HAWK: A Workload-driven Hierarchical Deadlock Detection Approach in Distributed Database System

Rongrong Zhang
East China Normal University
rrzhang@stu.ecnu.edu.cn

Zhiwei Ye
China Mobile Cloud Center
yezhiwei@cmss.chinamobile.com

Jun-Peng Zhu
East China Normal University
zjp.dase@stu.ecnu.edu.cn

Peng Cai
East China Normal University
pcai@dase.ecnu.edu.cn

Xuan Zhou
East China Normal University
xzhou@dase.ecnu.edu.cn

Dunbo Cai
China Mobile Cloud Center
caidunbo@cmss.chinamobile.com

Ling Qian
China Mobile Cloud Center
qianling@cmss.chinamobile.com

ABSTRACT

Distributed databases are widely used in various fields, such as financial services and e-commerce. These businesses generally exhibit characteristics of large-scale and rapid growth. However, these business systems often suffer from deadlocks that prevent them from operating normally for extended periods. Traditional deadlock detection methods face challenges in scalability and efficiency, especially as the number of nodes increases. Therefore, deadlock detection has always been a research area in distributed databases.

In this paper, we introduce an efficient deadlock detection algorithm called HAWK, leveraging a **H**ierarchical **A**pproach based on **W**or**K**load modeling. Our algorithm addresses these issues by constructing a dynamic hierarchical detection tree that adapts to transaction patterns, significantly reducing time complexity and communication overhead. HAWK first models the workload and generates a *predicted access graph* (PAG), transforming the problem of partitioning detection task in the basic hierarchical detection into partition detection zone (DZ) in the PAG by a graph-cutting algorithm. Then, leveraging the properties of strongly connected components (SCCs) and deadlock cycles, the SCC-cut algorithm naturally partitions the system-wide deadlock detection into multiple non-intersecting detection zones, thereby enhancing detection efficiency. We used the greedy SCC-cut algorithm to perform a more fine-grained partitioning of the complex PAG. Finally, by periodically sampling and updating the hierarchical structure, the algorithm remains responsive to dynamic workload variations, ensuring efficient detection. Our approach outperforms both centralized and distributed methods, offering a more efficient and adaptive solution. Extensive experimental results demonstrate the effectiveness of the HAWK algorithm, showing significant reductions in the duration of the deadlock and improved system throughput.

Proceedings of the VLDB Endowment, Vol. 18, No. 10 ISSN 2150-8097.
doi:10.14778/3748191.3748224

PVLDB Reference Format:
Rongrong Zhang, Zhiwei Ye, Jun-Peng Zhu, Peng Cai, Xuan Zhou, Dunbo Cai, and Ling Qian. HAWK: A Workload-driven Hierarchical Deadlock Detection Approach in Distributed Database System. PVLDB, 18(10): 3682-3694, 2025.
doi:10.14778/3748191.3748224

PVLDB Artifact Availability:
The source code, data, and/or other artifacts have been made available at https://github.com/rrzhang23/HAWK_txn.

1 INTRODUCTION

The two-phase locking (2PL) [9] protocol ensures isolation and consistency during the execution of concurrent transactions. Due to its flexibility and stability, 2PL has been widely integrated into many open-source and commercial database systems, such as Oracle [12], Spanner [3], MySQL [33], and Postgres [1]. Despite these advantages, 2PL can lead to performance decreases due to deadlock problems. Although deadlock detection and resolution are well-defined problems with extensive research in standalone systems, they have recently received increased attention in distributed systems [5, 21, 38, 44].

Existing deadlock detection algorithms can be classified into three types: centralized, distributed, and hierarchical. The modern distributed database (e.g., OceanBase [45]) can be deployed on more than a thousand nodes to provide extremely high-throughput performance, according to the TPC-C performance report [2]. Although both centralized and distributed perform well in small-scale clusters, their detection efficiency significantly declines in large-scale clusters (e.g., more than 100 nodes), especially when employing a multiple out-degree transaction model. Specifically, (1) centralized detection algorithms suffer from single-point bottlenecks in both network and computation at the detection node. (2) Although studies have shown that deadlock cycle length tends to be skewed [4, 6, 27], with most cycles being of length two, cycles involving many nodes can still occur. Distributed detection algorithms rely on serial message passing, which can take a large number of rounds to detect such deadlocks.

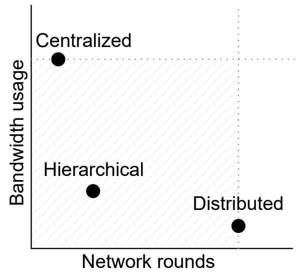


Figure 1: Performance trade-off between three deadlock detection algorithms.

As a compromise, to better adapt to large-scale clusters, hierarchical detection algorithms [23, 30, 36] were proposed to reduce the load on a single node in centralized detection and reduce the network overhead of distributed detection algorithms. The hierarchical detection algorithm partitions the centralized detection process into multiple detection zones (DZs), first detects deadlocks within the DZs, and then constructs a hierarchical detection tree to detect deadlocks across DZs. Figure 1 illustrates the trade-offs among the three algorithms in terms of the single node's bandwidth and the number of message transmission rounds. Existing hierarchical algorithms still face several challenges: (a) they provide a detection process based on a hierarchical tree, but lack an effective method for constructing the tree. The assumption of static partitioning is almost nonexistent, as node failures, scaling in or out, and changes in application access patterns all contribute to the dynamic nature of the workload, resulting in: (b) the fixed hierarchical tree structure does not align with the workload partitioning pattern, causing many deadlocks to be detected only at the root node, degenerating the algorithm into a centralized one, and (c) the static hierarchical tree cannot adapt to workload changes and fails to align with the workload partitioning pattern at runtime dynamically.

In this paper, we present HAWK, a *hierarchical approach based on workload modeling*, to detect deadlocks in large-scale clusters and dynamic workloads. First, different from the basic hierarchical algorithm, HAWK models the workload as a graph, referred to as the predicted access graph (PAG). The output of modeling is a set of DZs, which are generated by performing partitioning nodes in PAG using the graph-cut algorithm, where each DZ potentially involves deadlock cycles. The biggest difference between hierarchical algorithms and centralized algorithms is that hierarchical algorithms partition detection tasks on wait-for graph (WFG). Through modeling, we can partition nodes before WFG is generated, thereby transforming partition detection tasks into partition DZ by applying graph-cutting algorithms on PAG.

The PAG is a set consisting of database nodes (vertices) and inter-node resource request dependencies (edges). The WFG, on the other hand, is a set of transactions and resource request dependencies. Since we are concerned with distributed deadlock detection, we define the local WFG of a node as a compressed graph, *Cprs*(WFG). Through reasoning, we demonstrate that, under constant workload, *Cprs*(WFG) is exactly a subgraph of the PAG. In other words, if a distributed deadlock exists within *Cprs*(WFG), it will also exist in the PAG. Thus, detecting distributed deadlocks in the WFG can be transformed into detecting them in the PAG.

Second, we provide a specific method for partitioning DZs, which uses an SCC-cut graph-cut algorithm. This algorithm identifies the strongly connected components (SCCs) of a graph. The properties of SCC ensure that no cycle can span across two different SCCs, so each SCC naturally corresponds to a detection zone. Although partitioning detection zones improves the efficiency of deadlock detection algorithms in large-scale systems, not all PAGs can be partitioned into multiple uniform and sufficiently small DZs using a simple SCC-cut algorithm. The multiple out-degree model can lead to a more complex workload, causing the PAG itself to be a single large SCC, which leads the hierarchical detection algorithm to degrade into a centralized detection approach. To address this issue, we propose an optimized, fine-grained SCC-cut algorithm, which aims to decompose a single large SCC into as many smaller SCCs as possible while using a threshold $\mathcal{ST}$ to prevent overly fragmented cutting results.

Third, the DZs obtained from the fine-grained SCC-cut algorithm not only have the potential for deadlocks within them, but deadlocks may also exist between them. Therefore, simply performing deadlock detection within each DZ is insufficient. We design a dynamically constructed hierarchical tree to guide the hierarchical detection process based on the graph-cut results. The hierarchical algorithm detects deadlocks from the bottom up along the hierarchical tree, detecting deadlocks within each zone and then across zones until reaching the root node. A well-structured hierarchical tree ensures that most deadlocks are detected near the leaf nodes. However, changes in workload can lead to inefficiencies in detection, increasing the detection load on the root node. Therefore, the tree structure must be adjusted to adapt to the workload changes. We determine the timing of tree reconstruction based on the proportion of deadlocks detected at the root and lower layers.

Summarizing, the major contributions of our work are:

- *Efficient workload modeling*, through careful reasoning, effectively transforms the task of partition detection in WFG into partition DZs in PAG.
- *Accurate detection zone partitioning* based on the graph-cut algorithm to improve the accuracy of partition detection zones for the basic hierarchical algorithm.
- *Dynamic hierarchical tree construction* enhances the flexibility of the hierarchical detection algorithm, enabling it to better adapt to changing workloads.
- *Targeted optimization strategies* improve the efficiency of subgraph transmission and reduce the false positive rate in deadlock resolution during the detection process.

The remainder of this paper is organized as follows. We provide our motivation in Section 2, and then present the core ideas of our work in Section 3. In Section 4, we detail the optimizations made to the deadlock detection algorithm implementation. We show the experiments in Section 5, introduce related work in Section 6, and conclude the paper in Section 7.

2 MOTIVATION

In this section, we first provide the definition and terminology of the wait-for graph (WFG). Then, we present the price of centralized

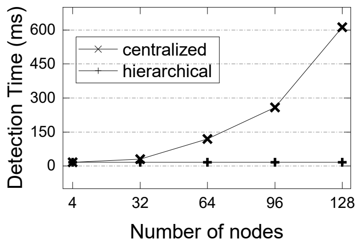


Figure 2: Illustration of deadlock detection on partitioned workloads.

algorithms in a cluster with over a hundred nodes. Finally, we use examples to demonstrate how the basic hierarchical detection algorithm operates and highlight its limitations.

2.1 Wait-for Graph

Deadlocks are accurately described using a directed graph called the *wait-for graph* (WFG). The WFG is represented as $G = (V, E)$, where V is the set of vertices and E is the set of edges. Each edge $(v_i, v_j) \in E$ represents a transaction T_i waiting for a resource currently held by another transaction T_j. The relationship is shown as $T_i \xrightarrow{D} T_j$ where D is the resource held by T_j, abbreviated as $T_i \rightarrow T_j$. An edge (v_i, v_j) is added into the wait-for graph when the transaction T_i is blocked by the transaction T_j, meaning that T_j holds the data item requested by T_i. An edge is removed from the wait-for graph only when transaction T_j no longer holds the data items required by transaction T_i. A path in the graph is denoted as $P = (V', E') \subseteq G$, where $V' = \{v_0, v_1, ..., v_k\}$ and $E' = \{(v_0, v_1), ..., (v_{k-1}, v_k)\}$. The path forms a cycle C if there is an edge from v_k to v_0. This cycle indicates a deadlock among certain transactions in the system. We simply denote this cycle by $C = (v_0, v_1, v_k, v_{k+1} \equiv v_0)$.

2.2 The Price of Centralized Algorithm

In distributed systems, each node N_i maintains a *local transaction wait-for graph* (LWFG), denoted as G_i, which can only detect cycles in the local. To detect deadlock cycles that cross nodes, it is necessary to construct a *global wait-for graph* (GWFG). In a system comprising N nodes, the GWFG is represented as $G_{global} = \bigcup_{i=1}^{N} G_i$. This centralized deadlock detection approach is widely used in commercial databases such as TiDB [24], Greenplum [29] and MySQL [33]. The algorithm merges LWFGs from multiple nodes into a GWFG and subsequently identifies cycles within the GWFG to detect global deadlocks. This algorithm effectively detects deadlocks with a few nodes, but its efficiency declines as the number of nodes increases. We demonstrate this with example 2.1 in Figure 2.

Example 2.1. In the sales business scenario like TPC-C, workloads are usually partitioned according to warehouses to improve the efficiency of ordering and payment transactions [17, 22]. Suppose that the workload is well partitioned, and we can group the database nodes according to the partitioned workloads. This means that the transaction on a node accesses only the resources within the same node group without cross-group resource requests. The detection zones in the hierarchical algorithm can perfectly match the workload partition (i.e., in such a TPC-C type workload, PAG can be well cut into multiple DZs in the algorithm we proposed in Section 3).

We ran this experiment on 128 nodes, each workload partition running on 4 nodes. The centralized algorithm only uses a single detection node, while the hierarchical algorithm has one detection node per partition. The results are shown in Figure 2. The centralized detection algorithm does not maintain constant detection efficiency as the hierarchical algorithm does due to workload partitioning, as there are network and computational bottlenecks for a single detection node.

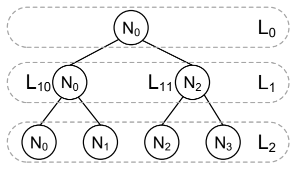


Figure 3: A basic hierarchical detection tree. The tree has three levels, L_0, L_1 and L_2. Nodes are partitioned into two DZs, with detecting nodes $L_{10} = N_0$ and $L_{11} = N_2$. N_0 is also responsible for detecting on a global scope at level L_0.

2.3 Basic Hierarchical Deadlock Detection

To address the performance degradation of centralized deadlock detection in systems with a large number of nodes, Özsu et al. introduced a hierarchical detection algorithm [36]. It utilizes the divide-and-conquer approach, partitioning the nodes into multiple detection zones (DZs), with one node in each zone designated to perform deadlock detection independently on WFG G_{zone}. Each zone prunes its detected G_{zone} before merging them all into a single GWFG for the final global deadlock detection due to the possibility of cross-zone transaction dependencies. If the number of nodes in a DZ remains too large, the DZ can be further partitioned. The entire detection process can be modeled using a tree structure, referred to as the hierarchical detection tree. The leaves represent individual database nodes, intermediate levels correspond to the detection nodes within each zone, and the root node is responsible for system-wide deadlock detection. Figure 3 shows the partitioning of DZs by node number and the construction of a complete binary tree responsible for the entire detection process.

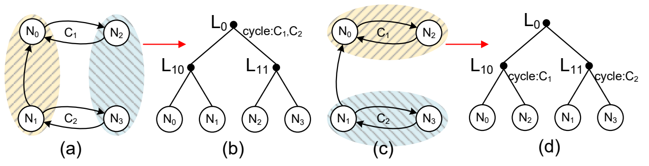


Figure 4: Two different DZ partitioning methods and the corresponding hierarchical detection tree.

2.4 Challenges of Hierarchical Detection

In Figure 4, four transactions $\{T_0, T_1, T_2, T_3\}$ are given, each running on a node N_i, and they are deadlocked due to resource dependencies. The vertices of the WFG are $\{N_0, N_1, N_2, N_3\}$, and edges are

$\{(N_0, N_2), (N_2, N_0), (N_1, N_0), (N_1, N_3), (N_3, N_1)\}$. There are two cycles present in this graph: $C_1 = (N_0, N_2, N_0)$ and $C_2 = (N_1, N_3, N_1)$.

Example 2.2. We first partition the nodes using the complete binary tree construction method of basic hierarchical detection algorithm, as shown in Figure 4(a). The nodes are partitioned into two zones: $\{N_0, N_1\}$ and $\{N_2, N_3\}$, and the constructed hierarchical tree is shown in Figure 4(b). While this method partitions nodes into multiple zones and performs deadlock detection independently within each zone, it is not the optimal solution. As shown in Figure 4(b), the detection nodes at level L_1 can not detect the existing deadlocks. Deadlocks C_1 an C_2 can only be detected at the level L_0. Compared to centralized detection, this hierarchical tree construction does not reduce the network transmission or the computation of detecting deadlocks at the node of L_0.

The *Example 2.2* reveals a significant issue with the basic hierarchical detection algorithm:

Issue 1: The fixed grouping method, based on node number order, does not align with the deadlock cycles in the WFG. More specifically, it does not match the workload patterns.

Workload assumptions and properties. In this work, our assumption is that the nodes participating in deadlock scenarios demonstrate behavior patterns that correlate with the observed workload distribution. By partitioning the accessing graph observed in the workloads, the nodes are grouped into different DZs, and deadlock detection can be effectively performed in each zone.

Example 2.3. An optimized example is illustrated in Figures 4(c) and 4(d). Assuming the locations of the cycles are known in advance, the nodes can be partitioned into the following DZs: $zone_1 = \{N_0, N_2\}$ and $zone_2 = \{N_1, N_3\}$. With this construction of the hierarchical tree, deadlocks C_1 and C_2 can be detected directly at level L_1. After pruning by the detection nodes at L_{10} and L_{11} the root node at L_0 no longer needs to process the remaining graph, including subgraph transmission and cycle detection.

Another issue with the *Example 2.3* is exposed:

Issue 2: It relies on the unrealistic assumption that exact deadlock information can be obtained before the detection process begins.

Workloads are rarely statically partitioned and unchanging. For instance, during e-commerce promotion events, warehouses in certain regions may experience being out of stock, causing new orders to be temporarily redirected to remote warehouses. On the other hand, when a database node fails or undergoes expansion/contraction due to application demands, it triggers data migration within the shard and alters the existing partitioning scheme. In addition, in some applications, a large fraction of transactions are cross-partitioned due to the lack of an effective static partitioning strategy. The dynamic workload makes it challenging for a statically constructed hierarchical detection tree to sustain high detection efficiency over time.

Issue 3: A statically constructed hierarchical detection tree cannot effectively adapt to dynamic workload variations.

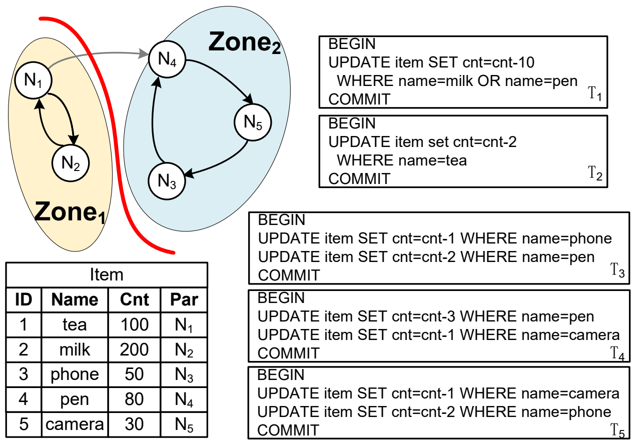
To address these issues, we need an efficient and adaptive method for constructing the hierarchical tree. The goal is to detect as many deadlocks as possible according to the workload access pattern, enabling quick detection and resolution, while ensuring the constructed hierarchical tree adapts to dynamic workloads.

3 EFFICIENT DEADLOCK DETECTION

In this section, we propose HAWK, an efficient and dynamic hierarchical deadlock detection algorithm. The key to hierarchical detection is partitioning nodes into multiple DZs, where each DZ contains a WFG_{zone} with deadlocks. We first provide the workflow, followed by the implementation of each step.

3.1 Workflow

The workflow of HAWK includes the following: First, we sample the workload through a short-term cross-node transaction access pattern and generate a graph PAG as input in Section 3.2. The output is a set of DZs, we cut the PAG by applying the graph-cutting algorithm SCC-cut in Section 3.3 and fine-grained SCC-cut in Section 3.4 to obtain it. Then, based on the obtained DZs, we construct a hierarchical detection tree in Section 3.5.1, and deadlock detection follows the hierarchical detection tree from bottom to top. Finally, we calculate the ratio of deadlocks detected at the root node and other detection nodes to determine whether the tree structure needs reconstruction to adapt to changing workloads in Section 3.5.2.



```
BEGIN
UPDATE item SET cnt=cnt-10
  WHERE name=milk OR name=pen
COMMIT
```

```
BEGIN
UPDATE item set cnt=cnt-2
  WHERE name=tea
COMMIT
```

```
BEGIN
UPDATE item SET cnt=cnt-1 WHERE name=phone
UPDATE item SET cnt=cnt-2 WHERE name=pen
COMMIT
```

```
BEGIN
UPDATE item SET cnt=cnt-3 WHERE name=pen
UPDATE item SET cnt=cnt-1 WHERE name=camera
COMMIT
```

```
BEGIN
UPDATE item SET cnt=cnt-1 WHERE name=camera
UPDATE item SET cnt=cnt-2 WHERE name=phone
COMMIT
```

Item

ID	Name	Cnt	Par
1	tea	100	N_1
2	milk	200	N_2
3	phone	50	N_3
4	pen	80	N_4
5	camera	30	N_5

Figure 5: Generation of the PAG. Each node N_i processes a transaction T_i. After all the first statements are executed, T_3, T_4 and T_5 execute their second statements, resulting in a distributed deadlock. According to the transaction access pattern, the PAG can be partitioned into two DZs.

3.2 Workload Modeling

We aim to partition the nodes involved in potential deadlocks into detection zones by analyzing the workload's access patterns rather than relying on the lagging WFG. Specifically, we first model the workload as a graph PAG.

We use an example to introduce the PAG representation. Although our example only uses a single table, our approach works with any schema and is independent of the complexity of the SQL statements in the workload. Modeling the workload has been validated as feasible by several studies [17, 37].

Example 3.1. Assume we have a database containing a single stock table with five tuples, and a workload consisting of five transactions, as shown in Figure 5. We focus on cross-node transaction resource requests, where each vertex in the figure represents a node rather than an individual transaction. An edge represents a cross-node resource request. The final resource access graph is shown in the upper-left corner of the figure.

The PAG is similar to WFG but not the same. There are two main differences between the two. Firstly, in WFG, the vertices represent transactions, whereas in PAG, the vertices represent database nodes. Secondly, the edges represent the waiting relationship between transactions during execution in WFG, while representing a cross-node resource requesting in PAG. Even if a cycle exists in a PAG, it does not necessarily indicate a deadlock, as deadlock detection requires further verification in the WFG. For instance, edges are pointing towards each other between nodes N_1 and N_2 in Figure 5, there is no resource access conflict between T_1 and T_2. Therefore there is no deadlock. A deadlock only exists when transactions T_3, T_4 and T_5 form a cycle in both the PAG and the WFG.

Despite the differences between PAG and WFG, we can still extract useful information from PAG for deadlock detection. In real-world scenarios, a single node may run multiple transactions, leading to a one-to-many relationship between nodes and transactions. Given that local and global deadlocks are independent of each other, we focus on the impact of inter-node dependencies that contribute to global deadlock formation. Specifically, we compress transactions within a single node and represent them with a single vertex. This process is referred to as compression, where $Cprs(G_i) = N_i$, and the resulting compressed WFG can be represented as $Cprs(WFG)$. The compressed WFG is a subgraph of the PAG when the workload remains unchanged. Therefore, the task of *partitioning nodes into DZs according to the WFG* can be transformed into the problem of *cutting the graph PAG*. To formally support this transformation, we present several lemmas and their proofs.

Definition 3.1. $Node(T)$, a vertex in $Cprs(WFG)$. The management node of a transaction, where a transaction can only be managed by one node.

The edges in WFG are denoted as (T_i, T_j). The edges in $Cprs(WFG)$ are denoted as $(Node(T_i), Node(T_j))$. The edges in PAG are denoted as (N_i, N_j).

Lemma 3.2. *If there exists an edge (T_i, T_j) in WFG, where $Node(T_i) \neq Node(T_j)$, then there is also an edge $(Node(T_i), Node(T_j))$ in the graph $Cprs(WFG)$. That is, $(T_i, T_j) \leftrightarrow (Node(T_i), Node(T_j)$. And vice versa.*

Proof of Lemma 3.2. Since $Cprs(WFG)$ does not affect the dependencies between transactions across nodes, this reasoning holds. □

Lemma 3.3. *If there exists an edge (T_i, T_j) in WFG, where $Node(T_i) \neq Node(T_j)$, then there is also an edge (N_i, N_j) $(T_i \in N_i, T_j \in N_j)$ in the graph PAG. That is, $(T_i, T_j) \rightarrow (N_i, N_j)$.*

Proof of Lemma 3.3. If transaction T_i depends on T_j, there must have been a request sent from T_i that was blocked by T_j before the dependency formed. □

Lemma 3.4. *According to Lemma 3.2 and 3.3, we obtain $(T_i, T_j) \leftrightarrow (Node(T_i), Node(T_j)) \rightarrow (N_i, N_j)$.*

Lemma 3.5. *If there is an edge $(N_i, N_j)(i \neq j)$ in PAG, we cannot guarantee the existence of an edge between two transactions T_i and T_j, where $T_i \in N_i$ and $T_j \in N_j$. That is, $(N_i, N_j) \nrightarrow (T_i, T_j) \leftrightarrow (Node(T_i), Node(T_j))$.*

Proof of Lemma 3.5. In Figure 5, there exist (N_1, N_2) and (N_2, N_1), but (T_1, T_2) and (T_2, T_1) do not exist. □

According to Lemma 3.4 and 3.5, we conclude that a dependency (or edge) in WFG is a sufficient but not necessary condition for the existence of an edge in the PAG. This leads to $Cprs(WFG) \subseteq PAG$. The goal of distributed deadlock detection is to search for cycles in *Cprs*(WFG), which is a subgraph of PAG. Therefore, the distributed deadlock detection problem can be summarized as the task of detecting cycles in the PAG.

3.3 Cut PAG based on SCC-cut Algorithm

To construct an efficient and accurate hierarchical detection tree, the first step is to partition the leaf nodes into DZs. Assuming there exists a cut Cut such that $PAG = PAG_1 \cup PAG_2 \cup \ldots \cup PAG_n$, where cycles only exist within each PAG_i, then the vertex set $Vertex(PAG_i)$ represents the *detection zone* we aim to identify. From Section 3.2, we know that although such a Cut cannot be obtained from the WFG as in Example 2.3, the Cut obtained by cutting the PAG can be considered equivalent.

After modeling the PAG, it becomes straightforward to partition the nodes into multiple detection zones using graph-cutting methods such as the SCC-cut algorithm. For example, in Figure 5, the generated PAG is partitioned into two DZs by cut edge (N_1, N_4): $zone_1 = \{N_1, N_2\}$ and $zone_2 = \{N_3, N_4, N_5\}$. Although $zone_1$ contains a cycle (N_1, N_2, N_1) in PAG, except for unnecessary detections, it does not affect the correctness of the detection results. Section 3.5.2 provides a solution for fine-tuning the tree structure when there are no potential deadlock cycles within certain DZs.

The intuition behind our algorithm is that every deadlock cycle must be contained within a strongly connected component (SCC) of the graph PAG, where $SCC(u)$ represents the SCC containing vertex u. We partition the PAG by identifying SCCs [44], which have 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 are vertices $u \in scc_1$ and $v \in scc_2$, and there is a path between u and v such that each can reach the other [28], then $scc_1 == scc_2$.
(3) If there is a path between u and v such that each can reach the other, then $SCC(u) == SCC(v)$.

These properties show that a cycle cannot span across two different SCCs. Therefore, detecting distributed deadlocks within each SCC is sufficient, and inter-SCC detection is not required. For a cycle C, $SCC(C)$ represents the strongly connected components containing C. Algorithms such as Tarjan [43], Kosaraju [40], and Gabow [20] can be used to identify SCCs. We use Tarjan's algorithm to find SCCs. The time complexity of this algorithm is $O(|V| + |E|)$, where $|V|$ is the number of vertices and $|E|$ is the number of edges in the graph. Even in a database system with over one hundred nodes, the time consumption of these algorithms is negligible.

Algorithm 1: GreedySccCutGraph(G)

```
Input: Directed graph $G$, maximum threshold of vertices $ST$
Output: $SCC_{list}[]$, a list of small $SCCs$
1  $SCC_{list}[] \leftarrow \emptyset$
2  $G_{SCC} \leftarrow$ StronglyConnectedComponents($G$)
3  Function Greedy($G_{SCC}$)
4      $SCCs \leftarrow$ MaxCut($G_{SCC}$)
5      for $S \in SCCs$ do
6          if Sizeof($S$) <= $ST$ then
7              $SCC_{list}[] \leftarrow SCC_{list}[] \cup S$
8              continue
9          end
10         Greedy($S$)
11     end
12 Function MaxCut($SCC$)
13     $max\_cnt \leftarrow 0$
14     $SCCs \leftarrow \emptyset$
15     for $e \in SCC$ do
16         $cnt, SCC\_set \leftarrow$
             StronglyConnectedComponents($SCC \setminus e$)
17         if $max\_cnt < cnt$ then
18             $max\_cnt \leftarrow cnt$
19             $SCCs \leftarrow SCC\_set$
20         end
21     end
22     return $SCCs$
```

If the workload is properly partitioned, the cutting results will create DZs of approximately equal size, with each zone handling no more than a predetermined threshold of nodes for detection. As a result, the height of the final hierarchical tree remains relatively low. For example, in the 128-node system from Example 2.1, a hierarchical tree with a height of 3 is sufficient to effectively reduce the deadlock duration to a desired time. At this point, all nodes will be detected within their respective DZs, e.g., level L_1, and no deadlocks will be detected at the root node. Nevertheless, we retain L_0, as it is responsible not only for deadlock detection but also for workload modeling and synchronization of the deadlock process(see Section3.5).

3.4 A Fine-grained SCC-cut Algorithm

Although partitioning the detection algorithm into multiple DZs based on the PAG is sufficient for most pre-partitioned workloads, some workloads are dynamic and hard to partition. Moreover, the multiple out-degree transaction models increase the complexity of access patterns, making it impossible to partition the PAG into sufficiently small SCCs when applying the basic SCC-cut algorithm. In extreme cases, the PAG itself becomes a single SCC. As a result, hierarchical deadlock detection degenerates into a centralized one.

We attempt to use a fine-grained greedy algorithm to remove certain edges in the PAG so that an oversized SCC is split into multiple sufficiently small SCC_i. The definition of "sufficiently small" here is that, given a threshold $\mathcal{ST}$, the graph-cutting process stops once the number of vertices in the SCC is less than it. While this approach cannot guarantee that the final height of the constructed hierarchical detection tree remains low and some deadlock detection tasks may still fall to the root node, our algorithm minimizes the network and computational burden on the root node as much as possible. For a graph with $|V|$ nodes and $|E|$ edges, this can be done in $O(|V| + |E|)$ time using Tarjan's algorithm.

Algorithm 1 shows the pseudo-code of the procedure. We begin by trimming the graph into an ultra-large SCC (line 2). The function Greedy then calls MaxCut to cut SCC into zones containing the most sub-SCCs (line 4). Greedy recursively processes each sub-SCC created by MaxCut (line 5-11). The function MaxCut attempts to cut only one edge of SCC at a time (line 15-21) and records the maximum number of sub SCCs generated after each cut (line 17-20). MaxCut returns the result corresponding to the maximum number of cuts (line 22). The termination condition for the function Greedy is that if the number of vertices in any SCC obtained is less than $\mathcal{ST}$, that SCC is merged into the output, and the recursion ends. Trimming and updating the graph after removing an edge takes $O(|E|)$ in total since each edge can be removed only once. In the worst case, i.e., such as in a fully connected graph, only one edge may be removed per iteration. Since finding SCC takes $O(|V| + |E|)$ per iteration, the overall time complexity of this algorithm is $O(|E|(|V| + |E|))$.

Example 3.2. Figure 6 illustrates an example of the Greedy algorithm that aims to reduce the number of vertices in the SCCs after cutting. The initial graph is a strongly connected component composed of 9 nodes and 14 edges (Figure 6(a)). The function MaxCut first attempts to cut each edge to maximize the number of resulting subgraphs in $G \setminus e$. The cuts made by edges e_3 and e_4 have the same effect, both cutting the graph into up to two distinct sub-SCCs (the shaded part in Figure 6(b)). We recursively apply this process until the preset termination condition is reached (in this example, $\mathcal{ST} = 2$, ensuring that each subgraph is an SCC containing more than one node). The final result, shown in Figure 6(c), is obtained by cutting the edges $\{e_3, e_5\}$ and trimming the edges $\{e_4, e_{13}\}$.

3.5 Tree Construction and Reconstruction

Compared to the basic SCC-cut algorithm, the fine-grained SCC-cut algorithm divides nodes into finer partitions, avoiding the degradation of hierarchical deadlock detection into a centralized one. But it also introduces cross-DZ detection, as shown in Figure 6, although the algorithm produces three detection zones: $DZ_1 = \{N_1, N_2\}$, $DZ_2 = \{N_3, N_6, N_7\}$, and $DZ_3 = \{N_4, N_5, N_8, N_9\}$, deadlocks may still exist across zones. For example, a deadlocks may occur along edges e_2, e_3, e_5 and e_4, crossing DZ_1, DZ_2 and DZ_3. Therefore, further cross-DZ deadlock detection is necessary.

3.5.1 Tree Construction. Sampling information includes not only cross node resource access, but also the resource utilization rate (e.g., network) of each machine. Deadlock detection first runs separately within each DZ and is the responsibility of the node (detection node) with the lowest resource utilization rate within the DZ. The detection node first collects the local WFG of each node to form a WFG_{zone} and then detects deadlocks within DZ. Then, regarding the example in Figure 6, cross-node deadlocks may still exist, and deadlock detection requires a second round. The node with

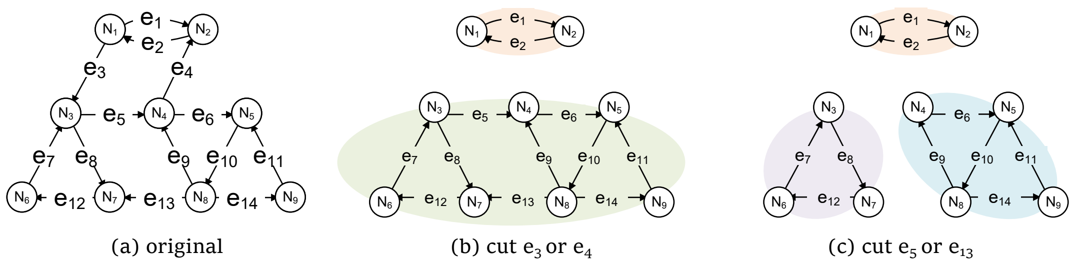


Figure 6: An example of the execution process of the greedy SCC-Cut algorithm.

the lowest resource utilization rate among all detection nodes is responsible for this round of detection. It collects WFG_{zone} from each detection node that has resolved deadlocks and pruned to form WFG_{global} for final detection. This process can be organized into a tree structure as shown in Figure 7(a), which is referred to as the hierarchical detection tree, where leaf nodes are transaction execution nodes. After the detection of the root node is completed, it immediately sends a signal to the detection nodes within each DZ for the next detection.

A hierarchical detection tree is constructed to detect deadlocks across DZs, and the efficiency of the tree is closely related to its height H. A high tree causes cross-DZ deadlocks that require multiple detections to be identified, wasting system resources. The height of the tree H is closely related to the number of branches B. When the number of nodes is N, the tree height is approximately $log_B N$. Therefore, without affecting the efficiency of detection performance, make B as large as possible.

The value of B is hard to determine. From another perspective, in Figure 2, there is a curve between the detection time and the number of nodes using a centralized algorithm. Therefore, a single node has an upper detection performance limit under specific system configurations. In other words, the average detection time $\overline{dt}$ is related to the number of nodes in the DZ that a single detection node is responsible for. Taking Figure 2 as an example, assuming that the expected $\overline{dt}$ does not exceed 50 ms, the maximum size of a single DZ is determined to be 32, that is, given $N = 128$, the tree height can be controlled with 3.

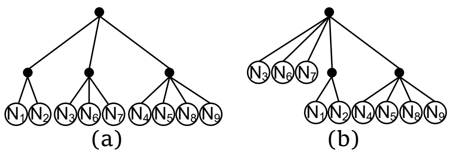


Figure 7: An example of tree structure fine-tuning.

3.5.2 Tree Reconstruction. The hierarchical tree structure can be dynamically adjusted according to changes in workload. The frequency of changes in workload is difficult to predict, so we cannot set a fixed period for tree reconstruction. We calculate the ratio α of the count of deadlocks detected by the root node cr and the total other detection nodes c in the past period (5 seconds in our experiment) to determine whether the tree needs to be reconstructed. When the workload changes and HAWK is difficult to run efficiently, most deadlocks are delayed until detected at the root node, resulting in a significant increase in the ratio of $\alpha = cr/c$.

We take $\alpha > 1$ as the condition for tree reconstruction. Once this condition is met, we resample and reconstruct the tree. When the tree structure matches the workload pattern, most deadlocks are detected within DZ, where α is significantly less than 1. However, when the tree structure severely does not match the workload pattern, most deadlocks can only be detected at the root node, where α is significantly greater than 1. In addition, we also fine-tune the tree because deadlocks do not always generate within a certain DZ partitioned by the greedy SCC-cut algorithm. When almost no detection occurs in a DZ, we raise the level of the DZ in the tree. Assuming that the number of deadlocks detected during a certain period (empirically set 15 seconds in our experiment) in $DZ_2 = \{N_3, N_6, N_7\}$ of Figure 6 is 0, we fine tune the tree from Figure 7(a) to Figure 7(b), as long as the number of nodes responsible by the upper level nodes is less than B.

3.6 Correctness

3.6.1 Safety. Transactions that are deadlocked will not be broken by timeout mechanisms or client access, aside from being aborted by the detection algorithm. Transactions are only aborted if there is a cycle in the WFG, ensuring that non-existent deadlocks are not detected. Each detection cycle is assigned a term and synchronized across nodes. During each detection, the $term$ of the transaction dependency list is checked, ensuring that no false deadlocks are detected.

3.6.2 Liveness. Local deadlocks at a node are detected by the current node, while deadlocks within a DZ are detected by a node within that DZ. Global deadlocks are ultimately detected by the root node. The entire detection process proceeds from the leaf nodes up to the root of the tree and is executed periodically, ensuring that deadlocks are detected within a finite amount of time.

4 OPTIMIZATION STRATEGIES

4.1 Reduce Subgraph Transmission

4.1.1 Pruning. The WFG-based deadlock detection algorithms typically prune the LWFG before sending it to the detection node to reduce network pressure. Typically, a coarse pruning algorithm determines whether the current LWFG has an outgoing edge pointing to or an incoming edge from another node. If such edges do not exist, nodes will not transmit the local subgraph to the detection node. We present a more granular pruning approach: *A node's*

LWFG G_i may have multiple subgraphs G_i^j. Only if a subgraph has both outgoing edges pointing to and incoming edges from other nodes will G_i^j be transmitted to the detection node.

Lemma 4.1. *When a local graph $G = (V, E)$ does not have incoming edges pointing from other nodes, i.e., all edges (u, v) in the global graph satisfy that $v \notin V$. Then there does not exist a path P that satisfies both $P \subseteq C$ and $P \subseteq G$, where $C \nsubseteq G$.*

Proof of 4.1. Suppose there exists such a cycle C and graph G (where $C \nsubseteq G$), and P is a common path of part of C and G. Then, there must exist an edge (u', v') such that $u' \in V$ and $v' \notin V$. Since C is a cycle, there must exist a path P' starting from v' and returning to u'. Therefore, there must exist another edge $(x, y) \subseteq P'$, and $(x, y) \neq (u', v')$ where $x \notin V$ and $y \in V$. This contradicts the given condition that all edges (u, v) in the global graph satisfy that $v \notin V$. Therefore, no such cycle and graph exist. □

According to Lemma 4.1, when a subgraph has no outgoing edges pointing to other nodes, no path is part of any global deadlock cycle. Therefore, the entire subgraph satisfying the aforementioned condition can be skipped during transmission. It is worth noting that, in the hierarchical detection algorithm, such subgraphs are not only subgraphs of a node's LWFG but can also be subgraphs of a WFG$_{zone}$. Therefore, at each level of the hierarchical detection algorithm, subgraph transfer can be optimized using this method.

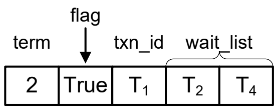


Figure 8: An example of an adjacency list item in the WFG.

4.1.2 Incremental Updates of the WFG. An adjacency list is typically used to store the WFG and represent dependencies between transactions. For example, in a multi-outdegree transaction model, a transaction can send multiple resource requests simultaneously and may wait for multiple other transactions to release locks. The following Figure 8 represents an item in the adjacency list.

The fields *txn_id* and *wait_list* indicate that transaction $T_1$is waiting for two other transactions T_2 and T_4. The hierarchical detection algorithm runs periodically, assigning each run a *term*. The root node maintains and synchronizes this term to all nodes. Once the transaction wait list is updated, the new *term* overwrites the old value, and the $flag$ is set to `True` to indicate that the item has been updated within the current term. After transmission, the $flag$ is reset to `False` again. If the wait list of transaction T_i does not change within a term, then during the next transmission, it is sufficient to update the term to $term + 1$ and send $(new\_term, False, T_i)$ to the detection node. This is an incremental method for updating the WFG. The detection node maintains a cached WFG through this optimization rather than retrieving the entire subgraph of all nodes each time. In addition to subgraph pruning, this method further reduces the size of the transmitted subgraphs.

4.2 Efficient Detection of Multi-outdegree WFG

In the single out-degree transaction model, each transaction waits on one resource at a time, and the maximum outdegree of the wait-for graph will be one. One deep traversal is sufficient to determine whether there is a cycle exists in the WFG. The complexity of a depth traversal is related to the number of vertices in the graph, which is $O(n)$. The deadlock detection of the multiple outdegree WFG is more complex. Consider a WFG that contains two cycles: $C_1 = (T_1, T_2, T_3, T_1)$ and $C_2 = (T_2, T_3, T_4, T_2)$. Among them, (T_2, T_3) are the common paths of two cycles. A flag array is used to indicate whether each node has been visited. When the flags for the corresponding vertices T_2 and T_3 are set to `True` after C_1 is detected, cycle C_2 will no longer be detected because the flags of T_2 and T_3 are marked as true. To address this issue, we change the flag to a positive integer, initialize it as `abs(` *outdegree − indegree*), and decrement it after each access.

Multiple detections are necessary for a multi-outdegree deadlock detection algorithm, requiring up to $n - 1$ detections in the worst case, where n is the number of vertices in the WFG, resulting in a computational complexity of $O(n^2)$. In centralized detection algorithms, a single detection node faces significant computational pressure when detecting complex WFGs. In contrast, the hierarchical detection algorithm partitions the computational cost.

4.3 Deadlock Resolution

4.3.1 Transaction Priority. The unique number represents priority, and the *M&M* algorithm compares by adding additional private priority numbers to promptly terminate the transaction with the lowest priority upon detecting a deadlock. Transaction IDs can be directly compared in the WFG-based detection algorithm with a self-increasing field. The smaller the ID, the higher the priority, and transactions with larger IDs are more likely to be terminated when deadlocks are detected. This method is simple and suitable for most scenarios but is inefficient in WFG with multiple outdegrees.

4.3.2 Mixed Transaction Priority. In Section 4.2, we demonstrate two deadlock cycles with a common path in which vertex set $V_{com} = (T_2, T_3)$. When only using transaction ID as the priority, two transactions need to be terminated to resolve all the deadlocks. First is the youngest T_4, and then C_1 remains deadlocked. T_3 needs to be terminated again to break the cycle C_1.

It is obvious that the termination of T_4 is unnecessary, and only terminating T_3 is enough to free the system from all the deadlock. We maintain a hash table during each detection to store the number of occurrences of transactions in multiple deadlock cycles. The transaction with the highest occurrence frequency will be terminated first, and the system will be restored to an active state at the minimum cost. The priority is defined as $<frequency, txn_{id}>$. Transactions with the highest $frequency$ have the highest priority. If multiple transactions have the same frequency, their transaction IDs are further compared.

5 PERFORMANCE EVALUATION

In this section, we evaluate HAWK by comparing it with other algorithms with TPC-C and micro benchmark workloads. Specifically, we first compare the impact of HAWK with several deadlock detection algorithms on throughput and average transaction latency. Then, we compare the improvement of the fine-grained SCC-cut algorithm to SCC-cut under the partitioned workload. We then evaluate the adaptability and shortcomings of HAWK under dynamic workloads. Finally, we analyzed the necessity of adopting a fine-grained SCC-cut algorithm and compared the consumption of network resources by different algorithms.

5.1 Implementation

Our work is built upon the open-source code provided in the LCL paper [44], which introduced an efficient deadlock detection framework. We have made adaptive adjustments and extensions to this framework to support the deadlock detection algorithm we evaluated. Concurrency control is based on the two-phase locking protocol (2PL), where transactions continuously request locks during execution and release all acquired locks upon transaction completion. Each of our transaction threads issued a new transaction as soon as its previous transaction was finished. The latency of each transaction was calculated as the time from when it was issued until it finished. For tree construction and reconstruction, the parameter $\overline{dt}$ was set to 50 ms, resulting in B being 32, according to Figure 2. The α was set to 1.

5.2 Experimental Setup

5.2.1 The Testbed. We conducted our experiments on up to 128 machines, all configured identically. Each node consists of two Intel Xeon Silver 4100 processors, 160 GB of main memory. The operating system on these machines was CentOS 7.4 64-bit. The machines are connected through a 10 Gbps Ethernet network.

5.2.2 Workloads.

TPC-C Workloads. For the TPC-C workload, we used 128 database nodes, with each node managing 10 warehouses. In the standard TPC-C operations, transactions access local and remote warehouses in a certain proportion, we adjust the proportion to 1:1. In addition, we added some restrictions to remote warehouse access. Specifically, we divide the nodes into 16 partitions, and all transactions' remote warehouse access is either within or outside of the partition, with a ratio of 8: 2. Only 20% of remote warehouse access cross partitions, causing most deadlocks to occur within each partition.

Microbenchmark. We created a microbenchmark for a more thorough evaluation of our algorithm in a scenario with a high probability of deadlock occurrence. Specifically, we reduce the data scale to 1000 per node. With small amounts of data, workload competition increases, resulting in more deadlock samples. Each transaction consists of multiple SQL statements, with each SQL statement accessing multiple records. The workload is generated following an exponential distribution. Each transaction consists of 10 to 50 randomly generated SQL statements, with an average of 30 (i.e., $\lambda = 1/30$). Each write SQL statement randomly acquires exclusive locks on one to five rows, with an average of 1.2 rows locked per statement (i.e., $\lambda = 1/1.2$). We use the polling mode to run a physical thread on multiple transaction queues, with the ratio being 1: 8.

5.2.3 Baselines. We compared the performance of our algorithm with several baselines, including a **centralized** algorithm and two distributed deadlock detection algorithms **path-pushing** and **LCL** [44, 45]. In addition, the baselines also include several variants of hierarchical algorithms, specifically:

range-cut-n: In the absence of knowledge about workload access patterns, constructing a hierarchical tree can be challenging. However, it is possible to partition nodes into several zones based on partitioning experience and other domain knowledge. Based on experience, the lengths of deadlock cycles follow an exponential distribution, with shorter lengths being more prevalent. We compared two fixed partitioning methods by node numbering, which uniformly partition the nodes into several zones containing n nodes. Specifically, we evaluated $n = 4$ and 8, with most deadlock cycles length not exceeding 8.

HAWK and HAWK-greedy: Corresponding to basic SCC-cut algorithm and fine-grained greedy SCC-cut algorithm respectively.

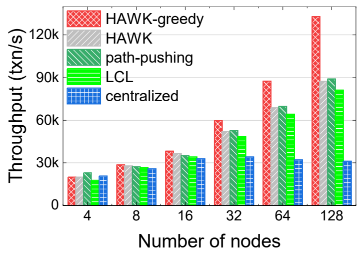
5.3 Throughput of TPC-C

We compared system throughput, average latency, and average deadlock detection time when using our proposed algorithms, HAWK and HAWK-greedy, along with three other deadlock detection algorithms, LCL, path-pushing, and centralized from previous works. We varied the number of database nodes from 4 to 128. The results of this experiment for TPC-C are shown in Figure 9.

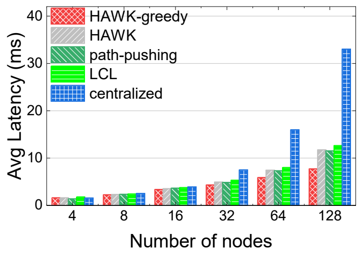
Figure 9(a) illustrates the scalability of these algorithms. Except for HAWK-greedy, the scalability of other algorithms gradually decreases as the number of nodes increases. Only HAWK-greedy maintains high scalability even with 128 nodes, achieving a throughput of over 120k, which is $4\times$ that of the centralized algorithm. The centralized algorithm experiences a decline in throughput when the number of nodes exceeds 32, indicating that the centralized deadlock detection node becomes a system bottleneck, limiting overall scalability. Figure 9(b) presents the average transaction latency for each algorithm. HAWK-greedy demonstrates high detection efficiency, minimizing its impact on transaction latency. In contrast, the detection efficiency of other algorithms decreases as the number of nodes increases, with the centralized algorithm being the most affected, reaching a worst-case latency of over 30 ms. Figure 9(c) shows the deadlock detection time for each algorithm. As the number of nodes increases, the detection time increases for all algorithms. However, HAWK-greedy can keep the detection time within 100 ms, even with 128 nodes. In contrast, other algorithms exhibit a significant increase in detection time, prolonging the resource acquisition time for successor transactions in the waiting chain of the lock request, thereby deteriorating transaction throughput.

5.4 Throughput of Microbenchmark

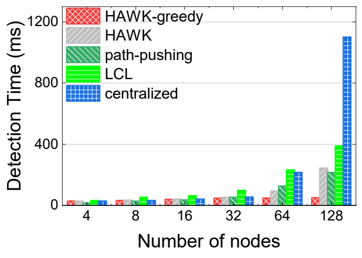
In this section, we compare five algorithms: *HAWK*, *HAWK-greedy*, *centralized*, *path-pushing*, and *LCL*, similar to the TPC-C experiment. All SQL queries randomly access resources across all nodes in the system. We focus on the impact of deadlock detection time on throughput by adjusting the number of database nodes. Figure 10 shows the average detection time and throughput of different algorithms under various numbers of nodes. *HAWK-greedy* reduces



(a) Throughput

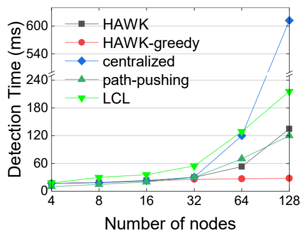


(b) Avg. Lantency

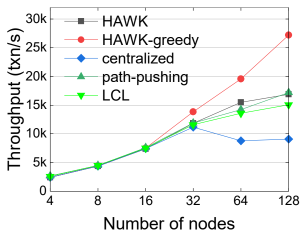


(c) Detection Time

Figure 9: (a) Throughput, (b) Average latency, and (c) Detection Time of five algorithms under TPC-C workload.

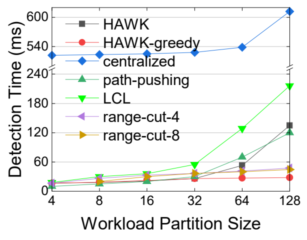


(a) Detection Time

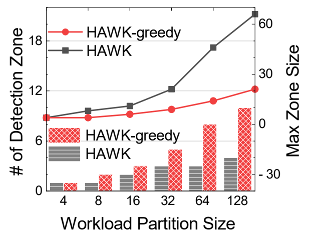


(b) Throughput

Figure 10: (a) Deadlock Detection Time and (b) Throughput under various numbers of nodes.



(a) Detection Time



(b) # of Zones and Max Zone

Figure 11: (a) Detection time and (b) The cutting results of *HAWK* and *HAWK-greedy*, where the bar chart corresponds to the left y-axis and the line chart corresponds to the right y-axis under different workload partition sizes.

the average detection time by up to 600 ms compared to other algorithms and increases throughput by up to 3×.

Figure 10(a) shows that the number of nodes 32 is a watershed; for the number of nodes from 4 to 32, all algorithms maintain low detection time and similar throughput. As the number of nodes exceeds 32, the overall transaction throughput decreases accordingly. From 4 to 16 nodes, *path-pushing* exhibits optimal performance in both average detection time and throughput. However, as the length of deadlock cycles increases, the efficiency of *path-pushing* decreases when the number of nodes exceeds 32. Besides, results show that LCL's detection time and throughput are slightly inferior to *path-pushing*. The algorithm *centralized*, due to the individual detection nodes becoming bottlenecks, exhibits an average deadlock detection time exceeding 600 ms. Due to the inability to partition DZs effectively, the gap between *HAWK* and *HAWK-greedy* widens as the number of system nodes increases. *HAWK-greedy* demonstrates its advantage when the number of nodes exceeds 32, outperforming all other algorithms. With 128 nodes, the average detection time of the other algorithms exceeds 120 milliseconds, while *HAWK-greedy* remains stable.

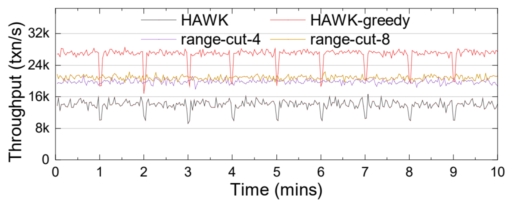
Figure 10(b) shows the throughput of the microbenchmark, corresponding to the transaction details described in Section 5.2.2. The overall trend in system throughput is inversely proportional to the average deadlock detection time. Due to decreased detection efficiency and the impact of distributed transactions, throughput does not scale linearly with the number of nodes. In addition to deadlock detection time, the number of deadlocks also influences system throughput. *HAWK-greedy* improves throughput by up to 3 × compared to the *centralized* and by up to 1.8 × compared to *LCL*.

5.5 The Impact on Workload Partition Size

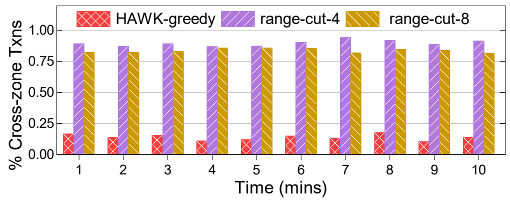
Next, we evaluate the impact of different workload partition sizes on the average deadlock detection time with a fixed configuration of 128 nodes. By partitioning the workload, we adjust the scope of transaction access to resources, thereby controlling the maximum length of generated distributed deadlock cycles. We primarily tested two proposed algorithms, *HAWK* and *HAWK-greedy*, alongside five baselines: *centralized*, *path-pushing*, *LCL*, *range-cut-4*, and *range-cut-8*.

Figure 11(a) shows the average detection time. The performance of the centralized deadlock detection algorithm is notably poor, with an average detection time exceeding 500 ms even though the workload partition size is 4. Due to the partitioning workloads and partitioning detection zones, all algorithms except for the *centralized* exhibit a trend similar to that shown in Figure 10(a).

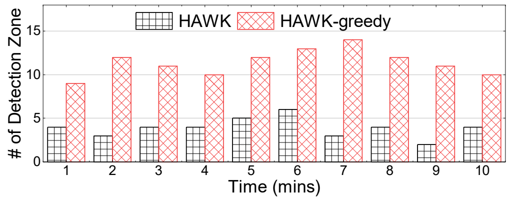
In this experiment, a three-level hierarchical tree was constructed, with L_0 as the root, L_1 as the partitioned detection zones, and L_2 corresponding to each database node. The *range-cut-4* and *range-cut-8* can quickly detect deadlocks at level L_1 when the workload partition size is 4 and 8 because the detection zones precisely match the workload partitions. However, when the workload partition size increases to 16, due to mismatches between detection zones and workload partitions, the performance of *range-cut-4* is slightly lower than *range-cut-8*, increasing deadlocks undetectable at the level L_1, with most deadlocks only detectable at the root level of the hierarchical tree. When the partition size exceeds 16, almost



(a) Performance Over Time



(b) Percentage of Cross-zone Deadlocks



(c) Number of Zones

Figure 12: The performance of deadlock detection algorithms where the workloads change over time.

every detection zone in *range-cut-4* and *range-cut-8* fails to perform adequately.

Workload-driven detection zone partitioning is not always the best approach for every scenario. *HAWK* performs worse than *range-cut* at 64 and 128 nodes. In Figure 11(b), we present the number of zones and the maximum zone size obtained by algorithms *HAWK* and *HAWK-greedy* for different partition sizes. Although *HAWK* consistently maintains a low number of zones, its maximum zone size reaches 66 at 128 nodes. This implies that the efficiency within this zone is comparable to centralized detection with 64 nodes. Even if there are relatively short deadlock cycles within the zone, their detection may be delayed significantly.

5.6 Adaptability

Next, we examine the adaptability of HAWK to a time-varying workload. We use the same configuration as in Section 5.4, with the only difference being that the workload changes every minute and the experiment runs for 10 minutes. The cost of cutting SCC in a PAG with 128 nodes will not exceed 10 ms, and the variation is not significant. Therefore, we did not display it separately. The time for calculating the deadlock ratio to reconstruct the hierarchical tree is 5 seconds, and we set α as 1. As shown in Figure 12(a) we observe that once the workload changes, both *HAWK* and *HAWK-greedy* experience a noticeable performance degradation, while *range-cut-n* remains stable. The reason is that workload-driven DZ partitioning fails to adapt to the new workload during sampling. Among the algorithms, *HAWK-greedy* maintains the best throughput performance, followed closely by *range-cut-8*, while *range-cut-4* detects fewer deadlocks within zones compared to *range-cut-8*, though its performance is slightly inferior. Surprisingly, *HAWK* exhibits the worst performance.

We calculated the proportion of deadlocks detected across DZs, defined as the percentage of deadlocks that remain undetected within a single DZ (at level L_1). Figure 12(b) presents the results, with *HAWK* excluded as it does not require secondary detection. Regardless of workload variations, *HAWK-greedy* consistently identifies the optimal PAG cutting, maintaining the proportion of deadlocks across zones below 17%. In contrast, *range-cut-n* fails to detect most deadlocks within DZs, resulting in over 82% deadlocks occurring across DZs.

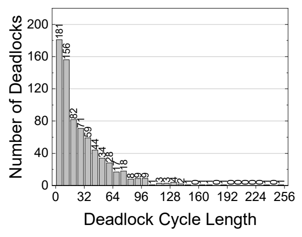
Figure 12(c) illustrates the number of detection zones for *HAWK* and *HAWK-greedy*. *HAWK-greedy* consistently subdivides the PAG more finely, ensuring that the burden of detecting deadlock nodes within each DZ is not too heavy. In contrast, *HAWK* produces fewer DZs in the PAG partitioning, leading to larger DZs. This results in reduced throughput performance for *HAWK*.

5.7 Comprehensive Dive into Greedy *SCC-Cut*

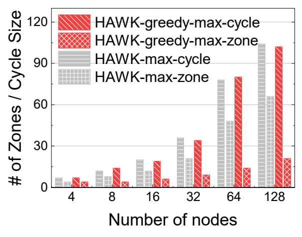
The previous experiment provided an overview of why *HAWK* performs worse than *HAWK-greedy* in scenarios with a large number of nodes (exceeding 32 nodes in our experiment), despite both being workload-driven PAG graph cutting algorithms. This section delves deeper to further clarify the advantages of *HAWK-greedy* and highlight the differences between it and other methods, e.g., *centralized* and fully distributed algorithm *path-pushing* and *LCL*.

First, we compare *HAWK* and *HAWK-greedy* to highlight the necessity of using *HAWK-greedy* for partitioning the graph PAG. We ran the workload for 5 minutes without partitioning it with 128 nodes. The results are shown in Figure 13(a). The longest deadlock cycle had a length of 252, while the shortest was 3. The majority of deadlock cycle lengths are concentrated between 0 and 64, with 75% of them falling within 0-32 and 14% within 32-64. Only 11% exceed 64. Intuitively, most deadlocks (length less than 32) can be detected quickly. Although the proportion of deadlocks with a length exceeding 64 is low, it may lead to larger DZs, resulting in reduced detection efficiency for these DZs. Figure 13(b) shows the detection zone partitioning results of *HAWK* and *HAWK-greedy*. Although the detected longest deadlock cycles length is approximately the same, the maximum detection zone size of *HAWK* is much larger than that of *HAWK-greedy*. Figure 14 provides detailed results of the detection zone partitioning for *HAWK* and *HAWK-greedy*.

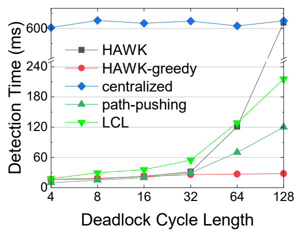
Then, we analyze *HAWK-greedy* compared with three traditional deadlock detection algorithms from the perspectives of deadlock cycle length and network utilization. In Figure 13(c), we measured the detection times for deadlocks of varying lengths within the detected deadlock results. Among these algorithms, *centralized* takes the longest time to detect deadlocks, even for very short cycles. The *path-pushing* and *LCL* are fully distributed algorithms, and their detection efficiency is independent of the number of nodes, only being related to the length of the deadlock cycle. In other words, the longer the cycle, the longer the detection time. As the



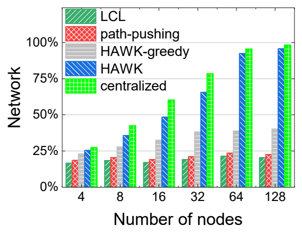
(a)



(b)



(c)



(d)

Figure 13: Algorithms *HAWK-greedy* compared with other algorithms in terms of detection zone partitioning, deadlock cycle length, and network utilization.

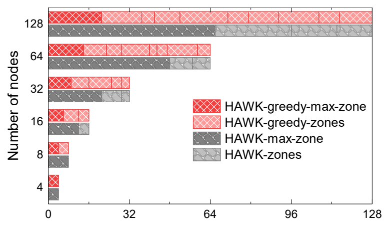


Figure 14: The deadlock detection zone decomposition diagrams for both *HAWK* and *HAWK-greedy*.

length of deadlock cycles affects the size of the largest detection zones, *HAWK*'s efficiency decreases significantly when the number of nodes exceeds 32. In this experiment, only *HAWK-greedy* is unaffected by both the number of nodes and the length of the deadlock cycle. In Figure 13(d), we measured the network usage of the node with the highest network occupancy rate. *path-pushing* and *LCL* always maintain low occupancy because they do not require the construction of WFG. *HAWK* and *Centralized*, because a single detection node is responsible for deadlock detection of a large range of nodes, resulting in high network occupancy when transmitting sub-WFGs. Although the *HAWK-greedy* has a higher network occupancy than distributed algorithms, it is much better than *HAWK* due to the finer granularity of DZ partitioning.

6 RELATED WORK

Due to the use of lock managers [46, 47] to manage distributed resources, distributed deadlocks are inevitably generated, causing significant harm to distributed applications. Many studies address deadlock detection in distributed systems [11, 13–15, 32, 39, 41]. Knapp [25] categorizes these works into four categories: *path-pushing*, *edge-chasing*, *diffusing computation*, and *global state detection*. In *path-pushing* [35, 42], the local waiting-for graph (LWFG) of the current node is pushed through the system along the edges pointing to other nodes, eventually reaching the initiator node, where the complete WFG can be constructed and checked for deadlock cycles. If termination occurs before reaching the initiator during the push phase, it indicates the absence of deadlock among the nodes. *Edge-chasing* [15] is a probe-based algorithm where a specialized probe message is transmitted along the reverse side of an edge. Upon the probe's return to the initiator node, it identifies the existence of a deadlocked cycle. *Diffusing computation* [10, 16, 19, 26, 31], used primarily in the k-out-of-n model, sends messages to successor nodes and detects knots or cycles through returned messages. *Global state detection* algorithms aim to capture a consistent snapshot of the wait-for graph (WFG) and identify deadlocks within this snapshot to prevent inconsistencies that could result in detecting phantom deadlocks. The AND model [7, 8, 18] and resource allocation graph (RAG) [34]/wait-for graph (WFG) are commonly used to model waiting relationships between transactions, with deadlock detection identifying cycles in the graph to determine deadlocks.

TiDB [24] uses a centralized detection algorithm where, after a transaction times out, it sends a detection request to the detection node. The detection node collects the WFGs from all nodes to form a global WFG, which is then used to detect and resolve deadlocks. CockroachDB's [42] transaction coordinator maintains a wait queue, regularly collecting and merging the wait relations of all transactions. The wait queue ultimately maintains a complete WFG for deadlock detection. LCL [44, 45] has two processes, *Proliferation* and *Spread*, to ensure values *LCLV* and $< PuAP, PuID >$ are propagated to every node in the graph. By propagating and comparing these two values, deadlock cycles and the transactions that should be aborted within them can be identified. The LCL algorithm is an edge-chasing algorithm.

7 CONCLUSION

In this paper, we presented HAWK, a novel workload-driven hierarchical deadlock detection algorithm that enforces efficiency and adaptability. By transforming the problem of partitioning detection zones in the hierarchical detection into a graph-cutting problem on the PAG, HAWK can organize the hierarchical tree before the WFG is formed. In addition, a fine-grained greedy SCC-cut method for cutting complex workloads is used to better partition the PAG into multiple appropriately sized detection zones, making HAWK more efficient in detection. Tree reconstruction ensures adaptability to workload changes, maintaining detection adaptability. Experimental results show that HAWK is superior at detecting deadlock in large-scale systems compared to other algorithms.

ACKNOWLEDGEMENTS

We thank the anonymous reviewers for their insightful comments and feedback. This work was supported in part by the National Natural Science Foundation of China under Grant No. U22B2020. Peng Cai is the corresponding author.

REFERENCES

[1] 2025. Postgres. https://www.postgresql.org/

[2] 2025. TPC-C Benchmark. https://www.tpc.org/tpcc/results/tpcc_results5.asp

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

[4] Dushan Z. Badal. 1986. The distributed deadlock detection algorithm. *ACM Transactions on Computer Systems (TOCS)* 4, 4 (1986), 320–337.

[5] Francesca Baldini, Faizan M Tariq, Sangjae Bae, and David Isele. 2024. Don't Get Stuck: A Deadlock Recovery Approach. *arXiv preprint arXiv:2408.10167* (2024).

[6] Roberto Baldoni and Silvio Salza. 1997. Deadlock detection in multidatabase systems: a performance analysis. *DIstributed Systems Engineering* 4, 4 (1997), 244.

[7] Valmir C Barbosa and Mario RF Benevides. 1998. A graph-theoretic characterization of AND-OR deadlocks. In *UFRJ Technical Report COPPE-ES-472/98, Rio de Janeiro, Brazil*. Citeseer.

[8] Valmir Carneiro Barbosa, Alan Diêgo A Carneiro, Fábio Protti, and Uéverton S Souza. 2016. Deadlock models in distributed computation: foundations, design, and computational complexity. In *Proceedings of the 31st annual ACM symposium on applied computing*. 538–541.

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

[10] Azzedine Boukerche and Carl Tropper. 1998. A distributed graph algorithm for the detection of local cycles and knots. *IEEE Transactions on Parallel and Distributed Systems* 9, 8 (1998), 748–757.

[11] Gabriel Bracha and Sam Toueg. 1984. A distributed algorithm for generalized deadlock detection. In *Proceedings of the third annual ACM symposium on Principles of distributed computing*. 285–301.

[12] Wilson Chan. 2010. Method and system for deadlock detection in a distributed environment.

[13] KM Chandy, J Misra, L Haas, and TEXAS UNIV AT AUSTIN DEPT OF COMPUTER SCIENCES. 1982. A Distributed Deadlock Detection Algorithm and Its Correctness Proof. *submitted to the Communications of the ACM* (1982).

[14] K Mani Chandy and Jayadev Misra. 1982. A distributed algorithm for detecting resource deadlocks in distributed systems. In *Proceedings of the first ACM SIGACT-SIGOPS symposium on Principles of distributed computing*. 157–164.

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

[16] Alok N. Choudhary, Walter H. Kohler, John A. Stankovic, and Don Towsley. 1989. A modified priority based probe algorithm for distributed deadlock detection and resolution. *IEEE Transactions on Software Engineering* 15, 1 (1989), 10–17.

[17] Carlo Curino, Evan Philip Charles Jones, Yang Zhang, and Samuel R Madden. 2010. Schism: a workload-driven approach to database replication and partitioning. (2010).

[18] JR González De Mendívil, Federico Fariña, JR Garitagotia, Carlos F Alastruey, and Jose M. Bernabeu-Auban. 1999. A distributed deadlock resolution algorithm for the AND model. *IEEE transactions on parallel and distributed systems* 10, 5 (1999), 433–447.

[19] Edsger W Dijkstra and Carel S Scholten. 1980. Termination detection for diffusing computations. *Inform. Process. Lett.* 11, 1 (1980), 1–4.

[20] Harold N Gabow. 1999. Path-based depth-first search for strong and biconnected components; CU-CS-890-99. (1999).

[21] Tarek Helmy. 2024. An Improved Deadlock Detection and Resolution Algorithm for Distributed Computing Systems. (2024).

[22] Benjamin Hilprecht, Carsten Binnig, and Uwe Röhm. 2020. Learning a Partitioning Advisor for Cloud Databases. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*.

[23] Gary S. Ho and CV Ramamoorthy. 1982. Protocols for deadlock detection in distributed database systems. *IEEE Transactions on Software Engineering* 6 (1982), 554–557.

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

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

[26] Ajay D. Kshemkalyani and Mukesh Singhal. 1994. Efficient detection and resolution of generalized distributed deadlocks. *IEEE Transactions on Software Engineering* 20, 1 (1994), 43–54.

[27] Soojung Lee. 2004. Fast, centralized detection and resolution of distributed deadlocks in the generalized model. *IEEE Transactions on Software Engineering* 30, 9 (2004), 561–573.

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

[29] Zhenghua Lyu, Huan Hubert Zhang, Gang Xiong, Gang Guo, Haozhou Wang, Jinbao Chen, Asim Praveen, Yu Yang, Xiaoming Gao, Alexandra Wang, et al. 2021. Greenplum: a hybrid database for transactional and analytical workloads. In *Proceedings of the 2021 International Conference on Management of Data*. 2530–2542.

[30] Daniel A Menasce and Richard R Muntz. 1979. Locking and deadlock detection in distributed databases. *IEEE Transactions on Software Engineering* 3 (1979), 195–202.

[31] Jayadev Misra and K. Mani Chandy. 1982. A distributed graph algorithm: Knot detection. *ACM Transactions on Programming Languages and Systems (TOPLAS)* 4, 4 (1982), 678–686.

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

[33] MySQL. 2024. https://www.mysql.com.

[34] Qinqin Ni, Weizhen Sun, and Sen Ma. 2009. Deadlock detection based on resource allocation graph. In *2009 Fifth International Conference on Information Assurance and Security*, Vol. 2. IEEE, 135–138.

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

[36] M Tamer Özsu, Patrick Valduriez, et al. 1999. *Principles of distributed database systems*. Vol. 2. Springer.

[37] Abdul Quamar, K Ashwin Kumar, and Amol Deshpande. 2013. SWORD: scalable workload-aware data placement for transactional workloads. In *Proceedings of the 16th international conference on extending database technology*. 430–441.

[38] Ahmed Elajeli Rgibi, Abdusamea Ibrahim Omer, Amany Khalifa Alarbish, and Amal Apojila Osha. 2024. EXPLORING DEADLOCK DETECTION ALGORITHMS IN CONCURRENT PROGRAMMING: A COMPARATIVE ANALYSIS AND EVALUATION. *Scientific Journal of Applied Sciences of Sabratha University* (2024), 106–117.

[39] Pooja Sapra, Suresh Kumar, and RK Rathy. 2013. Deadlock detection and recovery in distributed databases. *International Journal of Computer Applications* 73, 1 (2013).

[40] Micha Sharir. 1981. A strong-connectivity algorithm and its applications in data flow analysis. *Computers & Mathematics with Applications* 7, 1 (1981), 67–72.

[41] Selvaraj Srinivasan and Ramasamy Rajaram. 2011. A decentralized deadlock detection and resolution algorithm for generalized model in distributed systems. *Distributed and Parallel Databases* 29 (2011), 261–276.

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

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

[44] Zhenkun Yang, Chen Qian, Xuwang Teng, Fanyu Kong, Fusheng Han, and Quanqing Xu. 2023. LCL: A Lock Chain Length-based Distributed Algorithm for Deadlock Detection and Resolution. In *2023 IEEE 39th International Conference on Data Engineering (ICDE)*. IEEE, 151–163.

[45] Zhenkun Yang, Chuanhui Yang, Fusheng Han, Mingqiang Zhuang, Bing Yang, Zhifeng Yang, Xiaojun Cheng, Yuzhong Zhao, Wenhui Shi, Huafeng Xi, et al. 2022. OceanBase: a 707 million tpmC distributed relational database OceanBasesystem. *Proceedings of the VLDB Endowment* 15, 12 (2022), 3385–3397.

[46] Dong Young Yoon, Mosharaf Chowdhury, and Barzan Mozafari. 2018. Distributed lock management with RDMA: decentralization without starvation. In *Proceedings of the 2018 International Conference on Management of Data*. 1571–1586.

[47] Zhuolong Yu, Yiwen Zhang, Vladimir Braverman, Mosharaf Chowdhury, and Xin Jin. 2020. Netlock: Fast, centralized lock management using programmable switches. In *Proceedings of the Annual conference of the ACM Special Interest Group on Data Communication on the applications, technologies, architectures, and protocols for computer communication*. 126–138.