2023 IEEE 39th International Conference on Data Engineering (ICDE) | 979-8-3503-2227-9/23/$31.00 ©2023 IEEE | DOI: 10.1109/ICDE55515.2023.00019

LCL: A Lock Chain Length-based Distributed Algorithm for Deadlock Detection and Resolution

Zhenkun Yang, Chen Qian, Xuwang Teng, Fanyu Kong, Fusheng Han, Quanqing Xu
OceanBase, AntGroup
OceanBaseLabs@list.alibaba-inc.com

***Abstract*—The problem of deadlock detection and resolution in database systems has been studied for decades. While it has long been a mature feature of classical centralized database systems for many years, its use in distributed database systems remains in its infancy. Don P. Mitchell and Michael J. Merritt (*M&M*) proposed a simple and fully distributed deadlock detection and resolution algorithm, but its assumption that each process waits on only one resource at a time prevents it from being generally applicable. Inspired by this algorithm, we design and implement *LCL* (Lock Chain Length), an elegant and generally applicable algorithm for resource deadlock detection and resolution in distributed environments without a restriction of the above kind. Our extensive emulation experiments show that the proposed approach *LCL* significantly outperforms the state-of-the-art competitor *M&M*. In addition, it has been applied to the OceanBase distributed relational database system, and our extensive experiments in OceanBase illustrate that *LCL* is also more efficient than *M&M* in deadlock detection and resolution.**

***Index Terms*—Deadlock Detection, Transaction, Distributed Database, Wait for Graph.**

I. Introduction

Over the past decade, we have developed OceanBase [1] [2] [3], an open-source [4], distributed relational database system. Owing to its distributed characteristic, it recorded over 700 million tpmC on the TPC-C benchmark test in 2020 [5] and more than 15 million QphH@30,000 GB on the TPC-H benchmark test in 2021 [6]. Tens of thousands of OceanBase servers have been deployed in production systems of Alipay.com as well as banks and other commercial business organizations.

In addition to its distributed characteristic, OceanBase [1] supports almost all features of mainstream, classical, centralized relational database management systems (RDBMS), e.g., secondary index, view, trigger, cursor, constraints, functions, and stored procedures. Deadlock detection in OceanBase is an important issue, as in a classical centralized RDBMS. Compared with deadlock detection in a centralized database system, that in a distributed environment is much more challenging as each site has only a portion of the view of the whole system [7] [8] [9] [10]. Furthermore, even though a centralized deadlock detection scheme is possible in a distributed environment, a distributed scheme is much more natural and attractive owing to performance- and complexity-related issues [10].

Mitchell and Merritt [11] proposed a simple and fully distributed deadlock detection (*M&M*) algorithm. However, it assumes that each transaction process (vertex) waits on only one resource at a time, i.e., one outgoing edge per vertex in the wait for graph (*WFG*), whereas a vertex may depend on multiple vertices because it may wait for resource(s) occupied by vertices in use. For example, owing to performance-related considerations, parallel execution is critical, especially in a distributed database system. SQL *select* often accesses multiple parts or partitions of a table, and each part or partition executed in parallel may be blocked by other transaction locks, causing one transaction to wait for multiple locks. This above assumption prevents it from becoming a generally applicable deadlock detection method in a distributed environment.

In this paper, we propose *LCL*, an elegant, efficient and easy-to-implement distributed algorithm that can easily detect and resolve resource deadlocks in a distributed environment. We make the following contributions to distributed deadlock detection and resolution technology here:

- We propose a lock chain length (*LCL*)-based distributed deadlock detection and resolution algorithm in light of the requirements of OceanBase. *LCL* is fully distributed consisting of three stages: *LCL* proliferation, spread, and detection. It is implemented and used in OceanBase.
- We confirm that *LCL* has the following benefits: First, each vertex needs neither a WFG global view nor a local view, but only needs to send information to its direct downstream vertices (transaction processes that hold resources that it is waiting for), dozens of bytes each time. Second, every deadlock can be detected and each cycle is detected by just one vertex in it, thus simplifying the problem of the resolution. Third, only a real deadlock is detected and resolved, and transactions that are not deadlocked are not rolled back.
- We provide a strict mathematical proof to show that the *LCL* algorithm can easily detect and resolve deadlocks in distributed environments.
- The results of performance evaluation based on a distributed transaction processing emulator (TPE) on a server and OceanBase show that *LCL* significantly outperforms *M&M*.

The remainder of this paper is organized as follows: Section II gives definitions needed for describing and proving *LCL*. Section III describes the proposed algorithm and Section IV provides a strict proof for it. Section V illustrates *LCL* in OceanBase. Section VI provides the experimental results of *LCL*. Section VII provides a brief review of related work on distributed deadlock detection and resolution technologies. Section VIII summarizes the conclusions of this study.

2375-026X/23/$31.00 ©2023 IEEE
DOI 10.1109/ICDE55515.2023.00019

II. Definitions

In this section, we give some definitions.

A. Basic Definitions

Definition II.1 ($A \to B$)**.** *$A \to B$ represents an edge from vertex A to vertex B, which means vertex A (transaction process) is waiting for some resource being held by vertex B (transaction process). A is then said to be immediately upstream of B and B is said to be immediately downstream of A. $A(\neq B) \to B$ is used for $A \neq B$ and $A \to B$. Because it is trivial to find and fix a resource deadlock inside a transaction process, we assume there are no such edges (loops). For convenience of discussion, we denote $A \to A$.*

Definition II.2 ($A \Rightarrow B$)**.** *For two vertices A and B, $A \Rightarrow B$ if there exists a walk [12] from A to B. A is said to be an upstream vertex of B and B is said to be a downstream vertex of A, and A to B are said to be one-way connected. If $A \Rightarrow B$ and $B \Rightarrow C$, then $A \Rightarrow C$. For convenience of discussion, we denote $A \Rightarrow A$.*

Definition II.3 ($A \Leftrightarrow B$)**.** *For two vertices A and B, $A \Leftrightarrow B$ if both $A \Rightarrow B$ and $B \Rightarrow A$. A and B are then said to be two-way connected. If $A \Leftrightarrow B$, then $B \Leftrightarrow A$. If $A \Leftrightarrow B$ and $B \Leftrightarrow C$, then $A \Leftrightarrow C$. It is clear that any two vertices in a cycle [12] are two-way connected. For convenience of discussion, we denote $A \Leftrightarrow A$.*

Definition II.4 ($\langle x, y \rangle$)**.** *$\langle x, y \rangle$ is an integer with x as its high-order bits and y as its low-order bits.*

Definition II.5 ($Dist(A, B)$)**.** *This is only meaningful for $A \Rightarrow B$. Dist(A, B):= n. If there exists a walk of length n from A to B, for any walk of length m from A to B, $n \leq m$. Dist(A, A) == 0. If $A(\neq B) \to B$, Dist(A, B) == 1.*

B. Annulus subgraph distance

The strongly connected component [13] $SCC(A)$ indicates the SCC at which A is located, and has the following properties. Suppose scc_1 and scc_2 are two SCCs:

1) If $scc_1 \cap scc_2 \neq \emptyset$, then $scc_1 == scc_2$.
2) If there is $X \in scc_1$, $Y \in scc_2$, and $X \Leftrightarrow Y$, then $scc_1 == scc_2$.
3) If $X \Leftrightarrow Y$, then $SCC(X) == SCC(Y)$. So, for a cycle cyc, $SCC(cyc)$ is used to represent the strongly connected components containing cyc.
4) If X and Y are two vertices in scc_1, then any walk from X to Y is also in scc_1.
5) If X is not in any cycle, $SCC(X)$ is trivial, i.e., $SCC(X) == \{X\}$.

Definition II.6 ($USG(scc)$)**.** *The upstream subgraph (USG) of scc. $USG(scc) = (VS, ES)$, where $VS = \{X \in WFG : X \Rightarrow A, A \in scc\}$, $ES = \{X \to Y : X \in VS$ and $Y \in VS$ and $X \to Y \in WFG\}$.*

Suppose $A \in scc_1$ and $B \in scc_2$. If $A \Rightarrow B$, then $USG(scc_1) \subseteq USG(scc_2)$; if $A \Leftrightarrow B$, then $USG(scc_1) == USG(scc_2)$. If X is a vertex in $USG(scc_1)$, then any walk from some vertex to X is also in $USG(scc_1)$. Clearly, $scc_1 \subseteq USG(scc_1)$.

Definition II.7 ($ASG(scc)$)**.** *The annulus subgraph (ASG) of scc (as shown in Figure 1). $ASG(scc) = (VS, ES)$, where $VS = \{X \in WFG : X \in USG(scc)$ and $X \notin scc\}$, $ES = \{X \to Y : X \in VS$ and $Y \in VS$ and $X \to Y \in WFG\}$.*

Suppose $A \in scc_1$ and $B \in scc_2$. If $A \Leftrightarrow B$, $ASG(scc_1)$ == $ASG(scc_2)$. If we think that $USG(scc_1)$ is an outer circle and scc_1 is an inner circle, $ASG(scc_1)$ is the annulus.

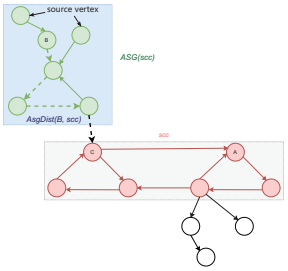


Fig. 1. *ASG* and *AsgDist*

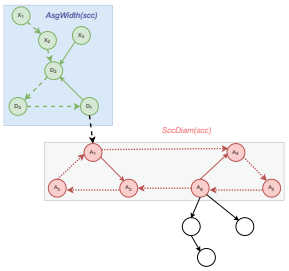


Fig. 2. *AsgWidth* and *SccDiam*

Definition II.8 (*AsgDist*(A, scc))**.** *For $A \in ASG(scc)$, AsgDist$(A, scc) := max\{m$: There exists m distinct vertices C_m(==A), $C_{m-1}, \ldots, C_1$ in ASG(scc) and a vertex $C_0 \in scc$ such that C_m(==A) $\to C_{m-1} \to \ldots \to C_1 \to C_0\}$.*

For $A \in scc$, *AsgDist*$(A, scc) := 0$. If A is in the annulus $ASG(scc)$, the length is a positive number; if A is in the inner circle scc, the distance is zero. As shown in Figure 1, *AsgDist*(B, scc) = 4.

III. *LCL* Algorithm

In this section, we give the detailed *LCL* algorithm.

A. Key idea of LCL

LCL is a fully distributed deadlock detection and resolution algorithm in OceanBase [1], as shown in Algorithm 1. All messages in *LCL* have a fixed length (tens of bytes), and each vertex sends messages only to its immediate downstream vertices, i.e., transaction processes that hold the resources for which it is waiting, and only one deadlocked process detects and resolves the deadlock in each deadlocked cycle. In *LCL*, each vertex requires neither a global view nor even a local view in WFG. We present the following definitions:

Definition III.1 (AsgWidth(scc))**.** *AsgWidth(scc) := $\max\{$AsgDist$(X, scc) : X \in USG(scc)\}$. AsgWidth$(scc)$ can be considered to be the width of $ASG(scc)$, the annulus.*

Definition III.2 (PrivateID, *PrID*)**.** *The PrID of a vertex is a fixed constant integer, and is unique for each vertex.*

Definition III.3 (PublicID, $PuID$)**.** *The $PuID$ of a vertex is an integer, and its initial value is set to the PrID of the vertex.*

Definition III.4 (Private Abortion Priority, $PrAP$)**.** *The $PrAP$ of a vertex is a fixed constant integer. During the resolution of a deadlock, the vertex with the largest $\langle PrAP,$ PrID$\rangle$ in the deadlock is chosen as the victim.*

Definition III.5 (Public Abortion Priority, $PuAP$)**.** *The $PuAP$ of a vertex is an integer, and its initial value is set to the $PrAP$ of the vertex.*

Definition III.6 (Lock Chain Length Value, $LCLV$)**.** *The $LCLV$ of a vertex is an integer with an initial value of zero. Usually, $A.LCLV == m$ implies that there is/was a walk of length m from some vertex to A.*

Definition III.7 ($max\{S.LCLV\}$)**.** *Let S be a subset of* WFG, *$max\{S.\ LCLV\}$:= $max\{X.LCLV$: X is a vertex in $S\}$. If S is an empty set, then $max\{S.LCLV\}$ is zero. The meanings of $max\{S.\langle PuAP, PuID\rangle\}$ and $max\{S.\langle PrAP,$ PrID$\rangle\}$ are similar.*

Definition III.8 ($SccDiam(scc)$)**.** *Let scc be an SCC, $SccDiam(scc)$:= max $\{Dist(X, Y) : X \in scc$ and $Y \in scc\}$. $SccDiam(scc)$ is the diameter of scc, the inner circle.*

Algorithm 1: *LCL*-based Distributed Algorithm for Deadlock Detection and Resolution

Input: vertex A, vertex B, A waits for B, $A \neq B$
Output: res

1 ▷ **Stage 1: *LCL* Proliferation**
2 **for** $i \leftarrow 1$ *to max(AsgWidth(scc),1)* **do**
3 $\quad A.\langle PuAP, PuID\rangle = A.\langle PrAP,$ PrID$\rangle$;
4 $\quad B.\langle PuAP, PuID\rangle = B.\langle PrAP,$ PrID$\rangle$;
5 $\quad B.LCLV = \max(B.LCLV, A.LCLV + 1)$;
6 ▷ **Stage 2: *LCL* Spread**
7 **for** $i \leftarrow 1$ *to* $2\times$*SccDiam(scc)* **do**
8 $\quad B.LCLV = \max(B.LCLV, A.LCLV)$;
9 $\quad$ **if** *B.LCLV == A.LCLV* **then**
10 $\quad\quad B.\langle PuAP, PuID\rangle = \max(B.\langle PuAP, PuID\rangle, A.\langle PuAP, PuID\rangle)$;
11 ▷ **Stage 3: *LCL* Detection**
12 **if** *B.LCLV == A.LCLV and* $B.\langle PuAP, PuID\rangle == A.\langle PuAP, PuID\rangle$ *and* $B.\langle PuAP, PuID\rangle == B.\langle PrAP,$ PrID$\rangle$ **then**
$\quad$ `// B is a detected deadlock vertex`
13 $\quad res \leftarrow B$;
14 **return** res;

The key idea of *LCL* is a "lock chain length" value ($LCLV$) of a transaction process, defined as Definition III.6. The $LCLV$ of a vertex is the length of a lock-dependent chain path from some other transaction to the given transaction. The key rule is that $LCLV$ of transaction process B is at least one unit greater than that of transaction process A if A depends on a resource held by B (i.e., $A \rightarrow B$). According to this rule, the "lock chain length" value ($LCLV$) of a transaction process in a cycle can proliferate to an unlimitedly large value owing to the characteristic of the cycle, and reaches an upper limit if any path ending at the vertex contains no cyclic part. The *LCL* algorithm contains three stages as follows:

B. Stage 1: LCL *Proliferation*

We define *LCL* proliferation as shown in Definition III.9.

Definition III.9 (*LCL* Proliferation)**.** *An* LCL *proliferation deduction on an edge $A(\neq B) \rightarrow B$ is defined as:* $A.\langle PuAP, PuID\rangle == A.\langle PrAP,$ PrID$\rangle$ *(Definition II.4) and* $B.\langle PuAP, PuID\rangle == B.\langle PrAP,$ PrID$\rangle$ *and* $B.LCLV = max(B.LCLV, A.LCLV + 1)$.

One can imagine that if there is a walk of length n ending at A, then there is a walk of length $n+1$ ending at B if $A(\neq B) \rightarrow B$. Figure 2 shows the following:

1) If there is a large $LCLV_1$ in $ASG(scc)$, then, after *AsgWidth*(scc) at most, this value may be increased by *AsgWidth*(scc) and disseminated to scc. Then, the $LCLV$ of the vertex in $ASG(scc)$ no longer increases.
2) Suppose that $A_1 \rightarrow A_2 \rightarrow A_3 \rightarrow A_1$ is a cycle in scc, and $A_1.LCLV$ is the largest. Then, after the proliferation of $A_1 \rightarrow A_2$, $A_2 \rightarrow A_3$, and $A_3 \rightarrow A_1$, $A_1.LCLV$ increases to at least three. Therefore, if the time available is long enough, the $LCLV$ of the vertex on the cycle can be arbitrarily large.

As shown in Figure 3, there are max(*AsgWidth*(scc),1) rounds of the *LCL* proliferation deduction in this stage (lines 2–5). Figure 3(a) shows the initial states of $LCLV$, $PuAP$, and $PuID$, where the $LCLV$ of each vertex is zero. Figure 3(b) illustrates the intermediate states of $LCLV$, $PuAP$, and $PuID$ (lines 4–5), where the $LCLV$ of each vertex is a number shown in bold orange. Figure 3(c) shows the final states of $LCLV$, $PuAP$, and $PuID$ after stage 1, where $LCLV$ of each vertex is modified and shown in bold orange, except that of the first vertex, which is still zero. The time complexity of stage 1 is $O(N\times M_1)$, where N is the number of directed edges in *WFG* and M_1 is *AsgWidth*(scc) (Definition III.1), which is the width of the largest annulus in Figure 2.

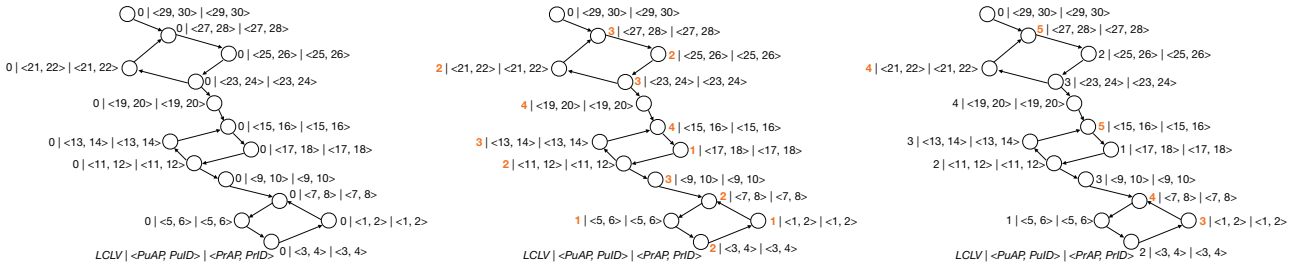
C. Stage 2: LCL *Spread*

We first give the definition of *LCL* spread, as shown in Definition III.10. One round of the proliferation/spread of *LCL* is defined as in Definition III.11.

Definition III.10 (*LCL* Spread)**.** *An* LCL *spread deduction on an edge $A(\neq B) \rightarrow B$ is defined as:* $B.LCLV = max(B.LCLV, A.LCLV)$, *if* $B.LCLV == A.LCLV$, $B.\langle PuAP, PuID\rangle$=$max(B.\langle PuAP, PuID\rangle, A.\langle PuAP,\ PuID\rangle)$ *(Definition II.4).*

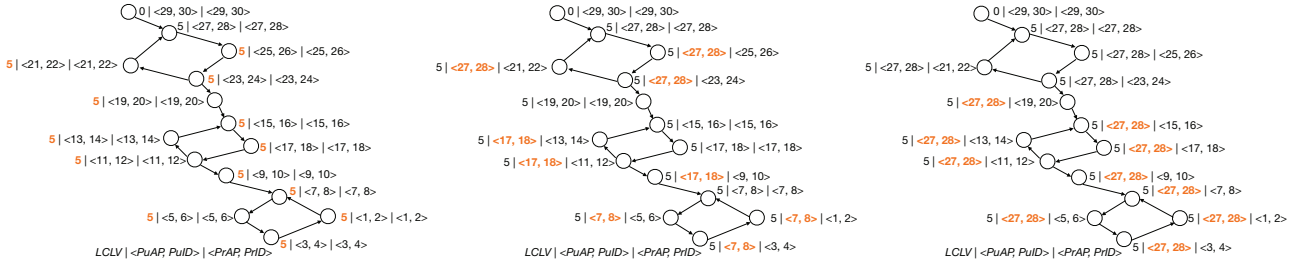
Definition III.11 (One round of the *LCL* proliferation/spread deduction)**.** *This implies that the* LCL *proliferation/spread deduction is applied on every edge in WFG at least once. The exact times of application on each edge and the order of application are arbitrary.*

As shown in Figure 4, there are $2\times$*SccDiam*(scc) rounds of the *LCL* spread deduction in this stage (lines 7–10). Figure 4(a) shows a completed state of $LCLV$ spread (line 6), where the changed $LCLV$ is shown in bold orange. Figure 4(b) illustrates intermediate states of the spread of $PuAP$ and $PuID$ (lines 9–10), where the changed values of $\langle PuAP, PuID\rangle$ are shown in bold orange. Finally, Figure 4(c) shows final states of spread of $LCLV$, $PuAP$, and $PuID$, where the changed $\langle PuAP, PuID\rangle$ is also shown in bold orange. As in stage 1,



(a) Initial state of $LCLV$, $PuAP$ and $PuID$ (b) An intermediate state of $LCLV$, $PuAP$ and $PuID$ (c) Final state of $LCLV$, $PuAP$ and $PuID$

Fig. 3. Stage 1: *LCL* Proliferation



(a) Completed state of $LCLV$ spread (b) An intermediate state of spread of $PuAP$ and $PuID$ (c) Final state of spread of $LCLV$, $PuAP$ and $PuID$

Fig. 4. Stage 2: *LCL* Spread

the time complexity of stage 2 is $O(N \times M_2)$, where N is the number of directed edges in *WFG*, and M_2 is $SccDiam(scc)$, the diameter of the largest SCC in *WFG* in Figure 2.

D. *Stage 3:* LCL *Detection*

In this stage, only one round of detection is needed, and is defined as (lines 12–13). As shown in Figure 5, for each edge $A(\neq B) \rightarrow B$ in *WFG*, if $B.LCLV$ == $A.LCLV$ and $B.\langle PrAP, \textbf{PrID}\rangle$ == $B.\langle PuAP, PuID\rangle$ and $B.\langle PuAP, PuID\rangle$ == $A.\langle PuAP, PuID\rangle$, then A and B are in a cycle, and B is the victim of the resolution of the cycle. The time complexity of stage 3 is $O(N)$, where N is the number of directed edges in *WFG*. Each deadlock can be detected, and only by the vertex with the largest $\langle \textbf{PrID}, PrAP\rangle$ in it. In addition, only a real deadlock is detected and unlocked, and transactions that are not in deadlock are not rolled back.

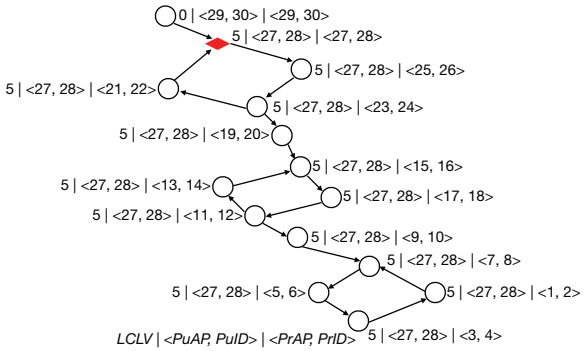


Fig. 5. Stage 3: *LCL* Detection

IV. Proof Of *LCL* Algorithm

In this section, we give a strict mathematical proof of the *LCL* algorithm.

A. *Correctness*

1) Safety: After vertexes depend on deadlocks, and they do not actively break the dependencies (e.g., the transaction will not end due to timeout and the user actively kills the session), in a view with N deadlock loops, up to no more than N nodes are killed during the process of no longer deadlocking in the view. It is said that *LCL* does not have the multi-kill feature. No vertex will be killed until transaction dependencies form a deadlock. It is said that *LCL* does not have the characteristics of manslaughter.

2) Liveness: If there is a deadlock path in the view, within a limited time, there must be a vertex that can discover this fact. If there are multiple deadlock paths in the view, and the deadlocked vertex can break at least one deadlock, there is no deadlock path in the final view.

B. *Topmost SCC*

We prove that there is at least one topmost SCC in WFG if there is an SCC.

Lemma IV.1. *Suppose* scc_1 *and* scc_2 *are two SCCs in WFG; if* $USG(scc_1)$ == $USG(scc_2)$*, then* scc_1 == scc_2.

The proof of IV.1 can be found in the appendix. Similarly, the proof of other lemmas and theorems can also be found in the appendix.

Definition IV.2 (Proper upstream SCC)**.** *An SCC* scc_2 *is said to be a proper upstream SCC (as shown in Figure 6) of* scc_1 *if* $scc_2 \subseteq ASG(scc_1)$.

Definition IV.3 (Topmost SCC)**.** *An SCC scc is said to be a topmost SCC if it has no proper upstream SCC.*

In Figure 6, scc_2 is a proper upstream SCC, and both scc_3 and scc_4 are topmost SCCs.

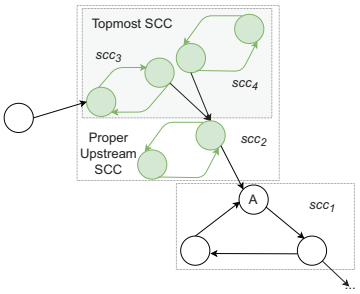


Fig. 6. Proper upstream SCC and topmost SCC in WFG

Lemma IV.4. *If scc_2 is a proper upstream SCC of scc_1, then:*

(i) $scc_2 \neq scc_1$

(ii) $USG(scc_2) \subset USG(scc_1)$

As shown in Figure 7, if scc_2 is a proper upstream SCC of scc_1, then: 1) $USG(scc_2)$ is a proper subset of $USG(scc_1)$ (①→③), and 2) scc_2 (④) is not the same as scc_1 (②).

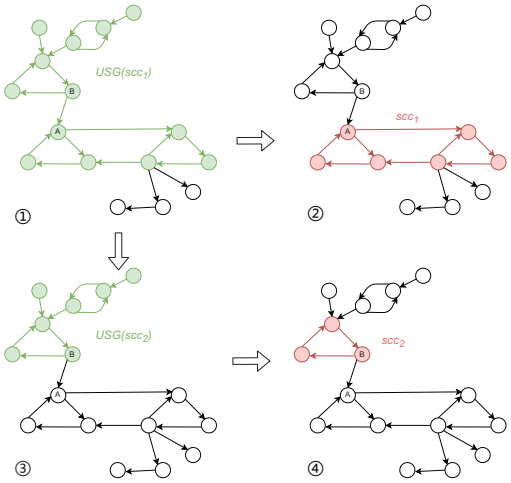


Fig. 7. Illustration of Lemma IV.1 and Lemma IV.4

Theorem IV.5. *In WFG, if there exists an SCC, then there exists at least one topmost SCC.*

C. Annulus subgraph

We first present the following lemmas (Lemma IV.6 ~ Lemma IV.8) in the annulus subgraph that have an important corollary (Corollary IV.9): For any vertex $X \in ASG(scc)$, where scc is a topmost SCC, there is a vertex $Y \in USG(scc)$, such that $X(\neq Y) \rightarrow Y$ and *AsgDist*(X, scc) == *AsgDist*$(Y, scc) + 1$.

Lemma IV.6. *Suppose scc is a topmost SCC in WFG, X_m, X_{m-1}, $\cdots$, X_1 are m ($m \geq 1$) distinct vertices in $ASG(scc)$, and $X_m \rightarrow X_{m-1} \rightarrow \cdots \rightarrow X_1$. If $X_{m+1} \in$ ASG(scc) and $X_{m+1}(\neq X_m) \rightarrow X_m$, then X_{m+1}, X_m, $\cdots$, X_1 are $m+1$ distinct vertices.*

Lemma IV.7. *Suppose scc is a topmost SCC in* WFG, $X \in ASG(scc)$ *and* $Y \in USG(scc)$, *and* $X(\neq Y) \rightarrow Y$; *then,* *AsgDist*$(X, scc) \geq$ *AsgDist*$(Y, scc) + 1$.

Lemma IV.8. *Suppose scc is a topmost SCC in* WFG, *vertex* $X \in ASG(scc)$, *and denote* *AsgDist*(X, scc) *by* m; *then,* $m \geq 1$, *and there exists a walk* $X_m(==X) \rightarrow X_{m-1} \rightarrow \ldots \rightarrow X_1 \rightarrow X_0$, *where* $X_m(==X)$, X_{m-1}, $\cdots$, X_1 *are* m *distinct vertices in ASG(scc) and* $X_0 \in$ *SCC(scc); then,* *AsgDist*(X_j, scc) == j, *for* $j = 0, 1, 2, \cdots, m-1$.

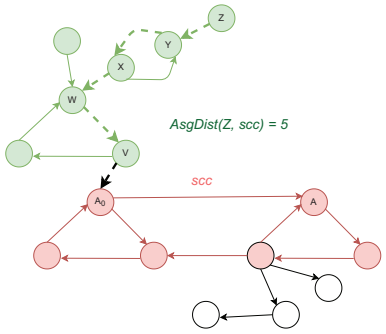


Fig. 8. Illustration of Lemma IV.8

In Figure 8, we can see that *AsgDist*(Z, scc) = 5, where the farthest path is $Z \rightarrow Y \rightarrow X \rightarrow W \rightarrow V \rightarrow A_0$. From Lemma IV.8, it is easy to obtain the following:

Corollary IV.9. *Suppose scc is a topmost SCC in* WFG. *Then, for any $X \in ASG(scc)$, there exists $Y \in USG(scc)$ such that $X(\neq Y) \rightarrow Y$ and* *AsgDist*(X, scc) == *AsgDist*$(Y, scc) + 1$.

D. Cycle detection and resolution

In this subsection, we use Lemma IV.10 to prove that the vertex with the largest $\langle PrAP, PrID \rangle$ in scc will detect the deadlock, after at least $\max(w, 1)$ rounds of *LCL* proliferation deductions and $2 \times d$ rounds of *LCL* spread deductions, where scc is a topmost SCC in WFG, $w =$*AsgWidth*(scc) and $d =$ *SccDiam*(scc).

Lemma IV.10. *During the* LCL *proliferation stage, after at least w rounds of* LCL *proliferation deductions,*

(i) *the $LCLV$ of every vertex in $ASG(scc)$ is invariant;*

(ii) *if* $\forall X \in ASG(scc)$ *and* $\forall Y \in USG(scc)$ *such that* $X(\neq Y) \rightarrow Y$, *then* $X.LCLV < Y.LCLV$; *and*

(iii) $max\{ASG(scc).LCLV\} < max\{scc.LCLV\}$, *where scc is a topmost SCC in WFG and* $w =$*AsgWidth*(scc).

Suppose scc is a topmost SCC, $w =$*AsgWidth*(scc). After at least w rounds of proliferation deductions, we can conclude:

1) The LCLV value of each vertex in $ASG(scc)$ is unchanged. scc is a topmost SCC; there is thus no cycle in $ASG(scc)$, and thus the LCLV values of all vertices have an upper limit.
2) If $X(\neq Y) \rightarrow Y$, X is a vertex in $ASG(scc)$ and Y is a vertex in $USG(scc)$. Since $X.LCLV$ is invariant, according to the definition of LCLV proliferation (Definition III.9), $Y.LCLV \geq X.LCLV + 1$.
3) The maximum $LCLV$ value of the vertices in $ASG(scc)$ must be less than the maximum value of $LCLV$ of those in scc.

Theorem IV.11. *Suppose scc is a topmost SCC in WFG, let* $w =$*AsgWidth*(scc) *and* $d =$*SccDiam*(scc), *then, after*

at least $\max(w, 1)$ *rounds of* LCL *proliferation deductions followed by at least* $2 \times d$ *rounds of* LCL *spread deductions, there is a unique vertex* B *in* $USG(scc)$ *and a vertex* $A \in USG(scc)$*, such that* $A(\neq B) \rightarrow B$ *and* $B.LCLV == A.LCLV$ *and* $B.\langle PrAP, \textbf{PrID}\rangle == B.\langle PuAP, PuID\rangle == A.\langle PuAP, PuID\rangle$*; Furthermore,* B *is the vertex with the largest* $\langle PrAP,$ PrID$\rangle$ *in* scc*.*

As shown in Figure 9, we can understand how the *LCL* algorithm works as follows: scc (red dashed vertices), the inner circle or a topmost SCC, is surrounded by its upstream vertices (dark-green) on the outside, i.e., its annulus subgraph, which contains no cycle. The whole graph, i.e., scc plus its annulus subgraph, can be regarded as an "outer circle", $USG(scc)$, the upstream subgraph. The most upstream vertex (dark-green) exists at the outermost periphery of the outer circle, and needs to pass through other upstream vertices (also dark-green) to reach scc over several steps. The maximum number of steps, i.e., $AsgWidth(scc)$=7, can be regarded as the width in the broad sense of the annulus subgraph. Each round of the deduction of *LCL* algorithm causes each vertex to affect its immediate downstream vertices. In one *LCL* proliferation deduction, LCLV is transmitted and increased step by step. After $AsgWidth(scc)$=7 rounds of *LCL* proliferation deduction, LCLV values of vertices in the annulus subgraph, i.e., $ASG(scc)$, no longer change and are smaller than their corresponding vertices in scc. Because $\langle PuAP, PuID\rangle$ of all vertices are re-initialized by *LCL* proliferation deduction, LCLV and $\langle PuAP, PuID\rangle$ of vertices in scc are no longer affected by the vertices outside of scc, i.e., the annulus subgraph, during *LCL* spread deduction. If $\langle PuAP, PuID\rangle$ is regarded as a kind of token, then $AsgWidth(scc)$ rounds of *LCL* proliferation deduction guarantee that all tokens passed in scc comes only from vertices in it during the spread stage of *LCL*. When the token issued by the vertex with the largest $\langle PuAP, PuID\rangle$ in the cycle returns to it after at most $2\times SccDiam(scc)$=$2\times$2=4 steps of *LCL* spread deduction, the deadlock detection of the cycle is done.

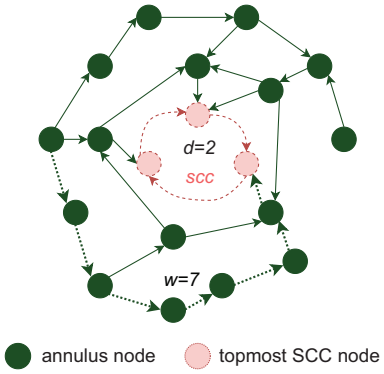


Fig. 9. Deduction of the *LCL* algorithm

V. LCL in OceanBase

In a distributed cluster of OceanBase, the *LCL* algorithm is used to realize distributed deadlock detection and resolution. As shown in Figure 10, a client connects to a node (*OBServer1*) in the OceanBase cluster, and a user starts a transaction on *OBServer1* through the client. This node is called the scheduler of the transaction. If the user executes a write statement involving modification in the transaction, the following operations will be performed. 1) The scheduler will analyze the SQL statement and parse the rows to be modified; 2) It finds the different partitions that these rows belong to; 3) It pinpoints a server where the Paxos replica leader of the corresponding partition is located; 4) It divides the modification behavior into multiple subtasks and sends them to these leader nodes for execution; 5) It waits to collect the execution results of all subtasks. If all subtasks are successful, continue to execute. If any subtasks fail, they will be processed or retried according to the failure results.

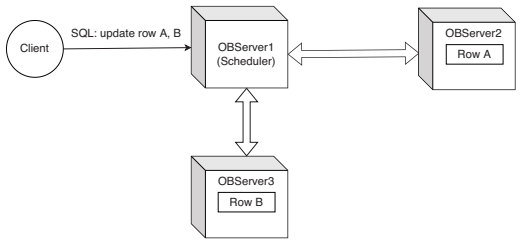


Fig. 10. Illustration of *LCL* in OceanBase

A SQL statement involving modification of several partitions may modify data on multiple OBServers in a distributed execution plan. These subtasks may not all succeed during execution. The most common cause of failure is lock conflict. After the subtask fails due to the lock conflict, the subtask will return the information of the collider to the scheduler, and the scheduler will establish the dependency relationship between the server and the collider according to the *LCL* algorithm.

The deduction of *LCL* will periodically pass a special token to the collider. If the collider also has its own collider, it will also deduce the *LCL* algorithm and continue to pass the token. If these colliders depend on each other, they will form a loop. The deduction of *LCL* will eventually allow the collider with the largest private abortion priority to discover this fact. It is worth noting that, for the consideration of execution efficiency, the subtasks of the distributed execution plan are executed in parallel, and the design of *LCL* multi-outdegree parallel detection can detect multiple dependent paths at the same time, and ensure that the occurrence of any one of the paths occurs. When a deadlock is detected, this fact is a huge advantage of *LCL* applied to the distributed parallel execution of SQL.

VI. Performance Evaluation

In this section, we verify the correctness of *LCL*, and study the performance of *LCL* comparing with *M&M* in a distributed transaction-processing emulator (TPE) that we implemented and a OceanBase cluster.

A. Setting

Each transaction consists of multiple SQLs, including: 1) type 1 SQL, which requires a lock, and 2) type 2 SQL, which does not require a lock. Each SQL needs to apply for a different number of resources, and the resources will

be distributed on different machines. Executing a transaction specifically refers to applying for the resources required by each SQL. After a transaction applies for a resource, the resource will be occupied. After the transaction executes all SQL statements, it will be committed and all the resources occupied will be released. When the resources requested by the current transaction are occupied by other transactions, the current transaction may be blocked and need to wait.

TABLE I
EXPERIMENTAL CONFIGURATION

Parameters	Setting
# of transaction threads	80
# of deadlock detection threads	15
Deadlock detection cycle	2.64 s
# of transaction processors	127,000
# of resources owned by each transaction processor	400, 410, ⋯, 490, 500, ⋯, 1000
The percentage of SQL statement requiring locks	50%
Minimum/maximum/average # of SQL statements per transaction (exponential)	10/50/30
The minimum/maximum/average number (exponential) of resources for each SQL request	1/5/1.2
Minimum/maximum/average # of SQL statements per transaction/variance (normal)	10/50/30/10
The minimum/maximum/average number/variance (normal) of resources for each SQL request	1/5/1.20/0.65

As shown in Table I, based on historical SQL queries in OceanBase [1], two distributions were employed in the TPE as follows: 1) **Exponential (Exp.) distribution.** Each transaction consisted of 10 to 50 randomly generated SQL statements with an average of 30 (i.e., $\lambda = 1/30$). Each type 1 SQL statement randomly exclusively locked one to five rows, with an average of 1.2 rows (i.e., $\lambda = 1/1.2$). 2) **Normal distribution.** Each transaction consisted of 10 to 50 randomly generated SQL statements with an average of 30 and a variance of 10. Each type 1 SQL statement exclusively locked one to five rows, with an average of 1.2 rows and a variance of 0.65. In both distributions, 50% of the SQL statements (type 1 SQL) require locks, and 50% of them (type 2 SQL) do not require locks. The lock attribute of each SQL statement was randomly generated as well. These rows were randomly chosen from the entire database. Each SQL statement took a little more than 2 milliseconds on average.

B. Evaluation in TPE

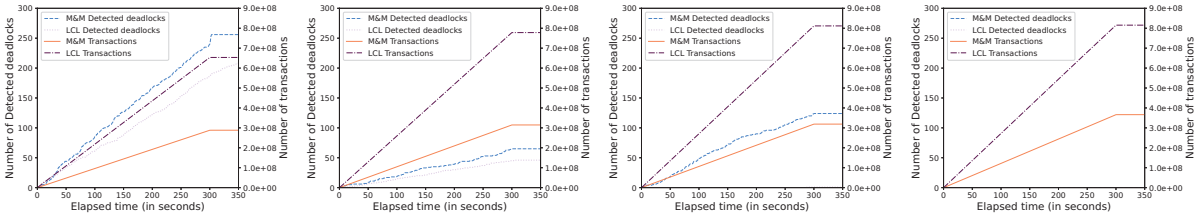
1) TPE: A TPE can continuously process transactions: 1) generate transactions, 2) execute transactions, and 3) commit transactions. The TPE was run on a two-way Intel® Xeon® Platinum 8163 CPU @ 2.50 GHz server. Table I shows the experimental configuration. Eighty of the server's 96 hyper-threads were used by the TPE and 15 by the *LCL* algorithm. All transaction processing and deadlock detection were carried out on the server's 768 GB RAM. The TPE emulated 127 (a prime number to avoid unevenly locked row selection among nodes of the database) database nodes and 1,000 transaction processes on each node. Only one table had the same number of rows on each node. Tables of different sizes (number of rows) were chosen to generate different deadlock statuses.

All transaction processes of the TPE were contained in one big array and the private ID of each was simply the array index. The private abortion priority of a transaction was generated by an auto-increment integer on each database node, similar to the start timestamp of a transaction. Each round of *LCL* detection and resolution lasted 2,640 ms: 1,200 ms for *LCL* proliferation, 1,200 ms for *LCL* spread, and 240 ms for *LCL* detection. The TPE and the *LCL* algorithm were each run for a predefined time, 300 seconds in our emulation. No transaction process generated any new transactions after this predefined time, but all processes continued their current transactions and the *LCL* algorithm continued until all transactions had been terminated, either committed or rolled back.

2) Exponential distributions of SQL and resources: We analyze the deadlock statuses of *M&M* and *LCL* on tables of different sizes using the exponential distributions of both SQL and resources. Figures 11(a), 11(b), 11(c), and 11(d) show the numbers of detected deadlocks and committed transactions during a 300s emulation period on tables with 400,000, 600,000, 900,000, and 1,000,000 rows, respectively. Figures 11(a)˜11(d) show that both *M&M* and *LCL* grow sublinearly in the four figures and their curvature did not decrease, indicating that *LCL* is better than *M&M* for the given scenario in deadlock detection and resolution. Figure 11 demonstrates that there are less deadlocks in *LCL* than *M&M* because *LCL* that is a multi-outdegree deadlock detection and resolution approach is more effective than *M&M* that is a single-outdegree one in deadlock detection and resolution.

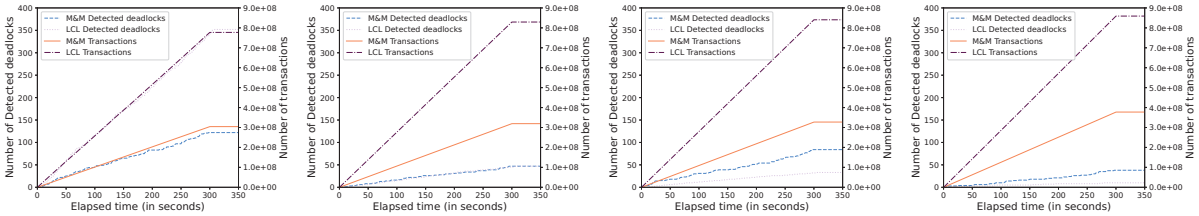
The number of committed transactions in case of 400,000, 600,000, 900,000, and 1,000,000 rows gradually increased, indicating that the row lock conflicts decreased with increasing number of rows for *M&M* and *LCL*. Figure 11 shows that the number of deadlocks gradually decreased when the number of rows increased. The probability of deadlock was low and few row lock conflicts occurred, as shown in Figure 11(b)˜11(d). The greater the number of rows, the greater the number of total committed transactions with a high probability. When the number of rows rose, the number of detected deadlocks and the length of the deadlock cycle roughly decreased because the numbers of row lock conflicts and deadlocks decreases. There were few deadlocks, and none in the case of more than 900,000 rows. *M&M* runs on a single-outdegree transaction processing system, and *LCL* runs on a multi-outdegree transaction processing system. The experimental results in Figure 11 show that *LCL* can detect deadlocks in a timely manner even on multi-outdegree transaction processing systems, and the number of transaction commits is also significantly improved.

3) SQL exp. distribution and resources normal distribution: We also performed evaluations of *M&M* and *LCL* on tables with 400,000, 600,000, 900,000, 1,000,000 rows with an exponential distribution of SQL and a normal distribution of resources. Figure 12 shows the numbers of detected deadlocks and committed transactions during the 300-second emulation period on tables with 400,000, 600,000, 900,000, and 1,000,000 rows, respectively. The four figures demonstrate that both *M&M* and *LCL* resolved the deadlocks within 300 seconds, but *LCL* is over 2 times more than *M&M* in transactions processed. The *M&M* algorithm has a single outdegree restriction, and the *LCL* algorithm supports multiple



(a) 400,000 rows (b) 600,000 rows (c) 900,000 rows (d) 1,000,000 rows

Fig. 11. Elapsed time on tables of different sizes with exponential distributions of SQL and resources



(a) 400,000 rows (b) 600,000 rows (c) 900,000 rows (d) 1,000,000 rows

Fig. 12. Elapsed time on tables of different sizes with exponential distribution of SQL and normal distribution of resources

out-degrees, thereby improving the parallelism of transaction processing, so *LCL* is more efficient than *M&M* in deadlock detection and resolution. The deadlock probability was low, and it did not take much time to identify and resolve the deadlocks. This probability thus increases sublinearly in Figure 12.

4) SQL normal distribution and resources exp. distribution: Figures 13(a), 13(b), 13(c), and 13(d) show the numbers of identified deadlocks and committed transactions for *M&M* and *LCL* during the 300-second emulation on table with 400,000, 600,000, 900,000, and 1,000,000 rows, respectively. The greater the number of rows was, the smaller was the time cost of solving deadlocks due to a reduction in the length of the deadlock cycle. Figure 13 shows that the *LCL* algorithm could deal with the submitted transactions with more excellent scalability than the *M&M* algorithm because of LCL's high parallelism of transaction processing.

5) Normal distributions of SQL and resources : Figures 14(a), 14(b), and 14(c) show the numbers of detected deadlocks and committed transactions for *M&M* and *LCL* during the 300s emulation period on tables with 400,000, 600,000, and 900,000 rows, respectively, for reasons similar to those for the first three scenarios. Figure 14(d) shows that the number of committed transactions during the 300-second emulation period on the table with 1,000,000 rows encountered few deadlocks, and even zero deadlocks in some instances. This is because there were enough rows such shat there was little or even no row lock conflict. Similarly, *LCL* is over 2 times more than *M&M* in the number of transactions processed, because a transaction in *LCL* can apply for multiple resources at the same time, and the transaction may be executed soon after the deadlock is released and the multiple resources are released by the transaction. However, after the deadlock is released in *M&M*, it is necessary to apply for the next resources. It may encounter more deadlocks when executing a transaction.

Note that the results of each run, and the numbers and sizes of the deadlocks might be slightly different in Figures 11˜14, due to the emulation randomness from the exponential and normal distributions of SQL and resources.

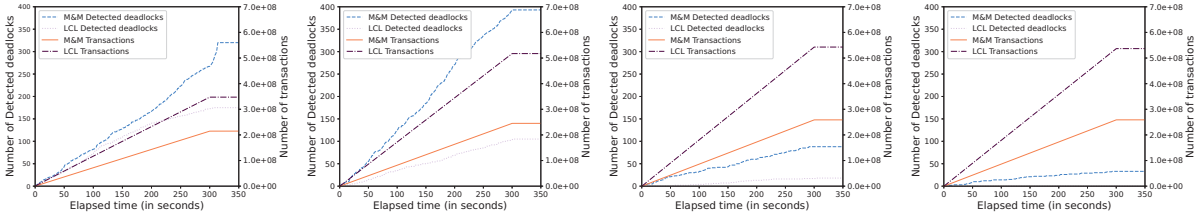
C. Evaluation in OceanBase

1) OceanBase cluster: We run *LCL* and *M&M* 300 seconds, respectively in a OceanBase cluster including nine servers. Each server is equipped with Intel(R) Xeon(R) Platinum 8269CY CPU @ 2.50GHz, 256 GB memory, gigabit ethernet, and operates AliOS 7.

2) Exponential distributions of SQL and resources: We first evaluate the number of successful transactions and their response time (RT) with different numbers of rows for *LCL* and *M&M*. It can be observed that *LCL* is over 40% more than *M&M* in the number of successful transactions in Figure 15(a). Referring to Figure 15(b), *M&M* is slower than *LCL* in the response time of successful transactions. Note that the RT of each transaction is relatively high due to lock conflicts. If there is no lock conflict, the RT of a transaction should be within 100ms. This is because the *M&M* algorithm only detects one out-degree, it is possible that a deadlock cannot be detected. In this case, it is necessary to rely on the transaction timeout time to release the deadlock. However, the *LCL* algorithm detects all out-degrees and releases the deadlock immediately when the deadlock is formed. It can release the occupied resources faster than *M&M* without depending on the timeout.

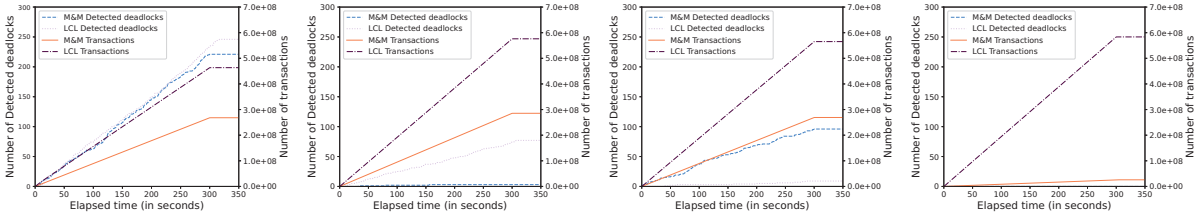
3) SQL exp. distribution and resources normal distribution: We present the results of *LCL* and *M&M* with exponential distribution of SQL and normal distribution of resources. Figure 16(a) shows that *M&M* only achieves as 74.2% as *LCL* in the number of successful transactions, and Figure 16(b) demonstrates that *M&M* is 49.7% slower than *LCL* in the response time of transactions. This significant performance gain is attributed to our multi-outdegree scheme in *LCL*, which results in releasing deadlocks more efficient than *M&M*.

4) SQL normal distribution and resources exp. distribution: In this experiment, we study the performance of *LCL* and *M&M* with normal distribution of SQL and exponential distribution of resources when varying the number of rows.



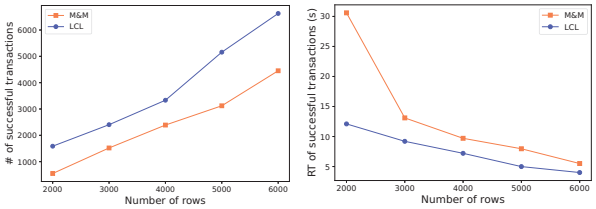
(a) 400,000 rows (b) 600,000 rows (c) 900,000 rows (d) 1,000,000 rows

Fig. 13. Elapsed time on tables of different sizes with normal distribution of SQL and exponential distribution of resources



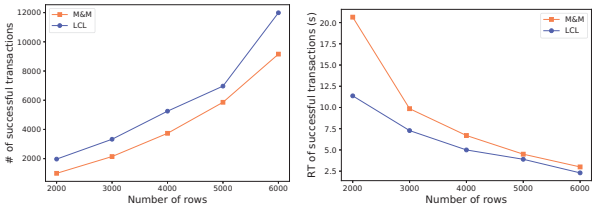
(a) 400,000 rows (b) 600,000 rows (c) 900,000 rows (d) 1,000,000 rows

Fig. 14. Elapsed time on tables of different sizes with normal distribution of SQL and normal distribution of resources



(a) # of successful transactions (b) RT of successful transactions

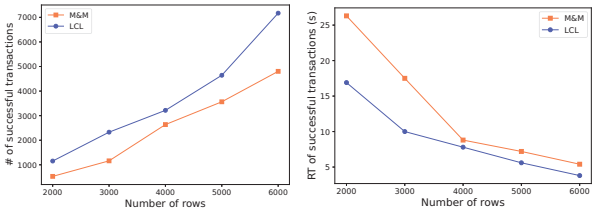
Fig. 15. *LCL* vs. *M&M* with exp. distributions of SQL and resources



(a) # of successful transactions (b) RT of successful transactions

Fig. 16. *LCL* vs. *M&M* with SQL exp. and resources normal distributions

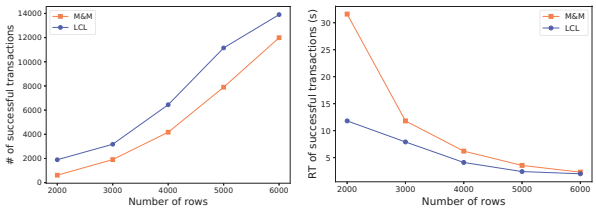
Figure 17(a) shows that *LCL* is around 45.8% more than *M&M* in the number of successful transactions because more out-degrees increase the throughput. More out-degrees also shorten the response time, as shown in Figure 17(b), because they can help to release deadlocks more efficiently.



(a) # of successful transactions (b) RT of successful transactions

Fig. 17. *LCL* vs. *M&M* with SQL normal and resources exp. distributions

5) Normal distributions of SQL and resources : In this experiment, we compare *LCL* with *M&M* when varying the number of rows with normal distributions of SQL and resources. In Figure 18(a), the advantage of the *LCL* approach is again confirmed: it is 32.4% more than *M&M* in the number of successful transactions as shown in Figure 18(a), and it is 49.1% less than *M&M* in the response time of successful transactions as shown in Figure 18(b), owing to the same previous reasons.



(a) # of successful transactions (b) RT of successful transactions

Fig. 18. *LCL* vs. *M&M* with normal distributions of SQL and resources

VII. Related Work

A. Distributed Deadlock Detection Algorithms

Knapp [10] divides distributed deadlock detection algorithms into the following four categories: 1) path-pushing algorithms, 2) edge-chasing algorithms, 3) diffusing computations, and 4) global state detection.

Obermarck et al. [14] proposed a distributed deadlock detection algorithm among transactions running concurrently in a distributed database system and presented its performance characteristics. CockroachDB's [15] deadlock detection is based on another well-known path-pushing distributed algorithm instead of the edge-chasing algorithm. It lets each transaction coordinator maintain its transaction waiting queue, regularly collecting the waiting relationships of all transactions in its waiting queue, it merges them into their own waiting relationships, and finally maintains a complete lock-waiting graph so that deadlocks can be detected. The disadvantage of this algorithm is that multiple vertices detect the same

deadlock, and each vertex tries to unlock it by killing a certain transaction in a cycle. However, many of the published path-pushing algorithms have turned out to be incorrect. For example, Elmagarmid et al. [16] stated that Phantom Deadlock is detected in the Obermarck's algorithm, since snapshots were taken asynchronously. For *LCL*, final results are deterministic in multiple out-degree scenarios, and each vertex only maintains its own information.

Oracle RAC also applies the edge-chasing algorithm. As stated in its patent [17], it uses frontward, backward, and concurrent searches based on both optimizations of frontward and backward searches. Forward search looks for transactions $\{P_2, P_3, \cdots\}$ that occupy resources waiting for transaction P_1, and uses the depth-first algorithm to recursively locate transactions that occupy its waiting resources. If there is no out-degree, it reduces the number of branches and tries again; if the depth-first algorithm returns to transaction P_1, the deadlock can be identified. Backward search is the opposite: By searching all resources held by transaction P_1, it finds the transactions $\{P_4, P_5, \cdots\}$ waiting for them, recursively searches backwards, and can find deadlocks in a similar way to forward search. Concurrent search uses concurrent queries instead of branch reduction for a depth-first search. It reduces the delay in finding the deadlock and, thus, the requisite network bandwidth. Oracle RAC selects the algorithm according to the corresponding rules. When the frequency of deadlocks is low, only forward or backward search is used. Otherwise, parallel optimized search that uses both is selected according to their frequency. *LCL* is also an edge-chasing algorithm.

Global state detection algorithms try to obtain a consistent snapshot of the WFG, and search for deadlocks in it to avoid inconsistency which may lead to the detection of phantom deadlocks. Barbosa et al. [18] revisited concepts of the deadlock model and proposed the simplified AND-OR model as a simpler alternative to the AND-OR model. Bracha et al. [19] presented a simple and efficient algorithm to detect deadlocks, which is a generalization of the well-known AND-OR request model in distributed systems. However, such algorithm assumes synchronous communication, which incurs a severe overhead. In comparison, system cost in *LCL* is as low as possible. Krivokapic et al. [20] proposed a deadlock detection agents (DDAs)-based distributed deadlock detection algorithm, and used a thorough simulation of deadlock detection algorithms to verify that the DDA scheme is better than other algorithms.

B. Other Deadlock Detection Algorithms

Deadlock detection in TiDB [21] is a central node algorithm. It involves choosing a node to collect all node-lock waiting relationships to generate a global lock-waiting graph. Obviously, it is not a distributed algorithm. Google Spanner [22] utilizes a simple wounded wait approach for deadlock detection. However, transactions in wound-wait and wait-die [23] may be selectively killed before deadlock occurs, which is pessimistic and thus accepted by very few customers. In contrast, *LCL* can keep as transactions alive as possible instead of killing them.

C. Lock Management

Yoon et al. [24] proposed decentralized and starvation-free lock management with RDMA (DSLR) as an RDMA-based decentralized lock manager. It targets distributed systems running on RDMA-enabled networks, and uses the notion of a lease to detect and resolve deadlocks via its advisory locking rules. NetLock [25] is a centralized lock manager that co-designs servers and network switches to achieve good performance without sacrificing flexibility in policy support. It uses the capability of emerging programmable switches to directly process lock requests in the switch data plane. Devulapalli et al. [26] presented a means of delegating lock management to the participating lock-requesting nodes by using advanced network primitives such as the RDMA and atomic operations. It complements the original idea of the distributed lock manager (DLM) in which lock space management is distributed. However, this design can only support exclusive mode locking. Narravula et al. [27] proposed a protocol for distributed locking services by using advanced network-level one-sided atomic operations provided by InfiniBand. This protocol augments prevalent approaches by eliminating the requirement for two-sided communication protocols in the critical locking path.

D. Deadlock Detection Algorithm Surveys

Bukhres et al. [28] presented performance evaluation study of three distributed deadlock detection approaches and demonstrated that the partially distributed one was the best. Elmagarmid et al. [16] surveyed research work performed from 1980 to 1986 in distributed deadlock detection, and only reviewed detection of resource deadlocks. Singhal et al. [29] also surveyed earlier work on distributed deadlock detection in distributed database systems. However, the surveys did not give quantitative analysis in terms of benchmarking on distributed deadlock detection. Lee et al. [30] presented a probabilistic performance analysis of a deadlock detection algorithm in distributed systems, and measured performance metrics, e.g., duration of deadlock and the number of algorithm invocations. It illustrated that the analytic estimates were nearly consistent with simulation results.

Compared with other approaches, *LCL* has the following advantages: 1) final results are deterministic in multiple out-degree scenarios; 2) each vertex only maintains its own information; 3) it can keep as many transactions alive as possible rather than killing them; 4) system cost is as low as possible.

VIII. Conclusion

In this paper, we proposed *LCL*, a fully distributed resource deadlock detection and resolution algorithm for a distributed database system. While motivated by the algorithm proposed by Mitchell and Merritt [11], *LCL* does not have any restriction on the number of resources that a transaction process can wait for at a time. The *LCL* approach has the following advantages:

1) Each vertex needs neither an overall view nor a local view of the system.
2) Each vertex needs only to send a fixed-length message (tens of bytes) to vertices immediately downstream from

it, i.e., those that have resources for which it is waiting during each round of detection and resolution.
3) Each vertex needs to add a fixed size of data (tens of bytes) for *LCL*.
4) Only one vertex in a cycle detects and resolves the deadlock, and the rollback of multiple transactions in a deadlock is thus avoided.
5) Every deadlock is detected and resolved.
6) Only real deadlocks are detected and resolved, and transactions that are not part of any cycle are not rolled back.

Our emulation experiments show that *LCL* outperforms the counterpart *M&M* in cases where both SQL and resources have an exponential distribution, where SQL has an exponential distribution and resources have a normal distribution, where SQL has a normal distribution and resources have an exponential distribution, and where both SQL and resources have a normal distribution. In addition, applications in production show that it is generally applicable and easy to implement in a distributed relational database, and our experiments in OceanBase also illustrate that *LCL* significantly outperforms *M&M*.

IX. Appendix

A. Detailed Proofs

1) Proof of Lemma IV.1:

Proof.

$\forall X \in scc_1, \forall Y \in scc_2,$
$\because X \in scc_1 \subseteq USG(scc_1) == USG(scc_2)$
$\therefore X \in USG(scc_2)$
$\because Y \in scc_2$
According to the definition of $USG(scc_2)$ (Definition II.6), $X \Rightarrow Y$ and $Y \Rightarrow X$. Therefore $X \Leftrightarrow Y$, so $scc_1 == scc_2$.

□

2) Proof of Lemma IV.4:

Proof.

$scc_2 \cap scc_1 == \emptyset$ (Definition IV.2)
$scc_2 \cap scc_2 == scc_2 \neq \emptyset$
$\therefore scc_2 \neq scc_1.$
$\therefore USG(scc_2) \neq USG(scc_1)$ (Lemma IV.1)
$\because scc_2 \subseteq USG(scc_1)$ (Definition IV.3)
$\therefore USG(scc_2) \subseteq USG(scc_1)$
$\therefore USG(scc_2) \subset USG(scc_1)$

□

3) Proof of Theorem IV.5:

Proof.

Suppose the opposite. Then, every SCC in *WFG* has at least one proper upstream SCC.
Let scc_1 be an SCC in *WFG* and let $USG(scc_1)$ contain n vertices.
Let scc_2 be a proper upstream SCC of scc_1. According to Lemma IV.4, $USG(scc_2) \subset USG(scc_1)$; therefore, $USG(scc_2)$ contains at most $n-1$ vertices.
Similarly, let scc_3 be a proper upstream SCC of scc_2; then, $USG(scc_3) \subset USG(scc_2)$, and $USG(scc_3)$ contains at most $n-2$ vertices.
.....
Let scc_n be a proper upstream SCC of $scc_{(n-1)}$; then, $USG(\ scc_n) \subset USG(scc_{(n-1)})$, and $USG(scc_n)$ contains at most one vertex. This is a contradiction because $scc_n \subseteq USG(scc_n)$ and scc_n contains at least two vertices.

□

4) Proof of Lemma IV.6:

Proof.

If $X_{m+1} == X_j$, $1 \leq j \leq m-1$, then $X_{m+1} \rightarrow X_m \rightarrow \cdots \rightarrow X_j$ $(== X_{m+1})$ is a cycle with at least two vertices. But scc cannot have any proper upstream SCC. Contradiction.

□

5) Proof of Lemma IV.7:

Proof.

Denote $\mathit{AsgDist}(Y, scc)$ by m.
If m == 0, the conclusion holds.
Let us consider m $\geq$ 1; then, there exists a walk $Y_m(== Y) \rightarrow Y_{m-1} \rightarrow \cdots \rightarrow Y_1 \rightarrow Y_0$, where Y_m(==Y), Y_{m-1}, ..., Y_1 are m distinct vertices in $ASG(scc)$ and $Y_0 \in scc$.
Since $X \in ASG(scc)$ and $X(\neq Y) \rightarrow Y$, then, according to Lemma IV.6, X, Y_m(==Y), Y_{m-1}, ..., Y_1 are $m+1$ distinct vertices in $ASG(scc)$. And X$\rightarrow Y_m$(==Y)$\rightarrow Y_{m-1} \rightarrow \cdots \rightarrow Y_1 \rightarrow Y_0$ implies that $\mathit{AsgDist}(X, scc) \geq m+1$.

□

6) Proof of Lemma IV.8:

Proof.

$\because X_0 \in scc,$
$\therefore \mathit{AsgDist}(X_0, scc) == 0.$
According to Lemma IV.7,
$\mathit{AsgDist}(X_1, scc) \geq \mathit{AsgDist}(X_0, scc) + 1 \geq 1,$
.....
$\mathit{AsgDist}(X_j, scc) \geq \mathit{AsgDist}(X_{j-1}, scc) + 1 \geq j,$
.....
$\mathit{AsgDist}(X_m, scc) \geq \mathit{AsgDist}(X_{m-1}, scc) + 1 \geq m,$
Since $\mathit{AsgDist}(X_m, scc) == m$, therefore,
$\mathit{AsgDist}(X_j, scc) == j$, for $j = 0, 1, 2, \cdots, m-1$.

□

7) Proof of Lemma IV.10:

Proof.

Let us assume that $ASG(scc)$ is nonempty, and let $m == \mathit{AsgWidth}(scc) > 0$. Then, $m \leq w$.
Let us define m sets:
S_j = $\{X : X \in ASG(scc)$ and $\mathit{AsgDist}(X, scc) == j\}$, for $j = 1, 2, \cdots, m$.
It is clear that these sets are all nonempty (Lemma IV.8), no pair among them intersects, and their union set is $ASG(scc)$.
Now, let us show that after the k-th round of the *LCL* proliferation deduction, the $LCLV$ of every vertex in S_m, S_{m-1}, ..., S_{m-k} is invariant, for $k = 0, 1, 2, \cdots, m-1$.
Let us consider round "0" and pay attention to S_m. $\forall X \in S_m$, since $\mathit{AsgDist}(X, scc)$ == m == $\mathit{AsgWidth}(scc)$, according

to Lemma IV.7, X cannot have any upstream vertex (other than X itself). Thus, X is a source vertex (a vertex with an indegree of zero). Therefore, $X.LCLV$ is invariant during the *LCL* proliferation stage.

Then, let us consider the k-th round of the *LCL* proliferation deduction, where $k = 1, 2, \cdots, m-1$.

Let us suppose that the previous $k-1$ rounds of *LCL* proliferation deductions have been completed. The $LCLV$ of every vertex in S_m, S_{m-1}, ..., $S_{m-(k-1)}$ is invariant.

During an *LCL* proliferation deduction, the alteration of the $LCLV$ of a vertex can occur only by an immediate upstream vertex. For any vertex Y in S_{m-k}, if $X \in ASG(scc)$ and $X(\neq Y) \rightarrow Y$, according to Lemma IV.7, $\mathit{AsgDist}(X, scc) \geq \mathit{AsgDist}(Y, scc) + 1 == m - k + 1 == m - (k-1)$. Therefore, X belongs to one of S_m, S_{m-1}, $\cdots$, $S_{m-(k-1)}$, which implies that $X.LCLV$ is invariant after $(k-1)$ rounds of the *LCL* proliferation deduction. So, after k-th round of the *LCL* proliferation deduction, $Y.LCLV$ is invariant.

So, after $m-1$ rounds of the *LCL* proliferation deduction, the $LCLV$ of every vertex in S_m, S_{m-1}, $\cdots$, S_1 is invariant. Furthermore, if X and $Y \in ASG(scc)$ and $X(\neq Y) \rightarrow Y$, because both $X.LCLV$ and $Y.LCLV$ are invariant, we must have $X.LCLV+1 \leq Y.LCLV$; otherwise, $Y.LCLV$ increases in the m-th round of the *LCL* proliferation deduction. Therefore, we have:

$$X.LCLV < Y.LCLV, \forall X \in ASG(scc) \text{ and } \forall Y \in ASG(scc) \text{ such that } X(\neq Y) \rightarrow Y. \quad (1)$$

We now attend to the m-th round of the *LCL* proliferation deduction: If $X \in ASG(scc)$ and $Y \in scc$, and $X(\neq Y) \rightarrow Y$, because $X.LCLV$ has already been invariant in the previous round of the *LCL* proliferation deduction, $X.LCLV + 1 \leq Y.LCLV$ after the m-th round of *LCL* proliferation deduction, which implies:

$$X.LCLV < Y.LCLV, \forall X \in ASG(scc) \text{ and } \forall Y \in scc \text{ such that } X(\neq Y) \rightarrow Y \quad (2)$$

According to the inequalities (1) and (2), (ii) holds. According to (i) and (ii), and Corollary IV.9, (iii) holds.

□

8) Proof of Theorem IV.11:

Proof.

$\max(w, 1) \geq 1$ guarantees that every $\langle PuAP, PuID \rangle$ is set to its initial value $\langle PrAP, \mathit{PrID} \rangle$ in the *LCL* proliferation stage even if $ASG(scc)$ is empty.

After the *LCL* proliferation stage, let $MAX\_LCLV = \max\{scc.LCLV\}$, and $MAX\_APID = \max\{scc.\langle PrAP, \mathit{PrID} \rangle\}$.

Assume $C \in scc$ and $C.LCLV == MAX\_LCLV$ and $B \in scc$ and $B.\langle PrAP, \mathit{PrID} \rangle == MAX\_APID$.

According to Lemma IV.10 (iii), $max\{ASG(scc).LCLV\} < max\{scc.LCLV\}$.

Therefore, $C.LCLV == MAX\_LCLV == max\{USG(scc).LCLV\}$

According to the definition of *LCL* spread in Definition III.10, $C.LCLV$, the maximum $LCLV$ value in $USG(scc)$, is invariant during the *LCL* spread stage. According to Lemma IV.10 (ii), the $\langle PuAP, PuID \rangle$ of any vertex in $ASG(scc)$ cannot alter the $\langle PuAP, PuID \rangle$ of any vertex in scc in an *LCL* spread deduction. Thus, $MAX\_APID == max\{scc.\langle PuAP, PuID \rangle\}$, and $B.\langle PuAP, PuID \rangle == MAX\_APID$. The maximum value of $\langle PuAP, PuID \rangle$ in scc is invariant during the *LCL* spread.

In the first d rounds of *LCL* spread deductions, since $C \in scc$, $\forall X \in scc$ $(X \neq C)$, then $C \Rightarrow X$. Let $m = Dist(C, X)$; then, $m \leq d$, and there is a walk of length m from C to X. Thus, along this walk, MAX_LCLV can be transmitted from C to X in m rounds of the *LCL* spread deduction. Therefore, after the first d rounds of *LCL* spread deductions,

$$X.LCLV == MAX\_LCLV, \forall X \in scc \quad (1)$$

During the second set of d rounds of *LCL* spread deductions, since $B \in scc$, $\forall Y \in scc$ $(Y \neq B)$, then $B \Rightarrow Y$. Let $n = Dist(B, Y)$. Then, $n \leq d$, and there is walk of length n from B to Y. Due to (1), MAX_APID can be transmitted from B to Y in at most n rounds of the *LCL* spread deduction along this walk. Therefore, after the second set of d rounds of *LCL* spread deductions,

$$Y.\langle PuAP, PuID \rangle == MAX\_APID, \forall Y \in scc \quad (2)$$

Thus, after $2 \times d$ rounds of *LCL* spread deductions,

$\because B \in scc$,

$\therefore \exists A \in scc$ such that $A(\neq B) \rightarrow B$. Therefore,

$A.LCLV == MAX\_LCLV == B.LCLV$ (equation (1)), and $A.\langle PuAP, PuID \rangle == MAX\_APID == B.\langle PuAP, PuID \rangle$ (equation (2)).

The uniqueness of B:

If $\exists X \in USG(scc)$, $\exists Y \in USG(scc)$, such that $X(\neq Y) \rightarrow Y$, and $Y.LCLV == X.LCLV$ and $Y.\langle PrAP, \mathit{PrID} \rangle == Y.\langle PuAP, PuID \rangle == X.\langle PuAP, PuID \rangle$.

If $X \in ASG(scc)$, according to Lemma IV.10 (ii), $X(\neq Y) \rightarrow Y$ implies $X.LCLV < Y.LCLV$. Thus, $X \in scc$, which implies $Y.LCLV == X.LCLV == MAX\_LCLV$; so, $Y \in scc$ (Lemma IV.10 (ii)). Thus, $Y.\langle PrAP, \mathit{PrID} \rangle == Y.\langle PuAP, PuID \rangle$ (assumption)

$Y.\langle PuAP, PuID \rangle == MAX\_APID$ (equation (2))

$B.\langle PrAP, \mathit{PrID} \rangle == MAX\_APID$ (assumption)

$\therefore Y.\mathit{PrID} == B.\mathit{PrID}$.

The uniqueness of *PrID* guarantees the uniqueness of B.

□

ACKNOWLEDGEMENT

We would like to thank the anonymous reviewers for their insightful comments, and our intern Haopeng Lu for his contribution in the distributed transaction processing emulator (TPE). We also would like to thank Professor Shan Wang and Professor Aoying Zhou for their insightful comments. The source code is available at https://github.com/oceanbase/LCL.

REFERENCES

[1] Zhenkun Yang, Chuanhui Yang, Fusheng Han, Mingqiang Zhuang, Bing Yang, Zhifeng Yang, Xiaojun Cheng, Yuzhong Zhao, Wenhui Shi, Huafeng Xi, Huang Yu, Bin Liu, Yi Pan, Boxue Yin, Junquan Chen, and Quanqing Xu. OceanBase: A 707 Million tpmC Distributed Relational Database System. *Proceedings of the VLDB Endowment*, 15(12):3385–3397, 2022.

[2] OceanBase. https://gitee.com/oceanbase, 2021.

[3] OceanBase. https://github.com/oceanbase, 2021.

[4] MulanPubL-2.0. https://license.coscl.org.cn/MulanPubL-2.0/index.html, 2021.

[5] TPC-C Result Highlights. http://tpc.org/1803, 2021.

[6] TPC-H V3 Result Highlights. http://tpc.org/3375, 2021.

[7] Philip A Bernstein, Vassos Hadzilacos, and Nathan Goodman. *Concurrency control and recovery in database systems*, volume 370. Addison-wesley Reading, 1987.

[8] K Mani Chandy, Jayadev Misra, and Laura M Haas. Distributed deadlock detection. *ACM Transactions on Computer Systems (TOCS)*, 1(2):144–156, 1983.

[9] Brian M Johnston, Ramesh Dutt Javagal, Ajoy Kumar Datta, and Sukumar Ghosh. A distributed algorithm for resource deadlock detection. In *Tenth Annual International Phoenix Conference on Computers and Communications*, pages 252–253. IEEE Computer Society, 1991.

[10] Edgar Knapp. Deadlock detection in distributed databases. *ACM Computing Surveys (CSUR)*, 19(4):303–328, 1987.

[11] Don P Mitchell and Michael J Merritt. A distributed algorithm for deadlock detection and resolution. In *Proceedings of the third annual ACM symposium on Principles of Distributed Computing*, pages 282–284, 1984.

[12] Eric Lehman, F Thomson Leighton, and Albert R Meyer. *Mathematics for Computer Science*, pages 317–327. 2015.

[13] Robert Tarjan. Depth-first search and linear graph algorithms. *SIAM journal on computing*, 1(2):146–160, 1972.

[14] Ron Obermarck. Distributed deadlock detection algorithm. *ACM Transactions on Database Systems (TODS)*, 7(2):187–208, 1982.

[15] Rebecca Taft, Irfan Sharif, Andrei Matei, Nathan VanBenschoten, Jordan Lewis, Tobias Grieger, Kai Niemi, Andy Woods, Anne Birzin, Raphael Poss, et al. Cockroachdb: The resilient geo-distributed sql database. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*, pages 1493–1509, 2020.

[16] Ahmed K Elmagarmid. A survey of distributed deadlock detection algorithms. *ACM Sigmod Record*, 15(3):37–45, 1986.

[17] Wilson Chan. Method and system for deadlock detection in a distributed environment, 2010. US Patent 7,735,089.

[18] Valmir Carneiro Barbosa, Alan Diêgo A. Carneiro, Fábio Protti, and Uéverton S. Souza. Deadlock models in distributed computation: foundations, design, and computational complexity. In Sascha Ossowski, editor, *Proceedings of the 31st Annual ACM Symposium on Applied Computing*, pages 538–541. ACM, 2016.

[19] Gabriel Bracha and Sam Toueg. Distributed deadlock detection. *Distributed Computing*, 2(3):127–138, 1987.

[20] Natalija Krivokapic, Alfons Kemper, and Ehud Gudes. Deadlock detection in distributed database systems: A new algorithm and a comparative performance analysis. *VLDB J.*, 8(2):79–100, 1999.

[21] Dongxu Huang, Qi Liu, Qiu Cui, Zhuhe Fang, Xiaoyu Ma, Fei Xu, Li Shen, Liu Tang, Yuxing Zhou, Menglong Huang, et al. TiDB: a Raft-based HTAP database. *Proceedings of the VLDB Endowment*, 13(12):3072–3084, 2020.

[22] David F Bacon, Nathan Bales, Nico Bruno, Brian F Cooper, Adam Dickinson, Andrew Fikes, Campbell Fraser, Andrey Gubarev, Milind Joshi, Eugene Kogan, et al. Spanner: Becoming a sql system. In *Proceedings of the 2017 ACM International Conference on Management of Data*, pages 331–343, 2017.

[23] R. Balter, P. Berard, and Paul Decitre. Why control of the concurrency level in distributed systems is more fundamental than deadlock management. In Robert L. Probert, Michael J. Fischer, and Nicola Santoro, editors, *ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing, Ottawa, CanadaAugust 18-20, 1982*, pages 183–193. ACM, 1982.

[24] Dong Young Yoon, Mosharaf Chowdhury, and Barzan Mozafari. Distributed lock management with RDMA: decentralization without starvation. In Gautam Das, Christopher M. Jermaine, and Philip A. Bernstein, editors, *Proceedings of the 2018 International Conference on Management of Data*, pages 1571–1586. ACM, 2018.

[25] Zhuolong Yu, Yiwen Zhang, Vladimir Braverman, Mosharaf Chowdhury, and Xin Jin. Netlock: Fast, centralized lock management using programmable switches. In Henning Schulzrinne and Vishal Misra, editors, *SIGCOMM '20*, pages 126–138. ACM, 2020.

[26] Ananth Devulapalli and Pete Wyckoff. Distributed queue-based locking using advanced network features. In *34th International Conference on Parallel Processing (ICPP 2005)*, pages 408–415. IEEE Computer Society, 2005.

[27] Sundeep Narravula, A. Marnidala, Abhinav Vishnu, Karthikeyan Vaidyanathan, and Dhabaleswar K. Panda. High performance distributed lock management services using network-based remote atomic operations. In *Seventh IEEE International Symposium on Cluster Computing and the Grid (CCGrid 2007), 14-17 May 2007, Rio de Janeiro, Brazil*, pages 583–590. IEEE Computer Society, 2007.

[28] Omran Bukhres and Kenneth Magel. Deadlock detection in distributed database systems: a performance evaluation study. In *1991 The Fifteenth Annual International Computer Software & Applications Conference*, pages 78–79. IEEE Computer Society, 1991.

[29] Mukesh Singhal. Deadlock detection in distributed systems. *Computer*, 22(11):37–48, 1989.

[30] Soojung Lee and Junguk L. Kim. Performance analysis of distributed deadlock detection algorithms. *IEEE Transactions on Knowledge and Data Engineering*, 13(4):623–636, 2001.