

# Consistent Main-memory Database Federations under Deferred Disk Writes

Rodrigo Schmidt<sup>\*,†</sup>

Fernando Pedone<sup>†</sup>

<sup>\*</sup>École Polytechnique Fédérale de Lausanne (EPFL), CH-1015 Lausanne, Switzerland

Phone: +41 21 693 6668 Fax: +41 21 693 6490

E-mail: rodrigo.schmidt@epfl.ch (contact author)

<sup>†</sup>Università della Svizzera Italiana (USI), CH-6904 Lugano, Switzerland

Phone: +41 91 912 4695 Fax: +41 91 913 8536

E-mail: fernando.pedone@unisi.ch

EPFL Technical Report IC/2005/017

27 Apr 2005

## Abstract

Current cluster architectures provide the ideal environment to run federations of main-memory database systems (FMMDBs). In FMMDBs, data resides in the main memory of the federation servers. FMMDBs significantly improve performance by avoiding I/O during the execution of read operations. To maximize the performance of update transactions as well, some applications recur to deferred disk writes. This means that update transactions commit before their modifications are written on stable storage and durability must be ensured outside the database. While deferred disk writes in centralized MMDBs relax the durability property of transactions only, in FMMDBs transaction atomicity may be also violated in case of failures. We address this issue from the perspective of log-based rollback-recovery in distributed systems and provide an efficient solution to the problem. Besides presenting a mechanism to ensure atomicity in FMMDBs, the paper bridges the gap between rollback-recovery in message-passing distributed systems and distributed transaction processing, and shows how results developed in the first context can be exploited in the second.

**Keywords:** dependency tracking, consistency, rollback-recovery, distributed transactions, main-memory database systems.

## 1 Introduction

Continuous technology improvements have reduced the cost and boosted the performance and memory capacity of commodity computers. As a consequence, powerful computer clusters are becoming increasingly affordable and common. These architectures provide the ideal environment for mechanisms targeting high-performance computing such as *main-memory database systems* (MMDBs [12]). MMDBs overcome the latency limitations of traditional disk-based databases by storing the data items in the main memory of the

servers [13]. By avoiding disk I/O, both transaction throughput and response time can be improved. Moreover, as transactions do not have to wait for data to be fetched from disk, concurrency becomes less important for performance and some approaches have considered lowering the overhead of transaction synchronization by reducing concurrency (e.g., locking tables instead of rows, executing transactions sequentially [12, 18]). Although most current MMDBs have been designed for single servers (e.g., [16, 21]), recent work has also suggested their use in the context of clusters of computers, where the existence of multiple components can increase reliability and performance [28].

For recovery reasons, MMDBs also keep a copy of the database in disk. Queries execute entirely using data in main memory, but update transactions have to modify the state in disk. In fact, accessing the disk is the main overhead incurred by update transactions executing in an MMDB. To maximize the performance in such cases, some applications recur to *deferred disk writes*. This means that update transactions commit before their modifications are written on stable storage. Since disk access is deferred until after transactions commit, various transaction logs can be grouped and asynchronously written at once on disk. This approach alone harms the durability property of transactions, but as we will show later in the paper, some applications may prefer to ensure durability outside the database for performance reasons. Section 5 presents an example in the context of database replication where deferred disk writes improve performance without relaxing any property of transactions from the end user’s perspective.

This paper considers a federation of main-memory database systems (FMMDB) where data is partitioned among different servers running local MMDBs. Global transaction termination is implemented by atomically grouping the commit decision of various local sub-transactions. As in a centralized database, applications can choose to use deferred disk writes in order to improve system’s performance. Deferred disk writes, however, introduce additional complexities in an FMMDB. In a single-server system, only the durability property may be violated in case of database crash; this happens as long as log writes respect the commit order of their respective transactions. By contrast, in a federation a crash may render a server inconsistent with respect to the others, compromising transaction atomicity as well. Consider a simple federation composed of two database servers. If a transaction  $t$  updates data in both servers, commits, and one of the servers crashes before making the updates locally persistent, when the failed server recovers from the failure, it will have forgotten  $t$ ’s local execution. In this case, atomicity is violated by the fact that only part of  $t$  persists: the one in the server that did not crash.

We address the problem of deferred disk writes in a federation of MMDBs using a novel approach

that borrows from the theory of rollback-recovery in distributed systems [9]. The basis of this theory is the identification of dependencies between process states. This allows the recognition of consistent global states (i.e., those composed of local states such that no one depends on the other) to which the application should be rolled back in case of failure. Efficiently applying these results in the context of transaction processing systems, however, is not straightforward and requires revisiting the original theory. Transaction processing systems create dependencies between database states differently from usual message-passing distributed systems. In the latter, dependencies are based on *causality*<sup>1</sup>; in the former, dependencies are created by *read* and *write* operations on database objects during the execution of transactions. As an example, consider a simple distributed transaction execution composed of two servers and one client. Two transactions execute sequentially:  $t_1$  and  $t_2$ . Figure 1 depicts the execution where read requests are denoted by R, write requests by W, and commit requests by C;  $\alpha$ ,  $\beta$ , and  $\gamma$  represent the database states at the servers. Database server  $S_i$  changes its state after an update transaction commits at  $S_i$ ; the state remains the same if the transaction only reads the local state or aborts. In a usual message-passing system, state  $\beta$  would precede  $\gamma$  since there is a causal path between the two states (depicted in bold in Figure 1). However, since  $t_1$  only reads  $\beta$ , it turns out that  $\beta$  and  $\gamma$  are in fact concurrent. This example shows that causality is actually too strong to capture database state dependencies, and a more appropriate formalism is needed.

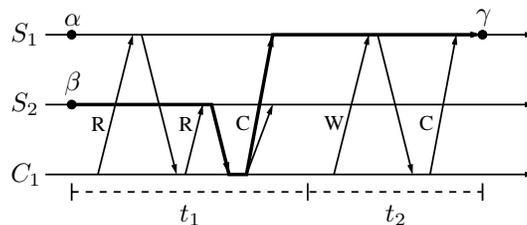


Figure 1: False (causal) dependency

Although there seems to be a clear overlap between research on rollback-recovery in distributed systems and distributed transaction processing, very few works have exploited their intersection—as we will discuss later in this paper, the only exceptions we are aware of are [5] and [11]. This paper bridges the gap between rollback-recovery in message-passing distributed systems and distributed transaction processing, showing more generally how results developed in the former can be exploited in the latter. We revisit the original dependency definitions, developed for message-passing systems, and propose a new one based on database

<sup>1</sup>Event  $e$  causally precedes  $e'$  iff (i) they execute in the same process,  $e$  before  $e'$ , or (ii)  $e$  refers to the sending of a message and  $e'$  refers to its receipt, or (iii)  $e$  and  $e'$  are related by the transitive closure of the two previous conditions [20].

states, minimal for distributed transaction environments and allowing efficient tracking implementation. In doing so, we hope to pave the way for further research on reusing existent work on rollback-recovery to build efficient distributed transactional systems. Moreover, this paper illustrates the applicability of our approach in the context of an FMMDDB with deferred disk writes. Our solution is optimistic in the sense that we do not force servers to synchronize their accesses to disk (e.g., using a two-phase commit-like protocol), but track dependencies between database states during normal execution and, in case of failure, bring the system to a consistent state during recovery.

This paper is structured as follows. Section 2 presents our computational and execution models. Section 3 explores consistency and dependencies in a transactional system. Our algorithms to ensure correctness of execution in a federation of main-memory databases with deferred disk writes are shown in Section 4. Section 5 exemplifies a scenario where deferred disk writes would be helpful and practical. We compare our approach with existent works in the field in Section 6 and conclude the paper in Section 7. Theorem and correctness proofs are found in the Appendix.

## 2 System model

We assume a system composed of two disjoint sets of processes: the set of servers  $\mathcal{S} = \{S_1, S_2, \dots, S_n\}$  and the set of clients  $\mathcal{C} = \{C_1, C_2, \dots, C_m\}$ . Servers are stateful—their state is given by the data values stored on them, and clients are stateless—their state can be recreated by the servers’ state in case of crash. We assume that clients interact only with servers by submitting transaction requests and waiting for their response. All communication between clients and servers is done through message exchanging.

The system is asynchronous: we make no assumptions about the time needed for processes to execute and messages to be transmitted.<sup>2</sup> Communication links may lose messages but if both sender and receiver remain up “long enough,” lost messages can be retransmitted and are eventually received. A process can fail by crashing, stopping its execution and losing its volatile state. Crashed servers must recover eventually to ensure the system’s availability at the federation level, otherwise clients who want to access data on some failed server cannot progress. Servers are equipped with stable storage whose contents survive crashes. The system execution alternates between normal execution periods and recovery sessions. A recovery session starts when a failure is noticed and ends after the servers are in a globally-consistent state.

---

<sup>2</sup>Notice that the implementation of a distributed transactional environment may require stronger system assumptions (e.g., to suspect failed clients and servers). The ideas described in this paper, however, are oblivious to such assumptions.

## 2.1 Database servers and transactions

Servers store disjoint subsets of the entire database accessible to the clients and run local main-memory databases. We call the complete set of servers  $\mathcal{S}$  a *main-memory database federation*. Each server executes local transactions, where a transaction is a (most likely short) sequence of read and write operations on data items, followed by a commit or an abort operation, but not both. A transaction is called *read-only* if it does not contain any write operations, and *update* otherwise. Transactions are abstracted by the following traditional properties [14]:

**Atomicity:** A transaction's changes to the database state are atomic: either all happen or none happen.

**Consistency:** A transaction is a correct transformation of the database state.

**Isolation:** Any execution of a set of transactions is *equivalent* to a serial execution of the same transactions.

Durability is relaxed as a result of deferred disk writes. If there is a failure before a transaction is made durable, but after its commit, such a transaction is *lost*. In that case, after recovery the execution has to proceed as if the transaction had never executed. Lost transactions differ from aborted ones because they actually commit and their results may have been seen by other transactions. A transaction that is not lost throughout the execution is called *persistent*. We redefine transaction durability under deferred disk writes through the two properties below:

**Weak Durability:** If an update transaction commits and the system does not crash for “long enough,” the transaction is persistent.

**Consistent Persistence:** A persistent transaction is *preceded* only by other persistent transactions.

In order to make the previous definitions sound, two things still have to be defined : equivalence between executions of sets of transactions and precedence between transactions. Let a *transaction history*  $H$  be a partial order on all the operations executed by a set of transactions, necessarily defined for all *conflicting operations*—two operations are said to *conflict* if they both operate on the same data item and one of them is a write [6].  $H$  represents a real execution (not necessarily serial) of the transactions in the system. Two histories over the same set of transactions are equivalent if they order *conflicting operations* of non-aborted persistent transactions in the same way. We say that a transaction  $t_1$  *directly precedes* a transaction  $t_2$  in  $H$  if there is a pair of conflicting operations,  $(o_1 \in t_1, o_2 \in t_2)$ , such that  $o_1$  precedes  $o_2$  in  $H$ . The precedence relation between transactions is given by the transitive closure of the direct precedence relation.

We assume the concurrency control in each server is based on shared read locks and exclusive write locks in the whole local database, characterizing the multiple-read single-write behavior found in some

MMDBs (e.g., [18]). This allows us to abstract client operations as Reads and Writes performed over an entire database state. Obviously, our approach can be extended to the case where concurrency control is done at a finer granularity by considering each piece of data (e.g., tables) as a separated “logical” database server inside a physical server in the federation. Our concern is to extend Weak Durability and Consistent Persistence from the local database servers to the federation, and ensure that none of the other transaction properties are violated in the presence of failures.

A server  $S_i$  updates its state to a new one after committing a transaction that wrote some value on the server. This creates a sequence of states  $\sigma_i^0, \sigma_i^1, \dots$ , where  $\sigma_i^j$  represents  $S_i$ 's state after committing the  $j$ -th local update transaction. Notice that this definition does not take into consideration the incarnation of a database state. Therefore, a database state  $\sigma_i^\gamma$  may be not unique throughout the system's execution. If the update transaction that created a certain database state  $\sigma_i^\gamma$  is lost, the server is rolled back to a previous state  $\sigma_i^{\gamma-k}$  ( $k > 0$ ) and as long as new update transactions are executed on  $S_i$ , a new state  $\sigma_i^\gamma$  will be created. However, as we assume that processes synchronize before resuming execution after a failure, no two states  $\sigma_i^\gamma$  can coexist in the same normal execution period and dependency tracking can be unaware of it.

## 2.2 Clients' execution model

Clients execute a sequence of steps. In each step, a client (a) performs some local computation, (b) submits a request to a database in the federation, and (c) waits for its response. We abstract the set of possible database requests by the following primitives, where  $op$  represents an operation to be submitted to the database. Details about their implementation are given in Section 4.2.

**Read**( $tid, S_i, op$ ): Operation  $op$  reads some data item stored in  $S_i$  on behalf of transaction  $tid$ .

**Write**( $tid, S_i, op$ ): Operation  $op$  updates some data item stored in  $S_i$  or creates it on behalf of  $tid$ .

**Commit**( $tid$ ): Requests the global commit of transaction  $tid$  in the federation.

**Abort**( $tid$ ): Requests the global abort of transaction  $tid$  in the federation.

To start a new transaction, a client simply generates a new unique transaction identification number ( $tid$ ), to be used in all servers. When the middleware in a server receives the first operation on behalf of  $tid$  (either a Read or a Write), it creates a new transaction abstraction in the local database and relates it to  $tid$  in order to submit future operations to the database in the same local transaction abstraction. When all the operations in all servers referent to a certain transaction have been executed, the client executes the Commit

request to ensure global commit. After a Commit or Abort request, no more requests with the same *tid* are executed by the client.

At any point during a transaction’s execution, a server that is participating in it can unilaterally abort its local sub-transaction. This is done, for example, if the local sub-transaction is involved in a deadlock or the server suspects that the client responsible for this transaction has crashed. To ensure transactions’ atomic commit in the absence of failures we use a simple blocking protocol: the client sends a message to all involved servers asking them to prepare to commit. Every involved server sends its committing/aborting vote to the client and the other servers. A server commits the transaction iff it receives a “commit” vote from every involved server. Moreover, if the client receives the “commit” vote from every server, it knows the transaction has been committed. To abort a transaction, a server simply sends an “abort” message to all involved servers. If the client fails and some server does not receive such a message, eventually this server will unilaterally abort the transaction. It is clear that this algorithm (derived from two-phase-commit [6, 14]) works in the absence of failures. Section 4.2 shows how Atomicity is preserved in the presence of failures albeit no disk write is executed during transaction commit.

### 3 Consistent global database states

When a failure occurs, we must make sure that the system will restart from a previous consistent global state. Our approach is motivated by research on log-based rollback-recovery mechanisms in the message-passing model [9]. In this section we precisely define the notion of consistency, analyze the conditions that make a global database state consistent, and show what must be done by our algorithm to have it recoverable.

#### 3.1 Database-state dependencies

When it comes to the creation of database-state dependencies, we are only interested in committed transactions, since aborted ones do not change the state of the database and their results are never seen by other transactions. Therefore, we consider only committed transactions in definitions and theorems presented in this section and, for simplicity, omit this condition in their statement. Additionally, some extra notation is necessary. We use  $RW(t)$  to represent the set of server states accessed by transaction  $t$  throughout its execution.  $W(t) \subseteq RW(t)$  is the set of server states updated by  $t$ . This means that if  $\sigma_i^\alpha \in W(t)$  and  $t$  commits, a new database state  $\sigma_i^{\alpha+1}$  is *created* by  $t$ . Furthermore, we define  $R(t) = RW(t) \setminus W(t)$  to be the set of server states read by  $t$ .

State dependencies in the transactional model are due to the three well-known types of transaction dependencies: write-read, write-write and read-write [6, 14]. Definition 1 below captures the notion of transaction dependency using our terminology in a simplified manner, where write-read and write-write dependencies are represented by condition (a), read-write dependencies by condition (b), and transitive dependencies by condition (c). In this context, a database state precedes another one if the former is overwritten by a transaction that either creates the latter or precedes the transaction that does it. This means that the first state will have already been overwritten by the time the second one is created and, therefore, no transaction (or external viewer) can see both of them together in the same global database state. Definition 2 presents this idea more formally. If  $\sigma_a^\alpha \rightarrow \sigma_b^\beta$ , given the transaction dependencies, any consistent global state containing  $\sigma_b^\beta$  must not contain a state  $\sigma_a^\gamma$  such that  $\gamma \leq \alpha$ .

**Definition 1** *Transaction  $t$  precedes  $t'$  ( $t \rightarrow t'$ ) iff*

- (a)  $\exists \sigma_c^\gamma \mid \sigma_c^{\gamma-1} \in W(t) \wedge \sigma_c^\gamma \in RW(t')$ ; or
- (b)  $\exists \sigma_c^\gamma \mid \sigma_c^\gamma \in R(t) \wedge \sigma_c^\gamma \in W(t')$ ; or
- (c)  $\exists t'' \mid t \rightarrow t'' \wedge t'' \rightarrow t'$ .

**Definition 2** *State  $\sigma_a^\alpha$  precedes  $\sigma_b^\beta$  ( $\sigma_a^\alpha \rightarrow \sigma_b^\beta$ ) iff*

- (a)  $\exists t \mid \sigma_a^\alpha \in W(t)$ ; and
- (b)  $\exists t' \mid \sigma_b^{\beta-1} \in W(t')$ ; and
- (c)  $t = t' \vee t \rightarrow t'$ .

### 3.2 Consistent and recoverable database states

A global state of the federation is a set composed of a local state for each database server in the system. We base our consistency criterion on the notion of *serializability* [6] and formalize it in Definition 3.

**Definition 3** *A global database state  $\{\sigma_1^{\alpha_1}, \dots, \sigma_n^{\alpha_n}\}$  in a given history  $H$  is consistent iff it represents the database state after the serial execution of an ordered set of transactions  $T = (t_1, t_2, \dots, t_l)$  such that:*

- (a) *all transactions in  $T$  are non-aborted persistent transactions in  $H$ ;*
- (b)  $\forall t \in T: t' \rightarrow t$  in  $H \Rightarrow t' \in T$ ; and
- (c)  $\forall t_a, t_b \in T: t_a \rightarrow t_b$  in  $H \Rightarrow a < b$ .

From Definition 3, a global state is consistent if it is created by the execution, in a correct order, of a subset of the executed transactions left-closed under the transaction dependency relation. Theorem 1 shows a simpler characterization of a consistent global state based on the database-state dependency relation we introduced in Definition 2.

**Theorem 1** A global database state  $G = \{\sigma_1^{\alpha_1}, \dots, \sigma_n^{\alpha_n}\}$  is consistent iff  $\forall \sigma_i^{\alpha_i}, \sigma_j^{\alpha_j} \in G : \sigma_i^{\alpha_i} \not\rightarrow \sigma_j^{\alpha_j}$ .

As an example, consider Figure 2(a), where we show a possible execution scenario in which five transactions are applied to a federation of two database servers. We omit message exchanges between clients and servers and depict only the operations performed against the databases grouped by transaction, where W means a database write and R means a database read. Figure 2(b) shows the dependencies between the database states created by the executed transactions. We depict only the direct dependencies and omit the transitive ones. Based on these dependencies, it is possible to identify a total of seven consistent global states according to Theorem 1, all of them depicted in Figure 2(b). Global state number 4 is reached after the serial execution of  $(t_1, t_2, t_3)$  and global state number 6 is achieved by  $T = (t_1, t_2, t_3, t_4)$ .

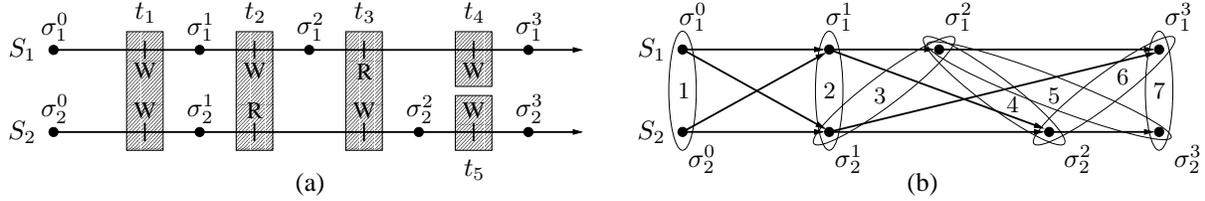


Figure 2: Example of consistent global states

Because of the weak durability property described in Section 2.1, if one server crashes, it might not recover in the same state it was just before the crash. According to Consistent Persistence, locally ensured by the MMDB running in the server, an entire suffix of the local sequence of states may be lost after a failure as a result of deferred disk writes. As this new local state may be inconsistent with the state of the other servers, to ensure Consistent Persistence globally the entire system may have to roll back to a previous consistent global state. Clearly, we want this state to be as recent as possible to roll back the least number of committed transactions. In order to satisfy this condition we have to distinguish between stable database states, already written on the server's disk, and volatile database states, whose local durability has not been ensured yet. A consistent global state is recoverable if it is composed of stable database states. When some database servers crash, the recovery algorithm must make the system roll back to its most recent recoverable consistent global state, or *recovery line*. A non-faulty server that wants to make its volatile states part of the recovery line should make them stable before executing the recovery algorithm.

The main determiner of the recovery line in some history  $H$  is the last stable state of each server  $S_i$ , which we denote by  $\sigma_i^{last}$ . As Theorem 2 shows, the recovery line for a given execution scenario is composed of the last persistent state not preceded by any state  $\sigma_i^{last}$ .

**Theorem 2** *The recovery line  $R$  for a given history is determined by*

$$R = \bigcup_{i=1}^n \{\sigma_i^k \mid k \text{ is the maximum value such that } \forall 1 \leq j \leq n : \sigma_j^{last} \not\rightarrow \sigma_i^k\}$$

Figure 3 depicts two examples of recovery line determination based on the scenario presented in Figure 2 (volatile states are depicted between square brackets, e.g.,  $[\sigma_i^j]$ ). Each figure shows a dependency graph with all the states dependent on some state  $\sigma_i^{last}$  as empty circles. Therefore, the recovery line is formed by the state represented by the last filled circle in each database server. In the first case (Fig. 3(a)), we suppose that after executing all transactions, both servers crash and states  $\{\sigma_1^2, \sigma_1^3, \sigma_2^3\}$  are lost due to deferred disk writes. Our second example (Fig. 3(b)) supposes that server  $S_2$  fails before making states  $\{\sigma_2^2, \sigma_2^3\}$  stable. Server  $S_1$  completes all the disk writes but has to roll back when  $S_2$  recovers because  $\sigma_1^3$  depends on  $\sigma_2^{last}$ . Notice that in both cases some stable database state had to be rolled back. Were this action not taken, the federation would be in a inconsistent state after recovery.

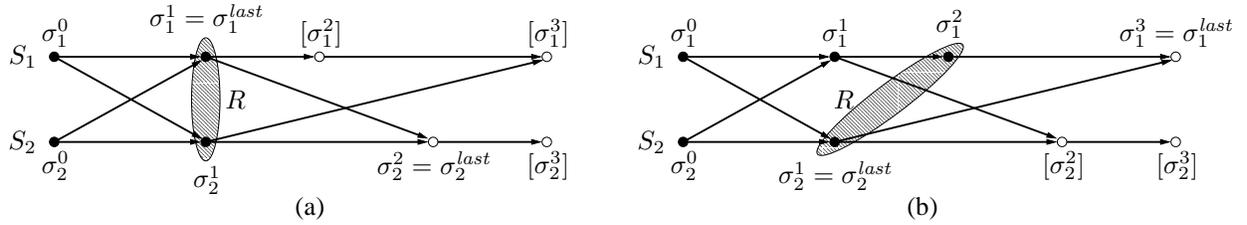


Figure 3: Recovery-line determination

## 4 Database-oriented rollback-recovery

Basically, our approach corresponds to propagating database-state dependencies during normal execution and orchestrating the global rollback to the recovery line after some failure. In this perspective, our algorithm can be seen as optimistic in the sense that no synchronization is performed during normal execution to ensure Consistent Persistence and databases can do the deferred disk writes at their own pace. Our algorithm synchronizes servers only when a failure indeed occurs and, therefore, must be treated.

### 4.1 Thrifty dependency tracking

Definition 2 relates database-state dependencies with transaction dependencies. Theorem 3 below shows that it is also possible to keep track of database-state dependencies without having to gather information about transaction dependencies.

**Theorem 3** Server state  $\sigma_a^\alpha$  precedes  $\sigma_b^\beta$  ( $\sigma_a^\alpha \rightarrow \sigma_b^\beta$ ) iff

- (a)  $\exists t \mid \sigma_a^\alpha \in W(t) \wedge \sigma_b^{\beta-1} \in RW(t)$ ; or
- (b)  $\exists t, \sigma_c^\gamma \mid \sigma_a^\alpha \rightarrow \sigma_c^\gamma \wedge \sigma_b^{\beta-1}, \sigma_c^\gamma \in RW(t)$ ; or
- (c)  $\exists t, \sigma_c^\gamma \mid \sigma_a^\alpha \rightarrow \sigma_c^\gamma \wedge \sigma_b^{\beta-1} \in RW(t) \wedge \sigma_c^{\gamma-1} \in W(t)$ .

Theorem 3 comes from the fact that a transaction  $t$  accesses a consistent partial state of the federation and generates, after its execution, another consistent partial state. These states work like partial snapshots of the execution and, therefore, incur constraints in the ordering of events. As in the real world, if an event is captured in a snapshot and another one is not (i.e., it took place after the snapshot was taken), then the snapshot is a “proof” that the first event happened before the second.

We exemplify conditions (a), (b) and (c) of Theorem 3 in Figure 4, where  $S_{Before}$  refers to the (partial) federation state accessed by transaction  $t$ , either a read-only or update transaction, and  $S_{After}$  refers to the federation state generated after  $t$ 's execution. In the figure, scenarios (a<sub>1</sub>) and (a<sub>2</sub>) correspond to condition (a) of Theorem 3, and scenarios (b) and (c) correspond to conditions (b) and (c), respectively. Figure 4(a<sub>1</sub>) depicts the situation where  $\sigma_a^\alpha \in W(t)$  and  $\sigma_b^{\beta-1} \in R(t)$ . When  $t$  commits, the new state it creates contains  $\sigma_a^{\alpha+1}$  and  $\sigma_b^{\beta-1}$ . Therefore, as  $\sigma_a^\alpha$  necessarily precedes this state and  $\sigma_b^\beta$  succeeds it, it is clear that  $\sigma_a^\alpha \rightarrow \sigma_b^\beta$ . Figure 4(a<sub>2</sub>) represents the case where  $\sigma_a^\alpha, \sigma_b^{\beta-1} \in W(t)$ . As  $\sigma_b^\beta$  is created by  $t$ , it did not exist before  $t$ 's commit; whilst  $\sigma_a^\alpha$  existed only until before  $t$  commits, since it is updated by  $t$ . This means that, as no other transaction can see a state between  $S_{Before}$  and  $S_{After}$ ,  $\sigma_a^\alpha \rightarrow \sigma_b^\beta$ . In Figure 4(b),  $\sigma_a^\alpha \rightarrow \sigma_c^\gamma$  and  $\sigma_b^{\beta-1}, \sigma_c^\gamma \in RW(t)$ . This means that  $\sigma_b^{\beta-1}$  and  $\sigma_c^\gamma$  belong to the federation state accessed by  $t$ . Similarly to the situation depicted in Figure 4(a<sub>1</sub>),  $\sigma_a^\alpha$  must precede  $\sigma_b^\beta$ . Lastly, let us consider the case where  $\sigma_a^\alpha \rightarrow \sigma_c^\gamma$  and  $\sigma_b^{\beta-1}, \sigma_c^{\gamma-1} \in W(t)$ , shown in Figure 4(c). The state generated after  $t$ 's commit contains  $\sigma_b^\beta$  and  $\sigma_c^\gamma$ . Since  $\sigma_a^\alpha$  precedes  $\sigma_c^\gamma$ ,  $\sigma_a^\alpha$  has been already updated before  $\sigma_c^\gamma$  is created. As  $\sigma_c^\gamma$  and  $\sigma_b^\beta$  are created together, surely  $\sigma_a^\alpha \rightarrow \sigma_b^\beta$ . The scenario of condition (c) where  $\sigma_a^\alpha \rightarrow \sigma_c^\gamma, \sigma_b^{\beta-1} \in R(t)$  and  $\sigma_c^{\gamma-1} \in W(t)$  resembles the situation depicted in Figure 4(b), just exchanging  $S_{Before}$  for  $S_{After}$ , and is not depicted in the figure.

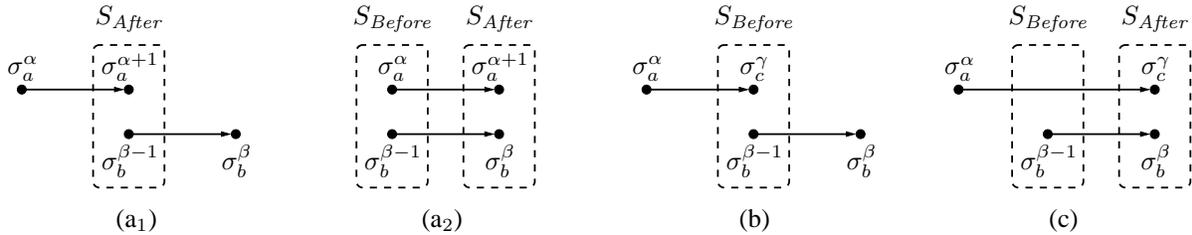


Figure 4: Dependencies based on the federation states accessed and created by a transaction

## 4.2 Dependency tracking algorithm

Theorem 3 leads to a simple way to gather database-state dependencies on-the-fly during the system's execution. Assume each state  $\sigma_i^\alpha$  has associated with it a data structure  $D(\sigma_i^\alpha)$  representing the set of states it depends on (we show later how this structure can be implemented efficiently). To update  $D(\sigma_i^\alpha)$ , upon committing, every transaction  $t$  executes the steps described in Algorithm 1, where  $D(S_i)$  is an auxiliary data structure local to  $S_i$ , initially empty.  $D(S_i)$  represents the dependencies that must be attributed to the next state to be created at server  $S_i$ . Lines 1–3 are directly associated with the three possible database-state precedences presented in Theorem 3. Line 4 associates a dependency data structure with every new database state created by the transaction.

---

### Algorithm 1 Dependency tracking

---

**During commit of transaction  $t$  at  $S_i$**

- 1:  $\forall \sigma_b^{\beta-1} \in RW(t) : D(S_b) \leftarrow D(S_b) \cup W(t)$
  - 2:  $\forall \sigma_b^{\beta-1}, \sigma_c^\gamma \in RW(t) : D(S_b) \leftarrow D(S_b) \cup D(\sigma_c^\gamma)$
  - 3:  $\forall \sigma_b^{\beta-1} \in RW(t) : D(S_b) \leftarrow D(S_b) \cup \bigcup_{\sigma_c^\gamma \in W(t)} D(S_c)$
  - 4:  $\forall \sigma_j^t \in W(t) : D(\sigma_j^{t+1}) \leftarrow D(S_j)$
- 

We now explain how Algorithm 1 can be implemented in practice. We start analyzing how MMDBs write database state changes on stable storage. In MMDBs, data changes are stored on disk only after an update transaction has issued a commit request. This means that no action must be undone in case of failures and the transaction log is typically redo-only, and can be implemented by simply storing the set of operations performed by each transaction [8]. Regardless its particular implementation details, each entry in a redo-only log represents the new state created by the respective update transaction executed. We can therefore associate the database state  $\sigma_c^\gamma$  with the  $\gamma^{th}$  entry in the log of Server  $S_c$ . To keep track of  $\sigma_c^\gamma$ 's dependencies, the only thing we have to do is to write the structure  $D(\sigma_c^\gamma)$  with its respective transaction's entry on  $S_c$ 's transaction log.

For a practical implementation, we must provide a way to implement the data structure  $D(\sigma_c^\gamma)$  efficiently with respect to space complexity. As dependencies are transitive and continuous in the sequence of states of a database server, it is not difficult to see that to keep track of the complete set of dependencies of a given state  $\sigma_c^\gamma$ , we need to store only the last state of each server on which  $\sigma_c^\gamma$  depends. If  $\sigma_c^\gamma$  depends on  $\sigma_a^\alpha$  ( $\alpha > 0$ ), clearly it also depends on  $\sigma_a^0, \dots, \sigma_a^{\alpha-1}$ . Therefore, a complete set of state dependencies can

---

**Algorithm 2** Complete algorithm for dependency tracking
 

---

CLIENT STUB	
1: <u>Data Structures</u>	25: The server continuously waits for an event:
2: $\Lambda$ : set of servers	26: <b>when</b> receive $\langle \text{READ}, tid, op \rangle$ from $C_i$
3: <u>Begin_Transaction</u> ()	27: $result \leftarrow submit(tid, op)$
4: $\Lambda \leftarrow \emptyset$	28:     send $\langle result \rangle$ to $C_i$
5: return <i>unique tid</i>	29: <b>when</b> receive $\langle \text{WRITE}, tid, op \rangle$ from $C_i$
6: <u>Read/Write</u> ( $tid, S_i, op$ )	30: $result \leftarrow submit(tid, op)$
7: $\Lambda \leftarrow \Lambda \cup \{S_i\}$	31:     append $op$ to $opSet_{tid}$
8: send $\langle \text{READ/WRITE}, tid, op \rangle$ to $S_i$	32:     send $\langle result \rangle$ to $C_i$
9: wait for $\langle result \rangle$ from $S_i$	33: <b>when</b> receive $\langle \text{PREPARE}, tid, \Lambda_i \rangle$ from $C_i$
10: return $result$	34: $\Lambda_{tid} \leftarrow \Lambda_i$
11: <u>Commit</u> ( $tid$ )	35: <b>if</b> willing to commit <b>then</b>
12: send $\langle \text{PREPARE}, tid \rangle$ to all $S_i \in \Lambda$	36: <b>if</b> $opSet_{tid} \neq \emptyset$ <b>then</b>
13: wait for $\langle \text{VOTE}, tid, v_i, DV_i \rangle$ from all $S_i \in \Lambda$	37: $DV_{aux} \leftarrow DV$
14: return $(\forall S_i \in \Lambda : v_i = \text{YES})$	38: $DV_{aux}[i] \leftarrow DV_{last}[i] + 1$
15: <u>Abort</u> ( $tid$ )	39: <b>else</b> $DV_{aux} \leftarrow DV_{last}$
16: send $\langle \text{ABORT}, tid \rangle$ to all $S_i \in \Lambda$	40:         send $\langle \text{VOTE}, tid, \text{YES}, DV_{aux} \rangle$ to $C_i \cup \Lambda_{tid}$
	41: <b>else</b> send $\langle \text{VOTE}, tid, \text{NO}, \perp \rangle$ to $C_i \cup \Lambda_{tid}$
<b>SERVER WRAPPER</b>	42: <b>when</b> $\exists tid$ such that $\forall S_i \in \Lambda_{tid} : \text{received}$
17: <u>Data Structures</u>	$\langle \text{VOTE}, tid, v_i^{tid}, DV_i^{tid} \rangle$ from $S_i$
18: $opSet_{tid}$ : ordered set of operations	43: <b>if</b> $\forall S_i \in \Lambda_{tid} : v_i^{tid} = \text{YES}$ <b>then</b>
19: $DV, DV_{last}$ : array[1..n] of integer	44:         submit( $tid, \text{COMMIT}$ )
20: $\Lambda_{tid}$ : set of servers	45: <b>for all</b> $S_i \in \Lambda_{tid}$ <b>do</b>
21: <u>Initialization</u>	$\forall j : DV[j] \leftarrow \max(DV[j], DV_i^{tid}[j])$
22: $\forall tid : opSet_{tid} \leftarrow \emptyset, \Lambda_{tid} \leftarrow \mathcal{S}$	46: <b>if</b> $opSet_{tid} \neq \emptyset$ <b>then</b>
23: $\forall 1 \leq j \leq n : DV[j] \leftarrow -1$	$DV_{last} \leftarrow DV$
24: $DV_{last} \leftarrow DV$	47:         asynchronously write entry $\langle opSet_{tid}, DV \rangle$
	in the transaction log
	50: <b>when</b> receive $\langle \text{ABORT}, tid \rangle$ from $C_i$
	51:     submit( $tid, \text{ABORT}$ )

---

be represented by a dependency vector  $DV$  with  $n$  entries, in which  $DV[i]$  stores the index of the most recent state dependency from server  $S_i$ . This idea and nomenclature is inspired by dependency tracking for rollback-recovery in the message-passing model [32].

We divide our dependency tracking algorithm into two parts: the client stub and the server wrapper, both shown in Algorithm 2. Only one *when* clause executes at a time, and only after its condition holds. If more than one *when*-clause's condition hold at the same time, any one is chosen to execute. We assume however that the execution is fair, that is, unless the server crashes, every *when* clause with a condition that holds will be executed. To submit transaction operations to the local MMDB, a server makes use of the *submit* interface. Moreover, to make it clear that our approach does not introduce any extra disk operations, all log operations are dealt by our algorithm, that is, all *submit* calls access only data in the server's main memory.

At the client side it is only necessary to keep track of the set of servers accessed during the execution of a transaction (line 2).<sup>3</sup> Basically, all operations performed by the client stub are straightforward and have little to do with dependency tracking. Dependency tracking takes place at commit making use of the synchronization messages exchanged by the servers to ensure transactions' atomicity. While analyzing the algorithm, remember that we assume Isolation is ensured by a simple database-locking mechanism and global Atomicity during normal execution is given by a variation of two-phase-commit, described in Sections 2.1 and 2.2, respectively. Although we make no explicit use of these two properties, they ensure the dependencies captured by our algorithm are consistent with the dependencies indeed created in the distributed database. Adapting our approach to different concurrency control and atomic commit mechanisms is possible, although we do not address this issue in the paper.

Briefly, each server keeps two dependency vectors during execution,  $DV$  and  $DV_{last}$ .  $DV$  implements  $D(S_i)$  (the dependencies to be attributed to the next state created) and  $DV_{last}$  stores the dependencies of the current database state. A server sends, together with the answer to the PREPARE request issued by the client, a dependency vector containing the dependencies the transaction should forward to all accessed servers based on the operations performed in the local database (lines 35-41). This information is sent not only to the client but also to the other involved servers. Finally, when a server  $S_i$  receives the messages from all servers involved in the transaction, it updates its  $DV$  (line 45-46) and, if the transaction wrote some data in the database, the server performs a local state transition (lines 48-49).

A correct implementation of Algorithm 1 is ensured by the dependencies propagated by the servers in the VOTE messages. Dependencies referent to line 1 of Algorithm 1 are gathered in line 38 of Algorithm 2. Dependencies given by line 2 of Algorithm 1 are gathered in line 39 of Algorithm 2 if the server was only read by the transaction, or in line 37 if the server was updated. Line 37 also captures dependencies referent to line 3 of Algorithm 1. Correctness proofs of Algorithms 1 and 2 appear in the Appendix.

As mentioned before, the atomic commit mechanism we assumed can block processes in case of failure, forcing them to wait for a message from a process that has crashed. A blocked process is unblocked when the crashed server upon which it depends recovers and starts the global recovery procedure explained in the next section. During the recovery phase, all running transactions (including those that caused a server to block) are aborted and global state consistency is ensured by the rollback-recovery mechanism. When execution resumes, no server is blocked any more. A blocked client has to wait for a recovery notification

---

<sup>3</sup>For code simplicity, let us assume a single client does not execute two transactions concurrently.

to unblock and check with the database servers whether some transaction was lost. Unblocked clients may also start some recovery procedure after receiving such a notification if they rely on something outside the database to ensure transaction durability.

### 4.3 Rollback-recovery

Once we have managed to perform dependency tracking efficiently during the execution, we can make use of one of the numerous existent approaches to orchestrate rollback-recovery in the message-passing model [17, 30, 32]. We illustrate the idea by extending the algorithm presented in [30], adapted to our execution model. The system runs as a sequence of incarnations, started after recovery from some failure. Each server keeps track of the current incarnation. In order to start a new one, an agreement among servers must be reached to determine the recovery line used for the federation restart. Therefore, processes exchange messages containing information about their last stable database state. When all information is received by a server  $S_i$ , it computes its local state that takes part in the recovery line based on Theorem 2 and rolls back to it by erasing inconsistent log entries. Due to the possibility of failures, information about the current incarnation and the last recovery line used for recovery must be kept in the stable storage of each server. A detailed description of this algorithm is presented in the Appendix.

### 4.4 Algorithm analysis

Algorithm 2 incurs no extra cost during transaction execution with respect to the number of messages and communication steps. The algorithm just piggybacks a vector timestamp in messages related to the transaction commit and updates local variables according to the timestamps received. Actually, the overhead is even lower than it would be if we had applied transitive dependency tracking for message-passing distributed systems, since the latter would piggyback a vector-based timestamp in every message exchanged [9].<sup>4</sup> Our approach ensures the minimum possible “window of vulnerability” for transactions, since it depends only on the time each server takes to physically write on stable storage the transaction’s log entry. Every server does that at its own pace without synchronizing with the others; as soon as all of them complete their writes the transaction is durable.

---

<sup>4</sup>Besides, as stated in the introduction, this latter approach would also capture inexistent dependencies.

It is possible to come up with alternative solutions to the problem of ensuring consistency in a federation of main-memory databases under deferred disk writes. For instance, non-blocking synchronous checkpointing approaches for the message-passing model, like [7] and [19] can be adapted to the transactional model considering database-state dependencies in the way we have defined. These algorithms, however, incur  $O(n^2)$  control messages during disk-write synchronization and may force the propagation of timestamps in the application messages to overcome the absence of FIFO communication channels [9] or two disk writes per synchronization to record the fact that the current instance has finished and new ones are allowed [19]. Although some difficulties can be avoided by stronger system assumptions as in [28], the problem of increasing the window of vulnerability and making it as large as the one of the slowest server for all servers will always be present in synchronous algorithms.

Table 1 summarizes the comparison between the approaches we have mentioned. We aggregate synchronous checkpointing protocols (e.g., [7] and [19]) since they present a similar behavior with respect to the variables analyzed in the table. Moreover, “MySQL Cluster” refers to the synchronous approach adopted in [28]. We represent the disk latency (i.e., the time it takes for a disk write request to be completed) of server  $S_i$  by  $dlat(S_i)$ ; and use  $MAX$  to refer to  $\max(\{dlat(S_i) \mid S_i \in \mathcal{S}\})$ . The network latency, used to quantify a communication step, is represented by  $\delta$ . Besides requiring FIFO channels, synchronous checkpointing protocols include the clients in their synchronization, since they are involved in the creation of database-state dependencies. MySQL Cluster assumes partially synchronous channels (i.e., with bounded message delivery) and have clients coordinate the task in order to simplify the algorithm. Differently, our approach makes no assumptions about communication channels and only propagates timestamps on some of the messages already exchanged by the system. As a result, our algorithm incur the minimum possible window of vulnerability in a single server and no extra message exchanges. As the role of the client in participating of synchronous approaches is not very clear, possibly forcing more messages to be exchanged, for such approaches we only show the lower bound on extra messages required for servers’ synchronization.

Algorithm	Communication channels	Client synchronization	$S_i$ 's window of vulnerability	Extra messages per execution
Sync. Checkpointing	FIFO	Clients participate	$MAX + 2\delta$	$\Omega(n^2)$
MySQL Cluster	Partially Sync.	Clients coordinate	$MAX + 3\delta$	$\Omega(n)$
Our approach	Any	None	$dlat(S_i)$	0

Table 1: Comparison of the different approaches

## 5 Improving replication performance

This section presents a practical scenario where relaxing durability inside the database engine with deferred disk writes increases performance significantly without violating any end user's transaction properties. We consider database replication based on group communication primitives [1, 3, 22, 23]. Transactions are processed locally at one database server and, during commit, update transactions are forwarded for certification to all servers using a total order broadcast primitive [15]. The order properties of this primitive allow the servers to run a deterministic certification test, resulting in the same outcome (commit/abort) in all database replicas [23]. However, in the crash-recovery model used for database replication, atomic broadcast should be able to deal with extra consistency issues. First, the recovery of a server raises the issue of keeping its local state consistent with what it did before crashing with respect to the total order broadcast algorithm, which boils down to writing state information on stable storage (e.g., in a disk log). Second, the possibility of crashing after delivering a message but before the application had time to use it in a way that the action would be remembered after recovery forces the group communication middleware to be able to provide the application with a whole suffix of the delivery history, during recovery, including the messages forgotten by the application or delivered by the others while the server was down [29]. In a modular implementation this can be solved by providing the application with the complete delivery history and letting it choose the transactions it does not know about after recovery [26]. An efficient implementation, though, would allow the application to access directly the logs written by the group communication middleware.

Since the total order broadcast properties ensure that all messages (that is, transactions in this context) must be delivered by the server application and the delivery order has to be the same in all servers, regardless of local or external failures, durability inside the database engine running in each replicated server can be relaxed. Note that the end user of the replicated database will see durable transactions, but this property is now granted by the total order broadcast primitive. A replicated database based on traditional disk resident database system (DRDBs) would not gain much in relaxing durability since database objects themselves are stored on disk. On the contrary, a replication scheme based on MMDBs that do not relax durability would at least double the latency of update transactions (which is already much bigger than the latency of read-only transactions). In this case, durability would be ensured "twice": first by the group communication, and then by the MMDBs.

MMDBs provide excellent performance for the price of storage limitations, since all data must fit in the

available main memory of a single server. This drawback can be overcome by partitioning data amongst a number of MMDB servers arranged in a database federation. A high-performance highly-available MMDB could be built using the replication scheme we described in this section and taking advantage of deferred disk writes inside local database engines. In this case, durability has to be relaxed at the federation level and some mechanism must guarantee that no other transaction property is violated. This is exactly what our algorithms ensure.

## 6 Related work

Although MMDBs do not represent a new concept in database design, only recently they have been applied to more general scenarios. Specifically, to our knowledge, the only work that makes use of MMDBs in a cluster of servers is [28] (derived from [27]), where performance and availability are enhanced by replicating and fragmenting the database among the database servers in the system. To ensure good performance for update transactions as well, the approach makes use of deferred disk writes, even for transactions that access multiple servers. In this case, consistency is ensured by synchronizing the servers' disk writes. Some drawbacks of synchronized approaches are avoided by assuming a stronger model. However, the algorithm still suffers from enlarging the transactions' window of vulnerability, as explained in the previous section.

Rollback-recovery has been extensively studied in the message-passing model and different approaches have been given to select the states to which the distributed application can be rolled-back (consistent states) and to orchestrate the recovery task [2, 7, 9, 17, 19, 30, 32]. Nevertheless, very few of these works have been exploited in different environments. In this context, the work in [4] presents a framework to analyze consistency in different shared-memory and message-passing systems. In [5], their results are extended to the transactional model, motivated by the problem of building a consistent snapshot of a centralized database without stopping the execution of transactions. Actually, the problem of building a consistent database snapshot has triggered a lot of research on the analysis of database-state dependencies [5, 10, 11, 24, 25, 31]. Some of the ideas presented in these works, specially in [5] and [11], resemble our transaction and state dependencies definitions. However, none of them present a practical characterization of database-state dependencies (e.g., Theorem 3). Our approach differs from these works by (a) assuming a distributed scenario where synchronization between different processes must be minimized, and (b) aiming at applying rollback-recovery techniques to bring the application back to a consistent state in case of failure.

## 7 Concluding remarks

In this paper we tackled the problem of deferred disk writes in federations of main-memory database systems. Our approach was motivated by previous research on rollback-recovery for message-passing distributed systems. We described how database-state dependencies are created in the transactional model and how they can be tracked efficiently during execution. Based on that, rollback-recovery mechanisms can be easily adapted to transactions. A possible extension to our algorithms is to use direct instead of transitive dependency tracking [9, 30], as this can possibly lead to smaller timestamps if transactions do not tend to access many servers. Moreover, our algorithms borrow from optimistic message logging. It is also possible to exploit other rollback-recovery techniques, like causal message logging and quasi-synchronous checkpointing, and compare their performance and advantages under different transaction scenarios. Research domains that may take advantage of this theory include optimistic concurrency control mechanisms and management of nested transactions. Investigating such issues is the subject of future work.

## References

- [1] D. Agrawal, G. Alonso, A. E. Abbadi, and I. Stanoi. Exploiting atomic broadcast in replicated databases. In *Proceedings of EuroPar (EuroPar'97)*, Passau (Germany), Sept. 1997.
- [2] L. Alvisi and K. Marzullo. Message Logging: Pessimistic, Optimistic, Causal and Optimal. *IEEE Trans. on Software Engineering*, 24(2):149–159, Feb. 1998.
- [3] Y. Amir and C. Tutu. From total order to database replication. In *International Conference on Distributed Computing Systems (ICDCS)*, July 2002.
- [4] R. Baldoni, J.-M. Helari, and M. Raynal. Consistent records in asynchronous computations. *Acta Informatica*, 35(6):441–455, June 1998.
- [5] R. Baldoni, F. Quaglia, and M. Raynal. Consistent checkpointing for transaction systems. *The Computer Journal*, 44(2):92–100, 2001.
- [6] P. Bernstein, V. Hadzilacos, and N. Goodman. *Concurrency Control and Recovery in Databases Systems*. Addison-Wesley, 1987.
- [7] M. Chandy and L. Lamport. Distributed Snapshots: Determining Global States of Distributed Systems. *ACM Trans. on Computer Systems*, 3(1):63–75, Feb. 1985.
- [8] D. DeWitt, R. H. Katz, F. Olken, L. D. Shapiro, M. Stonebraker, and D. A. Wood. Implementation techniques for main memory database systems. In *SIGMOD'84, Proceedings of Annual Meeting, Boston, Massachusetts, June 18-21*, pages 1–8. ACM Press, 1984.
- [9] E. N. Elnozahy, L. Alvisi, Y. M. Wang, and D. B. Johnson. A Survey of Rollback-Recovery Protocols in Message-Passing Systems. *ACM Computing Surveys*, 34(3):375–408, Sept. 2002.
- [10] M. J. Fischer, N. D. Griffeth, and N. A. Lynch. Global states of a distributed system. *IEEE Trans. on Software Engineering*, SE-8(3):198–202, 1982.
- [11] I. C. Garcia and L. E. Buzato. Asynchronous construction of consistent global snapshots in the object and action model. In *Proc. of the 4th IEEE Int. Conference on Configurable Distributed Systems*, 1998.

- [12] H. Garcia-Molina and K. Salem. Main memory database systems: An overview. *IEEE Transactions on Knowledge and Data Engineering*, 4(6):509–516, Dec. 1992.
- [13] J. Gray. The revolution in database architecture. Technical Report MSR-TR-2004-31, Microsoft Research, 2004.
- [14] J. N. Gray and A. Reuter. *Transaction Processing: Concepts and Techniques*. Morgan Kaufmann, 1993.
- [15] V. Hadzilacos and S. Toueg. Fault-tolerant broadcasts and related problems. In *Distributed Systems*, chapter 5. Addison-Wesley, 2nd edition, 1993.
- [16] M. Ji. Affinity-based management of main memory database clusters. *ACM Trans. on Internet Technology (TOIT)*, 2(4):307–339, 2002.
- [17] D. B. Johnson and W. Zwaenepoel. Recovery in distributed systems using optimistic message logging and checkpointing. *Journal of Algorithms*, 11(3):462–491, 1990.
- [18] K. Knizhnik. Fastdb: Main-memory relational database management system. <http://www.garret.ru/knizhnik/fastdb.html>.
- [19] R. Koo and S. Toueg. Checkpointing and rollback-recovery for distributed systems. *IEEE Trans. on Software Engineering*, 13:23–31, Jan. 1987.
- [20] L. Lamport. Time, clocks, and the ordering of events in a distributed system. *Commun. ACM*, 21(7):558–565, July 1978.
- [21] D. Morse. In-memory database web server. *Dedicated Systems Magazine*, 4:12–14, 2000.
- [22] M. P. no Martínez, R. Jiménez-Peris, B. Kemme, and G. Alonso. Scalable replication in database clusters. In *Proceedings of the 14th International Symposium on Distributed Computing (DISC'00, formerly WDAG)*, Oct. 2000.
- [23] F. Pedone, R. Guerraoui, and A. Schiper. The database state machine approach. *Journal of Distributed and Parallel Databases and Technology*, 14(1):71–98, 2003.
- [24] S. Pilarski and T. Kameda. Checkpointing for distributed databases: Starting from the basics. *IEEE Trans. on Parallel and Distributed Systems*, 3(5):602–610, 1992.
- [25] C. Pu. On-the-fly, incremental, consistent reading of entire databases. *Algorithmica*, 1(3):271–287, 1986.
- [26] L. Rodrigues and M. Raynal. Atomic broadcast in asynchronous crash-recovery distributed systems and its use in quorum-based replication. *IEEE Transactions on Knowledge and Data Engineering*, 15(5):1206–1217, 2003.
- [27] M. Ronström. The NDB cluster – A parallel data server for telecommunications applications. Ericsson Review no. 4, 1997.
- [28] M. Ronström and L. Thalmann. Mysql cluster architecture overview. MySQL Technical White Paper, 2004.
- [29] J. Saltzer, D. Reed, and D. Clarck. End-to-end arguments in system design. *ACM Trans. on Computer Systems*, 2(4):277–288, Nov. 1984.
- [30] A. P. Sistla and J. L. Welch. Efficient distributed recovery using message logging. In *Proceedings of the 8th ACM Symposium on the Principles of Distributed Computing*, pages 233–238, 1989.
- [31] S. H. Son and A. K. Agrawala. Distributed checkpointing for globally consistent states of databases. *IEEE Trans. on Software Engineering*, 15(19):1157–1166, 1989.
- [32] R. Strom and S. Yemini. Optimistic Recovery in Distributed Systems. *ACM Trans. on Computing Systems*, 3(3):204–226, Aug. 1985.

## Appendix

### A Theorem proofs

**Theorem 1** A global database state  $G = \{\sigma_1^{\alpha_1}, \dots, \sigma_n^{\alpha_n}\}$  is consistent iff  $\forall \sigma_i^{\alpha_i}, \sigma_j^{\alpha_j} \in G : \sigma_i^{\alpha_i} \not\rightarrow \sigma_j^{\alpha_j}$ .

**Proof sketch - Sufficiency**( $\Leftarrow$ ): If  $\forall \sigma_i^{\alpha_i}, \sigma_j^{\alpha_j} : \sigma_i^{\alpha_i} \not\rightarrow \sigma_j^{\alpha_j}$ , then we can build the ordered set of transactions  $T$  of Definition 3, whose serial execution leads to the corresponding global state. Let  $T_{bas}$  be the set  $\{t \mid \sigma_i^{\alpha_i-1} \in W(t)\}$ , that is, the set of transactions that create the individual states in our global state. It is guaranteed that no transaction in  $T_{bas}$  creates a database state in the future of  $G$ , that is, a database state whose index is bigger than its correspondent in  $G$ . Otherwise, by Definition 2, there would be two dependent states in  $G$ .  $T$  is composed of  $T_{bas}$  and all transactions that precede a transaction in  $T_{bas}$ , using any topological order given by the transaction dependency relation.

**Necessity**( $\Rightarrow$ ): Let us assume that  $\exists \sigma_i^{\alpha_i}, \sigma_j^{\alpha_j} \mid \sigma_i^{\alpha_i} \rightarrow \sigma_j^{\alpha_j}$  and it is possible to build the ordered set of transactions  $T$  of Definition 3 whose serial execution generates  $G$ . By Definition 2, there must be transactions  $t$  and  $t'$  such that  $\sigma_i^{\alpha_i} \in W(t)$ ,  $\sigma_j^{\alpha_j-1} \in W(t')$ , and  $t = t' \vee t \rightarrow t'$ . Since  $t'$  creates state  $\sigma_j^{\alpha_j}$ ,  $t'$  must appear in  $T$ . However, by Definition 3, if  $t' \in T$ , then  $t \in T$ . Since  $t$  creates  $\sigma_i^{\alpha_i+1}$ ,  $G$  will not contain  $\sigma_i^{\alpha_i}$ , which is a contradiction.  $\square$

**Theorem 2** The recovery line  $R$  for a given history is determined by

$$R = \bigcup_{i=1}^n \{\sigma_i^k \mid k \text{ is the maximum value such that } \forall 1 \leq j \leq n : \sigma_j^{last} \not\rightarrow \sigma_i^k\}$$

**Proof sketch:** We must show that  $R$  is a consistent global state and minimizes the number of database states rolled back. First consider, by contradiction, that  $R$  is not consistent and, thus, there are two database states  $\sigma_a^\alpha, \sigma_b^\beta \in R$  such that  $\sigma_a^\alpha \rightarrow \sigma_b^\beta$ . By Definition 2, there must be two transactions  $t$  and  $t'$  such that  $\sigma_a^\alpha \in W(t)$ ,  $\sigma_b^{\beta-1} \in W(t')$ , and  $t = t' \vee t \rightarrow t'$ . By the definition of  $R$  there must be a state  $\sigma_c^{last}$  such that  $\sigma_c^{last} \rightarrow \sigma_a^{\alpha+1}$  and, by Definition 2, a transaction  $t''$  such that  $\sigma_c^{last} \in W(t'')$  and  $t'' = t \vee t'' \rightarrow t$ . By transitivity, either  $t'' = t'$  or  $t' \rightarrow t''$  and  $\sigma_c^{last} \rightarrow \sigma_b^\beta$  by Definition 2, which contradicts the definition of  $R$ .

According to our definition,  $R$  clearly minimizes the number of database states rolled back since a state  $\sigma_c^\gamma$  is rolled back only if it is preceded by the last stable state of a database server  $S_j$ . In such case, by the

Consistent Persistence property, all stable states of  $S_f$  precede  $\sigma_c^\gamma$  and no recoverable consistent global state could be formed with it. As the necessary states are rolled back, minimization is ensured.  $\square$

**Theorem 3** *Server state  $\sigma_a^\alpha$  precedes  $\sigma_b^\beta$  ( $\sigma_a^\alpha \rightarrow \sigma_b^\beta$ ) iff*

- (a)  $\exists t \mid \sigma_a^\alpha \in W(t) \wedge \sigma_b^{\beta-1} \in RW(t)$ ; or
- (b)  $\exists t, \sigma_c^\gamma \mid \sigma_a^\alpha \rightarrow \sigma_c^\gamma \wedge \sigma_b^{\beta-1}, \sigma_c^\gamma \in RW(t)$ ; or
- (c)  $\exists t, \sigma_c^\gamma \mid \sigma_a^\alpha \rightarrow \sigma_c^\gamma \wedge \sigma_b^{\beta-1} \in RW(t) \wedge \sigma_c^{\gamma-1} \in W(t)$ .

**Proof sketch:** Let  $t_A$  be the transaction that overwrites state  $\sigma_a^\alpha$  ( $\sigma_a^\alpha \in W(t_A)$ ), and  $t_B$  the transaction that creates state  $\sigma_b^\beta$  ( $\sigma_b^{\beta-1} \in W(t_B)$ ).

*Sufficiency*( $\Leftarrow$ ): Let us prove the sufficiency of each of the three conditions separately. For every condition, we will show that either  $t_A = t_B$  or  $t_A \rightarrow t_B$ . In both cases, the condition of Definition 2 holds and  $\sigma_a^\alpha \rightarrow \sigma_b^\beta$ .

When condition (a) of Theorem 3 is satisfied, if  $\sigma_b^{\beta-1} \in W(t)$  then  $t_A = t_B$ , and if  $\sigma_b^{\beta-1} \in R(t)$  then the condition (b) of Definition 1 is satisfied and  $t_A \rightarrow t_B$ .

Now let us assume that condition (b) of Theorem 3 is fulfilled and let us call  $t_C$  the transaction that creates  $\sigma_c^\gamma$ . By Definition 2 we have that  $t_A = t_C \vee t_A \rightarrow t_C$ . As  $\sigma_c^\gamma \in RW(t)$ , condition (a) of Definition 1 is satisfied and  $t_C \rightarrow t$ . Therefore, by transitivity,  $t_A \rightarrow t$ . Using an argument similar to the one we used to prove the previous condition (analyzing whether  $\sigma_b^{\beta-1} \in W(t)$  or  $\sigma_b^{\beta-1} \in R(t)$ ) it is straightforward to show that  $t_A \rightarrow t_B$ .

Lastly, if condition (c) holds,  $t_A = t \vee t_A \rightarrow t$  since  $\sigma_a^\alpha \rightarrow \sigma_c^\gamma$  and  $\sigma_c^{\gamma-1} \in W(t)$ . If  $\sigma_b^{\beta-1} \in W(t)$ , then  $t = t_B$ . Otherwise, if  $\sigma_b^{\beta-1} \in R(t)$ , then  $t \rightarrow t_B$  as in the analysis of condition (a). By transitivity we conclude that  $t_A = t_B \vee t_A \rightarrow t_B$ .

*Necessity*( $\Rightarrow$ ): Here we have to prove that when  $t_A = t_B \vee t_A \rightarrow t_B$  one of the three conditions of the theorem is satisfied. If  $t_A = t_B$ , then the first condition is trivially fulfilled. Otherwise, there must be a chain  $t_A = t_1 \rightarrow t_2 \rightarrow \dots \rightarrow t_k \rightarrow t_B$ . We base our proof in Claim 1, presented next. Let us analyze the dependency relation between  $t_k$  and  $t_B$ . If it is given by condition (a) of Definition 1, then condition (b) of our theorem holds by Definition 1 and Claim 1. Otherwise, if it is given by condition (b) of Definition 1, then condition (c) of our theorem holds.  $\square$

**Claim 1** *If  $t_A = t_k \vee t_A \rightarrow t_k$ , then  $\forall \sigma_c^\gamma \in RW(t_k) : \sigma_a^\alpha \rightarrow \sigma_c^{\gamma+1}$ .*

**Proof sketch:** If  $\sigma_c^\gamma \in W(t_k)$ , the fact that  $\sigma_a^\alpha \rightarrow \sigma_c^{\gamma+1}$  follows directly from Definition 2. If  $\sigma_c^\gamma \in R(t_k)$ , then let  $t'$  be the transaction that creates  $\sigma_c^{\gamma+1}$  ( $\sigma_c^\gamma \in W(t')$ ). In this case, condition (b) of Definition 1 is satisfied and  $t_k \rightarrow t'$ . This implies, by Definition 2, that  $\sigma_a^\alpha \rightarrow \sigma_c^{\gamma+1}$ .  $\square$

## B Correctness proofs of dependency tracking algorithms

**Theorem 4** *If transactions execute Algorithm 1 during commit, then  $\sigma_a^\alpha \in D(\sigma_b^\beta) \iff \sigma_a^\alpha \rightarrow \sigma_b^\beta$*

**Proof sketch - Sufficiency( $\Leftarrow$ ):** We prove that  $\sigma_a^\alpha \rightarrow \sigma_b^\beta \Rightarrow \sigma_a^\alpha \in D(\sigma_b^\beta)$  by structural induction, considering it as our hypothesis and analyzing the three conditions of Theorem 3 by which one database state can precede another one, to verify if the implication follows in all of them.

If  $\sigma_a^\alpha \rightarrow \sigma_b^\beta$  by condition (a) of Theorem 3, when  $t$  commits, line 1 of Algorithm 1 makes sure that  $D(\sigma_b^\beta)$  will contain  $\sigma_a^\alpha$ , since  $D(S_b)$  will be assigned to  $D(\sigma_b^\beta)$  when the local database performs the next state transition.

If  $\sigma_a^\alpha \rightarrow \sigma_b^\beta$  by condition (b) of Theorem 3, our hypothesis says that  $\sigma_a^\alpha \in D(\sigma_c^\gamma)$ . In this case, when  $t$  commits, line 2 of Algorithm 1 adds  $\sigma_a^\alpha$  to  $D(S_b)$  for it to be assigned to  $D(\sigma_b^\beta)$  when the state is created.

If  $\sigma_a^\alpha \rightarrow \sigma_b^\beta$  derives from condition (c) of Theorem 3, our hypothesis states that  $\sigma_a^\alpha \in D(\sigma_c^\gamma)$ . As  $\sigma_c^\gamma$  is created by  $t$ , this means that  $\sigma_a^\alpha \in D(S_c)$  after the previous conditions have been considered. Therefore, line 3 of Algorithm 1 makes sure that this dependency will be inserted in  $D(\sigma_b^\beta)$  when the state is created.

**Necessity( $\Rightarrow$ ):** We prove that  $\sigma_a^\alpha \in D(\sigma_b^\beta) \Rightarrow \sigma_a^\alpha \rightarrow \sigma_b^\beta$  by structural induction based on the steps performed by Algorithm 1 to insert an element  $\sigma_a^\alpha$  on  $D(S_b)$  for the first time when  $\sigma_b^\beta$  is the next state to be created, since this means that such element will belong to  $D(\sigma_b^\beta)$ . If  $\sigma_a^\alpha$  is inserted in  $D(S_b)$  because of line 1 of Algorithm 1, then  $t$  satisfies condition (a) of Theorem 3 and, therefore,  $\sigma_a^\alpha \rightarrow \sigma_b^\beta$ .

Now suppose that  $\sigma_a^\alpha$  is inserted in  $D(S_b)$  because of line 2 of Algorithm 1, which means that  $\sigma_a^\alpha \in D(\sigma_c^\gamma)$ . By the induction hypothesis,  $\sigma_a^\alpha \rightarrow \sigma_c^\gamma$  and condition (b) of Theorem 3 states that indeed  $\sigma_a^\alpha \rightarrow \sigma_b^\beta$ .

Finally, consider the case where  $\sigma_a^\alpha$  is inserted in  $D(S_b)$  because of line 3 of Algorithm 1. This means that  $\sigma_a^\alpha$  will be assigned to  $D(\sigma_c^\gamma)$ , for some state  $\sigma_c^\gamma$  created by  $t$ . By our hypothesis,  $\sigma_a^\alpha \rightarrow \sigma_c^\gamma$  and condition (c) of Theorem 3 follows.  $\square$

**Theorem 5** *Algorithm 2 correctly implements Algorithm 1.*

**Proof sketch:** Correctness of Algorithm 2 comes from the propagation of the correct data structures from the servers to the client in response to a PREPARE request. Line 1 of Algorithm 1 is implemented by line 38 of Algorithm 2.  $DV_{last}[i]$  gives the index of the local state previous to the last one (the last local state on which the current state depends). Therefore, if we add 1 to such value we get the index of the current database state, which is being overwritten by transaction  $tid$ . The concurrency control mechanism ensures that no other state will be created before the COMMIT request is received.

Line 2 of Algorithm 1 is given by line 39 of Algorithm 2 if the server is read by  $tid$  and by line 37 if the server is written by  $tid$ . Line 39 propagates precisely the dependencies of the current state accessed by the transaction. Line 37 gives the dependencies of the local state accessed by  $tid$  together with the dependencies relative to line 3 of Algorithm 1 since the server was written by  $tid$ . In both cases, the concurrency control mechanism ensures that these data structures correspond to the dependencies of the states actually accessed by the transaction and that they do not change until the transaction is committed.

As no other dependencies are propagated by the servers to the client at commit time, strictness of implementation is ensured. □

## C Complete rollback-recovery algorithm

Algorithm 3 presents our complete algorithm for rollback-recovery in a single server. When a server recovers from a failure, it reads the record with the previous incarnation number and the index of its last stable state before such incarnation started (line 2). After that, the current incarnation number is calculated, the recovery flag is set, the dependency vector of the last entry in the log is recovered and the index of the most recent stable state is calculated (lines 3-6). At this point, the server must receive the index of the last stable state of each server, stored in vector  $LS$ . To trigger the emission of such information, the server keeps sending a recovery message to all the other servers, suggesting the creation of a new incarnation. A separate thread receives their answers and updates the recovery flag when  $LS$  is complete (lines 16-19). Then, according to our approach for recovery line determination, all entries in the database log that depend on some state  $\sigma_j^{last}$  are eliminated and the server is restarted to continue its normal execution (lines 10-14). It is possible that a

---

**Algorithm 3** Rollback-recovery

---

<pre>1: <u>Database restart:</u> 2:  recover <math>\langle previnc, prevstate \rangle</math> record from disk 3:  <math>incarnation \leftarrow previnc + 1</math> 4:  <math>recovering \leftarrow True</math> 5:  recover <math>DV</math> from last entry in the log 6:  <math>state \leftarrow DV[pid] + 1</math> 7:  <math>\forall 1 \leq j \leq n : LS[j] \leftarrow -1</math> 8:  <b>while</b> <math>recovering</math> <b>do periodically</b> 9:    send <math>\langle RECOVERY, incarnation, state \rangle</math> to all servers 10: delete all entries in the log where <math>\exists j \mid LS[j] \leq DV[j]</math> 11: recover <math>DV</math> from last entry in the log 12: <math>DV_{last} \leftarrow DV</math> 13: write <math>\langle incarnation, state \rangle</math> record on disk 14: load the database into memory and apply the log</pre>	<pre>15: <u>Concurrent tasks during recovery:</u> 16: <b>when</b> receive <math>\langle RECOVERY, incarnation, last_i \rangle</math> from <math>S_i</math> 17:   <math>LS[i] \leftarrow last_i</math> 18:   <b>if</b> <math>\forall 1 \leq j \leq n : LS[j] \neq -1</math> <b>then</b> 19:     <math>recovering \leftarrow False</math> 20: <b>when</b> receive <math>\langle RECOVERY, previnc, \_ \rangle</math> from <math>S_i</math> 21:   send <math>\langle RECOVERY, previnc, prevstate \rangle</math> to <math>S_i</math> 22: <u>Additional tasks for normal execution:</u> 23: <b>when</b> receive <math>\langle RECOVERY, incarnation, \_ \rangle</math> from <math>S_i</math> 24:   send <math>\langle RECOVERY, incarnation, state \rangle</math> to <math>S_i</math> 25: <b>before</b> receiving <math>\langle RECOVERY, incarnation + 1, \_ \rangle</math> 26:   abort running transactions 27:   restart database</pre>
---	--

---

server does not answer the current incarnation change request because it is still recovering from the previous incarnation (for simplicity, a server cannot jump an incarnation number). This problem is solved by keeping a separate thread during recovery (lines 20-21) whose function is simply to answer recovery requests for the previous incarnation with the information retrieved from stable storage in line 1.

Two additional threads are necessary during normal execution to complete our rollback-recovery algorithm. When a running server receives a recovery request from a different server related to the current incarnation, it responds with the information it calculated during the database restart (lines 23-24). If the server receives a recovery request related to the next incarnation, it simulates a failure and restarts (lines 25-27). Notice that in this case running transactions are aborted and communication channels are closed to avoid the interference from messages sent in past incarnations.

Algorithm 3 is also efficient. Due to the distributed calculation of the recovery line, the application is ready to resume its execution only two communication steps after the faulty servers recover. If no failure happens during recovery, only  $O(n^2)$  are exchanged and no process rolls back more than once (it does not suffer from exponential rollbacks [9]). All this features are inherited from the algorithm presented in [30], on which ours is based. Actually, this algorithm simply shows how rollback-recovery mechanisms can be applied in our model.