest also to other applications with similar needs.

Our future work addresses the encoding/decoding of data used pre computed algebraic signatures. The encoding/encoding according to either scheme occurs at the SDDS client. This concerns especially the non-key data and the aim is to prevent that users may even modify data.

Acknowledgements

We thank Jim Gray for fruitful discussions. 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-C0EN-RSCH-IG-IG09.

References