Journal of Computing and Information Technology - CIT 19, 2011, 2, 71–82
doi:10.2498/cit.1000911

Using Actors to Build a Parallel DBMS

Walid-Khaled Hidouci and Djamel Eddine Zegour

Ecole Nationale Supérieure d'Informatique, (ESI), Oued-Smar, Algeria

In this paper, we present the design and the architecture of a parallel main memory database management system. We focus on concurrency control scheme and recovery. Our prototype is based on the concept of "database actors", an object-oriented data model well suited for parallel manipulations. The storage sub system is built upon distributed Ram-files using SDDS (Scalable Distributed Data Structures) techniques. A nested transaction model is proposed and used to handle concurrency access and recovery. We have also proposed novel approach, based on wait-die, to implement a distributed deadlock prevention technique for our model of nested transactions.

Keywords: parallel DBMS, actor programming, SDDS (Scalable Distributed Data Structures), nested transactions, concurrency control, locking, indirect deadlocks, recovery, checkpoints, 2PC

1. Introduction

In distributed AI, actor programming languages were used to implement distributed inference engines and multi-agent systems. One of the main advantage of such an architecture, is to distribute the effort for problem solving over a number of autonomous agents, thereby reducing the complexity of the problem. Actor languages are also used in the development of distributed open systems where each component is described by an autonomous dynamic object (called 'actor') that encapsulates a part of the global knowledge. To achieve a common goal, actors must collaborate by means of messages passing. The system can be extended or modified dynamically by creating new actors and inserting them into the running system without global reorganization (Agha 1986; Hewitt 1977; Yonezawa 1990).

Actors are also recently used for concurrent applications that combine event-based and multithread architectures, such as GUI desktop applications and message passing in virtual machines (Schäfer, Poetzsch-Heffter 2010; Haller, Odersky 2009; Schippers et al. 2009)

In database field, main memory databases systems (MMDB) are an attractive solution for OLTP database applications which require very high throughput and fast response time (Kallman et al. 2008), because data reside in main memory and secondary memory accesses are only needed for recovery purposes (namely: logging and checkpointing) (Garcia-Molina, Salem 1992, Jagadish et al. 1993; LeGruenwald et al. 1996; Lin, Dunham 1997). Another way to improve performances in database systems is via parallelism and distributed computing (DeWitt, Gray 1992; DeWitt et al. 1996). However, building a parallel database system is known to be among the most complex tasks in software development.

The main idea in Act21[1] project is to apply the concepts of distributed AI, using actor paradigm, to build a parallel DBMS. In this context, we have developed an object-oriented data model based on actor model for concurrency programming (which we called DB-Act). We also used the Scalable Distributed Data Structures (SDDS), a class of dynamic fragmentation methods, to store data in distributed main memory (Litwin et al. 1993). These methods (SDDS) can effectively manage huge data sets over cluster nodes with some interesting features, such as (Alaei et al. 2010):

- Absence of a centralized repository, thereby avoiding bottlenecks;

[1] Act21, ANDRU research project of the Algerian Ministry of High Education and Research.

- Asynchronous updates of access functions, implying the absence of global reorganization;
- Dynamic distribution of data across all nodes in the network, facilitating load balancing in the system.

For this purpose, we have also developed an SDDS method called CTH* (distributed Compact Trie Hashing) to store the databases managed by Act21. As far as we are aware of, it is the first time that SDDS are used to build a parallel DBMS. There were two previous works that concern the coupling of SDDS techniques with existing DBMS (Litwin, Sahri 2004; Ndiyae 2000). Their main purpose is to add some scalability to the database system by maintaining a partitioning scheme dynamically adaptable to the size of the database.

In this paper, we focus on concurrency control and recovery issues that permit to manage nested transactions in Act21 parallel DBMS. The main functionalities of our transactional system are:

- The design of a nested transaction model adapted to our actor methods execution scheme. In this model, upward and automatic downward locks inheritance are provided for transactions a long a hierarchy path.
- The usage of "identifying strings" to timestamps transactions in the concurrency controller. This permits the "wait for commit" relationship to be straightforward handled and hence, facilitate the adaptation of deadlock prevention techniques (like wait-die or wound-wait) in distributed nested transactions.
- The adaptation of physical logging to the SDDS servers level, provides the ability to do fuzzy checkpoints without interfering with client transactions.

There are two major contributions of the present work:

- We show that nested transaction models can be adapted to actor programming paradigm to effectively build complex parallel database systems.
- We present an implementation of nested transactions and recovery techniques adapted to data management with SDDS methods.

The remainder of this paper is organized as follows: In Section 2 we present an architecture overview of Act21 parallel DBMS, we briefly present the concept of "database actors" and the SDDS-based storage sub-system. Section 3 is devoted to our nested transaction model implemented in Act21, concurrency control and recovery techniques are described. Section 4 presents some related works and Section 5 concludes the paper.

2. Architecture Overview

Act21 is a parallel DBMS running on a Linux cluster. It is functionally divided into three layers (see Figure 1): storage sub-system (SDDS servers), Act21 kernel (database actors: DB-Act) and external user layer (SQL and application programs).

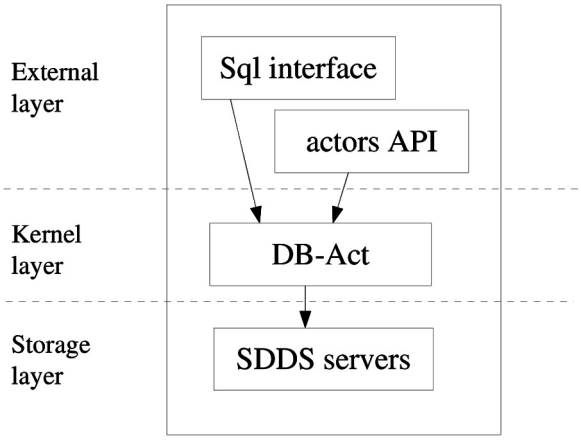

Figure 1. Act21 Architecture.

Act21 data model is very close to the object model, but is based on programming actors instead of the traditional class approach (Hidouci, Zegour 2008). It is mainly based on the notion of "databases actor" introduced below.

Database actors (DB-Act) are a kind of active objects oriented to database applications. There are three types of DB-Act:

- Type actors (T-Act): These actors represent user data types. Their role is to maintain and manage the stored data and to answer queries from other actors in the system. These are similar to classes in the object model.

- Collection actors (C-Act): These actors are containers. They allow to store a collection of data and can be used to represent some multivalued attributes or to store temporary results of queries.
- Request actors (R-Act): These are programming actors. They don't have predefined role as the first two, their behavior must be specified entirely by the user or by an application program. We use this kind of actors to make queries to the database.

These actors communicate through synchronous and asynchronous messages:

1. x=send method-name(parameters,. . .) to receiver-actor /* Synchronous message */
2. send method-name(parameters,. . .): continuation-actor to receiver-actor /* Asynchronous message */

In the first approach, method invocation is a blocking event, since the caller (sender actor) waits until the receiver actor finishes the execution of the called method and returns a response. In the second approach (asynchronous messages), method invocation is non blocking, the sender triggers the execution of the called method and continues in parallel with the called method. Eventually, the result of the invocation can be forwarded to another actor in the system. This is called 'continuation' in actor programming languages (Agha 1986).

The storage sub-system is composed of a set of SDDS servers managing distributed Ram-files using the distributed Compact Trie Hashing (CTH*) fragmentation technique (Zegour 2004). These SDDS servers store the instances of T-Act (Type actors) and the contents of collections (C-Act) i.e. the database.

Distributed Ram-files (files maintained in main memories of the cluster nodes) are composed by a set of SDDS-servers that store records in main memory at each node. Application programs that access these files are called SDDS-clients.

Each node of the cluster contains a set of database actors and one or more SDDS servers. The entry-point of the cluster distributes database actors using PVM[2] tasks and offers SQL and application program interface to the database users (Figure 2).

Some actors (namely T-Act and C-Act) communicate with SDDS-servers to store and manipulate data. These are considered as SDDS-clients. An SDDS-server manages a bucket consisting of a fixed number of pages (I/O transfer unit). Pages contain records of the form: <key, attribute_value>.

We assume that the size of main memory is sufficient for all the SDDS-servers and all actors for a particular node. The entire database is then kept in distributed main memories. The set of disks attached to each node are used only to perform checkpoints and logging activities.

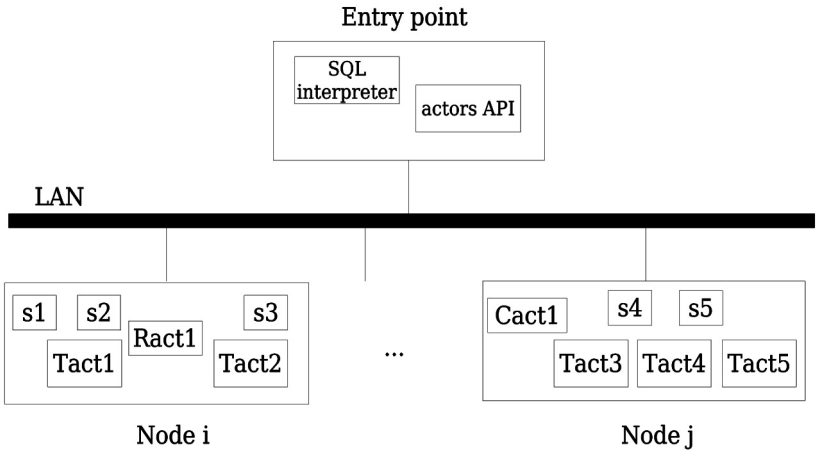

Figure 2. Nodes and entry-point of the cluster.

[2] PVM: Parallel Virtual Machine, a library to build clusters (http://www.csm.ornl.gov/pvm/)

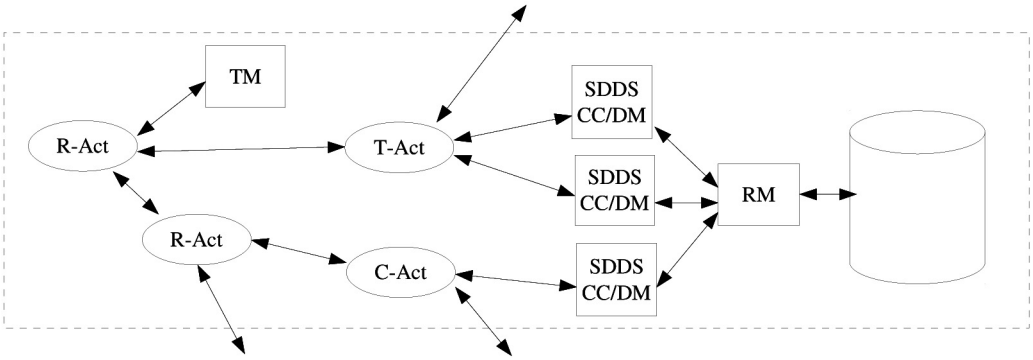

Figure 3. Node components.

3. Transaction Management

Transaction management and recovery concern specifically type actors (T-Act), collections (C-Act) and SDDS-servers, because they represent the distributed database. We recall that the whole T-Act instances and C-Act contents are managed by the SDDS-servers. Thus we have to adapt the conventional transaction architecture (Bernstein et al. 1987) where each site maintains one Transaction Manager (TM), one Concurrency Controller or scheduler (CC) and one Data and Recovery Manager (DRM), to an architecture where each site maintains a multitude of these modules (TM, CC,. . .). Each SDDS server implements one concurrency controller (using strict 2PL) to schedule the received data access operations (reads, writes, inserts, deletes. . .) and one data manager (DM) to execute them.

In Act21, there is one TM per site (or node), whose role is to coordinate the start and termination of transactions. The data access operations are issued in T-Act and C-Act (SDDS-clients managing data files) that are responsible for forwarding them to the correct SDDS-servers to be executed. Thus, T-Act and C-Act are doing some traditional TM's functions. There is also a Recovery Manager (RM) (one per site) that collaborates with all the DM located in the same site (SDDS-servers) to perform checkpointing during normal operations or recovering after a crash (see Figure 3).

The execution of an actor method is considered as a transaction. The nested model is then a good option to take into account the hierarchy of method calls (Figure 4). In this case, a parent transaction can run multiple sub-transactions, themselves can launch other sub-transactions, thus forming a transaction tree whose root is

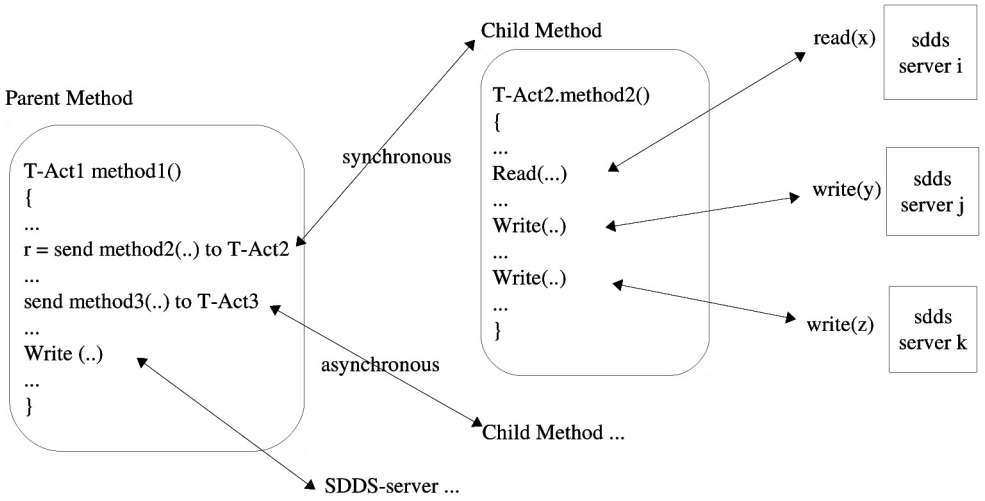

Figure 4. Nested transactions in Act21.

initiated by an external client (user or application program). The root transaction is called top-level transaction (TL-transaction).

Because synchronous messages are blocking calls, we consider that the execution of a method (Method2) following a synchronous message from a transaction T (Method1), does not generate a new transaction, unlike the model of Harder and Rothermel (Harder, Rothermel 1993). In fact, the same transaction T continues its execution in a different client space (that of the actor running Method2). When the called method (Method2) terminates, the execution of T returns to the initial client space (Figure 4).

However, the execution of a method (Method3 in Figure 4) following an asynchronous message (non blocking call) from a transaction T generates a new sub-transaction T' that runs in parallel with T. When T' terminates (precommits), the locks it had acquired during its execution will be "inherited" by its parent transaction T, offering the opportunity for other sub-transactions of T to ask for these locks if they need them. This is the principle of "upward inheritance of locks". If T' aborts, the locks it held (i.e. those that have been acquired through access operations) will be released.

Recall that a lock inherited by a transaction T does not give it the right to access the data, but just to limit its sphere of visibility to T and to its sub-transactions. Thus, T and its descendants will have the right to request the lock.

In Harder and Rothermel model (Harder, Rothermel 1993), the application programmer has the ability to transmit some of the locks of a parent transaction to its sub-transactions. This is known as "controlled downward inheritance". In our case, we opted for a slightly different approach, we used an "automatic downward inheritance" at the end of the parent transaction. That is, if a sub-transaction T' requests a lock held by its parent transaction T, it will proceed only when T reaches the end of its execution (just before the precommit state). This procedure seems simpler (and more realistic) to use for the programmer who does not need to know the types of locks acquired by each access operation it uses.

When a non TL-transaction finishes its execution, its locks become visible to its descendants. It then starts waiting for the termination (precommit) of its sub-transactions. When done, it precommits and signals its termination to its parent transaction.

On the other hand, when a TL-transaction T (root transaction) finishes its execution (reach the COMMIT statement), the 2PC protocol is started by the local TM to ensure that all SDDS-servers that have participated in the execution of data access operations issued from transactions in T hierarchy take the same decision (commit or rollback). At this point, all the precommited sub-transactions commit effectively.

This implies that the commitment of a root transaction (Top Level Transaction) is conditioned by the precommit of all its sub-transactions. If a sub-transaction is aborted due to an error or failure, all other transactions of the same hierarchy must abort too. But if a sub-transaction is aborted due to a deadlock, it will, just, automatically be reactivated.

3.1. Identifying the distributed transactions

When a TL-transaction begins its execution in a node, the local TM assigns it a unique identifier in the cluster (a timestamps formed by the concatenation of the clock value and the task ID of the TM). In a synchronous call, the called method receives the same identifier as that of the calling method (because it is part of the same transaction). In the case of an asynchronous message, a new sub-transaction is generated in the system. It registers with its local TM that assigns it a new timestamp (clock+TM's TID). The identifier of the new sub-transaction is then the result of the concatenation of the parent transaction identifier and the local timestamps. Thus each transaction Ti is globally identified by a sequence of Ids (T1, T2, . . ., Ti), describing a path in the tree hierarchy between T1 (the root transaction) and Ti. This identifiers sequence is what we call the "identification string" of Ti.

The "identification string" models the nesting of transactions in the system and will be used by the CC of an SDDS server for allocating locks and detecting indirect deadlocks. For instance, in Figure 5, the identification string for T2 is "T,T2" because T is the parent of T2, similarly, the identification string of T6 is "T,T1,T4,T6" and for T7 it is "T',T3,T7" and so on . . .

3.2. Distributed deadlock management

This way of identifying transactions facilitates detection of deadlocks, even those deemed difficult in the nested context.

In the example of Figure 5, we have 2 TL-transactions (T and T'). T1 and T2 are sub transactions of T, then T can't terminate until T1 and T2 precommit first. This is known as "wait-for-commit" relationship. Similarly, T' waits for T5 and T3, T1 waits for T4, who waits for T6. By transitivity of the wait-for-commit relationship, we have that T waits for T6 and T' waits for T9.

Recall that by the rule of upward locks inheritance, if a sub-transaction precommits, its locks are inherited by its parent. Now in the example of Figure 5, if transaction T2 requests a lock held by transaction T5, T2 waits for termination of T5 (since strict 2PL is applied) and also it waits for the termination of T', because, when T5 precommits, its locks are inherited by T' (its parent) and T2 cannot acquire the requested lock until T' terminates.

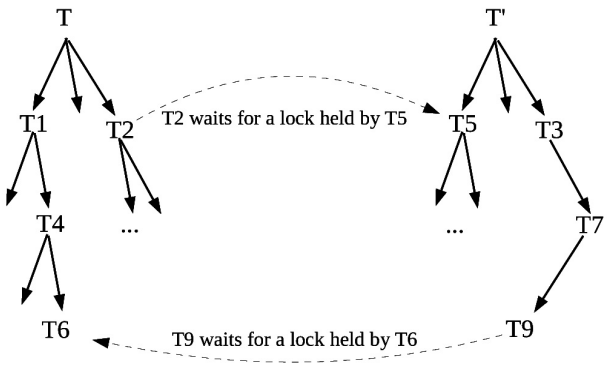

Figure 5. Indirect deadlock.

Since T waits for T2 (wait-for-commit relationship) and T2 waits for T' (lock request and upward inheritance between T5 and T'), we can actually say that T waits for T'. Moreover, if transaction T9 (a descendent of T') requests another lock held by T6 (a descendent of T), for the same reasons we see that T9 is actually waiting for all ascendants of T9 (T4, T1 and T) because of the upward locks inheritance rule and T' is actually waiting for T9 because of the wait-for-commit relationship. Thus we are in a situation where T waits for T' and T' waits for T. This is called an "indirect deadlock".

To prevent all deadlock situations, we have adapted wait-die technique to manipulate timestamps in the form of identification strings. In wait-die, when a transaction Ti with timestamps ts(Ti) requests a lock held by a transaction Tj, Ti is allowed to wait only if $ts(Ti) < ts(Tj)$, otherwise Ti is rolled back with unchanged timestamps ts(Ti).

The extension of the wait-die method concerns the generalization of the test $(ts(Ti) < ts(Tj))$ to take into account the identifying strings and to know what is the sub-tree of nested transactions to be aborted, if necessary.

The following pseudo-code illustrates this changes:

```
/* L1, L2 are the identifying strings of respectively Ti and Tj */

bool test( Identifying_String L1 , Identifying_String
L2 , Timestamps &Victim)
{
   if ( Empty(L1) && !Empty(L2) ) {
            Victim = Head(L2) ;
            return true ;
   }
   else if ( !Empty(L1) and Empty(L2) ) {
               Victim = Head(L1) ;
               return false ;
      }
      else if ( Head(L1) < Head(L2) ) {
            Victim = Head(L2) ;
            return true ;
         }
         else if ( Head(L1) > Head(L2) ) {
                  Victim := Head(L1) ;
                  return false ;
         }
         else /* Head(L1) == Head(L2) */
         return test ( Tail(L1), Tail(L2), Victim ) ;
}
```

If the test failed (return false), Victim is then the root of the sub-hierarchy to abort. It is the highest ascendant of Ti, for which Tj is not a descendant.

In the nested model, transactions belong to enclosing spheres (Figure 6). All transactions derived directly or indirectly from a transaction T (i.e. the whole sub-tree rooted at T), belong to the sphere of T. Each transaction defines an enclosing sphere. In the example of Figure 6, when a transaction T4 waits for the termination of a transaction (T6) to acquire a lock, T4 waits in fact, the termination all the transactions in the most enclosing sphere including T6 that does not include T4 (sphere of T2). When all transactions in this sphere terminate, all locks that were held or inherited by these transactions become free to other transactions in the next enclosing sphere (in this case, it is the T sphere).

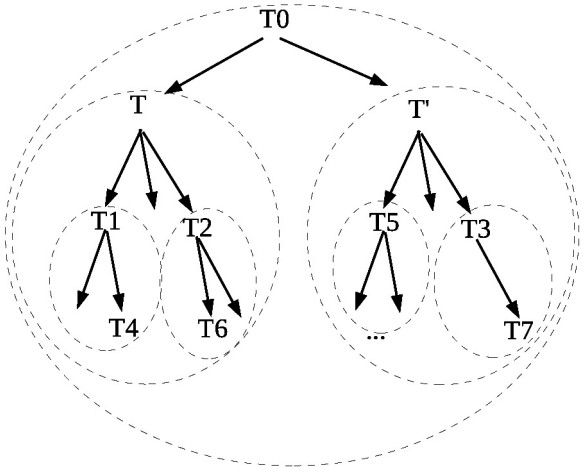

Figure 6. Enclosing spheres.

In deadlock prevention, the main reason which leads us to abort the highest possible ascendant that is not in the same sphere as the transaction holding the lock, is dictated by the wait-for-commit relationship.

In Figure 6, we suppose that T5 is holding a lock to a data item A, and that T2 is holding a lock to another data item B. We also suppose that Ti identifier is less than Tj identifier (i.e. ID(Ti) < ID(Tj) if and only if i < j. For the TL-transactions T and T', we suppose in this example that ID(T') < ID(T)).

In that case, if T5 requests a lock on B, it is allowed to wait for the termination of T2 (and thus its ascendant T) because "T0,T',T5" < "T0,T,T2" in the test function (T' < T). Now while T5 is waiting for the termination of T2 (and of its parent T), if another transaction in the sphere of T (say T4 for instance) requests a lock on A (which is held by T5), T4 must be rolled back (because the test "T0,T,T1,T4" < "T0,T',T5" is false: T > T'). If we only abort T4, the deadlock remains, because while T4 is automatically rolled back, its parent T can't terminate and then T5 remains blocked waiting for the lock on B to leave the sphere of T. To solve this problem, it would have had to cancel the whole sphere T instead of only T4. This is done in the test function by positioning Victim to be the first transaction ID in the identifying string L1 which is higher than its equivalent in L2 (in that case Victim == T).

Locks can be shared (S) or exclusive (X) and are managed at page level. The DM executes the (read/write) operations on pages located in its bucket. No I/O are needed to perform this task (data is memory resident), however the logging and checkpointing activities may require the RM of the node to access the local disks (e.g. for transaction commitment with 2PC).

3.3. Physical logging

SDDS servers maintain active transaction tables (ATT) (one per server) that keep track of transactions execution (see Figure 7).

All update operations on a page x generate physical undo record : <x, pos, len, old_value> and a physical redo record: <x, pos, len, new_value> where pos and len indicate respectively the position and the offset, in the page, of the updated bytes.

Each ATT entry contains both the undo and redo logs for one transaction. The undo log is used to rollback the effects of a transaction if it is aborted, whereas the redo log is flushed to the global stable log (on the disk) when the transaction commits.

When a transaction Ti commits, a record <Ti, Nb_rec> is added to the head of its redo log and then the whole redo log is flushed to the global stable log on the disk. Nb_rec indicates the number of log records pertaining to transaction Ti.

The stable log contains only the effects of committed transactions in serial order equivalent to

the concurrent (but serializable) execution of these transactions in the MMDB:

. . .<T1,3>,<x,. . .>,<y,. . .>,<z,. . .>,
|————T1's redo log————|

<T2,2>,<x,. . .>,<y,. . .>,
|———T2's redo log———|

<T3,4>,<x,. . .>,<y,. . .>,<z,. . .>,<t,. . .>. . .
|—————T3's redo log—————|

- the global stable log on the disk -

When a transaction terminates (commit/abort), its entry (and its log records) is discarded from the active transaction table ATT.

3.4. Fuzzy checkpointing

To recover from failure (system crash), where the content of main memory is lost, the global stable redo log can be replayed from the beginning or (if possible) from a "previous consistent copy" of the database to bring up the latest consistent state before the crash. There are two major drawbacks with this "naive" approach :

- Making a consistent copy of the database is a very time consuming task, because trans-

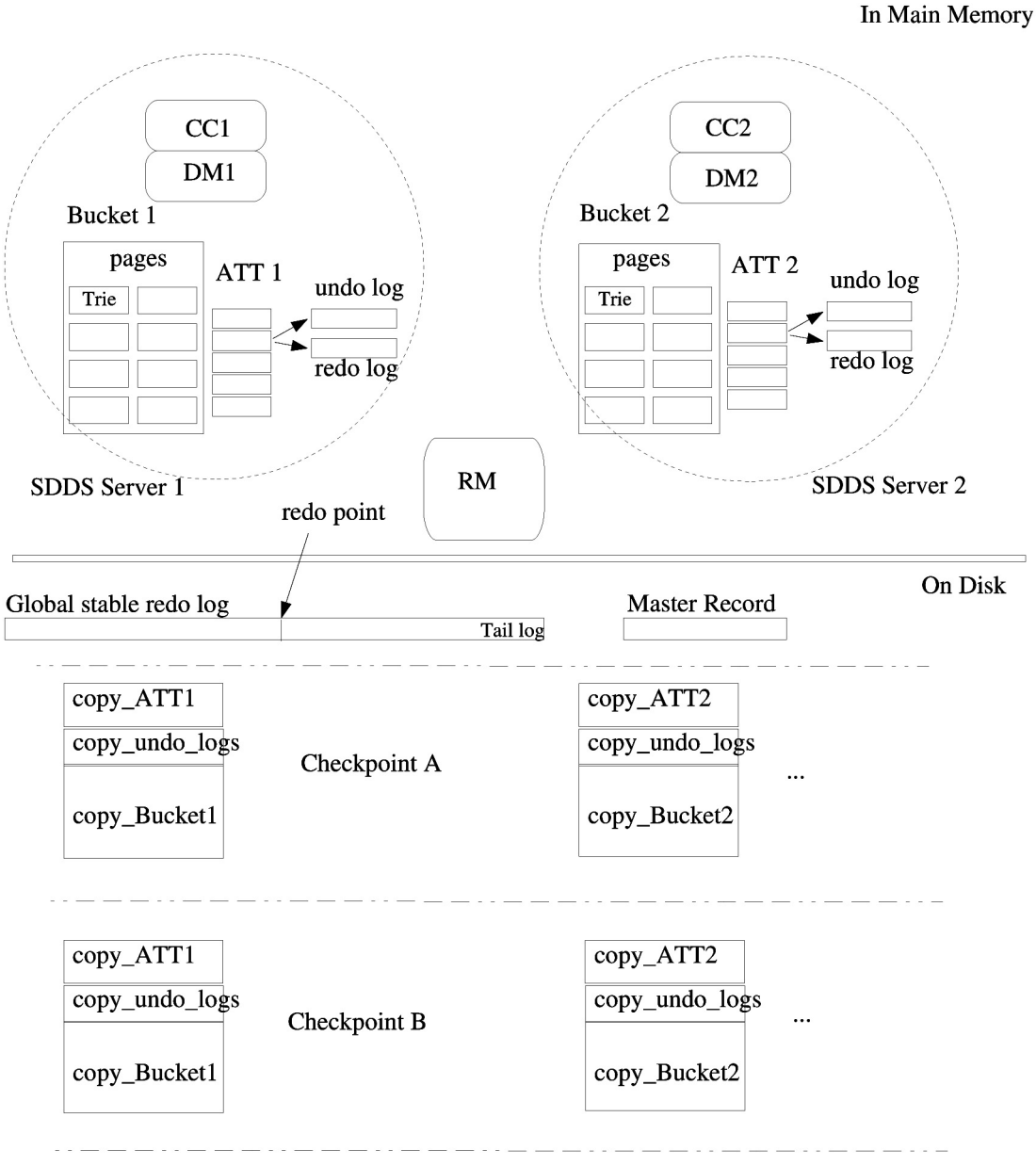

Figure 7. Node storage sub-system.

action processing has to be stopped during the flush of the bucket content.

- The huge number of records in the stable log makes the redoing procedure very inefficient, when recovering from a crash.

Instead of making a consistent copy, we can "emulate" it by doing a snapshot of the bucket content while active transactions are processed concurrently (the backup copy obtained is then not consistent, since uncommitted updates could be copied out). Then a copy of ATT with the undo logs is made. Indeed undoing the effects of uncommitted transactions from the inconsistent database copy makes it consistent. This is a "fuzzy" checkpoint, since the dump procedure does not require the system to be quiescent, i.e. a page locked (even exclusively) by an active transaction can be dumped during the checkpoint.

In Act21 we have adapted for the SDDS a variant of a fuzzy checkpoint (called ping pong checkpoint (LeGruenwald et al. 1996)) that keeps two copies of the database on the disk. Each one contains an inconsistent dump and the undo logs of active transactions during the dump.

For each SDDS server we maintain on the disk two copies (A and B) of the bucket, the ATT and the related undo-logs. A master record is also maintained and is composed of: an indicator of the current checkpoint (A or B), a pointer to the redo point in the stable log (the start point for scanning the stable log when recovering) and a "Directory" that keeps informations necessary for rebuilding all the SDDS servers and actors (SDDS clients: T-Act and C-Act) hosted in the site, when recovering.

Recall that buckets are composed of a set of pages (that contain data records). A page is "clean" when no updates have occured since the previous checkpoint. A page is "dirty" when one or more transactions have updated their content since the previous checkpoint.

The page state (clean/dirty) is indicated by two state bits: 00 for clean and 01 or 10 for dirty.

When a checkpoint occurs, all dirty pages (state bits = 01 or 10) are flushed to one copy of the disk database and their state bits are incremented modulo 3, i.e. those having state bits = 01 remain dirty (10) and those having state bits = 10 become clean (00).

When the next checkpoint occurs, the old dirty pages (10) and new ones (01) are flushed to the other copy of the disk database. Thus each dirty page is flushed twice, once to each copy of the database, on two consecutive checkpoints. If the system crashes before completing the current checkpoint, the other copy can be used to recover the database to the latest consistent state.

The checkpoint procedure below runs periodically without interfering with normal transaction processing:

1. *Let x = (A or B), such that x is different from the current checkpoint (stored in the master record)*
2. *Note the end of the stable log in a variable (redo point)*
3. *For each bucket in the node, do:*

   write dirty pages to the bucket copy x (on the disk)

   write ATT and the undo-logs to their copy x
4. *replace the master record with the new one: < x, redo-point, Directory > (in one atomic operation)*

To recover from a crash, RM loads the current checkpoint and rolls back the transactions that were active during this checkpoint. Finally we replay the global redo log beginning from the redo-point (saved in the master record) to reach the latest consistent state before the crash.

The recovery procedure is as follows:

1. *read the current checkpoint (A or B) in x, the redo point and the directory from the master record*
2. *rebuild the SDDS servers*
3. *For each SDDS server, do:*

   load its ATT and undo logs from copy x on the disk

   load the bucket pages and reset their state bits (00)

   roll back the active transactions from the loaded ATT, using the undo logs
4. *Replay the global redo log and update the ATT, undo logs, and dirty bits accordingly.*

3.5. Tests

We have implemented some modules of Act21 prototype in a virtual parallel environment (networked Linux boxes + PVM):

- An SQL interface that produces an execution plan from an SQL query, consisting of some R-Act and C-Act that cooperate with the existing T-Act to produce the query result.
- An interpreter of PACT language for executing actor's scripts.
- An actor manager that implements the actor's primitives (New_Act, Send, Broadcast,...) and the transaction management (TM) for T-Act (coordinators in 2PC).
- A storage manager based on CTH* SDDS method. Concurrency controllers, data managers and cohorts (in 2PC) are also implemented in the SDDS servers.
- A recovery manager that uses fuzzy checkpoints is also implemented.

We have conducted some preliminary experiments. The results are presented below:

The bucket size is fixed to 400 pages for each SDDS server. A page is a 1kb block.

Our parallel virtual machine is composed of 4 PC (Pentium 3 750Mhz – 256MB RAM) connected by an Ethernet switch (10-100 Mb/s). Clients and servers are PVM tasks.

Each client generates 1000 serial transactions, each one is composed of a random number of I/O operations varying from 1 to 10, example: read page(103) from server(1), write page(82) to server(4), write page(261) to server(3), ... page and server numbers are also randomly generated.

The tests consist of launching n parallel clients and observing some parameters such as response time, throughput, and the number of aborted transactions for dead-lock prevention (wait-die). When a transaction is aborted, the client waits 10 ms before restarting the transaction.

The response time is computed as the mean of the transaction execution time (from begin transaction to commit). The throughput is the number of successful commitments done in 1 second in the system. It's computed as the sum of all client throughputs in the same period.

The next tables (Tables 1, 2 and 3) resume the tests for respectively 2, 3 and 4 servers:

# clients	*Resp. Time*	*Throughput*	*Aborts*
10	23 ms	425 tps	1.6 %
20	41 ms	481 tps	3.9 %
30	66 ms	450 tps	7.3 %
40	89 ms	440 tps	10.7 %
50	114 ms	400 tps	15.4 %
60	140 ms	420 tps	20.6 %

Table 1. Performances with 2 servers.

# clients	*Resp. Time*	*Throughput*	*Aborts*
10	18 ms	544 tps	1.1 %
20	34 ms	566 tps	2.6 %
30	54 ms	540 tps	4.3 %
40	73 ms	521 tps	7.2 %
50	91 ms	510 tps	9.8 %
60	114 ms	481 tps	13.1 %

Table 2. Performances with 3 servers.

# clients	*Resp. Time*	*Throughput*	*Aborts*
10	18 ms	535 tps	0.5 %
20	29 ms	659 tps	1.9 %
30	44 ms	658 tps	3.1 %
40	59 ms	644 tps	4.7 %
50	76 ms	620 tps	7.4 %
60	93 ms	603 tps	9.6 %

Table 3. Performances with 4 servers.

We notice that the speed-up is about 66% to 79% for the response time when the number of servers doubles (from 2 servers to 4 servers). At the same time the percentage of aborted transactions is decreased by a factor of $1/3$ to $1/2$. The throughput computed in this experiment is relative to one server only and the global throughput is expected to be higher when the number of servers increases.

4. Related Works

A lot of work has been conducted in the area of high performance main memory database systems. (Bohannon et al. 1997) present the architecture of a main memory storage manager called Dali. It is a toolkit providing recovery and concurrency control features. Its primary goals are to serve as the lowest level of a database system and to support transaction processing in performance-critical applications. The main feature of the Dali storage manager is the use of a direct access to data in shared memory rather than via inter-process communication, which is relatively slow, but is not portable to a parallel shared nothing environment.

Scalable Distributed Data Structures (SDDS) are another way to use main memories as resident storage for databases. They are usually based on a dynamic hashing function as a partitioning scheme and provide good performances and scalability. In Act21 we use an SDDS method called CTH* (Zegour 2004) that is order preserving (like range partitioning (Litwin et al. 1994)) and less sensitive to data skew. We have implemented a fuzzy recovery technique in order to avoid loss of data in presence of failure. Other techniques have been studied to extend the SDDS methods with a high availability property, based on parity and reed-solomon codes (Cieslicki et al. 2010; Litwin, Schwarz 1997).

One open problem known to be the most serious limit to parallel MMDBs is the relatively high cost of the atomic commitment protocol for distributed transactions. In (Park, Yeom 1999) an approach that combines the advantages of the pre-commit and group commit in parallel MMDB while avoiding the consistency problem is presented.

5. Conclusion

In this article, we have presented an approach to build parallel Main Memory DBMS using the concepts of distributed open systems and of database actors. An overview of the architecture, including the data model and the storage sub-system, have been presented briefly.

We have also presented a model of nested transaction that is suited for our parallel DBMS. We used identifying strings to timestamp the transactions which permit to facilitate the management of locks in the tree hierarchy. We have also adapted the wait-die distributed deadlock prevention procedure to handle this kind of timestamps.

Recovery techniques (fuzzy ping pong checkpointing) are also adapted to the use of "Scalable Distributed Data Structures" SDDS as a storage manager for the parallel main memory DBMS.

Some improvements can be provided to our architecture, particularly for 2PC extensions to support "group commit". This can be done by keeping in main memory the tail of the global redo log and using techniques like those presented in (Lee et al. 2004; Park, Yeom 1999) where log records and some precedence information are centralized in one site.

References

[1] G. AGHA, ACTORS. a model of concurrent computation in distributed systems. *MIT press*, (1986).

[2] S. ALAEI, M. GHODSI, M. TOOSSI, Skiptree: A new scalable distributed data structure on multidimensional data supporting range queries. *Computer Communications*, vol. 33, Issue 1 (2010).

[3] P. A. BERNSTEIN, V. HADZILACOS, N. GOODMAN, *Concurrency Control and Recovery in Database Systems*. Addison-Wesley, 1987.

[4] P. BOHANNON, D. LIEUWEN, R. RASTOGI, A. SILBERSCHATZ, S. SUDARSHAN, The Architecture of the Dali Main Memory Storage Manager. *The Intl. Journal on Multimedia Tools and Applications*, 4(2) (March 1997).

[5] D. CIESLICKI, S. SCHAECKELER, T. SCHWARZ, Maintaining and checking parity in highly available Scalable Distributed Data Structure. *Journal of Systems and Software*, vol. 83, Issue 4 (2010).

[6] D. DEWITT, J. GRAY, Parallel Database Systems: The Future of High Performance Database Systems. *Communications of the ACM*, vol. 35, no. 6 (June 1992).

[7] D. DEWITT, J. NAUGHTON, J. SHAFER, SH. VENKATAMAN, Parallelizing OODBMS traversals: a performance evaluation. *VLDB journal*, vol. 5 n\r 1 (Jan 1996), pp. 3–18.

[8] H. GARCIA-MOLINA, K. SALEM, Main Memory Database Systems: An Overview. *IEEE Trans. Knowl. Data Eng.*, vol. 4, n\r 6 (Dec 1992), pp. 509–516.

[9] P. HALLER, M. ODERSKY, Scala Actors: Unifying thread-based and event-based programming. *Theoretical Computer Science*, vol. 410, Issue 2-3 (2009).

[10] T. HARDER, K. ROTHERMEL, Concurrency Control Issues in Nested Transactions. *The VLDB Journal*, vol. 2, n\r 1 (1993), pp. 39–74.

[11] C. E. HEWITT, Viewing Control Structure as Patterns of Passing Messages. *Artificial Intelligence*, (1977).

[12] W. K. HIDOUCI, D. E. ZEGOUR, An actor like data model for a parallel DBMS. *Journal of Digital Information Management*, vol. 6 issue 3 (June 2008).

[13] H. V. JAGADISH, A. SILBERSCHATZ, S. SUDARSHAN, Recovering from Main Memory Lapses. *Proceedings of the 19th VLDB*, Conf. Dublin, Ireland, (1993).

[14] R. KALLMAN, H. KIMURA, J. NATKINS, A. PAVLO, A. RASIN, S. ZDONIK, E. P. C. JONES, S. MADDEN, M. STONEBRAKER, Y. ZHANG, J. HUGG, D. J. ABADI, H-store: a high-performance, distributed main memory transaction processing system. *Proceedings of the VLDB Endowment*, vol. 1, Issue 2 (2008).

[15] L. GRUENWALD, J. HUANG, H. M. DUNHAM, J.-L. LIN, A. CH. PELTIER, Survey of Recovery in Main Memory Databases. *Engineering Intelligent Systems*, 4/3 (Sept. 1996), pp. 177–184.

[16] I. LEE, H. Y. YEOM, T. PARK, A New Approach for Distributed Main Memory Database Systems: A Causal Commit Protocol. *IEICE Trans. Inf. & Syst.*, vol. E87-D, n\r 1 (January 2004).

[17] J. LIN, M. H. DUNHAM, A Survey of Distributed Database Checkpointing. *Distributed and Parallel Databases*, 5 (1997), pp. 289–319.

[18] W. LITWIN, S. SAHRI, Implementing SD-SQL server: A Scalable Ditributed Database System. *Intl. Workshop on Distributed Data and Structures*, WDAS 2004, Carleton Scientific.

[19] W. LITWIN, J. E. SCHWARZ, LH*rs: A high availability scalable distributed data structure using Reed Solomon codes. *CERIA Res. Rep.*, 99-2 (1997), Paris 9.

[20] W. LITWIN, M. A. NEIMAT, D. SCHNEIDER, LH*: A Scalable Distributed Data Structure. *CERIA Res. Rep.*, (Nov. 1993), Paris 9.

[21] W. LITWIN, M. A. NEIMAT, D. SCHNEIDER, RP*: A Family of Order Preserving Scalable Distributed Data Structures. *Proc. Of 2O th conf. VLDB*, (1994), Chile.

[22] Y. NDIYAE, W. LITWIN, T. RISCH, Scalable Distributed Data Structures for High-Performance Databases, *Tech. Rep*, Ceria Paris Dauphine Univ. 2000.

[23] T. PARK, H. Y. YEOM, A Distributed Group Commit Protocol for Distributed Data Systems. *Tech. Rep.*, PDCS99-GC, Department of computer engineering, Sejong Univ., Korea, 1999.

[24] J. SCHÄFER, A. POETZSCH-HEFFTER, Writing concurrent desktop applications in an actor-based programming model. *Proc. of the 3rd International Workshop on Multicore Software Engineering*, (2010), New York, USA.

[25] H. SCHIPPERS, T. VAN CUTSEM, S. MARR, M. HAUPT, R. HIRSCHFELD, Towards an actor-based concurrent machine model. *Proc. of the 4th workshop on the Implementation, Compilation, Optimization of Object-oriented Languages and Programming Systems*, (2009), New York, USA.

[26] A. YONEZAWA, ABCL: an object-oriented concurrent system. *MIT press*, (1990), Cambridge MA.

[27] D. E. ZEGOUR, Scalable Distributed Compact Trie Hashing. *Inform. Soft. Tech.*, (2004), Elsevier.

Received: August, 2006
Accepted: March, 2011

Contact addresses:
Walid-Khaled Hidouci
Ecole Nationale Supérieure d'Informatique, (ESI)
BP 68M Oued-Smar, Algeria
e-mail: w_hidouci@esi.dz

Djamel Eddine Zegour
Ecole Nationale Supérieure d'Informatique, (ESI)
BP 68M Oued-Smar, Algeria
e-mail: d_zegour@esi.dz

WALID-KHALED HIDOUCI is an associate professor of computer science at ESI Graduate School of Computer Science in Algiers. His main topics of interests are: database systems, data structures, operating systems and parallel programming.

DJAMEL EDDINE ZEGOUR is a professor of computer science at ESI Graduate School of Computer Science in Algiers. His main research interests include data structures and algorithms, compilers, functional programming, object-oriented programming and distributed computing.