A Signature Based Concurrency Scheme for Scalable Distributed Data Structures

Thomas Schwarz, S.J.
Department of Computer Engineering
Santa Clara University
tjschwarz@scu.edu

JoAnne Holliday
Department of Computer Engineering
Santa Clara University
jholliday@scu.edu

Abstract

We evaluate theoretically a new validation scheme based on signatures with algebraic properties. In this proposal, we read records without regard to transactional isolation, but we validate these reads before we commit any writes. We evaluate the concurrency level of this basic scheme and propose a way to integrate it with a typical hash based SDDS that avoids phantoms.

Keywords: Scalable Distributed Data Structures, Transaction Processing, Algebraic Signatures, Concurrency Control.

1 Introduction

Transaction processing is a cornerstone of modern information technology. Techniques to allow concurrent transactions have been studied for at least the last three decades. In this paper, we investigate one recently proposed concurrency algorithm [11, 12] for distributed databases, especially Scalable Distributed Data Structures (SDDS), that uses signatures to validate reads. In particular, we investigate the degree of isolation and also how to achieve "the third degree", that is, isolated, serializable, and repeatable reads. As it turns out, the original proposal only implements the ANSI SQL Read Committed level, but an enhancement yields full isolation levels with excellent concurrency.

A transaction in our scheme reads records without regards for isolation control. It can determine its read set dynamically, that is, it does not have to predeclare reads. Before the transaction commits its writes and before it commits to the user, it verifies all its reads with a multicast. If all its reads have been successfully verified, the transaction performs all its writes with another multicast and returns to the client application.

When a transaction verifies a record, it just sends the signature – a small bit string calculated from the record contents – to the site. This technique allows very simple and fast verification. Our protocol works especially well for tables organized as LH* files. LH* is the distributed version of Linear Hashing (LH) [9]. We verify not individual records, but small regions, consisting of at most a few records. By verifying regions, we prevent phantoms.

Our fundamental result is that we can achieve full isolation (i.e. serializability) if we prevent conflicts with competing transactions during the verify-write phase of a transaction. We can think of the actual reads as "pre-reads" and of the read verifies as the "true reads" for the purposes of concurrency control. The actual critical section of a transaction, during which it can conflict with other transactions is then extremely short, consisting only of a few multicasting rounds, but without read data transfers and calculations. We consider this an important property. We can use locking for the critical verify-write phase to gain the properties of a conservative, strict two-phase locking protocol.

2 Signature Scheme: Definition and Properties

2.1 Galois Field Fundamentals

Our algebraic signatures use Galois field calculations. The elements of the Galois field we are using are symbols, that is, bit strings of length f. Typically, $f = 8$, and a symbol is simply a byte. Sometimes it is advantageous to use $f = 16$ (half words) or even $f = 32$ (words). We can add, multiply, and divide symbols just as we manipulate real numbers or rational numbers (fractions). Mathematically, our field consists of all polynomials of degree up to $f - 1$ in an undeterminate x over the field of binary numbers $\{0, 1\}$ modulo a certain *generator polynomial* $g(x)$, which is irreducible and of degree f. The bit string encodes the coefficients of the polynomial. For example the polynomial $x^7 + x^3 + x + 1$ corresponds to the string 1000 1011. Addition is the addition of two such polynomials. At the bit string level, we merely add the coefficients, that is, we take the XOR of the bit string. To multiply two symbols, we mathematically multiply the two corresponding polynomials and then calculate the remainder of the division by g of the product. This method is too cumbersome to be used in practice. A much more preferable method uses logarithms and antilogarithm. A Galois field abounds with primitive elements α so that every non-zero field element β can be written as a power α^i. The power i is uniquely determined within $\{1, \ldots 2^f - 1\}$ and is called the logarithm $i = \log_\alpha(\beta)$ of β. Reversly, we call β the antilogarithm $\beta = \text{antilog}_\alpha(i)$ of i. The product of two non-zero elements β and

γ is then $\beta \cdot \gamma = \text{antilog}_\alpha(\log_\alpha(\beta) + \log_\alpha(\gamma))$, where the addition is taken modulo $2^f - 1$. We implement multiplication then by using a logarithm and an antilogarithm table. By using longer logarithm tables, we can replace the lengthy modulo $2^f - 1$ addition with normal integer addition and even replace the tests whether the factors are zero.

2.2 Record Signatures

The records in our database consists of a key c and a the non-key data field R. In practice, R is often divided into many different fields, it might contain various types of data such as images, and it might be of non-uniform length. By padding if necessary, we assume that R is a string of symbols r_ν that are elements in our Galois field. We calculate a *signature* (a.k.a. checksum or hash) from the contents of R in the way we now describe.

Definition 1 *Let β be a non-zero element of the Galois field, and let $R = (r_\nu)_{\{\nu=1,\ldots,N\}}$ be a record field. Define the β-signature of R to be*

$$\text{sig}_\beta(R) = \sum_{\nu=1}^{N} \alpha^{\nu-1} r_\nu.$$

If α is a primitive element and if m is an integer $m \geq 1$, define the m-fold α-signature of R to be

$$\text{sig}_{\alpha,m}(R) = (\text{sig}_{\alpha^0}(R), \text{sig}_\alpha(R), \text{sig}_{\alpha^2}(R), \ldots, \text{sig}_{\alpha^{m-1}}(R)).$$

It follows from the definition that $\text{sig}_{\alpha^0}(R)$ is the XOR of all the symbols making up R. The m-fold α-signature of R is a bit string of length mf. The definition of the β-signature is closely related, but not identical to the signature used in [8, 5] and falls into a general class called a *Karp-Rabin polynomial.* We call our signatures *algebraic* because they have algebraic properties that we are going to exploit. We are interested in validating records and record fields. In particularly, we are interested that the signature detects a change, that is, that the signature changes when we change the record. The first proposition below shows that this is true for small changes while the second one applies to major changes. The third proposition shows that we can calculate the change in signature from the change in the record. The first and the third proposition do not apply to cryptographically secure signature like the Secure Hash Algorithm (SHA1 et al.) or MD5.

Proposition 1 *If the record field length n of R is smaller than $2^f - 1$ then the m-fold α-signature discovers any changes in R of up to m symbols.*

Proposition 2 *The probability that the m-fold α-signature of two random records fields coincides is 2^{-mf}.*

Proposition 3 *Let us change record field R to R' by replacing the symbols starting in position $s+1$ and ending in position t with a string $q_{s+1}, q_{s+2}, \ldots, q_t$. Define the Δ-string to be the string $\delta_1, \delta_2, \ldots, \delta_{t-s}$ with $\delta_i = p_{i+s} \bigoplus q_{i+s}$. Then*

$$\mathrm{sig}_\beta(R') = \mathrm{sig}_\beta(R) + \beta^s \mathrm{sig}_\beta(\Delta)$$

Proposition 4 *If we add zero characters to the end of R, then the signature of R does not change.*

The proof of the last proposition is trivial, the other proofs as well as additional properties can be found in [12].

2.3 Record and Region Signatures

A record consists of a key c and a non-key field R. By conceptually padding with zeroes, we do not change signatures (Prop. 3) and so we can assume that the non-key fields in the data base have all the same length. We assume a mapping ϕ that maps each key c into a non-zero Galois field element $\phi(c)$. A *region* $\mathcal{R}$ is a set of records $\{(c_j, R_j) | j \in J\}$. We define the *region signature* with respect to a Galois field element β to be

$$\mathrm{sig}_\beta(\mathcal{R}) = \sum_{j \in J} \phi(c_j) \mathrm{sig}_\beta(R_j).$$

Essentially, the region is a sum of the signatures of the fields in the regions, but modified such that the keys contribute to the signature calculation. If we switch the keys of two fields, then the region signature changes as long as the ϕ values of the keys differ. Since regions are typically composed of records with keys that have the same binary postfix and since we typically set $\phi(c)$ to be the first f digits, the latter will almost always be the case. We define the m-fold signature of a region $\mathcal{R}$ to be

$$\mathrm{sig}_{\alpha,m}(\mathcal{R}) := (sig_1(\mathcal{R}), sig_\alpha(\mathcal{R}), \ldots sig_{\alpha^{m-1}}(\mathcal{R}))\,.$$

The m-fold region signature inherits properties from the record signature, but also satisfies an analogue of Proposition 3 when adding, deleting, or modifying a member record.

Proposition 5 *Let $\beta = \alpha^i$ for some Galois field element α and some power i. Let $R = (c, F)$ be a record and let $\mathcal{R}$ be a region. Then*

1. *If $R \notin \mathcal{R}$ then $\mathrm{sig}_\beta(\mathcal{R} \cup \{R\}) = \mathrm{sig}_\beta(\mathcal{R}) + \phi(c)\mathrm{sig}_\beta(R)$.*

2. *If* $R \in \mathcal{R}$ *then* $\text{sig}_\beta(\mathcal{R} \setminus \{R\}) = \text{sig}_\beta(\mathcal{R}) + \phi(c)\text{sig}_\beta(R)$.

3. *If* $R \in \mathcal{R}$ *and we change* (c, R) *to* (c, R') *and by that change* $\mathcal{R}$ *to* $\mathcal{R}'$ *then* $\text{sig}_\beta(\mathcal{R}') =$

$$\text{sig}_\beta(\mathcal{R}) + \phi(c) \cdot (\text{sig}_\beta(R) + \text{sig}_\beta(R')).$$

The proof of the proposition follows immediately from the definitions. Readers not familiar with calculations in a Galois field $\mathcal{GF}(2^f)$ need to be warned that in these fields, addition is the same as subtraction.

In the context of scalable, distributed data structures, we encounter regions in two varieties. If the SDDS is range partitioned, then regions will be formed by records with keys in an interval $[c_0, c_1]$, if SDDS is a LH* variant, then a region would be a bucket or a sub-bucket. We consider this case in the next section.

2.4 LH* Regions

We assume some familiarity with Linear Hashing and its distributed version, LH*. An LH* bucket with bucket number n consists of all records in the LH* file with a key c such that the last l bits of the key make up the binary number n. Parameter l is the level, which scales up and down with the file. Consequentially, we define a region $\mathcal{R}(n, k)$ to consists of all records such the last k digits of the binary representation of their key c equals n. A region can thus consists of several buckets, a single bucket, or parts of a bucket. If the LH* buckets are organized as Linear Hashing files, then a region might consists in the latter case of several LH buckets, an LH bucket, or parts of an LH bucket. We choose regions with k small enough so that on average a region is made of zero, one, or a few records. Our scheme treats regions as atomic isolation units. If we do locking, then we lock and unlock regions instead of individual records. This avoids problems with *phantoms*. Phantoms are records that appear or disappear from the sets read or written by a transaction because some other transaction puts them in (because we do not have predicate locking). It is easy to lock regions, independently whether a region actually contains records or not. The alternative to region locking is to describe the records that are being locked, i.e. to use predicate locking. We could also lock the next record after record to be inserted or deleted, as in Aries. Predicate locking is cumbersome and often too pessimistic. The drawback of region locking is that a record about to be created needs to have a key, by which it can be referred to. This might limit our solution to SDDS.

As the SDDS file grows and shrinks, we need to adjust the size of a region by splitting or merging regions. These operations are exactly like the corresponding operations for buckets, with the important exception that we do not actually move records (in LH*) or even pointers to records (in most LH implementations)

around. Instead, we need to merge or divide region signatures. Merging signatures is simple, as the following proposition shows. To calculate the signature of a split region, we need to recalculate the signature of one of the new regions, but only one, as the following proposition shows.

Proposition 6 *Let $\mathcal{R}_1$ and $\mathcal{R}_2$ be two regions such that $\mathcal{R}_1 \cap \mathcal{R}_2 = \emptyset$. Then* $\text{sig}_\beta(\mathcal{R}_1 \cup \mathcal{R}_2) = \text{sig}_\beta(\mathcal{R}_1) + \text{sig}_\beta(\mathcal{R}_2)$.

3 Algebraic Signature Based Concurrency Scheme

We develop transaction support for a distributed database representing a single SDDS file. The records in the SDDS are stored in buckets, and the records are grouped into regions within the bucket. We maintain the algebraic signature of the regions with the bucket. Each server site in the distributed database contains one or more buckets. The client site manages transactions from the users and submits operations to the appropriate server sites.

Transactions are executed in two phases, the *Read and Calculate phase* and the *Verify and Write phase*. In the first phase, a transaction reads records, makes calculations, possibly followed by more reads and more calculations. Whenever a transaction reads a record (or even checks for the existence of a record), it also receives the algebraic signature of the region in which the record was. In the second, typically much faster phase, the transaction checks that the region signatures of its reads have not changed. If that is the case, then the transaction does all its writes. If however a region signature has changed, then we restart the transaction, because the read was probably of a record that another transaction has changed. When a transaction restarts, it does not do so from scratch, because it does not need to redo a read that the scheme has just confirmed to be still valid.

4 Serialization Properties

We adapt the following notation for execution histories. We use $x, y, z, \ldots$ for records, we number transactions with arabic numerals $1, 2, \ldots$ and we write R for a read operation, W for a write operation, and V for the verification of signature operation. Thus, $V_1(x)$ means that transaction 1 verified its previous read of record x.

4.1 Basic Scheme

According to our rules in the basic scheme, transaction i aborts (written as A_i) if even one verify fails, and commits (written as C_i) after all writes are done. In more detail, we have the following rules for execution histories.

1. All read operations in a transaction precede all verify operations, which proceed all write operations.

2. A verify operation $V_i(x)$ is successful if there was no write operation $W_j(x)$ between $R_i(x)$ and $V_i(x)$.

3. If all verify operations are successful, then the transaction does all its writes, if any, and terminates successfully by committing. Otherwise, the transaction aborts without writing.

We compare our basic scheme to the three ANSI SQL isolation levels in the strict interpretation of [3]. Since transactions that write have already successfully verified all their reads and hence will commit, and since transactions that abort (= restart) never write, the scheme prevents phenomena P0 (dirty writes) and P1 (dirty reads). However, since reads are not protected, our scheme allows a partial history such as

$$R_1(x) \quad W_2(x) \quad C_2 \quad R_1(x) \quad \ldots \quad V_1(x) \quad C_1.$$

Thus, our scheme allows thus non-repeatable (fuzzy) reads and this constitutes phenomenon P2. Our scheme implements the ANSI Read Committed Level of Isolation. Since we do not protect reads, we can also have phantoms.

Assume now that we modify our scheme such that we have the following additional property:

4. Every actual partial execution history is conflict-equivalent to one where all verifies and writes of a transaction i happen in a single, atomic block.

Theorem 1 *A schedule subject to rules 1–4 is serializable.*

Proof: We recall that two schedules are conflict equivalent ([14], Def. 3.12,[6]) if they contain the same operations and have the same conflict relations. This happens exactly if we can transform one of the schedules into the other ones by commuting (switching) two transactions in the schedule subject to some rather obvious rules. For example, we can switch two read transactions or two transactions that affect different objects. See Weikum and Vossen [14], 3.8.3 for an exact formulation. Let S be a schedule subject to rules 1-4. A schedule is an abstraction that orders all operations. In an actual history of a distributed database, there are operations that cannot be ordered, because they happen on different servers that do not immediately communicate with each other, but in these cases the ordering does not matter for the final state of the

database. Without loss of generality, all verify and write operations of a transaction are done in a single block, with all verifies preceeding all writes. If i and j are transactions, then we write $i <_S j$ if the verify-write block of transaction i comes before the verify-write block of transaction j. Since all transactions in a schedule either read, and hence verify, or write, $<_S$ is a complete ordering of $tr(S)$, the set of all transactions in S. Assume now two transactions $i, j \in tr(S)$ with $i <_S j$ and two operations in conflict with the first one being an operation of j. The ordering $i <_S j$ excludes the case $(\ldots W_j(x), W_i(x)\ldots)$. If $(\ldots R_j(x), W_i(x)\ldots)$, then $V_j(x)$ comes after $W_i(x)$ and hence fails. Therefore, transaction j aborts without writing any data, thus removing all potential conflicts. Finally, a schedule $(\ldots W_j(x), R_i(x)\ldots)$ is excluded by rule 1, since $V_i(x)$ would follow after $W_j(x)$ which contradicts $i <_S j$.

A more intuitive argument is that we consider all our reads to be merely *pre-reads* of an advisory character and that the verifies count as the true reads for concurrency.

4.2 A Locking Scheme

To fully support transaction serializability, we need to make the verify–write phase atomic. We propose a locking scheme where a transaction acquires all locks quasi-simultaneously and releases them simultaneously. This is a variation of well known locking schemes, but one in which all locks are short-lived. In more detail, a transaction accesses data records during its read–calculate phase without regard to locks. In this, our scheme is very optimistic. When a transaction reads from a region, it also requests the region signature. After the transaction has finished all its reads, it sends a multicast message to all members of its read and write set, requesting read and write locks respectively. It sends the region signature(s) to the members of the read set. Similarly, it sends the update to a server at which it wants to update a record. A server receiving a read lock request grants it if the region signature sent by the transaction is the same as that calculated locally and if there is no write lock on the region. A server receiving a write lock request grants it if there are no other locks on the region. A server for a region in both the read and the write set grants the request and sets the write lock if the signatures match and there are no other locks on the region. All servers accessed send their decision to the transaction. If the transaction receives one negative answer, it aborts, that is, it restarts. It also sends an "abort and release locks" message to all other service sites that it accessed. Otherwise, the transaction sends a multicast commit message to all sites. This causes a lock at a read region to be released. The receipt of the commit message at a write site causes the new value to be written and the lock released.

Our locking variant combines conservative and strict two-phase locking, because it sets all locks "simulta-

neously" at the beginning of the critical phase and releases them "simultaneously" at the end of the critical phase. The first property implies that the locks are short-lived; the second one gives us *strict execution*, which makes recovery easy. Because locks are short-lived, the degree of concurrency should be high. On the negative side, a transaction with a very large read set might suffer from live-lock (many repeated aborts) in an environment with many writes.

Because reads in our scheme are advisory, a transaction that aborts because a verify operation fails does not need to restart from scratch. Since the data obtained from reads have just been verified, the transaction only needs to read data items that have been invalidated. Some transactions might actually be able to recover parts of the calculation, but this capability has to be engineered at the database query processing level.

4.3 Combining the Locking Scheme with SDDS

In order to avoid phantoms, we do not want to lock individual records but regions of records. We therefore conceptually divide the SDDS files into small regions that we lock. For each region we maintain a signature. When we read a data item, we acquire the signature of the region in which the record resides. When we verify the value of the record, we merely submit the signature to the bucket server of the bucket that contains the region and ask whether the signature has changed. If the region were to contain many records, then we could not tell whether the signature changed because the read record changed or because of an unrelated change. For this reason, regions should on average contain a single item. We can achieve this by dynamically merging and splitting regions as the file size shrinks and grows. When we split or merge a region, we should keep the signature of the old region(s) around for the little time it takes for transactions that have obtained this signature to clear.

We can even store the region signatures in a Merkle tree, whose leaves consists of region signatures and whose inner nodes consist of sums of the signatures of the regions. In a database with very few writes, we can then verify multiple reads at the same server with a single signature. The details of this scheme are subject to continuing research.

4.4 A Non-Locking Scheme

Modern distributed systems have loosely synchronized clocks, that is, clocks that are synchronized with a given maximum difference at the order of a few milliseconds at most. We then can use an optimistic

concurrency scheme based on a timestamp acquired at a local server such as the one in [1]. When a transaction T enters the verify phase, it determines the server with the lowest bucket number and gets a time stamp from this server. The transaction coordinator (TC) then sends verify/write/vote requests simultaneously to all servers. The verify/write/vote request contains the following information: T.ts, the timestamp of T; T.ReadSet, the regions at that bucket that were read by T; T.WriteSet, the regions of that bucket that are to be written by T; and the identifier of the client. The server maintains a validation queue (VQ) of recently validated transactions and the VQ is used to ensure that T is serialized by timestamp with all conflicting transactions in the VQ. If T is not in serial order, the server aborts T by returning a "no" vote to the TC. If T is in proper order, the server puts the transaction information in the VQ and votes "yes". If the TC receives all "yes" votes, it tells all servers to commit, at which time the writes are made permanent.

5 Conclusion and Future Work

Using signatures alone to verify reads allows unserializable histories. However, by adding standard database concurrency mechanisms, region signatures can enforce transactional isolation. The use of signatures allows us to shorten considerably the time in which a transaction might contend with another transaction and thus should considerably improve throughput. Using regions deals with phantom records effectively. We assume that a comparison with predicate locks will be favorable to our scheme.

In the future, we will investigate under what circumstances (if any) our schemes perform better than normal schemes. We expect to have a significant advantage for situations with few writes and long calculation phases. We also need to design rules for dealing with failure, because currently, a transaction that encounters an unavailable site needs to wait for the data items to be reconstructed. This might not be good policy, since it increases lock duration when the system as a whole experiences considerable network load.

References

[1] A. Ayda, R. Gruber, B. Liskov, U. Maheshwari. Efficient optimistic concurrency control using loosely synchronized clocks. *Proceedings of the 1995 ACM SIGMOD International Conference on Management of Data*. p. 1-10, 1995.

[2] A. Ayda, B. Liskov, P. ONeil. Generalized isolation level definitions. *P*roc. of the 16th Intern. Conf. on Data Engineering, ICDE, p. 67-80, 2000.

[3] H. Berenson, P. Bernstein, J. Gray, J. Melton, E. ONeil, and P. ONeil A critique of ANSI SQL isolation levels. In *Proceedings of the 1995 ACM SIGMOD International Conference on Management of Data*. San Jose, CA, p. 23-34, 1995.

[4] M. Blaum, G. Farrell, and H. van Tilborg. Array Codes. In Pless, Huffman (ed.) Handbook of Coding Theory II, p. 1855 - 1909. North Holland, 1998.

[5] A. Broder. Some applications of Rabin's fingerprinting method. In Capocelli, De Santis, and Vaccaro, (ed.), Sequences II: Methods in Communications, Security, and Computer Science, p. 143 - 152. Springer-Verlag, 1993.

[6] J. Gray and A. Reuter. Transaction Processing: Concepts and Techniques, Morgan Kaufmann, San Francisco, 1993.

[7] L. Hellerstein, G. Gibson, R. Karp, R. Katz, and D. PattersonD. Coding techniques for handling failures in large disk arrays. In *Algorithmica Vol. 2, Nr. 3,* 182-208 (1994)

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

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

[10] W. Litwin and T. Schwarz. LH*RS, A high availability scalable distributed data structure using Reed Solomon codes. In *Proceedings of the 2000 ACM SIGMOD International Conference on Management of Data,* May 16-18, 2000, Dallas, Texas. Also *S*IGMOD Records, vol. 29 (2), June 2000, p. 237-248.

[11] W. Litwin and T. Schwarz. Algebraic signatures. In *Proceedings of the Fifth Workshop on Distributed Data and Structures,* Thessaloniki, June 2003 (WDAS 2003).

[12] W. Litwin and T. Schwarz. Algebraic signatures for scalable distributed data structures. In *Proceedings of the 20th International Conference on Data Engineering (ICDE),* Boston, March 30 - April 2, 2004.

[13] F. J. MacWilliams and N. J. A. Sloane. The theory of error correcting codes. North Holland, Amsterdam, 1977.

[14] G. Weikum and G. Vossen. Transactional information systems. Morgan Kaufmann, San Francisco, 2002.

This work has been supported in part by IBM Research Grants 41102-COEN-RSCH and EIBM0010 at Santa Clara University