

# ON DEFERRED CONSTRAINTS IN DISTRIBUTED DATABASE SYSTEMS

Yousef J. Al-Houmaily

Department of Computer and Information Programs,  
Institute of Public Administration, Riyadh, Saudi Arabia

## **ABSTRACT**

*An atomic commit protocol (ACP) is a distributed algorithm used to ensure the atomicity property of transactions in distributed database systems. Although ACPs are designed to guarantee atomicity, they add a significant extra cost to each transaction execution time. This added cost is due to the overhead of the required coordination messages and log writes at each involved database site to achieve atomicity. For this reason, the continuing research efforts led to a number of optimizations that reduce the aforementioned cost. The most commonly adopted optimizations in the database standards and commercial database management systems are those designed around the early release of read locks of transactions. In this type of optimizations, certain participating sites may start releasing the read locks held by transactions before they are fully terminated across all participants. Hence, greatly enhancing concurrency among executing transactions and, consequently, the overall system performance. However, this type of optimizations introduces possible “execution infections” in the presence of deferred consistency constraints; a devastating complication that may lead to non-serializable executions of transactions. Thus, this type of optimizations could be considered useless, given the importance of preserving the consistency of the database in presence of deferred constraints, unless this complication is resolved in a practical and efficient manner. This is the essence of the “unsolicited deferred consistency constraints validation” mechanism presented in this paper.*

## **KEYWORDS**

*Atomic Commit Protocols, Database Recovery, Database Infection, Database Systems, Distributed Transaction Processing, Two-Phase Commit, Voting Protocols*

## **1. INTRODUCTION**

Preserving the atomicity property of transactions at the local level of individual database sites is not sufficient in a distributed database system. This is because a distributed transaction might end up committing at some participating sites and aborting at others due to a site or a communication failure. This jeopardizes *global atomicity* and, consequently, the consistency of the entire database. For this reason, an *atomic commit protocol* (ACP) has to be used in any distributed database system.

An ACP is essentially a distributed synchronization algorithm that guarantees a unanimous and deterministic final outcome for each distributed transaction. This outcome represents either the execution of a transaction as a whole, across all participating sites, or none at all.

ACPs, such as the basic *two-phase commit* (2PC) [1, 2] or one of its widely known variants (see [3, 4] for surveys of the most common 2PC variants and optimizations), ensure atomicity but with a significant extra cost added to each transaction execution time. This added cost is due to the overhead of the required coordination messages and log writes at each involved database site to achieve atomicity. To minimize the adverse effects of this overhead on the overall system performance, there is a continuing interest in developing more efficient ACPs and optimizations, albeit for different distributed database system environments with inherently different characteristics. These include main memory databases (e.g., [5]), mobile and ad hoc networks (e.g., [6, 7, 8] ), real-time databases (e.g., [9, 10, 11]) and component-based architectures (e.g., [12]); besides traditional distributed database systems (e.g., [3, 13]).

One of the most notable results of these continuing efforts is a type of optimizations in which transactions are allowed to release their read locks at certain participants before they have fully terminated across all participants. An example to this type of optimizations is the (traditional) *read-only optimization* [14]. This optimization is considered one of the most pronounced optimizations and is currently part of the distributed database standards [15]. In this optimization, an *exclusively* read-only participant in a transaction execution terminates the transaction and releases the read locks held on behalf of the transaction before the transaction has fully terminated at the update participants. Hence, this early release of read locks at read-only participants significantly enhances concurrency among executing transactions and, consequently, the overall system performance. This is especially important considering that read-only transactions (i.e., retrieval transactions that do not contain any update operations) are the majority of executing transactions in any general database management system. The performance enhancement of the read-only optimization on the overall system performance has been addressed in a number of simulation studies (e.g., [4, 16]) although basic analytical evaluations can still reveal the fact of enhanced performance using this optimization (e.g., [14]).

In spite of the significance of the above type of optimizations on the overall performance of distributed database systems, its applicability is curtailed due to the fact that it may lead to nonserializable executions of transactions. This is a devastating possibility which could occur whenever transactions are allowed to continue to execute at some participants, acquiring further locks, after they have already terminated at others [15, 17]. These possible *infected executions* of transactions at individual participating sites cannot be captured by any (local) concurrency control mechanism without a sort of an additional global synchronization mechanism among all involved participating sites. Furthermore, these infected executions could occur not only in the context of the more general *non-request/response* processing paradigm, but also in the context of the simpler, widely accepted and standardized *request/response* paradigm where each operation processing request is acknowledged [15].

In the context of the latter paradigm, infected executions of transactions arise when the ACP used in the system incorporates an optimization that is designed around the early release of read locks and, at the same time, consistency constraints are allowed to be evaluated in *deferred mode*. This mode for evaluating consistency constraints is currently part of the SQL (Structured Query Language) standards [18] and is defined to enhance *performance*, in certain situations, and to resolve some *applicability* limitations of consistency constraints, in others. When this mode is

used, a transaction may need to acquire further locks at its commit time/point (i.e., the time/point at which the transaction issues its final commit DCL (Data Control Language) statement). While the transaction is still executing at some participants to evaluate deferred constraints, it may terminate and start releasing its read locks at others, using the above type of optimizations, providing an opportunity for possible execution infections.

Thus, to benefit from any optimization that is designed around the early release of read locks in the presence of deferred consistency constraints, the above possible infections have to be eliminated in a practical and efficient manner. This is the essence of the “*unsolicited deferred consistency constraints validation*” (UDCCV) mechanism presented in this paper. UDCCV is designed to *detect* the above possible execution infections during the execution of individual transactions and, once detected, UDCCV *prohibit* each susceptible transaction from being infected by disallowing it from using any early read lock release optimization at commit time. Thus, UDCCV allows for the use of both deferred constraints as well as early read lock release optimizations in the system without penalizing those transactions that are not susceptible to execution infections. In UDCCV, this is accomplished through the use of an inexpensive *piggybacking* of control information in the messages exchanged among involved sites during the course of the execution of each transaction (before the transaction reaches its final commit point).

The rest of this paper is structured as follows. First, Section 2 discusses consistency constraints and highlights their impotence on performance and applicability. Section 2 also explains, through an example, the two operating modes of consistency constraints: *immediate* and *deferred*, and explains the differences between them. Then, Section 3 thoroughly explains execution infections in the presence of deferred consistency constraints. After that, Section 4 presents the UDCCV mechanism. Lastly, Section 5 summaries this paper and provides some concluding remarks.

## 2. HIGHLIGHTS OF CONSISTENCY CONSTRAINTS

In any organization, data is governed by predefined set of rules called *consistency constraint (data integrity)* rules. These rules are extracted by data administrators or application developers from the business rules of the organization and used to ensure the quality of data. Consistency constraints could be managed (i.e., both checked and enforced) at the application level or at the database management system level. One of the most important advantages of managing consistency constraints at the database management system level is that it relieves application programmers from performing this tedious, slow and error-prone task at their application level once defined within the database management system.

Relational database management systems allow for the definition of certain types of consistency constraints in a declarative manner using SQL. Once defined, the database management system is responsible about their management. In the SQL standards, a consistency constraint can be defined in an *immediate* or *deferred* mode. The immediate mode for the management of consistency constraints is the default mode and the most commonly used in practice. In this mode, a consistency constraint is evaluated at the level of each individual SQL DML (Data Manipulation Language) statement that manipulates the data governed by the consistency constraint. If a DML statement of a transaction violates a consistency constraint, the transaction is immediately aborted (i.e., rolled-back).

## 2.1 Deferred Constraints

Unlike the immediate mode, when a consistency constraint is defined in deferred mode, it is evaluated at the commit time of the transaction. Thus, the consistency constraint is evaluated at the transaction level instead of each individual DML statement. At commit time of a transaction, all deferred constraints that governs the data manipulated by the transaction are evaluated. If any of the constraints is violated, the transaction is aborted. Otherwise, the transaction is allowed to commit. The SQL standards incorporate the deferred mode for evaluating consistency constraints for two essential reasons:

- I. *Performance*: deferred constraints enhance system performance when transactions tend to abort because of users' requests to abort or because of congestion over the system resources (other than the locks on data). Thus, in such systems, by using deferred constraints, the system can avoid wasting computing resources on evaluating consistency constraints at the level of individual DML statements for transactions that will most probably end up aborting before reaching their final commit points. Similarly, system performance is enhanced when inserting (or loading) large amounts of data associated with consistency constraints into the database.
- II. *Applicability*: deferred constraints solve the problem of "cyclic consistency constraints" [19]. This problem arises whenever there is a circular chain of referential consistency constraints between two or more relations (i.e., tables). Once exist, it is impossible to insert any tuple (i.e., row or record) into any of the relations unless a method is used to temporarily breaks the chain.

Next, the above two reasons are discussed in further details.

### 2.1.1. Performance of Deferred Constraints

In deferred mode, any locks that are exclusively needed for the evaluation of a deferred consistency constraint are acquired at commit time of transactions. This is in contrast with immediate constraints where all the locks needed for the execution of a DML statement are acquired during the execution of the DML statement. This includes any exclusively needed locks for the evaluation of the consistency constraint. Thus, there is a clear trade-off between the two modes of consistency constraints. Deferred constraints hold the locks that are exclusively needed to evaluate the constraints for shorter periods of time compared with immediate constraints, allowing for more concurrency in the system. This is besides not wasting any time on evaluating the constraints during the execution of a transaction until the transaction issues its final commit statement. On the other hand, deferred constraints introduce the risk of conflicts among concurrently executing transactions at commit time. This is due to the contention over the locks that are exclusively needed to evaluate the deferred constraints compared with immediate constraints where all needed locks are acquired during execution time (before reaching the final commit point). This increases the possibility of transactions that are associated with deferred constraints to abort at commit time after they have already consumed all the required computing resources for their execution.

Based on the above trade-off, it can be concluded that deferred constraints significantly enhance the system performance when transactions tend to abort because of users' requests to abort or

because of congestion over the system resources (other than the locks on data). Similarly, system performance is enhanced when loading large amounts of data into the database, delaying the validation of constraints until the loading of the data is completed. Otherwise, immediate constraints are best to use.

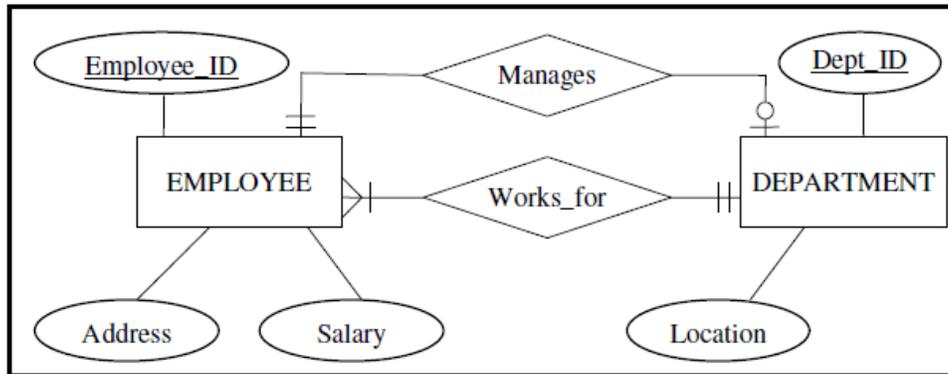


Figure 1. A cyclic consistency constraints example.

### 2.1.2. Applicability of Deferred Constraints

Consider the ER (Entity-Relationship) diagram shown in Figure 1. The diagram consists of two entities: “Employee” and “Department”; and two relationships between them: “Works\_for” and “Manages”. The “Works\_for” relationship states that “each employee works for one and only one department; and each department is worked for by one or more employees”. The “Manages” relationship states that “each employee manages zero or one department; and each department is managed by one and only one (managing) employee”.

Usually, when transforming the conceptual ER diagram shown in Figure 1 to the relational model, two relations are created: one that corresponds to the “Employee” entity and one that corresponds to the “Department” entity. We also need to observe the cardinality constraints when transforming the two relationships in the ER diagram to the relational model. That is, each employee has to be assigned to an exactly one department and each department has to be assigned to an exactly one (managing) employee. This can be accomplished by placing a “NOT NULL” constraint in the “Dept\_ID” field that corresponds to the department for which an employee works, and placing another “NOT NULL” constraint in the “Mgr\_ID” field that corresponds to the employee who manages the department, respectively. This is, of course, besides defining two referential consistency constraints: one that ties each employee to a valid department, and one that ties each department to a valid (managing) employee. This leads to the SQL DDL (Data Definition Language) statements shown in Figure 2. Notice that the two referential consistency constraints are added after the two relations are defined. This is because of the presence of a cyclic referential consistency constraints that prohibits the definition of one referential consistency constraint within the definition of one relation before the creation of the second relation, and vice versa.

```

CREATE TABLE Employee(
Emp_ID      INTEGER    PRIMARY KEY,
...
...
Dept_ID     INTEGER    NOT NULL
);

CREATE TABLE Department(
Dept_ID     INTEGER    PRIMARY KEY,
...
...
Mgr_ID      INTEGER    NOT NULL
);

ALTER TABLE Employee ADD CONSTRAINT Emp_FK FOREIGN KEY (Dept_ID)
REFERENCES Department(Dept_ID) DEFERRABLE INITIALLY IMMEDIATE;

ALTER TABLE Department ADD CONSTRAINT Dept_FK FOREIGN KEY (Mgr_ID)
REFERENCES Employee(Emp_ID) DEFERRABLE INITIALLY IMMEDIATE;

```

Figure 2. SQL DDL statements for the two relations.

Based on the SQL definitions shown in Figure 2, consider a transaction T that attempts to insert two tuples: one into the EMPLOYEE relation and one into the DEPARTMENT relation, as shown in Figure 3. Such a transaction will be aborted by the system during the execution of its first DML statement (regardless of the order of the two INSERT statements within the transaction). This is because both of the Dept\_ID in the EMPLOYEE relation and the Mgr\_ID in the DEPARTMENT relation cannot be “NULL”, and a tuple in one of the two relations requires the existence of a valid tuple in the other one (according to the rules of referential consistency constraints), and vice versa. That is, if an employee tuple is to be inserted, it has to reference a pre-existing department tuple and, if a department tuple is to be inserted, it has to reference a pre-existing (managing) employee tuple. In fact, no transaction can insert any tuples in either of the two relations at all. This dilemma is called “*cyclic consistency constraints*” and could span any number of relations, and not just two, in its general case.

```

INSERT INTO Employee values(100,...,1);
INSERT INTO Department values(1,...,100);
COMMIT;

```

Figure 3. The two DML statements of transaction T.

The only way to resolve the above dilemma is by breaking the cycle in the consistency constraints such that one of them is delayed (i.e., violated temporarily) until the end of T. This represents the other reason for using deferred consistency constraints. That is, deferred constraints solve the above dilemma by allowing a transaction to delay the validation of some or all the consistency constraints until its commit time. This is, of course, so long as the validation of each constraint that is to be delayed is defined “deferrable” (as shown in the definition of the two referential consistency constraints (i.e., foreign keys) in Figure 2). To defer the validation of a constraint until commit time or to set it back to the immediate mode, the SQL standards provide the statement shown in Figure 4 which can be invoked from within a transaction.

```
SET CONSTRAINTS ConstName DEFERRED | IMMEDIATE
```

Figure 4. SQL statement to set the constraint mode

Once one of the consistency constraints is set to deferred, the cycle in the consistency constraints is broken. Hence, T can insert an employee tuple and a department tuple, as shown in Figure 5. At commit time of the transaction, all deferred constraints are validated and the transaction is allowed to commit if it does not violate any of them. Otherwise, the transaction is aborted. In our example, only the Dept\_ID in the EMPLOYEE relation is validated at commit time and the transaction is allowed to commit. This is because T has already inserted a department tuple that satisfies the foreign key referential consistency constraint in the DEPARTMENT relation before reaching its commit point and this constraint was evaluated in immediate mode (during the INSERT DML statement of the department tuple).

```
SET CONSTRAINTS Emp_FK DEFERRED;
INSERT INTO Employee values(100, .., 1);
INSERT INTO Department values(1, .., 100);
COMMIT;
```

Figure 5. Breaking the cyclic consistency constraints.

### 3. CONSTRAINTS IN DISTRIBUTED DATABASES

The need for deferred constraints in distributed database systems is similar to their need in centralized database systems. However, their use in distributed database systems introduces a significant complication. This complication stems from the fact that any release of locks prior to the full termination of transactions across all participants may introduce non-serializable executions, jeopardizing the consistency of the entire database. This scenario arises when the ACP used incorporates optimizations that are designed around the early release of read locks, which is the case of the majority of current implementations of commercial database management systems. This scenario is thoroughly explained in the next section.

#### 3.1. 2PC and Early Release of Read Locks

In distributed database systems, once a transaction finishes its execution and issues its final “commit” request, the *transaction manager* at the site where the transaction was first initiated acts as the *coordinator* for the termination of the transaction across all participating sites. This is achieved by initiating an ACP such as 2PC.

As the name implies, 2PC consists of two phases. These two phases are the *voting phase* and the *decision phase*. During the voting phase, the coordinator of a transaction requests all the participants in the execution of the transaction to *prepare-to-commit*, whereas, during the decision phase, the coordinator either decides to commit the transaction if all the participants are prepared to commit (voted “yes”), or to abort if any participant has decided to abort (voted “no”). On a

commit decision, the coordinator sends out commit messages to all participants whereas, on an abort decision, it sends out abort messages to only those (required) participants that are prepared to commit (voted “yes”). When a participant receives a decision, it enforces the decision and sends back an acknowledgment.

When a participant receives a prepare-to-commit message for a transaction, it validates the transaction with respect to data consistency. If the transaction can be committed, the participant responds with a “yes” vote. Otherwise, the participant responds with a “no” vote and aborts the transaction, releasing all the resources held by the transaction including the locks held on behalf of the transaction.

When the coordinator receives acknowledgments from all the participants that had voted “yes”, it *forgets* the transaction by discarding all information pertaining to the transaction from its *protocol table* that is kept in main memory.

The resilience of 2PC to system and communication failures is achieved by recording the progress of the protocol in the logs of the coordinator and the participants. Specifically, the coordinator of a transaction force-writes a *decision* record for the transaction prior to sending out its final decision to the participants. Since a *forced write* of a log record causes a flush of the log onto a stable storage that survives system failures, the final decision is not lost if the coordinator fails. Similarly, each participant force-writes a *prepared* record before sending its “yes” vote and a *decision* (i.e., either commit or abort) record before acknowledging a final decision. When the coordinator completes the protocol, it writes a non-forced *end* record in the volatile portion of its log that is kept in main memory. This record indicates that all (required) participants have received the final decision and none of them will inquire about the transaction’s status in the future. Hence, allowing the coordinator to (permanently) forget the transaction, with respect to the 2PC protocol, and to garbage collect the log records of the transaction when necessary.

### **3.1.1. The Read-Only Optimization**

The read-only optimization significantly reduces the cost of 2PC for read-only transactions. This is because any *exclusively* read-only participant, a participant that has not performed any updates to data at its site on behalf of a transaction, can be completely excluded from the decision phase of the transaction.

Specifically, in this optimization, when a read-only participant in the execution of a transaction receives a prepare-to-commit message, it simply releases all the locks held by the transaction and responds with a “read-only” vote (instead of a “yes” vote). This vote means that the transaction has read consistent data and the participant does not need to be involved in the second phase of the protocol because it does not matter whether the transaction is finally committed or aborted. Consequently, this optimization allows each read-only participant to terminate and to release all the resources held by the transaction, including the read-locks, earlier than its update counterparts and without having to write any log records. This represents the essence of the traditional read-only optimization [14].

$T_1: r(x) \ w(y) \ commit$ $T_2: w(x) \ w(c) \ commit$
--

Figure 6. Two representative executing transactions.

### 3.1.2. Dangers of Early Release of Locks

The significance of the read-only optimization on the overall system performance is faced with the possibility of execution infections in the presence of deferred constraints. This possibility arises even if each participant deploys *strict two-phase locking* (S2PL) for concurrency control, the one that guarantees serializability in distributed database systems and the *de facto* concurrency control mechanism in the industry. In fact, any optimization that terminates a transaction at some participants while the transaction is still executing at others, acquiring further locks, may lead to execution infections. This represents a major complication that may jeopardize the consistency of the entire database if not resolved in a practical manner.

To illustrate execution infections and their possible consequences, consider two transactions:  $T_1$  and  $T_2$  that contain read ( $r$ ) and write ( $w$ ) operations. Each transaction also contains a final *commit* request, as shown in Figure 6.

Assume that the data items ( $y$ ) and ( $c$ ) are located at one participant ( $P_1$ ) and data item ( $x$ ) is located at another participant ( $P_2$ ). Furthermore, assume that ( $y$ ) is associated with a deferred consistency constraint that has to examine the value of ( $c$ ) for consistency before any transaction that modifies ( $y$ ) can commit.

One possible execution scenario for the two transactions is shown in Figure 7. In the figure, each operation ( $Op$ ) of a transaction ( $T$ ) submitted to a participant ( $P$ ) is represented as ( $Op_{P\_ID, T\_ID}$  ( $data\_item$ )) and when an operation pertaining to a transaction is executed successfully at a participant, it is acknowledged ( $Ack_{P\_ID, T\_ID}$ ) by the participant.

In this transaction execution scenario,  $T_1$  executes its first operation at  $P_2$  and then its second operation at  $P_1$ . Once the two operations of  $T_1$  are executed and acknowledged, the coordinator initiates 2PC for  $T_1$ . Participant  $P_2$  of  $T_1$  is read-only and releases the read lock that it holds on behalf of  $T_1$  when it receives the prepare ( $Prep_{P\_ID, T\_ID}$ ) message. It also sends a “read-only” vote ( $ROP\_ID, T\_ID$ ) to the coordinator in response to the prepare message. Participant  $P_1$  does not receive the prepare message of  $T_1$  until some latter time (possibly due to queuing or communication delays). Meanwhile,  $T_2$  starts executing and changes the released data ( $x$ ) of  $T_1$  at  $P_2$ . Thus,  $T_2$  is “serialized after”  $T_1$  at  $P_2$  ( $T_1 \rightarrow T_2$ ).  $T_2$  also changes ( $c$ ) prior to  $T_1$  had the chance to examining it at  $P_1$  (according to the consistency constraint that is placed on ( $y$ )).

After the commitment of  $T_2$  at both participants,  $P_1$  receives the delayed prepare message of  $T_1$  and acquires a read lock on ( $c$ ) to evaluate the deferred constraint associated with ( $y$ ) that  $T_1$  has modified during the course of its execution. Then,  $P_1$  proceeds with the commit processing of  $T_1$ . Thus,  $T_2$  is “serialized before”  $T_1$  at  $P_1$  ( $T_2 \rightarrow T_1$ ). By merging the two local schedules, this scenario results in a global schedule that is not serializable ( $T_1 \rightarrow T_2 \rightarrow T_1$ ), although local serializability is preserved at each participant.

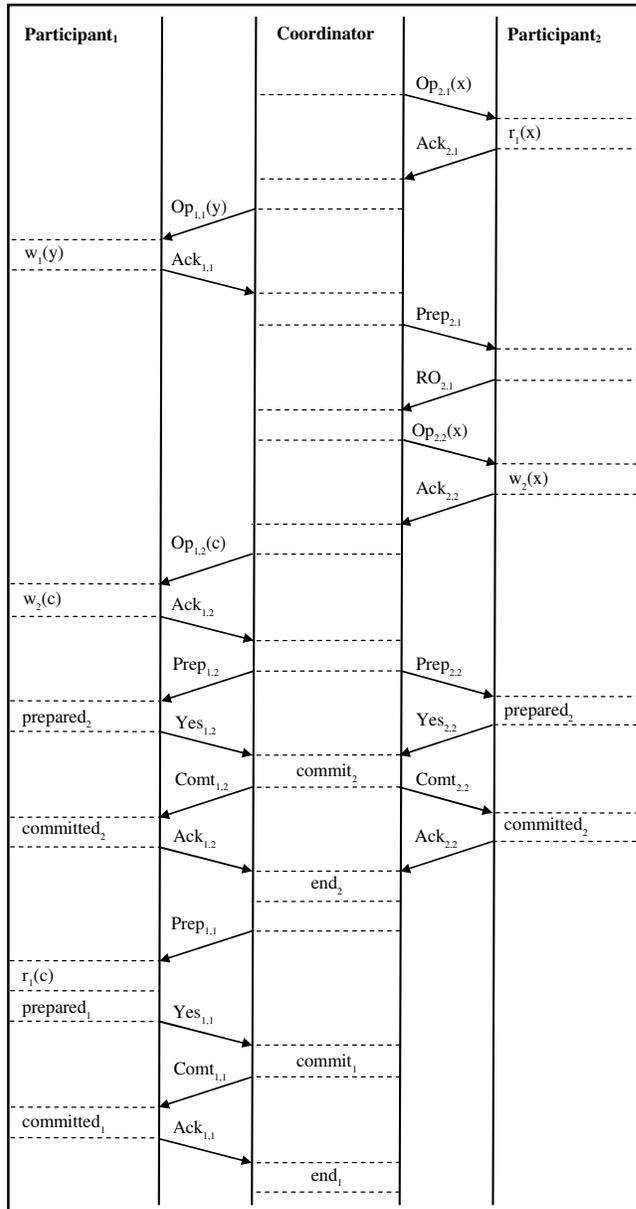


Figure 7. An example to an infected execution

The above possible execution infection could happen not only with the read-only optimization but with any optimization designed around the early release of read locks before a transaction is fully terminated across all participants. Consequently, as this type of optimizations has been shown to significantly enhance system performance and so widely popular in the implementation of commercial database management systems, there is an absolute necessity for a mechanism that is able to *detect* and to *prohibit* scenarios similar to the one above from occurring. This is the essence of the *unsolicited deferred consistency constraint validation* mechanism.

#### 4. THE UDCCV MECHANISM

As explained in the previous section, when deferred constraints are used in conjunction with optimizations designed around the early release of read locks, there are possibilities for execution infections. These infections may occur because the time at which a transaction terminates and starts releasing its read locks at some participants is not the same as the time it terminates at the other participants. This creates a different *termination time/point* for the same transaction at different participants, providing an opportunity for execution infections. Consequently, this difference in termination time is called *infection prone time period* and is formally stated as follows:

**Definition:** The *infection prone time period* is the time period between the earliest terminating time of a transaction at one participant and the latest terminating time of the same transaction at another participant.

To exclude the possibility of execution infections, the infection prone time period has to be eliminated, disallowing any further locking between the earliest terminating time at one participant and the latest terminating time at another participant. This can be accomplished by prohibiting the use of either any optimization that is designed around the early release of read locks, or deferred consistency constraints. The first approach makes the earliest terminating time of a transaction at any participant to be the same as the latest terminating time for the same transaction at any other participant; whereas, the second approach makes the latest terminating time of a transaction at any participant to be the same as the earliest terminating time for the same transaction at any other participant. In other words, regardless of which approach is used, the termination time of each transaction will be the same across all participants. However, neither of these two approaches are acceptable from performance point of view. This is because both of them deprive the system of the anticipated performance enhancement gain. Not only that, but the second approach also deprives the system of a solution to a consistency constraint applicability limitation, as explained in Section 2.1.2.

A third approach that also eliminates the infection prone time period is also visible. In this approach, all exclusively needed locks for the evaluation of deferred constraints are acquired during the execution of individual DML statements and postponing the (actual) validation process of deferred constraints until commit time of transactions. This approach enhances system performance by not performing the actual validation of deferred constraints until the end of transactions but reduces the level of concurrency among executing transactions. This is because it leads to the holding of the locks exclusively needed for the evaluation of deferred constraints, unnecessarily, from the time that a DML statement is executed until the termination time of the transaction. Thus, all the three approaches above fall short of providing the full anticipated benefits of deferred constraints and, at the same time, the optimizations designed around the early release of read locks, motivating the design of the UDCCV mechanism.

Unlike the first two approaches above, UDCCV eliminates the infection prone time period on a per *transaction* basis and not on the *system design level*. That is, UDCCV allows for optimizations that are designed around the early release of read locks to co-exist with deferred constraints and resolves any possible execution infections on a per transaction basis. More specifically, in UDCCV, when a participant executes an operation pertaining to a transaction and

the operation is associated with a deferred consistency constraint that needs to be validated at commit time of the transaction, the participant sends an UDCCV *flag* as part of the operation acknowledgment message. When the coordinator of the transaction receives such a message from a participant, it means that the transaction may need to acquire further locks at the participant at commit processing time. In this way, the coordinator, with the help of the participant, can *detect* possible execution infections at the participant and it has to *prohibit* it from occurring at commit processing time of the transaction.

At the commit time of a transaction (i.e., when the transaction issues its final commit statement), the coordinator knows which participants have executed operations associated with deferred consistency constraints. If any participant has executed such an operation, the coordinator needs to *prohibit* the possibility of execution infections. This is accomplished by sending an UDCCV flag as part of each prepare message that the coordinator sends to each participant not aware of the possible execution infections. Notice that more than participant may have already sent UDCCV flags during the execution of the transaction as they have deferred constraints to be validated at their sites at commit time of the transaction. Thus, these participants are already aware of possible execution infections and do not need to be informed about this possibility.

When a participant receives a prepare message with an UDCCV flag for a transaction, the participant becomes aware of possible execution infections. Based on that, the participant continues to hold all the locks, including the read-only ones, until the final decision is received. That is, the participant does not apply any of its implemented early read lock release optimizations with this specific transaction. This means that the participant will follow 2PC until the completion of both of its phases and not just the first phase as though it does not implement any early read lock release optimization. Thus, when a potential case of execution infection is encountered, all participants continue to hold all locks until the final decision is made, eliminating the possibility of execution infections.

If none of the participants had executed any operation that is associated with deferred constraints, the received prepare message by a read-only participant will not contain an UDCCV flag. Hence, the participant can utilize the read-only optimization and releases the read locks earlier than its update counterparts. Thus, the UDCCV delays the early release of read locks only when there is a possibility of execution infections, eliminating the infection prone time period on a per transaction basis.

Returning to the previous example with the UDCCV in mind, when  $P_1$  executes the second operation of  $T_1$ , it will send an UDCCV flag as part of its acknowledgment ( $Ack_{1,1}$  in Figure 7). Based on that, the coordinator will send an UDCCV flag as part of the prepare message to  $P_2$  ( $Prep_{2,1}$ ). Hence,  $P_2$  will be aware of the possibility of the execution infection and, instead of releasing the read lock held on behalf of  $T_1$  and sending a “read-only” message ( $RO_{2,1}$ ),  $P_2$  will continue to hold the read lock and sends a “yes” vote, awaiting the final decision from the coordinator. Based on that,  $T_2$  will not be able to execute its operation at  $P_1$  as the lock needed on (x) at  $P_2$  is still held on behalf of  $T_1$ , preventing the execution progress of  $T_2$ . This eliminates the possibility for the two transactions to engage in a non-serializable execution due to an execution infection. This result can be generalized with the following theorem.

**Theorem:** In any distributed database system that is based on a request/reply paradigm and implements deferred consistency constraints, the UDCCV mechanism guarantees infection-free executions of transactions when combined with S2PL and an ACP that incorporates an optimization designed around the early release of read locks.

**Proof:** The proof proceeds by contradiction and consists of two parts.

For an execution infection to occur, there has to be a lock on a data item ( $d$ ) that is released by a transaction  $T_i$  before its commit point at some participant  $P$  and ( $d$ ) is locked and *modified* by another transaction  $T_j$  which commits before  $T_i$  in the global serialization schedule as follows:  $lock_i(d) < lock_j(d)$  and  $T_j \rightarrow T_i$

I. If  $lock_i$  is a “write” lock then it is impossible for  $T_j$  to acquire a lock on ( $d$ ) and reaches its commit point before  $T_i$  reaches its commit point first because, in S2PL with an ACP, the *write locks* are released after transactions are terminated across all participants.

II. If  $lock_i$  is a “read” lock then it is possible for  $T_j$  to acquire a lock on ( $d$ ) and reaches its commit point before  $T_i$  terminates across all participants according to any early read lock release optimization incorporated in the ACP used and not S2PL. But, as UDCCV is used,  $lock_i$  cannot be released until  $T_i$  terminates across all participants. Thus, it is impossible for  $T_j$  to precede  $T_i$  in the serialization order.

#### 4.1. The UDCCV Mechanism and Other Optimizations

The UDCCV can be used with other optimizations designed around early release of read locks such as early *release of read locks* (ERRL) [17] and the *unsolicited update vote* (UUV) [20, 21].

In ERRL, a participant in a transaction execution does not have to be an exclusively read-only participant for it to release the read locks held on behalf of the transaction at prepare time of the transaction. That is, in ERRL, each update participant is also allowed to release the read locks held on behalf of a transaction once it receives the prepare message for the transaction.

UUV is another read-only optimization that further reduces the costs associated with read-only participants. In UUV, each transaction starts as a read-only transaction at each participant and when the transaction executes the first operation that updates data at a participant, the participant sends an unsolicited update-vote. The “*unsolicited update-vote*” is a flag that is sent as part of the operation acknowledgment. Thus, at the end of a transaction, the coordinator precisely knows all update participants and infers that the rest of the participants are read only. In contrast to the traditional read-only optimization, this is accomplished, in UUV, without having to explicitly pull the vote of each participant. Consequently, the coordinator only needs to inform each read-only participant that the transaction has terminated without requiring the participant to send back any reply message, reducing the number of messages with read-only participants, compared with the traditional read-only optimization, to the half.

The UDCCV mechanism can be used in conjunction with ERRL in a manner similar to its use with the traditional read-only optimization. That is, when a participant executes an operation that is associated with a deferred constraint, the participant sends an UDCCV flag as part of the successful operation acknowledgment. This flag informs the coordinator about a possible

execution infection. Consequently, at commit processing time, the coordinator informs each other participant about this possibility using an UDCCV flag that it sends as part of each prepare message. Once a participant receives such a flag, it continues to hold the read locks until the end of the commit protocol (i.e., the second phase of the protocol). If the transaction does not perform any operation that is associated with deferred constraints, the participants can still utilize ERRL as usual. Thus, UDCCV prevents possible execution infections only with transactions that are associated with deferred constraints while still able to utilize ERRL with the other transactions.

The UDCCV mechanism can be also used in conjunction with UUV but in a slightly different manner than its use with the traditional read-only optimization. This is because, in UUV, a coordinator does not pull the votes of read-only participants through prepare messages. Consequently, in the case of possible execution infections, the coordinator cannot request from any participant to continue to hold the read locks until the commit decision is made. Instead, the coordinator delays the termination message, which is called “read-only” in UUV, that it is supposed to send to each read-only participant until after it has received the votes of the update participants and made the final decision for the transaction. Thus, ensuring full termination of transactions across all participants before any locks are released. Similar to the previous two optimizations, in UUV, the UDCCV mechanism is used on a per transaction basis and only in the presence of potential execution infections.

## 5. SUMMARY AND CONCLUSIONS

The most commonly adopted *atomic commit protocol* (ACP) optimizations in the database standards and commercial database management systems are those designed around the early release of read locks. However, when this type of optimizations is used in conjunction with deferred consistency constraints that are evaluated at commit time of transactions, execution infections may occur. These infections could lead to non-serializable executions of transactions, jeopardizing the consistency of the entire database. Thus, there is a need for global synchronization mechanisms to detect execution infections and to prohibit them from occurring. The *unsolicited deferred consistency constraints validation* (UDCCV), presented in this paper, represents one such mechanism.

The UDCCV mechanism relies on the *piggybacking* of control information among all involved database sites in a transaction’s execution to detect and to prohibit any execution infection. Hence, UDCCV allows for the use of deferred consistency constraints while eliminating possible execution infections when adopting those widely advocated ACP optimizations that are designed around the early release of read locks. UDCCV achieves this on a per transaction basis and without adding any extra cost to the coordination messages and log writes of the used ACP.

## REFERENCES

- [1] Gray, J. (1978) “Notes on Database Operating Systems”, in Bayer, R., R.M. Graham & G. Seegmuller, (Eds.), *Operating Systems: An Advanced Course*, LNCS, Vol. 60, pp.393–481, Springer-Verlag.
- [2] Lampon, B. (1981) “Atomic Transactions”, in Lampon, B., M. Paul & H.J. Siegert, (Eds.): *Distributed Systems: Architecture and Implementation – An Advanced Course*, Vol. 105, pp.246–265, Springer-Verlag.

- [3] Al-Houmaily, Y. (2010) "Atomic Commit Protocols, their Integration, and their Optimisations in Distributed Database Systems", *Int'l J. of Intelligent Information and Database Systems*, Vol. 4, No.4, pp. 373-412.
- [4] Chrysanthis, P., G. Samaras & Y. Al-Houmaily (1998) "Recovery and Performance of Atomic Commit Processing in Distributed Database Systems", (Chapter 13), in Kumar, V. & M. Hsu (Eds.), *Recovery Mechanisms in Database Systems*, Prentice Hall.
- [5] Lee, I. & H. Yeom (2002) "A Single Phase Distributed Commit Protocol for Main Memory Database Systems", in *Proc. of the 6th Int'l Parallel and Distributed Processing Symposium*.
- [6] Ayari, B., A. Khelil & N. Suri (2011) "On the Design of Perturbation-Resilient Atomic Commit Protocols for Mobile Transactions", *ACM Transactions on Computer Systems*, Vol. 29, No. 3, pp.7:1-7:36.
- [7] Nouali-Taboudjemat, N., F. Chehbour & H. Drias (2010) "On Performance Evaluation and Design of Atomic Commit Protocols for Mobile Transactions", *Distributed and Parallel Databases*, Vol. 27, No. 1, pp. 53-94.
- [8] Obermeier, S., S. Böttcher, M. Hett, P. Chrysanthis & G. Samaras (2009) "Blocking Reduction for Distributed Transaction Processing Within MANETs", *Distributed and Parallel Databases*, Vol. 25, No. 3.
- [9] Haritsa, J., K. Ramamritham & R. Gupta (2000) "The PROMPT Real-Time Commit Protocol", *IEEE Transactions on Parallel and Distributed Systems*, Vol. 11, No. 2.
- [10] Qin, B. & Y. Liu (2003) "High Performance Distributed Real-Time Commit Protocol", *J. of Systems and Software*, Vol. 68, No. 2.
- [11] Shanker, U., M. Mesra & A. Sarje (2008) "Distributed Real Time Database Systems: Background and Literature Review", *Distributed and Parallel Databases*, Vol. 23, No. 2, pp. 127-149.
- [12] Rocha, T., A-B. Arntsen, A. Eidsvik, M. Toledo & R. Karlsen (2007) "Promoting Levels of Openness on Component-Based Adaptable Middleware", in *Proc. of the 6th Int'l Workshop on Adaptive and Reflective Middleware*.
- [13] Al-Houmaily, Y. (2013) "An Intelligent Adaptive Participant's Presumption Protocol for Atomic Commitment in Distributed Databases", *Int'l J. of Intelligent Information and Database Systems*, Vol. 7, No. 3, pp. 242-277.
- [14] Mohan, C., B. Lindsay & R. Obermarck (1986) "Transaction Management in the R\* Distributed Data Base Management System", *ACM TODS*, Vol. 11, No. 4, pp. 378-396.
- [15] X/Open Company Limited (1991) "Distributed Transaction Processing: The XA Specification", (X/Open Doc. No. XO/CAE/91/300).
- [16] Samaras, G., G. Kyrou & P. Chrysanthis (2003) "Two-Phase Commit Processing with Restructured Commit Tree", in *Proc. of the Panhellenic Conf. on Informatics, LNCS*, Vol. 2563, pp.82-99.
- [17] Lomet, D. (1993) "Using Timestamping to Optimize Two Phase Commit", in *Proc. of the 2nd Parallel and Distributed Information Systems*.
- [18] ISO (2008) "Information Technology - Database Languages - SQL - Part 2: Foundation (SQL/Foundation)", *ISO/IEC 9075-2*.
- [19] Garcia-Molina, H., J. Ullman & J. Widom (2008) "Database Systems: The Complete Book", 2nd edition, Prentice-Hall.
- [20] Al-Houmaily, Y., P. Chrysanthis & S. Levitan (1997) "Enhancing the Performance of Presumed Commit Protocol", in *Proc. of the 12th ACM Annual Symposium on Applied Computing*.
- [21] Al-Houmaily, Y., P. Chrysanthis & S. Levitan (1997) "An Argument in Favor of the Presumed Commit Protocol", in *Proc. of the 13th ICDE*.

**AUTHOR**

Yousef J. Al-Houmaily received his BSc in Computer Engineering from King Saud University, Saudi Arabia in 1986, MSc in Computer Science from George Washington University, Washington DC in 1990, and PhD in Computer Engineering from the University of Pittsburgh in 1997. Currently, he is an Associate Professor in the Department of Computer and Information Programs at the Institute of Public Administration, Riyadh, Saudi Arabia. His current research interests are in the areas of database management systems, mobile distributed computing systems and sensor networks

