Consistent Main-Memory Database Federations under Deferred Disk Writes*

Rodrigo Schmidt*,†   Fernando Pedone†

*École Polytechnique Fédérale de Lausanne (EPFL), CH-1015 Lausanne, Switzerland
†Università della Svizzera Italiana (USI), CH-6904 Lugano, Switzerland

E-mails: `rodrigo.schmidt@epfl.ch`, `fernando.pedone@unisi.ch`

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, significantly improving 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.

***Keywords :** dependency tracking, consistency, rollback-recovery, distributed transactions, MMDBs.*

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 [11]). Although originally designed for specific classes of applications (e.g., telecommunication) running in single servers, recent work has suggested that MMDBs can be also used in broader contexts (e.g., web servers [18]) and environments (e.g., clustered architectures [24]). Shortly, MMDBs overcome the latency limitations of traditional disk-based databases by storing the data items in the main memory of the servers [12]. 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 [11, 15]).

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 some applications may prefer to ensure durability outside the database for performance reasons. As an example of such applications, database replication schemes based on atomic broadcast primitives (e.g., the database state machine approach [19]) in the crash-recovery model will have durability ensured by the group communication primitive (see the work in [22]) and, therefore, it is redundant to also have it in each database replica.

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 subtransactions. 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 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* [1]; in the former, dependencies are created by *read* and *write* operations on database objects during the execution of transactions.

Consider, for example, 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; α, β, and γ 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 β would precede γ since there is a causal path between the two states (depicted in bold in Figure 1). However, since t_1 only reads β, it turns out that β and γ 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.

We revisit the original dependency definitions, developed for message-passing systems, and propose a new one based on database states, minimal for distributed
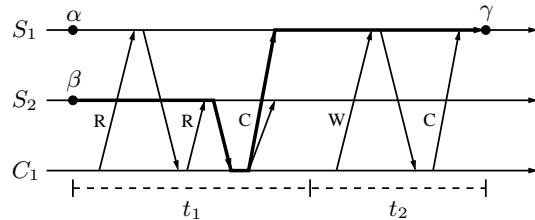

Figure 1. False (causal) dependency

transaction environments and allowing efficient tracking implementation. Moreover, this paper illustrates the applicability of our approach in the context of an FMMDB 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 introduces our computational and execution models. Section 3 explores consistency and dependencies in a transactional system. Section 4 presents our algorithms to ensure correctness of execution in a federation of main-memory databases with deferred disk writes. We compare our approach with existent works in the field in Section 5 and conclude the paper in Section 6.

Due to space limitations, theorems and correctness proofs are presented in the full paper [25].

2. System model

We assume a system composed of two disjoint sets of processes: the set of servers $\mathcal{S} = \{S_1, S_2, \ldots, S_n\}$ and the set of clients $\mathcal{C} = \{C_1, C_2, \ldots, C_m\}$. 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.[2] 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, but

---

[1]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 [17].

[2]The implementation of a distributed transactional environment may require stronger assumptions (e.g., failure suspicion). The ideas described in this paper, however, are oblivious to such assumptions.

it eventually recovers. 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 ensured to be in a globally-consistent state.

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 [13]:

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 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 [4]. 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. Having clarified our definitions, we would like to reinforce that 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.

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., [15]). This allows us to abstract client operations as Reads and Writes performed over an entire database state. We show how our approach can be extended to more complicated concurrency control mechanisms such as two-phase-locking in Section 4.5.

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 $\sigma_i^0, \sigma_i^1, \ldots$, where σ_i^j represents S_i's state after committing the j-th local update transaction.

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 generates a new unique transaction identification number (tid), to be used in all servers. When 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 client 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 [4, 13]) 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. 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. 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 at server S_i. 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 [4, 13]. 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.

Definition 1 *Transaction t precedes t' ($t \to t'$) 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) $\exists t'' \mid t \to t'' \ \wedge \ t'' \to t'$.

Definition 2 *State σ_a^α precedes σ_b^β ($\sigma_a^\alpha \to \sigma_b^\beta$) 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) $t = t' \ \vee \ t \to t'$.

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 *serializabilty* [4] and formalize it in Definition 3.

Definition 3 *A global database state $\{\sigma_1^{\alpha_1}, \ldots, \sigma_n^{\alpha_n}\}$ in a given history H is consistent iff it represents the database state after the serial execution of an ordered set of transactions $T = (t_1, t_2, .., t_l)$ such that:*
(a) all transactions in T are non-aborted persistent transactions in H;
(b) $\forall t \in T : t' \to t \ in \ H \Rightarrow t' \in T$; and
(c) $\forall t_a, t_b \in T : t_a \to t_b \ in \ H \Rightarrow a < b$.

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 state $G = \{\sigma_1^{\alpha_1}, \ldots, \sigma_n^{\alpha_n}\}$ is consistent iff $\forall \sigma_i^{\alpha_i}, \sigma_j^{\alpha_j} \in G : \sigma_i^{\alpha_i} \not\to \sigma_j^{\alpha_j}$.*

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 $T = (t_1, t_2, t_3, t_4)$.

By 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 execution may be lost after a failure. 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.
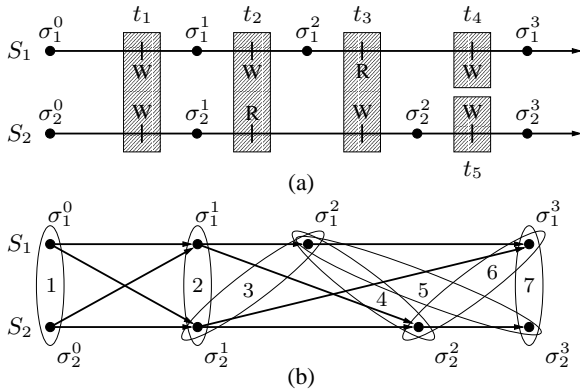

(a)

(b)

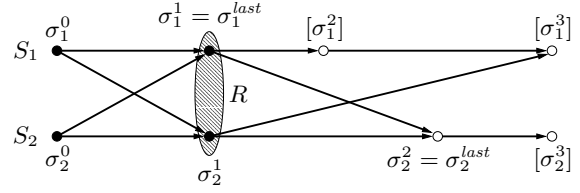**Figure 2. Consistent global states**

Figure 3. Recovery-line determination

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 σ_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 σ_i^{last}.

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

$$R = \bigcup_{i=1}^{n} \{\sigma_i^k \mid k = max(\gamma \mid \forall S_j : \sigma_j^{last} \not\rightarrow \sigma_i^{\gamma})\}$$

Figure 3 depicts an example of recovery line determination based on the scenario presented in Figure 2 (volatile states are depicted between square brackets, e.g., $[\sigma_i^j]$). The figure shows a dependency graph with all the states dependent on some state σ_i^{last} as empty circles. Therefore, the recovery line is formed by the state represented by the last filled circle in each database server.

4. Database-oriented rollback-recovery

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 σ_a^α precedes σ_b^β ($\sigma_a^\alpha \rightarrow \sigma_b^\beta$) 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.

5

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_1) and (a_2) correspond to condition (a) of Theorem 3, and scenarios (b) and (c) correspond to conditions (b) and (c), respectively. Figure 4(a_1) 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 σ_a^α necessarily precedes this state and σ_b^β succeeds it, it is clear that $\sigma_a^\alpha \rightarrow \sigma_b^\beta$. Figure 4($a_2$) represents the case where $\sigma_a^\alpha, \sigma_b^{\beta-1} \in W(t)$. As σ_b^β is created by t, it did not exist before t's commit; whilst σ_a^α 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 σ_c^γ belong to the federation state accessed by t. Similarly to the situation depicted in Figure 4(a_1), σ_a^α must precede σ_b^β. 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 σ_b^β and σ_c^γ. Since σ_a^α precedes σ_c^γ, σ_a^α has been already updated before σ_c^γ is created. As σ_c^γ and σ_b^β 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}.

4.2. Dependency tracking algorithm

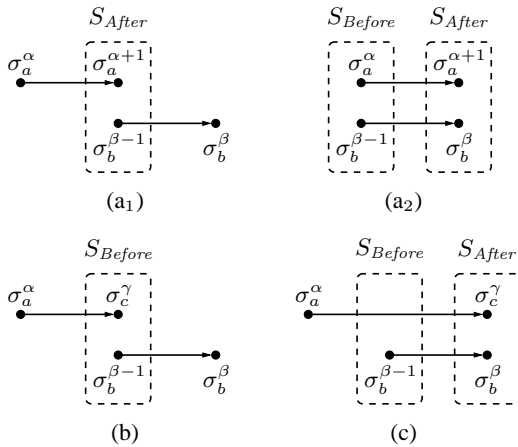Theorem 3 leads to a simple way to gather database-state dependencies on-the-fly during the system's execution. Assume each state σ_i^α has associated with it a data structure $D(\sigma_i^\alpha)$ representing the set of states it depends on (we show later how this structure can be implemented efficiently). 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: $\forall \sigma_j^\iota \in W(t) : D(\sigma_j^{\iota+1}) \leftarrow D(S_j)$

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 σ_c^γ with the γth entry in the log of Server S_c. To keep track of σ_c^γ'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 σ_c^γ, we need to store only the last state of each server on which σ_c^γ depends. If σ_c^γ depends on σ_a^α ($\alpha > 0$), clearly it also depends on $\sigma_a^0, \ldots, \sigma_a^{\alpha-1}$. Therefore, a complete set of state dependencies can 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

Figure 4. Dependencies based on the server states accessed by a transaction

inspired by dependency tracking for rollback-recovery in the message-passing model [28].

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).[3] 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.

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 [25].

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 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 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 [14, 26, 28]. We illustrate the idea by extending the algorithm presented in [26], 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 [25].

4.4. Algorithm analysis

Algorithm 2 incurs no extra cost during transaction execution with respect to the number of messages and

---

[3] For code simplicity, let us assume a single client does not execute two transactions concurrently.

Algorithm 2 Complete algorithm for dependency tracking

<div style="display:flex">

CLIENT STUB

1: Data Structures
2:    Λ : set of servers

3: Begin_Transaction ()
4:    $\Lambda \leftarrow \emptyset$
5:    return *unique tid*

6: Read/Write (tid, S_i, op)
7:    $\Lambda \leftarrow \Lambda \cup \{S_i\}$
8:    send $\langle \text{READ/WRITE}, tid, op \rangle$ to S_i
9:    wait for $\langle result \rangle$ from S_i
10:   return *result*

11: Commit (tid)
12:   send $\langle \text{PREPARE}, tid \rangle$ to all $S_i \in \Lambda$
13:   wait for $\langle \text{VOTE}, tid, v_i, DV_i \rangle$ from all $S_i \in \Lambda$
14:   return ($\forall S_i \in \Lambda : v_i = \text{YES}$)

15: Abort (tid)
16:   send $\langle \text{ABORT}, tid \rangle$ to all $S_i \in \Lambda$

SERVER WRAPPER

17: Data Structures
18:   $opSet_{tid}$: ordered set of operations
19:   DV, DV_{last} : array$[1..n]$ of integer
20:   Λ_{tid} : set of servers

21: Initialization
22:   $\forall\, tid : opSet_{tid} \leftarrow \emptyset,\ \Lambda_{tid} \leftarrow \mathcal{S}$
23:   $\forall\, 1 \leq j \leq n : DV[j] \leftarrow -1$
24:   $DV_{last} \leftarrow DV$

25: The server continuously waits for an event:
26:   **when** receive $\langle \text{READ}, tid, op \rangle$ from C_i
27:     $result \leftarrow \text{submit}(tid, op)$
28:     send $\langle result \rangle$ to C_i

29:   **when** receive $\langle \text{WRITE}, tid, op \rangle$ from C_i
30:     $result \leftarrow \text{submit}(tid, op)$
31:     append op to $opSet_{tid}$
32:     send $\langle result \rangle$ to C_i

33:   **when** receive $\langle \text{PREPARE}, tid, \Lambda_i \rangle$ from C_i
34:     $\Lambda_{tid} \leftarrow \Lambda_i$
35:     **if** willing to commit **then**
36:       **if** $opSet_{tid} \neq \emptyset$ **then**
37:         $DV_{aux} \leftarrow DV$
38:         $DV_{aux}[i] \leftarrow DV_{last}[i] + 1$
39:       **else** $DV_{aux} \leftarrow DV_{last}$
40:       send $\langle \text{VOTE}, tid, \text{YES}, DV_{aux} \rangle$ to $C_i \cup \Lambda_{tid}$
41:     **else** send $\langle \text{VOTE}, tid, \text{NO}, \perp \rangle$ to $C_i \cup \Lambda_{tid}$

42:   **when** $\exists\, tid$ such that $\forall S_i \in \Lambda_{tid}$: received
                           $\langle \text{VOTE}, tid, v_i^{tid}, DV_i^{tid} \rangle$ from S_i
43:     **if** $\forall S_i \in \Lambda_{tid} :\ v_i^{tid} = \text{YES}$ **then**
44:       submit(tid, COMMIT)
45:       **for all** $S_i \in \Lambda_{tid}$ **do**
46:         $\forall j : DV[j] \leftarrow max(DV[j], DV_i^{tid}[j])$
47:       **if** $opSet_{tid} \neq \emptyset$ **then**
48:         $DV_{last} \leftarrow DV$
49:         assynchronously write entry $\langle opSet_{tid}, DV \rangle$
                                in the transaction log

50:   **when** receive $\langle \text{ABORT}, tid \rangle$ from C_i
51:     submit(tid, ABORT)

</div>

---

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. 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.

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 [6] and [16] 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 synchro-

nization to record the fact that the current instance has finished and new ones are allowed [16]. Although some difficulties can be avoided by stronger system assumptions as in [24], 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., [6] and [16]) 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 [24]. 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 δ. 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

8

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

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 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.

4.5. Dealing with complex concurrency control

So far, we have assumed a very simple concurrency control mechanism inside every single database server, with concurrent access for read-only transactions and exclusive access for update transactions. However our results can be easily extended to more general cases. For example, the well-known two-phase-locking (2PL) algorithm can be seen as an extension of our simple concurrency control where each piece of data plays the role of a "virtual database": multiple transactions can read the data concurrently but only one can update it at a time. As a consequence, though, vector timestamps will have as many entries as the number of virtual databases.[4] Clearly the implementation of such a system can be simplified since all virtual databases inside the same physical one will be always synchronized with each other. Reducing the size of the timestamps will involve either the use of direct instead of transitive dependency tracking (and a more complex recovery algorithm [9, 26]), or the identification of false dependencies as it happens when logical clocks are used instead of vector clocks to gather causal dependencies between events [17]. Studying such alternatives is out of the scope of this paper, and subject to further work.

5 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 [24] (derived from [23]), 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 as mentioned in the previous section.

Rollback-recovery has been extensively studied in the message-passing model [1, 6, 9, 14, 16, 26, 28]. Nevertheless, very few of these works have been exploited in different environments. The work in [2] presents a framework to analyze consistency in different shared-memory and message-passing systems. In [3], 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 [3, 10, 20, 21, 27]. Different approaches have considered dependencies created between transactions due to concurrency control [5] or between data accessed within a single process which should be consistently transfered to stable storage [7]. Some of the ideas presented in these works, specially in [3] and [10], 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, 5]

6. 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

---

[4]Although in practice this might not incur in large overheads since in most MMDBs concurrency control is usually performed at a coarse granularity [11].

be tracked efficiently during execution. A possible extension to our algorithms is to use direct instead of transitive dependency tracking [9, 26], 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.

Acknowledgments

References

[1] L. Alvisi and K. Marzullo. Message Logging: Pessimistic, Optimistic, Causal and Optimal. *IEEE Trans. on Software Engineering*, 24(2):149–159, Feb. 1998.

[2] R. Baldoni, J.-M. Helari, and M. Raynal. Consistent records in asynchronous computations. *Acta Informatica*, 35(6):441–455, June 1998.

[3] R. Baldoni, F. Quaglia, and M. Raynal. Consistent checkpointing for transaction systems. *The Computer Journal*, 44(2):92–100, 2001.

[4] P. Bernstein, V. Hadzilacos, and N. Goodman. *Concurrency Control and Recovery in Databases Systems*. Addison-Wesley, 1987.

[5] B. Bhargava. Concurrency control in database systems. *IEEE Transactions on Knowledge and Data Engineering*, 11(1):3–16, 1999.

[6] M. Chandy and L. Lamport. Distributed Snapshots: Determining Global States of Distributed Systems. *ACM Trans. on Computer Systems*, 3(1):63–75, Feb. 1985.

[7] F. Cristian, S. Mishra, and Y. S. Hyun. Implementation and performance of a stable-storage service in Unix. In *Proceedings of the 15th IEEE Symposium on Reliable Distributed Systems*, 1996.

[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] 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.

[11] 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.

[12] J. Gray. The revolution in database architecture. Technical Report MSR-TR-2004-31, Microsoft Research, 2004.

[13] J. N. Gray and A. Reuter. *Transaction Processing: Concepts and Techniques*. Morgan Kaufmann, 1993.

[14] D. B. Johnson and W. Zwaenepoel. Recovery in distributed systems using optimistic message logging and checkpointing. *Journal of Algorithms*, 11(3):462–491, 1990.

[15] K. Knizhnik. Fastdb: Main-memory relational database management system. http://www.garret.ru/ knizhnik/fastdb.html.

[16] R. Koo and S. Toueg. Checkpointing and rollback-recovery for distributed systems. *IEEE Trans. on Software Engineering*, 13:23–31, Jan. 1987.

[17] L. Lamport. Time, clocks, and the ordering of events in a distributed system. *Commun. ACM*, 21(7):558–565, July 1978.

[18] D. Morse. In-memory database web server. *Dedicated Systems Magazine*, 4:12–14, 2000.

[19] 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.

[20] 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.

[21] C. Pu. On-the-fly, incremental, consistent reading of entire databases. *Algorithmica*, 1(3):271–287, 1986.

[22] 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.

[23] M. Ronström. The NDB cluster – A parallel data server for telecommunications applications. Ericsson Review no. 4, 1997.

[24] M. Ronström and L. Thalmann. Mysql cluster architecture overview. MySQL Technical White Paper, 2004.

[25] R. Schmidt and F. Pedone. Consistent main-memory database federations under deferred disk writes. Technical Report IC/2005/17, School of Computer and Communicaiton Sciences, EPFL, 2005.

[26] 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.

[27] 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.

[28] R. Strom and S. Yemini. Optimistic Recovery in Distributed Systems. *ACM Trans. on Computing Systems*, 3(3):204–226, Aug. 1985.