A Triggerless Approach to Writeset Extraction in Multiversioned Databases

Raúl Salinas-Monteagudo, Francesc D. Muñoz-Escoí

Instituto Tecnológico de Informática
Universidad Politécnica de Valencia
Camino de Vera, s/n
46022 Valencia (Spain)

{rsalinas,fmunyoz}@iti.upv.es

Technical Report ITI-ITE-08/11

A Triggerless Approach to Writeset Extraction in Multiversioned Databases

Raúl Salinas-Monteagudo, Francesc D. Muñoz-Escoí

Instituto Tecnológico de Informática
Universidad Politécnica de Valencia
Camino de Vera, s/n
46022 Valencia (Spain)

Technical Report ITI-ITE-08/11

e-mail: {rsalinas,fmunyoz}@iti.upv.es

Abstract

Efficient writeset extraction and application is crucial in modern database replication protocols since they execute all transaction operations in a delegate replica, collecting and propagating the updates to all replicas in total-order when transactions request commitment. In order to ensure portability, such replication protocols can be implemented in a middleware and some of such middleware systems use triggers for managing writeset collection. MADIS is an example of such systems. It has been developed assuming PostgreSQL as its underlying DBMS. Thus, its protocols are usually oriented to enforce snapshot isolation and rely on multiversioned concurrency control. PostgreSQL provides in its catalog information regarding its multiversioned control and such information can be used by the replication middleware for improving its writeset collection and application performance, without requiring any knowledge about DBMS's internals nor compromising the independence between the middleware and the DBMS layer. This paper describes an almost triggerless approach for writeset management in the MADIS middleware database replication system, comparing its performance with its previous release based on triggers.

1 Introduction

Database replication has been an interesting research field for many years [2]. In its first stages, database replication already used the ROWA [2] approach, combined with pessimistic concurrency control –e.g., distributed locking, based on local 2PL– and 2PC as its termination protocol. Following the ROWA approach, a first enhancement consisted in using an *optimistic two-phase locking (O2PL)* [4] concurrenty control, executing the transaction locally in its delegate replica and requesting remote write locks at update propagation time, instead of at each write access. A second important optimization was proposed in two papers [1, 12] that suggested the use of atomic broadcast [5] as a possible replacement of the traditional 2PC termination protocol. As a result of this, different classes of database replication protocols based on atomic broadcast and on writeset propagation can be found nowadays [20], being the *weak-voting* and *certification-based* ones able to provide the best performance. In both classes, all transactions should be locally executed in their receiving delegate replica. Once all their operations have completed and when transactions locally request their commitment, their writesets should have been collected and need to be transferred using an atomic broadcast to all replicas. In some cases, e.g., with *certification-based* protocols providing a serializable isolation, readsets should also be collected and transferred.

Note that in a modern replication protocol, the most time-consuming steps will be those related to writeset collection in the delegate replica and writeset application in the remote replicas, since in a non-replicated setting none of such tasks is needed and they demand a non-negligible effort. Due to this, several

database replication systems (e.g., Postgres-R [9]) have been built modifying the underlying DBMS in order to get access to the internal database log for retrieving writeset contents and for applying them in the receiving replicas. Thus, good performance can be achieved, but the replication system has a restricted portability since it heavily depends on its underlying DBMS.

Portability can be guaranteed using a middleware architecture that only relies on the underlying DBMS public interfaces, but in this case writeset collection and application techniques should be carefully designed. An accepted solution is to base such techniques on triggers or any other standard mechanism provided by relational DBMSes. But now we find the reverse problem: performance is seriously compromised. Despite this, multiple database replication systems have been based on a middleware architecture: GlobData [17], Middle-R [11], MADIS [8], Ganymed [14, 13], Sprint [3],...

This paper describes the second release of the MADIS database replication middleware. Instead of relying entirely on triggers for readset/writeset management like its first release –that was carefully described in [8]–, this new middleware edition is based on some version-management attributes of the underlying PostgreSQL [15] DBMS. Note however that this new writeset management can coexist with all the mechanisms already used in the previous release. As a result, if MADIS is deployed on top of a non-multiversion-based DBMS it still may use all the mechanisms already described in [8]. As we will see in the following sections, this new writeset management does only rely on publicly accessible DBMS features; i.e., it does not demand any knowledge of the DBMS internals. Besides this, it eliminates the need of using database triggers for managing both insert and update operations, although delete operation management is still trigger-based. With this new management, transaction completion times can be highly reduced, improving thus the overall system performance in a significant way.

The rest of this paper is structured as follows. Section 2 explains the writeset management system. A performance comparison with the trigger-based MADIS previous release is shown in Section 3. Some related works are presented in Section 4. Finally, conclusions end the paper at Section 5.

2 The New MADIS Middleware

The MADIS Middleware relies on a component named DbLayer for managing writeset collection [8] and transaction conflict detection [10]. This section describes the new writeset collection and application management implemented by DbLayer_v2. It takes advantage of some well-documented transparent system columns offered by PostgreSQL in every table, and allows an *almost* triggerless mechanism for writeset extraction. Delete sentences do need trigger executions. Since in most applications inserts and updates occur much more often than deletes, a significant amount of time is saved.

2.1 PostgreSQL's Multiversioning Features

PostgreSQL is a *multiversioned* DBMS offering snapshot isolation. Every time a new transaction starts, a virtual snapshot of the database is taken. The transaction will work with this frozen view of the state of the database by the time it started. Changes are not immediately seen by other transactions before commit. At commit time, all those changes become atomically visible to new transactions.

In PostgreSQL, each transaction has a unique identifier in the system (XID, "trid" in MADIS nomenclature), which is a sequentially increasing integer. Each row of a PostgreSQL table contains several additional fields, that are invisible unless explicitly requested in the query.

- $xmin$. The XID of the last transaction that updated that row. If the row was modified by the active transaction, this value will be that of our XID (as obtained with *txid_current()*).
- $xmax$. The XID of the transaction that deleted (or tried to delete and later aborted) this row.
- $cmin$. The order number of the sentence modifying this row. Every sentence has a number that uniquely identifies it inside the transaction. This number is written onto every touched row. This allows ordered writeset application in the receiving replicas. It also makes easy an incremental writeset extraction in order to deal with replication protocols based on *linear interaction* [19].

The field $xmin$ allows a triggerless extraction of inserted and updated rows. These rows satisfy that their $xmin$ equals the XID of the running transaction. So, we only need to *select* –once commit has been requested– all rows whose value in such field is equal to the current XID.

Deleted rows should be theoretically distinguishable by their $xmax$. Every row deleted by a transaction T will have $xmax$ set to its XID. Obviously those rows are not visible inside the transaction that has deleted them, since they have disappeared from its view, but they are indeed visible from any other transaction.

XIDs can be obtained with the native PostgreSQL function *txid_current()*. Other DBMSes provides similar functions; e.g., Oracle gives access to this datum by means of its *dbms_transaction.local_transaction_id*.

2.2 The MADIS Catalog

The DbLayer_v2 writeset extraction subsystem requires an extra table per database –containing the catalog– to be created. This catalog table contains one tuple per database table with these fields:

- The table name (a character string).
- The table identifier (a 16-bit integer).

This table must be synchronized across all replicas in the system. When changing the schema, the catalog must be updated as well. Since MADIS servers cache the database schema of each repository, catalog updates must invalidate the cached data.

The main reason why DbLayer_v2 needs a separated catalog table is the need to map table names to integers. While DbLlayer_v1 used full name tables in its global tuple identifiers, the DbLayer_v2 uses integers instead.

Since DbLayer_v1 will only work on those tables on which *initmeta()* was run, and since DbLayer_v2 will restrict itself to those tables listed in this MADIS catalog, a conflict-free coexistence of both systems is, although uninteresting, theoretically –and trivially– possible.

2.3 Writeset Extraction Mechanism

In order to explain the new writeset extraction mechanism, several dessign issues should be described. To begin with, the *global object identifiers* (or GOIDs, for short). They are 64-bit integers and are needed in order to assign to each tuple the same identifier in all system replicas. These identifiers are not stored in a separated metatable as done in DbLayer_v1, but in the user tables themselves. When preparing the tables for writeset extraction, a *bigint* column called *GOID* defaulting to NULL is appended as the last column.

GOID (64 bits)		
Node id (16 bits)	Table id (16 bits)	Local OID (32 bits)

Table 1: GOID structure

As seen in Table 1, GOIDs are constructed from the delegate replica's identifier (highest 16 bits), the table identifier (next 16 bits, as described above) and the local OID (last 32 bits, automatically assigned by PostgreSQL for its own management). To simplify the notation, we will use on the sequel the function *buildGoid(int node, int tableId, int OID)* that returns a 64-bit integer.

Using a single 64-bit integer as GOID allows a more efficient management than character strings. First, it is more compact. Secondly, comparing long integers for equality is far more efficient than comparing strings. Modern 64-bit processors can namely use a single processor operation to compare them, without breaking the instruction pipeline. Hashing integers has also lesser overhead, which is important when finding conflicts between writesets, operation that is performed in the certification step of all database replication protocols; i.e., when such replication protocol checks whether a transaction requesting commit can be accepted or not.

The second issue to be described is the extraction of inserts and updates. Once all the sentences shown in Table 2 have been applied in order to adapt the original database schema to MADIS, we can extract

For each database, once	`CREATE OR REPLACE FUNCTION t_delete()`
	`RETURNS trigger AS '`
	`  begin`
	`  -- If object was not created by`
	`ourselves`
	`  if old.goid is not null then`
	`  insert into deleted_goids`
	`values(old.goid);`
	`  end if;`
	`  return old;`
	`  end;`
	`  ' LANGUAGE plpgsql;`

For each table *tableX*, once	`CREATE INDEX tableX_goid_idx ON tableX`
	`  using btree(goid);`
	`CREATE INDEX tableX_xmin_idx ON tableX`
	`  using hash(xmin);`
	`CREATE TRIGGER trigger_delete`
	`  AFTER DELETE ON tableX`
	`  FOR EACH ROW`
	`  EXECUTE PROCEDURE t_delete();`

Table 2: Preparing the database for writeset extraction

the inserts and updates made by the current transaction by executing a sentence like (this has been also summarized in Table 3 describing the prepared statements being set at connection start):

```
SELECT buildGoid(node, tableId, oid) newgoid,* FROM table WHERE
xmin = txid_current();
```

for each modified table. Note that inserted rows have a null GOID. Before committing, it must be set to the corresponding value, by issuing:

```
UPDATE table SET goid= buildGoid(node, tableId, oid) WHERE
xmin = txid_current() AND goid IS NULL;
```

This operation has to be performed some time between the writeset extraction and the final commit. It can be done in parallel with the rest of the transaction processing (broadcast, certification, etc), which can increase performance.

This approach allows a triggerless extraction of the inserted and updated rows. In addition to that, by putting the GOID in the same tuple, an access to a second table and the need to join both tables is saved.

The third feature to be described is how the deleted tuples can be collected. Theoretically, it should be possible to retrieve the deleteset of a transaction by opening a new transaction and retrieving from all tables every row having as *xmax* the XID of the transaction requesting commit. If two transactions would have tried to delete the same object, the second one would have had to wait to lock this field. If the first one committed the delete, the second one would not be able to delete that object. If the first one aborted, upon abortion the lock would be released and then the second transaction would set its *xmax* to its own XID. All this mechanism works indeed, but surprisingly, an index cannot be built on *xmax* whereas indexes can indeed be made on *xmin*. To be more precise, such index is allowed, but it is only updated when it is created, not in normal operation, which renders it unusable. If PostgreSQL allowed indexes on the $xmax$ field, the deleteset of a transaction could be quickly extracted in parallel with its insert- and update-set, even reusing any other already open connection (such as that of the block detector described in [10]), without any triggers at all. Without an index, this operation would perform a sequential scan, which is unacceptable on big tables.

Because of the mentioned limitation, deletes are tracked by means of triggers. A procedure is triggered for each deleted row. It checks its GOID, and, if not null (that is, if that row was not created in the same transaction), the GOID is appended to a temporary table *deleted_goids* having a single field *GOID*. A single

At connection start	Create temporary table for deleted items `CREATE TEMP TABLE deleted_goids (goid BIGINT) on commit delete rows;`
	Prepare statement getWS: `SELECT serialize(buildGoid(node,table,oid), goid, *) FROM t1 WHERE xmin=txid_current() UNION ALL SELECT serialize(0, goid) FROM deleted_goids ORDER BY cmin`
	Prepare statement update_GOIDs: `UPDATE t1 SET goid=buildGoid(node,table,oid) WHERE goid IS NULL`
Writeset extraction	`EXECUTE getWS;`
GOID setting (some time before commit)	`EXECUTE update_GOIDS;`

Table 3: Extraction mechanism

table is used to register the identifiers of the deleted objects of all tables.

Finally, note that an incremental writeset extraction can be performed in the new DbLayer_v2 as follows:

- Inserts and updates. Only those rows with a $cmin$ field greater than the highest $cmin$ already seen in previous collections are returned.
- Deletes. After extracting the accumulated deletions, the *deleted_goids* table is emptied.

2.4 Row Serialization

Internally, a DbLayer_v2 writeset consists of:

- An array of modified objects. Every object includes an operation code (insert, update, delete), a GOID, and the serialized data (only for inserts and updates).This configuration is easily extensible to support other kinds of modifications, such as schema changes, by using new operation types. This array is used when applying the writeset. Insert and update entries are allowed to have a null payload array, meaning that a compactation function has cancelled it, which will be explained further on.
- A *HashSet* with the GOIDs of the updates and the deletes is kept. Since inserts have always new GOIDs, it would be pointless to check them against other modifications. This hash table is used when checking for conflicts between writesets. This structure, which derives from the previous one, is not transmitted over the network, but it is regenerated at the receiver side.

In order to be able to extract with a single SQL query all modified rows, regardless of the table they were made on, all fields of each extracted row are bundled together and encoded into a single byte array inside the DBMS. Otherwise, a separate query would be needed for each table, since the returned result sets would not be UNION-compatible. Since experiments have shown that JDBC's performance decreases as the number of columns increase, not only variable-type data is encoded to a byte array, but also the common fields (operation type and GOID). For this purpose, PostgreSQL functions allowing the creation of new user-defined datatypes are used. Every datatype in PostgreSQL must provide `send()` and `receive()` functions. A *type*`_send()` function converts a datum of type *type* to a byte array. Each such function has its inverse counterpart. A great advantage of these functions is that they return a compact representation of the data. When analyzed in detail, we can see that it is basically XDR encoding. Integers are stored in network order (big endian), independently of the endianness of the machine. In clusters with nodes having different endiannesses this could bring some benefit.

In order to allow null values, a byte indicating whether the field is null is inserted before every nullable field. A fast way to do this is to use PostgreSQL's *coalesce()* function. This function receives one or more arguments, and it returns the first non-null one. Considering that a send()-function returns null when given a null value, and since concatenating (||) null with anything else yields null, a nullable field "fieldN" of type "typeX" is serialized as follows:

`COALESCE(byteasend(E'1')||`**`typeX`**`send(`**`fieldN`**`||d),byteasend(E'0'))`

That is to say, the first non-null value from the following pair:

- A byte array consisting of the byte 0x01, followed by the serialized field. The result of this concatenation is null if the given field is null.
- A byte array consisting of a single byte 0x00. *coalesce()* will choose this part if the first one is null.

In order to avoid calling *coalesce()* innecessarily, the database schema finds out the field nullability. Non-nullable fields are directly appended without previous checks, which saves one byte per non-nullable field, and a condition check when applying the row.

When a server repository manager starts, the database schema (the enumeration of all tables in the database with their respective fields and their respective types) is retrieved. Using this information, a SQL query retrieving the serialized modified (inserted, updated and deleted) rows is built. Such query is prepared at the start of each connection, and it will be reused for all transactions in that connection. A query that updates the GOIDs is also prepared at the connection start. See Table 3 for details on this.

The writeset extraction query includes several subqueries unioned with each other. If there are n tables under change control, the resulting query has $n + 1$ subqueries: one for each table, extracting the inserts and updates, and one for all tables, extracting the deleted GOIDs. `UNION ALL` is used instead of `UNION` because it avoids the overhead of sorting and removing duplicates. Duplicates could never appear in such query anyway.

A useful property of the serialization scheme being used is that by omitting the clause "`where xmin = txid_current()`" a full database dump is obtained. This can be used to trivially create a single writeset containing the whole database. By ordering by GOID and checksumming the returned byte arrays, full consistency among replicas can be verified at critical points, such as, for example, after recovering from a crash.

Figure 1 shows a summarized synopsis on how to use writeset extraction on a given database connection.

2.5 Writeset Application

The application of remote writesets is performed by means of prepared statements. Since items are referred by GOID, an index has to be created on this attribute to make writeset application efficient. Triggers are entirely disabled when applying remote writesets. If not disabled, the trigger used to collect the deletes would make the system slower, but would be otherwise harmless. However, if triggers having accumulative side effects were used, chances are that the delegate replica would run those procedures once, while the rest of the replicas would apply those changes twice, thus breaking consistency.

Ordered writeset application. The field $cmin$ of the extracted writeset can be used to apply the modifications in the original order that they took place at the delegate replica. Entries in the table *deleted_goids* also have their $cmin$, so that a total ordering between inserts, updates and deletes in every table can be achieved.

Figure 2 summarizes the way how remote writesets are applied on the server side. First, a normal connection is obtained. Then, the updating statements are prepared. This optimizes the performance in most DBMSes. The generic `WritesetApplier` interface has been constructed with the two-phase commit model into account, needed by the weak-voting protocols. Remote replicas start applying the broadcast changes, without knowing whether they will be finally committed or not. The application work is performed, and then the final decision is waited for. Protocols implementing the certification termination strategy have another method at their disposal, that applies and commits the changes in a single step.

Having such clear interface allows a clean decoupling between the writeset appliers and the consistency protocols. This permits writing such appliers in other languages, being C the fastest candidate.

1. Middleware's client side prepares the connection.

   Create table `deleted_GOIDs`.

   Prepare statement `getWS`.

   Prepare statement `update_GOIDs`.

   Disable autocommit.

2. While the connection is not closed:

   (a) The client works.

   (b) The client requests commit. (If rollback, restart at (a)).

   (c) The middleware's client side extracts the writeset: `EXECUTE getWS`. The client side sends the writeset to the server side.

   (d) The consistency protocol at the server side broadcasts the transaction with its writeset.

   (e) The middleware's client side updates the GOIDs, in parallel: `EXECUTE update_GOIDs`.

   (f) The middleware's client side performs a final `COMMIT` or `ABORT`.

Figure 1: Writeset extraction in DbLayer_v2

2.6 Metadata Hiding

DbLayer_v2 has a drawback that must be addressed. Namely, the inclusion of the GOID field into the data tables does not only make the system faster, but it also introduces the need to hide this field to the users' eyes. This field should not be accessible by the user for two reasons. First, because applications will get, if they do not specify the fields in the queries, an extra GOID field, which could confuse them, or pass through up to the user application. A more serious problem is that a client program could directly set GOIDs to any value. This could easily happen when importing into a table data from another table.

As a consequence, this field must be hidden in production servers. At least two approaches can be used:

- Mangling the *ResultSets* in order to hide the field. Little or negligible performance impact, but it only prevents the first undesirable phenomenon.
- Creating a view on each table, containing every field except the object tracking ones. The SQL would be automatically rewritten in order to change the name of the specified table to the name of the view. Alternatively, the tables could be renamed when preparing the schema for writeset extraction. This would avoid both phenomena, with some time expense. In PostgreSQL, using such a view instead of the original tables introduces an overhead of around 20%.

2.7 Other Features

DbLayer_v2 includes new features fulfilling requirements that were not evident by the time DbLayer_v1 was designed.

Generic interfaces. The writeset implementation has been strongly decoupled from other parts of the MADIS middleware. Specifically, the consistency protocol does not need to know any internal details of the writeset extraction and application mechanism in order to:

- Serialize/deserialize the writesets. The generic interface *Writeset* includes the methods *writeTo(DataOutputStream)* and *readFrom(DataInputStream)*. By using streams, instead of fully serializing and

1. The server gets a connection to the suitable repository, and prepares it.

   $\forall T_i, T_i \in MADIS\_catalog$, being T_i a table: prepare insert, update and delete statements for T_i

   Disable autocommit.

2. When a new remote writeset ws arrives:

   (a) The **consistency protocol** invokes `WritesetApplier.apply(Writeset` ws`)`.

   (b) The **writeset applier** applies the changes of ws and leaves the connection just about commit or abort.

   (c) The **consistency protocol** decides the outcome of the transaction and invokes either `commit()` or `abort()` on the applier.

   (d) The **applier** either commits or aborts, and leaves the remote connection ready for the next writeset.

Figure 2: Writeset application in DbLayer_v2. Performed at the server side.

deserializing to and from byte arrays, paralellism can be achieved, by sending/receiving the writesets and encoding/applying them simultaneously.

- Check for conflicts with other writesets (*conflictsWith(Writeset ws)*).
- Check for conflicts with a given readset (*conflictsWith(Readset ws)*).
- Compact a series of writesets; i.e., remove all item repetitions but the last item update action (*AggregateWriteset.compact(ArrayList<Writeset> wsl)*).
- Get the cardinality of the encapsulated changeset. The case *Writeset.getSize() == 0* is specially significative, since most consistency protocols will stop further transaction processing if the writeset is zero-sized.

Writeset compactation. A generic interface to compact sets of writesets has been implemented. Consistency protocols can this way be independent of the writeset management system. Writeset grouping and compactation is useful for the recovery process, but, in normal operation, merging together several writesets in order to apply them in a single transaction can also give a performance benefit [7]. The compactation algorithm carries out the following rules (assuming all shown operations are applied on the same single tuple):

- If an UPDATE appears after another UPDATE, the first one is cancelled.
- If an UPDATE appears after an INSERT, the first one is cancelled, and the second one is turned into an INSERT.
- If a DELETE and one or more UPDATEs appear in the same writeset list, but not its INSERT, every UPDATE is cancelled.
- If a DELETE and its corresponding INSERT appear in the same writeset list, both operations and every related UPDATE that might have appeared in the middle are cancelled.

Cancelled row modifications are marked by setting their data array to null. The rest of the data used to check for conflicts is left untouched, just in case the certification procedures need them.

3 Performance Analysis

First we tested the performance of the writeset collection and extraction procedures alone, without involving MADIS, which allows a precise comparison of both mechanisms. In the next subsection we show some results obtained when using the studied DbLayers inside MADIS. Finally, in subsection 3.3 future improvements are discussed.

In all tests we have used PCs with a Pentium 4 processor at 3.0 GHz with 1 GB of RAM, with Linux Fedora and PostgreSQL 8.2. In Sect. 3.2, a cluster with 4 machines of this kind has been used.

3.1 DbLayer Alone

In Table 4 we show a comparison of the time costs of writeset management for both DbLayers. For this experiment only the DbLayer was used, without any actual replication. DbLayer_v2 outperforms its antecessor in every case.

	Bare PostgreSQL	DbLayer_v1	DbLayer_v2	Speedup
1. Inserting 10000 rows [ms]	1305	13294	1797	7.40
2. Updating 10000 rows [ms]	1251	13886	1725	8.05
3. Deleting 10000 rows [ms]	914	12524	1754	7.14
4. WS extraction (upd) [ms]	-	22818	184	124
5. WS extraction (ins) [ms]	-	16606	1595	10.41
6. Applying a writeset [ms]	-	63297	2343	27.02
7. Precommit actions [ms]	-	0	10	-

Table 4: Timing measures for writeset extraction and application

As we can see in the table, the overhead of writeset operation is noticeably decreased. Normal transaction work involves almost one tenth of the time used before. Writeset extraction uses very little time per extracted row, being 124 times faster for update operations and 10 times faster for insertions. Writeset application is 27 times faster, which can dramatically speed up the recovery process.

Items 1, 2 are only slowed down by the presence of indexes. Appart from that, DbLayer_v2 imposes no overhead on inserts and updates. Deletes do involve trigger executions, but even if deletes take twice as much time as without any writeset-collection mechanism, they use 7 times less time than the older mechanism.

In DbLayer_v2, GOIDs must be set before committing the transaction. This lasts about 10 ms in our experiments with 10000-update transactions.

	DbLayer_v1	DbLayer_v2	Saved space
Serialized WS's size [bytes] (10.000 rows)	350180	210016	40%

Table 5: Writeset size comparison

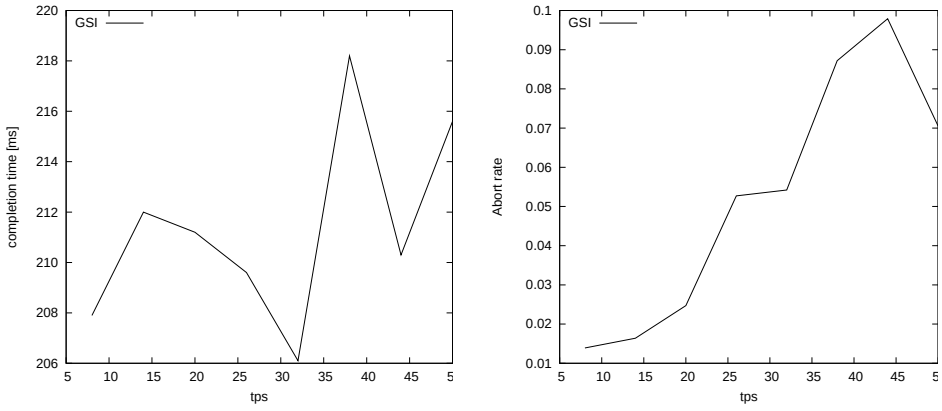
Space usage. In Table 5 we see the size of the encoded writesets of each system. We can see that —with the used table structure—, the new writeset encoding system saved 40% of space. Since DbLayer_v2 encodes writesets into a binary form, they are much more compact. This optimization reduces the size of the messages on the network, as well as the memory used at the replicas.

3.2 Performance inside MADIS

In order to compare both releases of MADIS (based, respectively, on DbLayer_v1 and DbLayer_v2) we have used the SIRC consistency protocol [18] running in a cluster of 4 PCs. To this end, the load consisted

of update transactions performing updates on 20 different tuples from a single table. No read-only transactions were used. Results for DbLayer_v1 were already presented in [18]. In such results, the average completion time for GSI [6] transactions in a 2 TPS load was 1041 ms, raising up to 1292 ms with 4 TPS. The system was unable to exceed a load of 5 TPS with such DbLayer. Abortion rates in a pure GSI load were always greater than 18%.

The performance of the new system is shown in Figure 3. We can observe that when using the new approach, the system can easily reach 50 TPS, which is around 10 times faster than the previous implementation working on the same cluster.



a) Response time. b) Abortion rate.

Figure 3: Experimental results with DbLayer_v2.

On the plot 3.a we can see that the commit time varies between 206 and 220 ms, which represents a small deviation of around 6%. The system does not exhibit overload symptoms in this load range. Note that this is an average completion time 6 times smaller than that achieved with DbLayer_v1 for the same configuration.

The abort rate, shown in Figure 3.b, is very low at every tested transaction rate with the given load (that of the experiment of SIRC): between 1% and 10%. It increases as the system load gets higher, which is due to the fact that, as more transactions exist simultanously in the system, more chances are that conflicts between them arise. Note that in its worst case, these values are almost 50% lower than those obtained with DbLayer_v1. Such results can be explained by the fast completion time with this new DbLayer, minimizing thus the risk of conflicting with other concurrent transactions since the length of all transactions is very short. This proves that the overall performance that can be achieved with the new DbLayer is significantly better than that obtainable with a trigger-based writeset management.

3.3 Future Work

Although the results shown in this section are better than those of the previous system, they can still be improved –both at extracting and at applying writesets– as follows:

- Performing the row serialization in a C function inside the PostgreSQL server. Such function would receive a whole row and would return a compact representation as a byte array. Tricks such as coalescing null values are certainly performed more quickly in C.
- Applying the writesets from a C helper process. Applying writesets from a C client can be up to 4 times faster than performing the same task from Java. Moreover, the COPY command is not usable with the mainstream JDBC driver, but only with the C *pglib* [15] instead. COPY is 150 times

faster than INSERT. Every UPDATE can be replaced by an insertion and a DELETE. By performing DELETE sentences with multiple clauses (WHERE goid=X1 OR goid=X2 ... OR goid=XN), UPDATEs can benefit of COPYing.

- Applying the writesets by directly connecting to the DBMS via sockets. The performance can be similar to that of a C client, but working entirely in the Java domain would make the system easier to maintain.

4 Related Work

As already discussed in the introduction, trigger-based middleware database systems ensure application portability and could be used even for interoperability purposes, like in the *Progress DataXtend RE* middleware [16] that was able to seemlessly *replicate* data among different DBMS (it supported some old versions of IBM DB2, Microsoft SQL Server, and Oracle).

Multiple database replication middleware systems have used trigger-based mechanisms for writeset management. Several examples have already been presented in Section 1. Our group developed part of such systems. Concretely, a performance comparison between the GlobData and MADIS –with its DbLayer_v1 module– systems was already made in [8]. In that case, MADIS performed better than GlobData, since the latter was further penalized by its object-oriented to relational translation. Recall that GlobData provided an ODMS-compliant interface whilst MADIS provides a JDBC interface. In such comparison GlobData finished insert operations with the same performance than MADIS, but it was 50 times slower for updates and 180 times slower for deletions. Note that MADIS results in such paper were better than those presented here for its DbLayer_v1, since in that first deployment its block detection mechanism was not yet developed and no replication protocol was used for those tests. In this paper, we have preferred to also show the system performance using actual replication protocols accompanied by all middleware components (that introduce additional overhead; compare the values shown in Table 4 and Fig. 3 to this end) instead of showing only direct client-to-database interactions that were not filtered by any replication support.

Ganymed [14] is another middleware system based on triggers, although helped with an extension module written in C that was plugged into PostgreSQL. So, it combines the advantages of a solution based on triggers and those of an internals-based solution, although this also inherits some problems of this last approach (some adaptations of the pluggable modules are needed in order to ensure portability). As a result, its performance is much better than that of the first relase of MADIS. Its performance figures [13] –compared with a bare access to the underlying DBMS– are only penalized in a 20% in the best cases up to a 61% in the worst ones. This implies an overall performance also better than that achievable with the system described in this paper. Note, however, that in our MADIS settings we have not especifically tuned PostgreSQL in order to improve its performance; i.e., we have used its default settings since they are usually employed in small-medium enterprises for their DBMS deployments, and they are the common target for our MADIS middleware. Moreover, we have not used any pluggable module for accessing the internals of the underlying DBMS.

5 Conclusions

In this paper we have explained an almost triggerless mechanism to extract writesets in those multiversioned database managers that give read access to internal attributes of the versioning. The resulting performance is significatively better.

The contributions of the explained mechanism over the previous system used in MADIS are multiple:

1. Inserted and updated rows are triggerless distinguished by means of their *xmin* field. Trigger invocation is only used for deleted rows.

2. The GOID is stored along with the payload, thus saving the time to join both tables. Also, the presence of memory caches makes faster accesses to items if they are stored closer to each other.

3. Writesets are encoded in a binary form. Serialization is more efficient. Writesets are smaller. Binary data types in the user data are trivially allowed. Integers are extracted as the DBMS itself stores them. PostgreSQL encodes integers in network order, which can be useful for exchanging data among heterogeneous nodes. These PL/SQL conversion functions perform very quickly.

4. Writesets are applied with prepared statements referencing the rows by GOID.

As we could see in the performance comparative study in Section 3, this approach to writeset extraction is faster than its antecessor in every respect. When using MADIS, we reached a performance up to ten times higher than with the older system. Deletes, which still use triggers, are even though much faster than in DbLayer_v1. Systems allowing indexes on the identifier of the deleting transaction of a row will allow a completely triggerless writeset extraction.

Performance benefits are expected to be obtained even if the metadata are separated again into additional tables. In some cases it could be desirable, to simplify upper layers. The system is flexible enough to enable such heterogeneous configurations, by means of a proper definition of the MADIS catalog. The detection of modified rows, as well as the row encoding would still yield a noticeable performance improvement, in comparison to DbLayer_v1.

Table 6 shows a summarized comparative between the older and the new systems' features.

The quantitative limitations shown in the table are inherent to using 64-bit GOIDs. These limits could be easily overcome by using longer GOIDs, which would not essentially change the system properties.

Acknowledgments

This work has been supported by EU FEDER and the Spanish MEC under grants TIN2006-14738-C02-01 and BES-2004-6500 and by IMPIVA under grant IMIDIC/2007/68.

References

[1] Divyakant Agrawal, Gustavo Alonso, Amr El Abbadi, and Ioana Stanoi. Exploiting atomic broadcast in replicated databases. In *3rd International Euro-Par Conference*, pages 496–503, Passau, Germany, August 1997.

[2] Philip A. Bernstein, Vassos Hadzilacos, and Nathan Goodman. *Concurrency Control and Recovery in Database Systems*. Addison Wesley, Reading, MA, USA, 1987.

[3] Lásaro J. Camargos, Fernando Pedone, and Marcin Wieloch. Sprint: a middleware for high-performance transaction processing. In *ACM EuroSys Conf.*, pages 385–398, Lisbon, Portugal, March 2007.

[4] M. J. Carey and M. Livny. Conflict detection tradeoffs for replicated data. *ACM Trans. on Database Systems*, 16(4):703–746, 1991.

[5] Gregory Chockler, Idit Keidar, and Roman Vitenberg. Group communication specifications: a comprehensive study. *ACM Comput. Surv.*, 33(4):427–469, 2001.

[6] S. Elnikety, W. Zwaenepoel, and F. Pedone. Database replication using generalized snapshot isolation. In *SRDS*, pages 73–84, Orlando, FL, USA, October 2005.

[7] Sameh Elnikety, Steven G. Dropsho, and Fernando Pedone. Tashkent: uniting durability with transaction ordering for high-performance scalable database replication. In *ACM EuroSys Conf.*, pages 117–130, Leuven, Belgium, April 2006. ACM Press.

[8] Luis Irún-Briz, Hendrik Decker, Rubén de Juan-Marín, Francisco Castro-Company, José Enrique Armendáriz-Iñigo, and Francesc D. Muñoz-Escoí. MADIS: A slim middleware for database replication. *Lecture Notes in Computer Science*, 3648:349–359, August 2005.

	DbLayer_v1	DbLayer_v2
MVCC	Not required (generic SQL system)	Required (uses MV attributes)
Needed DBMS features	Triggers, temporary tables	Triggers on delete, temporary tables, a field in each table indicating the last updater, externalization functions
GOID datatype	Text concatenation of (nodeId, tableName, 32-bit local OID)	64-bit integer
GOID storage	In an extra metadata table No effort to hide the metadata	In the same table The metadata must be hidden from the user
Extraction procedure	For-each-row triggers for INSERT, DELETE and UPDATE	Usage of hidden system fields with MVCC information. For-each-row triggers used for deletes
Row serialization	Text-based escaped aggregated string No support to binary fields Field separator using 4 bytes. Need to escape special characters	XDR,minimal binary form, directly usable from C programs Support to binary fields (support to every kind of type supported in Postgres) No field separator (the database schema is known) No need to escape, no in-band signaling
Indexes	On LOID and GOID	On GOID and on *xmin*
Readset extraction support	No special mechanisms	Readset extraction becomes much easier
Need for database schema	No need of schema extraction	Needs the database schema. Table identifiers must be consistent across all nodes
C modules in the server	Yes in the optimized "native" version. Code inside psql is hard to maintain.	Not at all
Incremental ws extraction	Allowed, by emptying the temp tables after extracting	Allowed. Inserts and updates: by using the cmin field; deletes: by truncating the deleted GOID table
Ordered writeset application	Unfeasible unless incrementally extracting writesets after every sentence (costly)	Trivial to do by using $cmin$
Maximum number of tables	No limit	2^{16}
Maximum number of nodes	No limit	2^{16}
Maximum number of items	2^{32} rows per database (Postgres's OID limitation)	2^{32} rows per database (Postgres's OID limitation), extensible to 2^{64} by adding a *serial8* field to every row, instead of the OID. Then, the limit would be per table, not per database.

Table 6: Comparative analysis

[9] Bettina Kemme and Gustavo Alonso. Don't be lazy, be consistent: Postgres-r, a new way to implement database replication. In *26th International Conference on Very Large Data Bases*, pages 134–143, Cairo, Egypt, 2000.

[10] Francesc D. Muñoz-Escoí, Jerónimo Pla-Civera, María Idoia Ruiz-Fuertes, Luis Irún-Briz, Hendrik Decker, José Enrique Armendáriz-Iñigo, and José Ramón González de Mendívil. Managing transaction conflicts in middleware-based database replication architectures. In *Symposium on Reliable Distributed Systems*, pages 401–410, Leeds, UK, 2006.

[11] M. Patiño-Martínez, R. Jiménez-Peris, B. Kemme, and G. Alonso. MIDDLE-R: Consistent database replication at the middleware level. *ACM Trans. Comput. Syst.*, 23(4):375–423, 2005.

[12] Fernando Pedone, Rachid Guerraoui, and André Schiper. Exploiting atomic broadcast in replicated databases. In *4th International Euro-Par Conference*, pages 513–520, Southampton, UK, September 1998.

[13] Christian Plattner. *Ganymed: A Platform for Database Replication*. PhD thesis, Swiss Federal Institute of Technology, Zurich, Switzerland, 2006. Diss. ETH No. 16945.

[14] Christian Plattner and Gustavo Alonso. Ganymed: Scalable and flexible replication. *IEEE Data Eng. Bull.*, 27(2):27–34, 2004.

[15] PostgreSQL. PostgreSQL, the world's most advanced open source database. `http://www.postgresql.org`, 2008.

[16] Progress Software. Progress DataXtend RE documentation. Accessible at: *http://www.progress.com/realtime/docs/faq/peerdirect/index.ssp*, June 2008.

[17] L. Rodrigues, H. Miranda, R. Almeida, J. Martins, and P. Vicente. The GlobData fault-tolerant replicated distributed object database. In *EurAsia-ICT*, pages 426–433, 2002.

[18] Raúl Salinas-Monteagudo, Josep M. Bernabé-Gisbert, Francesc D. Muñoz-Escoí, J. Enrique Armendáriz-Íñigo, and J. R. González de Mendívil. SIRC: A multiple isolation level protocol for middleware-based data replication. In *Intnl. Symp. on Comp. Inform. Sciences*, Ankara, Turkey, November 2007.

[19] M. Wiesmann, A. Schiper, F. Pedone, B. Kemme, and G. Alonso. Database replication techniques: A three parameter classification. In *Proc. of the 19th IEEE Symposium on Reliable Distributed Systems*, pages 206–217, October 2000.

[20] Matthias Wiesmann and André Schiper. Comparison of database replication techniques based on total order broadcast. *IEEE Trans. on Knowledge and Data Engineering*, 17(4):551–566, April 2005.