Scaling distributed transaction processing and recovery based on dependency logging

Chang Yao1 · Meihui Zhang2, Qian Lin1 · Beng Chin Ooi1 · Jiatao Xu3

Received: 29 June 2017 / Revised: 26 February 2018 / Accepted: 10 March 2018
© Springer-Verlag GmbH Germany, part of Springer Nature 2018

Abstract
Dependency graph-based concurrency control (DGCC) protocol has been shown to achieve good performance on multi-core in-memory system. DGCC separates contention resolution from the transaction execution and employs dependency graphs to derive serializable transaction schedules. However, distributed transactions complicate the dependency resolution, and therefore, an effective transaction partitioning strategy is essential to reduce expensive multi-node distributed transactions. During failure recovery, log must be examined from the last checkpoint onward and the affected transactions are re-executed based on the way they are partitioned and executed. Existing approaches treat both transaction management and recovery as two separate problems, even though recovery is dependent on the sequence in which transactions are executed. In this paper, we propose to treat the transaction management and recovery problems as one. We first propose an efficient distributed dependency graph-based concurrency control (DistDGCC) protocol for handling transactions spanning multiple nodes and propose a new novel and efficient logging protocol called dependency logging that also makes use of dependency graphs for efficient logging and recovery. DistDGCC optimizes the average cost for each distributed transaction by processing transactions in batches. Moreover, it also reduces the effects of thread blocking caused by distributed transactions and consequently improves the runtime performance. Further, dependency logging exploits the same data structure that is used by DistDGCC to reduce the logging overhead, as well as the logical dependency information to improve the recovery parallelism. Extensive experiments are conducted to evaluate the performance of our proposed technique against state-of-the-art techniques. Experimental results show that DistDGCC is efficient and scalable, and dependency logging supports fast recovery with marginal runtime overhead. Hence, the overall system performance is significantly improved as a result.

Keywords  Transaction management · Concurrency control · Recovery · Logging · Distributed system

1 Introduction
Database systems process transactions [10] to effect online updates. They serve as the infrastructure of interactive applications such as stock trading, banking, e-commerce and inventory management. Naturally, online transaction processing (OLTP) plays a key role in the database systems as well as applications built on top of it. In-memory systems have been gaining tractions in recent years due to factors such as the increased capacity of main memory and its decreased price and the widening gap in memory bandwidth with respect to the disk storage. Consequently, the cost of buffer management is further reduced or even eliminated [16], and the performance of in-memory OLTP systems is now mainly constrained by latching, locking and logging [12,50]. Accordingly, many recent research efforts for in-memory OLTP systems have been focusing on the design
and optimization of concurrency control protocols [31,37,48] and logging techniques [28,46,51].

Most of the existing concurrency control protocols resolve contentions by examining the conflicts among individual transactions. As an optimization, various batching strategies may be employed. However, batching is traditionally considered to be supplementary in the design of concurrency control protocol. In contrast, to reduce frequent disk I/Os, database systems typically write logs in a batch manner [11]. As a consequence, the tuple-oriented concurrency control and the batched transaction logging are a mismatch that may counter each other’s optimizations.

To address the aforementioned issue, we advocate that batching should be treated as a first-class citizen in the design of concurrency control for OLTP systems. It is a pragmatic consideration for efficient in-memory OLTP in a distributed environment based on the following key observations. First, batch-oriented strategies are already widely adopted in transaction processing to optimize the runtime performance. For instance, client usually sends a group of requests to servers to optimize network throughput and reduce latency [27]. Moreover, as frequent disk I/Os restrict system’s performance especially for in-memory systems, group commit and batch logging become a standard optimization [6,41]. Second, as shown in Fig. 1a, the performance of a distributed OLTP system is mainly affected by distributed transactions, which introduce much thread blocking (time intervals highlighted in Fig. 1a) due to coordination and distributed synchronization and as a consequence hinder the full use of computational resources. With batch processing enabled, as shown in Fig. 1b, the blocking synchronization processes can overlap, leading to better utilization of the computational resources.

It has been the practice that the concurrency control and logging are treated as two independent tasks. This could have been the consequence that data logging has been effective and has been regarded as the de facto protocol for logging needed for recovery. Moreover, distributed transaction management is more complex than single-node transaction management, since the data are distributed and a transaction may access data from multiple nodes. Transactions are often partitioned based on the data locality so that the number of subtransactions accessing multiple nodes is kept low. Such information is useful for recovery in the distributed environment. Therefore, we re-examine the transaction management and recovery problems with the aim of achieving high throughput and fast recovery.

Our overall idea is to exploit the same data structure, the dependency graph, for both concurrency control and logging. We first make use of the dependency graph to resolve transaction contentions both within one node and among different nodes, and derive a set of serializable transaction schedules. In particular, transactions are executed in discrete temporal batches, each of which is processed through a dependency resolution phase followed by an execution phase. In the dependency resolution phase, each transaction is parsed into a set of transaction pieces and the operation

![Fig. 1 Life cycle of distributed transaction processing.](image-url)
dependency graphs are constructed in parallel. With the constructed dependency graphs, a set of serializable transaction schedules can be derived. In the execution phase, each transaction schedule is processed by a single thread and thus the overhead due to locks can be eliminated. While transactions in the same schedule are processed sequentially, multiple schedules can be processed in parallel with multi-threading and distributed computing to maximize system resource utilization. Moreover, the above two processing phases can be conducted in a pipelined manner with respect to the batched transactions.

We note that the dependency graphs derived during transaction processing capture sufficient information for recovery as well. We therefore propose a new type of logging strategy, namely dependency logging. Dependency logging reduces the time of log construction by reusing the dependency graphs and hence improves the logging efficiency. Compared with traditional ARIES logging [29], dependency logging captures the dependency information among committed transactions. Instead of replaying log records in serial order, parallel replaying is enabled by reconstructing the dependency relations among log records. Consequently, a bigger degree of parallelism can be exploited and on-demand recovery can be supported when failures occur.

The contributions of this paper are threefold:

- We extend the dependency graph-based concurrency control protocol designed for multi-core systems [45] for transaction management in distributed environment. The proposed distributed dependency graph-based concurrency control (DistDGCC) makes use of dependency graphs for concurrency control and facilitates efficient transaction processing and reduces transaction aborts due to conflicts by resolving transaction dependencies ahead of transaction execution.

- Based on the dependency graphs, we present a new type of logging, dependency logging, which improves logging efficiency and supports on-demand recovery when a system failure occurs. This novel dependency logging reduces the log construction time and speeds up the recovery process by reusing the dependency graphs constructed during transaction processing.

- We conduct extensive experiments to evaluate both DistDGCC and dependency logging in distributed environment. The results show that DistDGCC is efficient and scalable, and dependency logging is effective for fast recovery upon a system failure and even further improves the processing efficiency during the runtime.

The rest of the paper is organized as follows. Section 2 introduces some backgrounds. Section 3 describes the mechanism of the original dependency graph-based concurrency control (DGCC) protocol [45]. We introduce the enhanced DistDGCC for a distributed in-memory OLTP system in Sect. 4. The dependency logging is elaborated on in Sect. 5. The recovery mechanism of dependency logging is presented in Sect. 6. We conduct an experimental study in Sect. 7 to evaluate both the effectiveness and efficiency of our proposed techniques. We review the related work in Sect. 8 and conclude the paper in Sect. 9.

2 Background

With the DRAM price decreasing in the recent decades, DRAM is replacing disk as the primary storage. More and more systems attempt to maintain the whole data in memory to support faster data accesses. Consequently, buffer management is no longer a main performance bottleneck for in-memory systems [16], and the efficiency of concurrency control and logging becomes a critical performance issue. Traditionally, concurrency control and logging have been treated as two separate problems and designed independently, which may compromise the computation resource utilization as a consequence.

Concurrency control Fundamentally, concurrency control protocols can be broadly classified into two categories: two-phase locking and timestamps ordering.

As a pessimistic protocol, two-phase locking (2PL) [8] assumes transactions tend to conflict and thus it requires transactions to acquire all locks for a particular data record before releasing them. With locks, read and write operations are allowed to be executed. By following the 2PL scheme, potential conflicting data accesses can be prevented. However, the pessimistic concurrency control scheme of 2PL suffers from high overhead associated with synchronization on concurrency meta-data (i.e., locks) [37]. Moreover, deadlock detection and resolution are also expensive especially in a distributed setting. As a consequence, optimistic concurrency control (OCC) protocols [22] based on timestamp ordering are preferred by recent high-performance OLTP systems [31]. The design of timestamp ordering concurrency control schemes is based on the monotonically increasing timestamps, which are exploited to process conflicting read/write operations in a proper order. While OCC exhibits great success with low-contended workloads, it performs poorly under high contention due to excessive transaction aborts [9]. OCC usually aborts conflicting transactions before the commit time and results in wasting precious CPU cycles spent on these transactions that are destined to abort [2]. Moreover, a centralized timestamp assignment component is usually required in the distributed environment, which also compromises the system performance and scalability.

1 http://www.anandtech.com/show/10512/price-check-q3-2016-dram-prices-down-over-20-since-early-2015.
By taking a batch of transactions rather a single transaction as the processing unit, DGCC achieves good performance on both low contention and high contention workloads by reducing the transaction aborts due to conflicts on multi-core system [45]. It is therefore a natural consideration to extend it to distributed settings. In Table 1, we compare 2PL and OCC with DistDGCC on average latency of distributed transaction and conflicts handling. As illustrated, both 2PL and OCC take 2 round-trip time (RTT) to commit (or abort) a distributed transaction. By aggregating messages for a batch of distributed transactions, DistDGCC reduces the network latency significantly.

3 DGCC overview

A dependency graph-based concurrency control was proposed in (DGCC) [45] to facilitate lock-free transaction processing and reduce transaction aborts caused by contention for multi-core systems. By separating the contention resolution from the transaction execution, DGCC processes transactions in discrete temporary batches. Each batch is processed through a contention resolution phase followed by an execution phase. In this section, we shall briefly describe DGCC as it is the background work for this paper.

DGCC first constructs dependency graphs in parallel according to the transactions’ timestamps and logics. Specifically, each worker constructs one dependency graph for a set of transactions and then decomposes the constructed graph into subgraphs. During the actual transaction execution, each worker thread executes according to one subgraph. With dependency graphs, workers can process the transactions with the guarantee of conflict serializability. DGCC also exploits single-threaded model to the fullest, and consequently, the single-threaded model is adopted in the dependency graph construction and actual transaction execution.

3.1 Dependency graph construction

Before constructing a dependency graph for a set of transactions, DGCC has to resolve dependency relations within one single transaction. It parses the statements of a transaction and decomposes it into a set of record actions, where each record action only accesses a single record in the database. The record action is defined as below:

Definition 1 (Record Action) A transaction is a unit of operations performed on the database. Operations consecutively conducted on the same record within one transaction (with no
Fig. 2 An example of dependency graph construction and partitioning

intervening operation on another distinct record) constitute a single record action \( \alpha_i \).

Given a record action \( \alpha_i \), \( r(\alpha_i) \) denotes the record on which it acts upon, and \( c(\alpha_i), f(\alpha_i), \rho(\alpha_i) \), respectively, denote the transaction’s timestamp, function set and parameter set.

Figure 2 illustrates an example running with a single type of transactional procedure, namely Transfer, on a single table. In particular, transaction \( t_5 \) attempts to transfer \$1 from account \( D \) to account \( E \). DGCC firstly decomposes \( t_5 \) into two record actions, \((5 : D, \text{minus}, 1)\) and \((5 : E, \text{add}, 1)\). Record action \((5 : D, \text{minus}, 1)\) operates on account \( D \) by subtracting \$1 from its balance. Record action \((5 : E, \text{add}, 1)\) attempts to add \$1 to account \( E \).

There are two types of dependency relations on record actions: logical dependency \( \succ \) and temporal dependency \( \succ \).

**Definition 2 (Logical Dependency)** Record action \( \alpha_i \) logically depends on \( \alpha_j \), denoted as

\[
\alpha_i \succ \text{logic} \alpha_j
\]

if and only if both \( \alpha_i \) and \( \alpha_j \) belong to the same transaction, i.e., \( c(\alpha_i) = c(\alpha_j) \), and \( \alpha_i \) must be executed after \( \alpha_j \).

It is obvious that \( \succ \) determines the logical execution order of record actions within one transaction. In the previous example, record action \((5 : E, \text{add}, 1)\) is logically dependent on \((5 : D, \text{minus}, 1)\), since transaction \( t_5 \) has to ensure that the balance in account \( D \) is sufficient. Apart from the logical dependency relation, we also need to resolve the contention of record actions from different transactions, which is defined by the temporal dependency relation \( \succ \).

**Definition 3 (Temporal Dependency)** A temporal dependency exists between record actions \( \alpha_i \) and \( \alpha_j \), denoted as

\[
\alpha_i \succ \text{temporal} \alpha_j
\]

if and only if \( \alpha_i \) and \( \alpha_j \) belong to different transactions with \( c(\alpha_i) > c(\alpha_j) \), and \( r(\alpha_i) = r(\alpha_j) \).

As shown in Fig. 2, record actions \((5 : E, \text{add}, 1)\) temporally depend on \((4 : E, \text{add}, 3)\). In the dependency graph, a vertex represents a record action and an edge represents a dependency relation between two record actions. It is defined as follows:

**Definition 4 (Dependency Graph)** Given a set of transactions \( T = \{t_1, t_2, \ldots, t_n\} \), and the associated sets of record actions \( \varphi_{t_1}, \varphi_{t_2}, \ldots, \varphi_{t_n} \), the dependency graph \( G = (V, E) \) is a directed graph and consists of

- \( V = \varphi_{t_1} \cup \varphi_{t_2} \cup \ldots \cup \varphi_{t_n} \), and
- \( E = \{(\alpha_i, \alpha_j) \mid \alpha_i \succ \text{logical} \alpha_j \text{ or } \alpha_j \succ \text{temporal} \alpha_i\}, \alpha_i \in \varphi_{t_i}, \alpha_j \in \varphi_{t_j}\)
By decomposing a transaction into record actions, logical dependency relations are naturally resolved. When record action \( \alpha \) is inserted into the dependency graph, an edge from \( L(r(\alpha)) \) to \( \alpha \) is created, since \( \alpha \succ_{\text{temporal}} L(r(\alpha)) \). DGCC maintains an action queue \( \phi_k \) for each record \( k \). Given a record action \( \alpha \), it should be appended to the end of \( \phi_r(\alpha) \). The record actions in a queue should satisfy either \( \succ_{\text{temporal}} \) or \( \succ_{\text{logical}} \). Record actions in different queues may only have \( \succ_{\text{logical}} \). Note that \( \succ_{\text{temporal}} \) never exists between record actions in different queues according to its definition. Algorithm 1 summarizes the process of dependency graph construction for a single transaction.

**Algorithm 1** dependencyGraphConstruction(\( t, \mathcal{G} \))

1: Input: transaction \( t \) and dependency graph \( \mathcal{G} \) before inserting \( t \)
2: Output: dependency graph \( \mathcal{G} \) after inserting \( t \)
3: \( \phi_i \leftarrow \{\alpha_1, \alpha_2, ..., \alpha_m\} \) decomposed from \( t \)
4: for \( i \leftarrow 1 \) to \( m \) do
5: \( \mathcal{G}.\text{AddVertex}(\alpha_i) \)
6: if \( L(r(\alpha_i)) \) exists then
7: \( \mathcal{G}.\text{AddEdge}(L(r(\alpha_i)), \alpha_i) \)
8: \( L(r(\alpha_i)) \leftarrow \alpha_i \)
9: for \( i \leftarrow 1 \) to \( m - 1 \) do
10: for \( j \leftarrow i + 1 \) to \( m \) do
11: if \( \alpha_i \succ_{\text{logic}} \alpha_j \) then
12: \( \mathcal{G}.\text{AddEdge}(\alpha_j, \alpha_i) \)
13: return \( \mathcal{G} \)

During the dependency graph construction, each worker thread maintains a constructor to resolve dependency relations among a batch of transactions, which is essentially a consecutive number of transactions from its transaction queue, and build the dependency graph accordingly. To better exploit the parallelism in the CPU, several graphs can be constructed in parallel by different worker threads. Each thread can construct a dependency graph asynchronously since dependency graph construction for a batch of transactions is a completely independent process. Before starting the transaction execution phase, each worker thread needs to decompose each constructed dependency graph \( \mathcal{G} = (\mathcal{V}, \mathcal{E}) \) into evenly partitioned subgraphs for distribution over all the available worker threads. The constructed graphs are first converted to weighted graphs. A vertex in the weighted graph is a record action queue, whose weight equals to the number of record actions in the queue. The weight of one edge equals to the number of logical dependency relations between the two queues. For optimal performance, it is necessary to ensure that, as far as possible, each worker thread gets the same amount of work. Furthermore, execution of a record action in one worker thread may result in sending information to its dependent record actions in other worker threads. Hence, the objective of dependency graph decomposition is to get partitions about the same size with minimal number of cross-partition edges. The graph decomposition problem is isomorphic to the classical uniform graph partitioning problem [18]. We adopt a greedy algorithm to accelerate the dependency graph partitioning, which first evenly partitions the weighted graph according to key range and minimizes edge cuts based on the partitioning result.

### 3.2 Transaction execution

A global synchronization operation is enforced to ensure that all the worker threads will complete the graph construction and partitioning before entering the transaction execution phase simultaneously. Given a set of partitions \( \Psi = \{\mathcal{Y}_1, \mathcal{Y}_2, ..., \mathcal{Y}_n\} \), DGCC distributes it to the available workers to perform the actual transaction execution in parallel. Each worker \( i \) executes the record actions of \( \mathcal{Y}_i \) in a greedy manner, as summarized in Algorithm 2. Initially, the worker selects record actions with no in-edges in the subgraph and inserts them into an executable set that it maintains. Any record action in the executable set will not conflict with any other remaining record actions and can be executed correctly.

A worker thread then iteratively selects record actions from the executable set to execute. After one record action is executed, it will be removed from the subgraph together with its outgoing edges from the original dependency graph \( \mathcal{G} \). Those record actions without incoming edges in the residual graph should then be inserted into the executable set. Figure 3 illustrates an example of the above procedure. In this example, record action (2 : A, minus, 7) has no incoming edges and can be inserted into the executable set. After the record action (2 : A, minus, 7) is executed, the worker thread would remove it and its outgoing edge. The process keeps repeated till all the record actions in \( \mathcal{Y}_i \) are fully executed.

**Algorithm 2** dependencyGraphExecution(\( \mathcal{G} \))

1: Input: dependency graph \( \mathcal{G} = (\mathcal{V}, \mathcal{E}) \)
2: \( \mathcal{G} \leftarrow \emptyset \)
3: while \( \mathcal{G} \neq \emptyset \) and \( \mathcal{G}.\text{Size} > 0 \) do
4: \( \mathcal{G}.\text{AddVertex}(\mathcal{G}.\text{RootVertex}) \)
5: \( \alpha_q \leftarrow \mathcal{G}.\text{Pop}() \)
6: \( \alpha_q.\text{Execute}() \)
7: \( \mathcal{G}.\text{RemoveOutEdge}(\alpha_q) \)
8: \( \mathcal{G}.\text{RemoveVertex}(\alpha_q) \)

DGCC evenly partitions the workload for the transaction execution phase, thus addressing the problem of workload skew. In addition, DGCC reorders operations before transaction execution so that operations on the same record can be executed within a short time interval, thereby improving cache hit rate.

As there is more than one dependency graph available during the execution, it is possible that there exist conflicts among dependency graphs. The conflicts are resolved by pro-
4 Distributed DGCC

While the aforementioned DGCC protocol fits the multi-core and main-memory architecture well, it is nontrivial to apply to the distributed setting. The main challenges are twofold: (1) The original DGCC constructs dependency graphs in parallel for several batches of transactions. However, dependency graphs should be executed in a serial order. When the cluster is large, the graphs executed latter may have to wait for a long time before they are executed, which may increase the latency dramatically. (2) While node failures are infrequent in reality, an increasing number of nodes in distributed environment probabilistically leads to more failures and results in expensive recovery cost [35,38,39]. Thus, an efficient recovery mechanism that supports DGCC is essential. Therefore, instead of treating both concurrency control and recovery as two distinct problems, we make use of the information used in DGCC in logging for supporting fast recovery. However, for ease of explanation, we shall first describe our concurrency control protocol for distributed transaction management, followed by recovery based on dependency logging.

We therefore first propose a transaction management solution, DistDGCC, for distributed environment. DistDGCC introduces a coordination mechanism to adjust the execution order of distributed transactions, which not only guarantees the serializability, but also facilitates high execution parallelism. By aggregating network messages for one batch of transactions, DistDGCC reduces the thread waiting time for network blocking and thus optimizes the average latency. Since local transactions are executed differently from distributed transactions, we distinguish the two as below.

**Local transaction** It accesses data accessed stored in a single compute node. For a local transaction $t_{\text{local}}$ running on one node, we add $t_{\text{local}}$’s corresponding vertices and edges into the dependency graph constructed in the node by directly following the protocol as described in Sect. 3. Note that both $t_{\text{local}}$’s intra-transaction dependencies and inter-transaction dependencies related to $t_{\text{local}}$ are automatically resolved by the resulting dependency graph.

**Distributed transaction** it accesses data stored in multiple compute nodes. For a distributed transaction $t_{\text{dist}}$ whose execution spans multiple compute nodes, we add $t_{\text{dist}}$’s corresponding vertices and edges into the dependency graphs of the related nodes. Although the inter-transaction dependencies related to $t_{\text{dist}}$ are resolved by the resulting dependency graphs, $t_{\text{dist}}$’s intra-transaction dependencies among its inter-node actions are not captured by the dependency graphs due to the distribution. To address this issue, we define an additional type of dependency, namely node dependency, to track the dependencies among inter-node actions of the same transaction.

**Definition 5** Node Dependency. A node dependency exists between vertex $\alpha_i$ and vertex $\alpha_j$, denoted as

$$\alpha_i \succ_{\text{dist}} \alpha_j$$

if and only if $\alpha_i \succ_{\text{logical}} \alpha_j$ and $\alpha_i, \alpha_j$ are executed in different compute nodes.

In a distributed system, it is important to take advantage of data locality to optimize system’s overall performance. The ideal case is that each compute node works independently. As batching strategy is considered as a high priority in DistDGCC, it is easy to separate local transactions from distributed transactions during the dependency graph construction. In the implementation, local transactions in each batch are executed independently following the original DGCC [45] to achieve high parallelism both among cluster and in a single node. For distributed transactions with node dependency, DistDGCC distributes the vertices to relevant nodes according to the data locality in a batch manner, which reduces the number of network messages. We introduce a **Distributed Graph Coordinator** in each node to help distribute and receive vertices with dependency information to relevant compute nodes. Along with this information, some meta-data (e.g., node ID) are also sent to resolve temporal dependency among vertices from different compute nodes. After receiving all vertices in one batch, the Distributed Graph Coordinator is responsible for constructing a new dependency graph for those vertices.

Figure 4a illustrates a running example on two compute nodes. A batch of transactions $t_{11}, t_{12}, t_{13}, t_{14}, t_{15}$ runs on the node 1. $t_{11}$ and $t_{12}$ are local transactions, and the rest
are distributed transactions. During the contention resolution phase, each node constructs the dependency graphs for local transactions and distributed transaction, respectively. The dependency graph of all local transactions contains logical dependency and temporal dependency. Nodes are not required to communicate with each other for the execution of local transactions. However, node dependency may exist in distributed transactions that cannot be executed locally. Taking \( t_{13} \) as an example, it reads record \( B \) and updates records \( C \) and \( F \). The challenge is that \( t_{13} \) does not know any information about \( F \), and thus it is hard to resolve the conflicts among transactions from other nodes. Therefore, before the transaction execution, the Distributed Graph Coordinator partitions the original dependency graph along the node dependency edges and distributes subgraphs to relevant nodes. As shown in Fig. 4b, node 1 sends the yellow vertices along with their relevant edges to node 2. The Distributed Graph Coordinator on node 2 collects the subgraph and detects the temporal dependency relations between the local subgraphs. Then it builds a global dependency graph for all distribute transactions that touch data on node 2. The procedure of DistDGCC is summarized in Algorithm 3, and the system architecture of DistDGCC is shown in Fig. 5.

**Algorithm 3** Transaction management with DistDGCC

1: Input: transaction set \( \mathcal{T} \)
2: Initialize an empty local dependency graph \( \mathcal{G}_{\text{local}} \)
3: Initialize an empty distributed dependency graph \( \mathcal{G}_{\text{dist}} \)
4: for \( t \in \mathcal{T} \) do
5: if \( t \) is local transaction then
6: dependencyGraphConstruction(\( t, \mathcal{G}_{\text{local}} \))
7: else
8: dependencyGraphConstruction(\( t, \mathcal{G}_{\text{dist}} \))
9: coordinator.send(\( \mathcal{G}_{\text{dist}} \))
10: coordinator.receive()
11: dependencyGraphExecution(\( \mathcal{G}_{\text{local}} \))
12: dependencyGraphExecution(\( \mathcal{G}_{\text{dist}} \))

**Correctness** The dependency graph \( \mathcal{G} \) constructed by DistDGCC works as a schedule \( \eta \) for transaction set \( \mathcal{T} \). We need to prove that DistDGCC guarantees conflict serializability for transaction set \( \mathcal{T} \).

We first define the conflict graph for transaction set \( \mathcal{T} \).

**Definition 6** (Conflict Graph) Given the dependency graph \( \mathcal{G} = (V, E) \), let \( \eta \) be its schedule. The conflict graph \( G(\eta) = (V, E) \) of \( \eta \) is defined by

\[
V = \mathcal{T} \quad \text{and} \quad E = \{(t_i, t_j) | i < j \quad \text{and} \quad \exists \alpha_i, \alpha_j \in \mathcal{V}, \alpha_j \succ_{\text{temporal}} \alpha_i\}
\]

According the definition, each vertex in the conflict graph represents a transaction. Edge \( (t_i, t_j) \) exists only when record action in \( t_j \) temporally depends on record action in \( t_i \).

**Lemma 1** If the conflict graph \( G \) of a schedule \( \eta \) has no cycles, schedule \( \eta \) is conflict-serializable.

**Proof** We prove Lemma 1 by induction on the number of vertices \( n \) in the conflict graph \( G \).

Base case \( n = 1 \): There is only one transaction and the schedule \( \eta \) is a schedule of one transaction. Obviously, \( \eta \) is conflict-serializable. So Lemma 1 holds when \( n = 1 \).

Inductive hypothesis: Suppose Lemma 1 holds for all values of \( n \) up to \( k \) and \( n \geq 1 \).

Inductive step: Now, we consider the conflict graph consisting of \( k + 1 \) vertices. If this conflict graph has no cycles, there exists at least one vertex \( t_i \) that has no incoming edges. Any record action in \( t_i \) does not depend on record actions from the rest \( k \) transactions. Otherwise, \( t_i \) has incoming edges due to the definition of conflict graph. Thus, schedule \( \eta(k + 1) \) for the \( k + 1 \) transactions is conflict equivalent to \( \{t_i, \eta(k)\} \), where \( \eta(k) \) is the schedule for the rest \( k \) transactions. By the induction hypothesis that \( \eta(k) \) is conflict-serializable, we can conclude \( \eta(k + 1) \) for \( k + 1 \) transactions is still conflict-serializable and Lemma 1 holds for \( n = k + 1 \).
Scaling distributed transaction processing and recovery based on dependency logging

**Fig. 5** System architecture of DistDGCC protocol

**Lemma 2** The dependency graph $\mathcal{G}$ constructed by DistDGCC has no cycles.

**Proof** According to our previous definition of dependency graph, the dependency relation in the dependency graph $\mathcal{G}$ should be either temporal dependency $\succ_{temporal}$ or logical dependency $\succ_{logic}$ (noted that node dependency $\succ_{dist}$ is a special kind of logical dependency $\succ_{logic}$).

Since logical dependency $\succ_{logic}$ only handles dependency relations within one transaction, it will not lead to any cycle.

Let us consider temporal relation $\succ_{temporal}$. If there is a directed edge from $a_i$ to $a_j$, then $c(a_i) < c(a_j)$ where $c(a_i)$ and $c(a_j)$ represent transaction timestamps. If the dependency graph has one cycle, we can always find the edge path $(a_{i_0}, a_{i_1}, a_{i_2}, \ldots, a_{i_{v-1}}, a_{i_v})$, where $c(a_{i_0}) < c(a_{i_1}) < \cdots < c(a_{i_{v-1}}) < c(a_{i_v}) < c(a_{i_0})$. Obviously, this violates the initial condition, namely $c(a_{i_0}) < c(a_{i_v})$. Thus, we can conclude the dependency graph has no cycles.

**Theorem 1** DistDGCC guarantees conflict serializability.

**Proof** The dependency graph constructed by DistDGCC works as a schedule $\eta$ for a set of transactions. Based on Definition 6 and Lemma 2, the conflict graph $G(\eta)$ of the schedule $\eta$ has no cycles. Thus, the schedule $\eta$ is conflict-serializable according to Lemma 2. Hence, we conclude that DistDGCC guarantees conflict serializability.

### 5 Dependency logging

Fault tolerance and recovery are important to database systems which have to guarantee the ACID (Atomicity, Consistency, Isolation, Durability) properties. We adopt the log-based fault-tolerant scheme, which is a tried-and-tested approach in database literature. Operationally, it generates transaction logs at runtime and performs recovery upon system failure based on the logs. Specifically, each transaction saves the recovery-oriented information into logs along its execution and then flushes the logs to persistent storage before it commits. When the system failure occurs and the recovery process is launched, partially processed transactions need to be undone and committed transactions need to be redone according to the materialized logs. Generally, two aspects of performance are concerned in the design of log-based fault tolerance: One is for runtime logging, and the other is for failure recovery. On the one hand, transactions that construct their logs at runtime definitely introduce performance overhead, which affects the system throughput and processing latency. Therefore, optimized designs aim to make runtime logging as efficient as possible. On the other hand, undoing and redoing transactions during failure recovery typically incur a significant system downtime. Toward better system utilization, the impact of system downtime ought to be reduced in terms of the affected nodes and the checkpointing interval.

While the high-level idea of log-based fault-tolerant scheme for distributed OLTP systems is straightforward, effi-
ciently supporting low-overhead logging at runtime and fast recovery upon system failure is nontrivial, especially with the traditional fault-tolerant schemes, e.g., ARIES logging [29]. ARIES logging is widely adopted and is considered a "heavy-weight" logging approach, as it generates fine-grained logs to describe how data records have been changed by the transactions. Since all the recovery-oriented information is stored in the log records, its log structure is complicated, which usually takes more CPU cycles to generate. Also, more disk I/Os are required to flush the logs to disk, further affecting the system runtime performance. While ARIES logging supports independent recovery when system failure happens, it is hard to exploit parallelism during the recovery due to its sequential log processing.

To maximize the performance gain of DistDGCC and optimize failure recovery process, a new type of dependency logging is proposed based on dependency information generated during transaction processing. Dependency logging is a variant of write-ahead logging. It not only logs how the data are changed, but also logs the dependency information among the updated data. We therefore propose to consider transaction management and fault tolerance within the same framework. Specifically, the dependency graphs that are constructed during transaction processing can be reused and transformed to the log records with little overhead. The dependency graphs used for concurrency control are at record level, which may generate a large amount of log data. To further improve the logging efficiency, we then propose a coarse-grained optimization. We distinguish them as fine-grained dependency logging and coarse-grained dependency logging, respectively, and detail them in subsections that follow.

5.1 Fine-grained dependency logging

Traditional logging schemes usually generate log records during the runtime that incurs additional cost to create and maintain log data structures. In order to generate log records more efficiently, fine-grained dependency logging reuses the dependency graphs derived for concurrency control at the runtime. The data structure of log record written for each record action is shown in Fig. 6. Most of the fields in the log structure are also maintained in the dependency graphs, e.g., the incoming and outgoing edge list. As a consequence, the fine-grained dependency logging saves many CPU cycles for generating log records.

Fine-grained dependency logging writes a unique Logical Sequence Number (LSN) for each log record. An active transaction table is maintained on each node to help track the LSN of the last flushed log record for each transaction. Besides, each log record also contains the operation function name and its relevant parameters, with which the system can restore the data record to a correct state. Each fine-grained dependency logging record maintains the change information for one data record that can be uniquely referred by the (table-name, primary-key) pair.

In each log record, fine-grained dependency logging also keeps track of the dependency information, incoming and outgoing edges, with which fine-grained dependency logging can support on-demand recovery and further improve the recovery parallelism on the failed node. As shown in Fig. 7a, when a failure occurs, ARIES logging first loads the logs from disk to memory and redoes log records generated by committed transaction in sequence. It is hard for ARIES logging to exploit more parallelism in the redo process, since there is little information that can be used to resolve the possible conflicts. In this example, ARIES logging has to redo the six log records in order to recover the database to the correct state. By saving the dependency information in each log record, the fine-grained dependency logging first loads the log records from disk to memory and rebuilds the dependency structures among log records. Then the redo process can be executed in parallel. As the example shown in Fig. 7b, four threads can work in parallel to finish the recovery. Moreover, fine-grained dependency logging can further reduce the downtime compared to other logging approaches. For example, system with ARIES logging cannot start to execute new transactions before the failed node is fully recovered. By rebuilding the dependency relations for the committed transactions in advance, the system with fine-grained dependency logging can execute a newly arrived transaction immediately.

\[ \text{Checksum} \mid \text{LSN} \mid \text{Operation Function} \mid \text{Transaction Id} \mid \text{Partition Id} \mid \text{Table Name} \mid \text{Primary Key} \mid \text{Params} \mid \text{Dist-Flag} \mid \text{Incoming edge list} \mid \text{Outgoing edge list} \mid \text{Before Image of Updated Columns} \mid \text{After Image of Updated Columns} \]

Fig. 6 Fine-grained dependency logging record structure

Fig. 7 Example of dependency logging. a ARIES logging. b Dependency logging
The newly arrived transaction first recovers all its dependent data and then does the execution, which reduces the system downtime.

The field Dist-Flag in fine-grained dependency logging’s record structure indicates its locality attribute, and it distinguishes the fine-grained dependency logging records into two classes: local dependency logging record and remote dependency logging record. The log records produced by local transaction are all local dependency logging records. For a distributed transaction that has a coordination node, the log records maintained on the coordination node are local dependency logging records and the rest are remote dependency logging records. Like ARIES logging, the remote dependency logging record also stores the data images before and after the change, with which the dependency relation among different nodes is resolved. While this design increases the size of remote dependency logging record, it improves the recovery efficiency. More details are discussed in Sect. 6.

### 5.2 Coarse-grained dependency logging

While fine-grained dependency logging increases parallelism and improves the efficiency during the recovery, it generates record-level log records that not only increases the log size, but also takes more CPU cycles and incurs more number of disk I/Os. Hence, the system runtime performance with fine-grained dependency logging could be affected, especially when transactions touch more number of tuples. Toward better runtime performance, we propose the coarse-grained dependency logging strategy to reduce the log size. Instead of tracking the dependency information among updated data records, coarse-grained dependency logging tracks dependency information among transactions by parsing the record-level dependency graphs. Intuitively, it needs to traverse the whole dependency graph to complete the transformation, which restricts the performance, especially when the dependency graph is large. Instead, coarse-grained dependency logging does the transformation during the dependency graph construction. Specifically, if a temporal dependency edge is inserted between record actions $\alpha_i$ and $\alpha_j$ where $\alpha_i \in t_p$ and $\alpha_j \in t_q$, an edge should be also added between $t_p$ and $t_q$ in the dependency transaction graph. As shown in Fig. 8, in the dependency graph construction phase, there are three transactions $t_1$, $t_2$ and $t_3$. When $t_2$ is parsed, a temporal dependency edge should be inserted into the dependency graph. Similarly, there should be an edge from $t_1$ and $t_2$ in the transaction dependency graph.

The coarse-grained dependency logging simply tracks how transaction works and all its dependency relations. Besides the transaction information, each coarse-grained dependency logging record also contains the dependency information among other transactions. Each log record that is written out for each transaction has the structure as shown in Fig. 9. Like the structure of the fine-grained dependency logging record, there is also a Dist-Flag field here that indicates whether the transaction is distributed transaction or not. For all local transactions, the fields “Before Image of Updated Columns” and “After Image of Updated Columns” are empty. Otherwise, the changed data image should be saved in the log record. Unlike traditional Command logging that only stores a single log entry for distributed transaction (usually on its coordination node), coarse-grained dependency logging creates log entries on all its participating nodes. Instead of recording all the changes of the distributed transaction, each log record only saves recovery-oriented information where it locates. By doing this, dependencies among nodes can be resolved and independent recovery can be achieved. Compared to fine-grained dependency logging, coarse-grained dependency logging not only keeps all its strengths but also reduces the log size, e.g., the number of log record in Fig. 7 can be reduced to 2.

### 5.3 Dependency log recording

As discussed in Sect. 4, DistDGCC executes local transactions and distributed transactions separately even in one batch of transactions. It is simple to write the dependency log records for local transactions, since each node does the

```
t_1 = \{\text{update}(A), \text{update}(B)\} \quad t_2 = \{\text{update}(A), \text{update}(C)\} \quad t_3 = \{\text{update}(B), \text{update}(D)\}
```

![Fig. 8 Dependency graph transformation](image)

![Fig. 9 Coarse-grained dependency logging record structure](image)
Committed Transactions:
\[ t_1 = \{\text{update}(A), \text{update}(B), \text{update}(C)\} \]
\[ t_2 = \{\text{update}(A), \text{update}(B), \text{update}(D)\} \]

Newly Arrived Transaction:
\[ t_3 = \{\text{update}(A), \text{update}(C)\} \]

Fig. 10 Single-node recovery from dependency logging. (a) Fine-grained dependency logging records. (b) Coarse-grained dependency logging records.

execution independently. When all the local transactions are processed, system writes all the dependency log records to the disk at one time to maximize the performance. As all data can be fetched locally, each dependency logging record mainly saves dependency information, specifically the incoming and outgoing edges either among log records or transactions. For distributed transaction, the stored information of local dependency logging record differs from that of remote dependency logging record. Both local and remote log records write incoming and outgoing edges. For remote dependency logging record, it also writes the data images before and after the change.

To further improve the performance of our proposed dependency logging, we introduce two optimizations.

5.3.1 Log compression

Since the dependency graphs constructed during the runtime are directed graphs, it is sufficient to store such edge information in one column instead of two columns. Like the example shown in Fig. 7b, the dependency logging record 0 contains the information that its outgoing edge is to record 3. Record 3 also knows that its incoming edge is from record 0. Hence, dependency logging can remove the field “Incoming edge list” from the log structure to reduce the log size.

5.3.2 Batch-oriented optimizations

Group commit that groups multiple log records and flushes to disk at one time is already widely adopted. It reduces the number of disk writes and thus improves the logging performance. However, systems that execute transactions individually usually write log data to a shared log file, simplifying the recovery at the expense of higher logging overhead. In DistDGCC, transactions are executed in a batch manner and both fine-grained and coarse-grained dependency loggings rely on dependency information rather than commit sequence to perform the recovery. Hence, each worker thread maintains an individual log buffer to avoid lock contention.

6 Recovery

When a system performs failure recovery, system consistency and availability are the two main concerns. While traditional logging approaches (e.g., ARIES logging) have proven guarantee of durability, systems using such logging approaches inevitably incur longer system downtime during recovery, which significantly degrades the system availability (even replication-based approaches provide higher availability in database cluster, they still suffer from the whole cluster failure). In contrast, the proposed dependency logging not only guarantees the durability, but also provides a better availability during recovery. We justify the above claim by first looking into how single-node failure is handled under the dependency logging scheme and then extending the discussion to the case of cascading failure.

6.1 Recovery of single-node failure

The recovery for the dependency logging starts by reloading the latest database snapshot from the disk. After which, the system reloads log files and restores necessary data structures (e.g., indexes). Since the dependencies among committed transactions are tracked by the dependency logging, redoing these transactions can be performed by replaying their execution based on the corresponding rebuilt dependency graphs. Compared to coarse-grained dependency logging in which each committed transaction has only one log record, fine-grained dependency logging records of a transaction may not be fully written to the disk. Thus, the system needs to first remove the log records from the incomplete transaction before starting the recovery process, which is like the undo operation for ARIES logging. As such recovery process is equivalent to the normal execution phase, the system can still accept new transactions during recovery. That is, the redoing transactions form a batch which is processed first, and the newly arrived transactions form more batches which will be processed subsequently. However, this does not lead to any improvement on system availability, since newly arrived transactions cannot be executed before all committed transactions are totally redone. Instead, the system executes newly arrived transactions sequentially (i.e.,
enforcing batch size to be 1). In order to guarantee the consistency, the newly arrived transaction is executed only after all its dependent data are recovered. As shown in Fig. 10, node 1 encounters a failure after \( t_1 \) and \( t_2 \) commit. By recovering data record \( A \) and \( B \) to the correct state, the newly arrived transaction \( t_3 \) can be executed without waiting for the whole database to be fully recovered. This on-demand recovery mechanism enables normal transaction processing and recovery work in parallel, which reduces the downtime for failure and improves the system availability. When the failed node is fully recovered, the system increases the batch size that optimizes the overall performance.

The example in Fig. 10 illustrates the trade-off between fine-grained and coarse-grained dependency logging. While coarse-grained dependency logging usually achieves better runtime performance due to smaller log size, it sacrifices some degree of parallelism for the recovery and usually leads to a higher latency for the transaction that is processed during the recovery.

6.2 Recovery of cascading failure

In distributed environment, cascading failures are common. There are two different cases for cascading failures in a cluster. One is that a new failure happens on the failed node before the completion of its recovery. In this case, if there are no newly arrived transactions, the system can handle such cascading failure by simply restarting the recovery on the failed node. Otherwise, the next recovery should also consider those new generated log records. The second case is that a system failure happens on another node during the recovery. Although dependency logging is a form of logical logging approach, the dependency relations between different nodes are already resolved by saving data images in remote logging records, and hence, it facilitates independent recovery of failed nodes; cascading failure is therefore also handled as a result.

7 Evaluation

In this section, we first evaluate the performance of DistDGCC against 2PL and OCC. Then we evaluate the performance of dependency logging in both runtime and recovery phase.

7.1 Experimental setup

**Evaluation environment** We run DistDGCC and all logging experiments with our system which is implemented with 18,151 lines of C++. To evaluate the performance of DistDGCC, we compare it against 2PL and OCC which are originally implemented in an open-source DBMS [47]. To enable a fair comparison, we modified the codes to use the same storage and network system as our implementation. The performance of dependency logging is evaluated against ARIES logging both in runtime and in recovery. System failure is simulated by killing the daemon process on a node, and the recovery process is then invoked immediately. All the experiments are conducted on an in-house cluster of 8 nodes. Each node has an Intel(R) Xeon(R) 1.8 GHz 4-core CPU, 8GB RAM and 500GB HHD.
Benchmarks We adopt two popular benchmarks, namely YCSB [5] and TPC-C,² to conduct the evaluations. YCSB transaction performs 10 mixed read/write operations, and the key access follows the Zipfian distribution. To generate both low contention and high contention workloads, we set the Zipfian parameter to 0.6 and 0.8, respectively. TPC-C is adopted to simulate a real and complete order-entry environment. TPC-C workload mixes five kinds of transactions: New-Order (44%), Payment (45%), Delivery (4%), Order-Status (4%) and Stock-Level (3%).

7.2 DistDGCC evaluation

In this section, we evaluate the performance of DistDGCC against 2PL and OCC on both single node and cluster. Evaluation on a single node Figure 11a, b shows the throughput of the three protocols on a single node with YCSB workload. In summary, DistDGCC shows the best performance in both low contention and high contention settings. The performance gain mainly comes from the separation of contention resolution and transaction execution, which reduces the time on thread blocking. Moreover, the acyclicity of the dependency graph eliminates transaction aborts due to contentions and further improves the computation efficiency. 2PL also shows good scalability in the low contention setting. However, its performance drops with increasing contentions due to the higher cost of lock acquisition and deadlock resolution. Compared to DistDGCC and 2PL, OCC performs the worst, since it resolves conflicts during a validation phase using timestamps, which are usually assigned by a centralized component and may result in a performance bottleneck. Moreover, the aborted transactions waste the computation resources and also incur higher penalty. Figure 11c shows the results with TPC-C workload that contains 1 warehouse. In this scenario, the contention rate is high, since all the NewOrder and Payment transactions need to update the data record in the warehouse table. As DistDGCC is more resilient to the workload contentions, it still exhibits superiority over 2PL and OCC.

In Fig. 12, we compare the latency of the three protocols running on a single node. When the contention rate is low, the latency of the three methods varies within a small range. However, the latency of 2PL and OCC increases with increasing contentions, because 2PL needs to spend more time on acquiring the locks, while high contention workload leads to more transaction aborts in OCC. In contrast, the latency of DistDGCC is mainly affected by the batch size.

Evaluation on a cluster Figures 13 and 14 show the performances of the three protocols on an 8-node cluster by varying the percentage of distributed transaction. As shown in Figs. 13a and 14a, the throughput of 2PL and OCC is affected significantly by distributed transactions. Even a small portion of distributed transactions leads to a dramatical performance loss. Compared to a local transaction, a dis-

---

² http://www.tpc.org/tpcc/.

---
Scaling distributed transaction processing and recovery based on dependency logging

Fig. 15 Effects of batch size on an 8 nodes cluster. a Throughput on YCSB. b Latency on YCSB. c Throughput on TPC-C. d Latency on TPC-C

Distributed transaction accesses data records in multiple nodes and incurs extra network cost. More importantly, the worker thread has to be blocked until the distributed transaction commits or aborts, which degrades the computation utilization and hence leads to the performance loss. Since DistDGCC processes transactions in a batch manner, network messages for one batch of transactions are aggregated, which leads to an improved network performance. Moreover, worker threads will not be blocked during the processing. Instead, they construct the dependency graphs for the next batch of transactions, and the computation utilization is improved as a result. Hence, DistDGCC exhibits good performance superiority over 2PL and OCC in the distributed setting. While DistDGCC shows good robustness to the percentage of distributed transactions, its performance also drops as more distributed transactions are involved due to the fact that more network messages are generated.

In Figs. 13b and 14b, we show the latency variations. The latency of both 2PL and OCC increases significantly when there are more distributed transactions. The reason are twofold. First, a distributed transaction is more expensive and always incurs higher latency. Second, a worker thread has to wait until the distributed transaction commits or aborts and thus increases the latency for those blocked transactions. DistDGCC reduces the network overheads by aggregating network messages and also avoids long waiting time caused by thread blocking. Consequently, its latency increases only slightly with the increasing percentage of distributed transactions.

Effects of batch size The effects of batch size on the throughput and latency for DistDGCC are shown in Fig. 15. We fix the number of threads on each node to 8. As shown in Fig. 15a, c, the throughput of DistDGCC first increases with the batch size due to better exploitation of computation resources. It subsequently reaches a plateau since computation resources on each node are limited. For the same reason, as illustrated in Fig. 15b, d, the latency increases almost linearly with the batch size.

7.3 Dependency logging evaluation

In this subsection, we evaluate the efficiency of dependency logging through the comparison against Aries and Command logging. For ease of explanation, we shall use the following representative names.

- No logging—disable the logging during the runtime.
- DPLogging-F—fine-grained dependency logging approach proposed in Sect. 5.1.
- DPLogging-C—coarse-grained dependency logging approach proposed in Sect. 5.2.
- Aries logging—ARIES logging approach.
- Command logging—Command Logging approach.
Runtime evaluation  We shall first evaluate the overhead of the proposed dependency logging approach during the runtime. Two kinds of workloads are used: One workload only contains local transactions, while the other workload contains both local and distributed transactions.

Figure 16a, c shows the throughputs of the five logging approaches for the workload that only local transactions are involved. When the number of worker threads is small, all these logging approaches achieve similar throughputs, since disk I/Os caused by logging do not cause the performance bottleneck. As more worker threads are adopted, Aries logging takes more time to construct the logs and results in more disk I/Os. Hence, its throughput increases slower than the other four approaches. Compared to the ideal case that is represented as the No Logging approach, dependency logging and Command Logging achieve comparative performances. Unlike Aries logging that generates one log record for each updated data record, DPLogging-C and Command logging create one log record for each transaction. Moreover, each log record only tracks the transaction information instead of the updated data records. Thus, DPLogging-C and Command logging generate much less log data and reduce the number of disk I/Os. While DPLogging-F adopts a fine-grained logging strategy like the Aries logging, it reduces the number of log records using the dependency information during the runtime. If DPLogging-F saves data image for one update in its log record, there is no need to generate log records for the previous updates on the same record.

Figure 17a, c shows the results for the workload when distributed transactions are involved. In this set of experiments, we fix the number of worker threads on each node to 8, while we vary the percentage of distributed transaction from 0% to 100%, to study the effects of logging and network communication on the system performance. DPLogging-C, DPLogging-F and Command logging show comparative performance as the No Logging approach. The gaps between Aries logging and the other approaches become narrower as more distributed transactions are involved, since the extra network cost lightens the effect of logging. Even when all transactions are distributed transactions, DPLogging-C and DPLogging-F still achieve 1.4X and 1.8X higher runtime performance than that of Aries logging with YCSB workload and TPC-C workload, respectively.

Figure 16b, d, 17b, d show the effects of logging approaches on latency. While the average latency of DPLogging-C and DPLogging-F is slightly higher than that of No Logging, they are much lower than that of Aries logging in both local and distributed settings. In pure local transaction processing, the latency of all approaches changes within a small range. In the distributed setting, the latency of DPLogging-C and DPLogging-F increases with the percentage of distributed transactions, since they are mainly dominated by the network communication cost. However, the latency of Aries logging
Scaling distributed transaction processing and recovery based on dependency logging

Fig. 17 Effects of logging with workload mixed of local and distributed transactions. a Throughput on YCSB. b Latency on YCSB. c Throughput on TPC-C. d Latency on TPC-C

appears fairly stable irrespective of the percentage of distributed transactions, because its logging cost dominates its latency.

Dependency logging comparison Compared to fine-grained dependency logging that directly transforms the used dependency graphs into log records, coarse-grained dependency logging reduces the number of log records and thus further reduces the log size. In previous experiments with YCSB workloads, each transaction touches only 10 tuples. Hence, the log size reduction is not that significant (the log size of fine-grained dependency logging is about 2 times bigger than that of coarse-grained dependency logging). Coarse-grained dependency logging shows its superiority when transaction touches more tuples. As shown in Fig. 18, the performance of coarse-grained dependency logging is 1.2 times higher than that of fine-grained dependency logging, when each transaction touches 50 tuples.

Cost analysis To study the overhead of different methods, we conduct a cost analysis using the YCSB workload with 8 threads. In Fig. 19, the cost is presented as the percentage of CPU cycles. We measure the percentages of CPU utilization contributed by the individual components at runtime. As can be seen, most of the CPU cycles are spent in resolving and executing transactions (i.e., dependency graph construction and execution). Compared to Aries logging, the dependency logging approaches generate much less logs and hence incur less amount of disk I/Os. Moreover, Aries log-
Recovering takes much more CPU cycles to generate log records, which further degrades its performance. Both dependency logging approaches and Command logging take a small fraction of CPU cycles for logging, and thus they achieve comparative runtime performance.

**Recovery evaluation** We now evaluate the recovery performance with different logging approaches. We simulate the failure by killing the daemon process of one node after the system runs for 60 seconds, and then we measure the time span for recovering the failed node. Before the failed node starts to replay log records, it must first load the latest database snapshot into memory. Thus, the recovery time mainly consists of three parts: data loading, log loading, and replaying. As shown in Figs. 20 and 21, the time for data loading is almost the same for all logging approaches. Compared to DPLogging-C and DPLogging-F, Aries logging spends more time to load the logs into memory, since the log size of Aries logging is much larger.

When only one worker thread is enabled, Aries logging and Command logging perform slightly better than DPLogging-C and DPLogging-F. With only one worker thread available, all the four approaches replay their log records sequentially. However, Command logging, DPLogging-C and DPLogging-F need to re-execute functions instead of updating the database directly. As Aries logging saves data images before and after each update, read operations and transaction logics are not required to be redone during the recovery. Thus, the recovery time with dependency logging is comparative with Command logging, but is slightly higher than Aries logging. When 8 worker threads are enabled, the recovery performance of Aries logging and Command Logging drops significantly. In the runtime, each worker thread maintains a log buffer to avoid contention. However, in the recovery phase, all the log records in these log files have to be replayed one by one, since no dependency information can be used. Both DPLogging-C and DPLogging-F can achieve higher parallelism during the recovery by resolving the dependency relations among log records. DPLogging-F achieves the best recovery performance which is almost 5X faster than that of Aries logging. As shown in Fig. 20b, DPLogging-C achieves a comparative performance to DPLogging-F on YCSB workload. However, as shown in Fig. 21b, DPLogging-F is 2X faster than that of DPLogging-C on TPC-C workload. This is because each log record in DPLogging-C represents a transaction and most transactions update the data record in the warehouse table, restricting the re-execution parallelism as a result.
7.4 Overall performance evaluation

In this section, we evaluate the overall performance of our proposed DistDGCC and dependency logging. We run both YCSB and TPC-C workloads with 10% distributed transactions for 500 s on the 8-node cluster. After the systems run for 60 s, we kill the daemon process on a random node and invoke the recovery process.

Figure 22a, b summarizes the throughputs of the whole cluster. Command logging performs slightly better than the dependency logging approaches before failure occurs. However, when there is a failure, Command logging requires all the nodes to stop the processing and enter recovery mode. Moreover, committed transactions on all the nodes need to be replayed in a global order. As a consequence, Command logging incurs very high cost for recovery. When a failure occurs, the system with Aries logging cannot process transactions that access data on the failed node, and therefore aborts them. After the failed node is fully recovered, the throughput of the system returns to a normal level. Compared to 2PL with Aries logging and OCC with Aries logging, we observe that DistDGCC with Aries logging takes more time to complete the recovery process. This is because DistDGCC is more robust to distributed transactions and has a higher runtime throughput. Running for the same amount of time (60 s in the experiment), DistDGCC commits more transactions before the failure. Thus, it requires more time to replay the committed log records. For systems with dependency logging (both DPLogging-C and DPLogging-F), the failed node can be recovered according to the dependency graphs and process new incoming transactions at the same time. Thus, the throughput gradually increases as the failed node is being recovered. As discussed above, transactions in TPC-C workload contend on a small set of data records. Hence, system with DPLogging-C takes more time to recover than system with DPLogging-F.

In a nutshell, DistDGCC works more efficiently than OCC and 2PL in both standalone and distributed versions. By considering transaction management and recovery as one, dependency logging supports fast recovery with marginal runtime overhead. As a consequence, DistDGCC with dependency logging exhibits superior overall performance compared to the state-of-the-art techniques.
8 Related work

Concurrency control protocols High-efficient concurrency control protocol that ensures correct execution of concurrent transactions is vital for in-memory database systems. Two-phase locking (2PL) [8] and optimistic concurrency control (OCC) [22] are most widely adopted. As a pessimistic protocol, 2PL needs to acquire the lock before accessing a tuple and release it after transaction commits or aborts. With 2PL, conflict operations are resolved in advance and are executed in sequence. On the contrary, OCC assumes that conflicts are rare and does not check the conflicts during the transaction execution. Each transaction maintains read and write sets and conducts a conflict validation. Transaction commits only when the validation phase is passed; otherwise, it restarts or directly aborts. With the advancement of new hardware techniques, many research efforts have been devoted to improve the efficiency of concurrency control protocols.

In multi-version concurrency control (MVCC) [4,30], read operation does not block write operations. Hekaton [7,23] makes use of MVCC together with a lock-free hash table to improve its performance. Hyper [17,32] and BOHM [9] extend MVCC to enforce serializability and avoid shared memory writes for read tracking, respectively. For lock-based concurrency control protocols, the lock manager is typically very complex and incurs performance bottleneck. Light-weight Intent Lock (LIL) [21] introduces light-weight counter in a global lock table to ease the management. Very Lightweight Lock (VLL) [36] maintains a lock state with each tuple and removes centralized lock manager. However, LIL has to block transaction that cannot obtain all the locks and the performance of VLL is seriously affected by workloads that cannot be well partitioned. Silo [42] optimizes OCC by adopting a batched timestamp allocation. [19,20,44,48,49] show that OCC suffers for high contention workloads. They identify the bottlenecks and propose new approaches to improve its performance. However, all these optimizations mainly focus on multi-core systems rather than distributed systems. H-Store [16] and Hyper [17] adopt single-threaded model on partitioned databases to eliminate the overhead caused by concurrency control. However, their performance may suffer for workloads with more cross-partition transactions.

Fault-tolerant schemes ARIES [29] is the most widely used logging approach in traditional database systems. By maintaining data in memory, new recovery techniques [13, 14,24,51] were proposed, most of which inherit the idea from ARIES. Logical logging techniques [26,28] were proposed recently aiming to reduce the log size. While they improve the runtime performance by reducing the number of disk I/Os to an extent, they usually incur expensive cost for recovery, especially in distributed environment [46]. Many optimizations have also been proposed to increase the efficiency of logging and recovery. [25] makes use of shadow pages to reduce the log size during the runtime. [15] reduces the lock contention on the log buffers and the effects of context switching to improve the logging performance.

Recently, many research efforts have also been devoted to combine logging techniques with nonvolatile memory (NVM). [34] attempts to reduce the number of writes on NVM, and [43] introduces a distributed logging protocol on NVM by making use of group commit approaches. Write-Behind Logging [3] flushes changes of databases before flushing the logs by tracking where the databases are changed instead of how they are changed. As our design mainly considers using general commodity machines, it does not apply these techniques.

Replication-based techniques [33,40] provide fast recovery and high availability. However, these techniques incur expensive coordination cost during the runtime to achieve strong consistency. Moreover, given a limited amount of memory, it is expensive to maintain replicas, especially for very large databases. In most cases, replication-based approaches do not work for the whole cluster failure. Further, logging techniques are usually orthogonal to replication-based techniques.

9 Conclusion

In this paper, we have designed transaction management and logging within the same framework. We propose a distributed graph-based concurrency control protocol that reduces the cost for distributed transactions by aggregating network messages for a batch of transactions. Subsequently, we also propose a new dependency logging technique and associated fine-grained dependency logging and coarse-grained dependency logging approaches. By tracking the dependency relations among the transactions, dependency logging enables parallel recovery that helps to speed up the recovery and reduce the system downtime. Extensive experiments on both YCSB and TPC-C workloads confirm that system with dependency-based partitioning distributed transaction management and logging exhibits superiority in both runtime throughout and recovery performance.

Acknowledgements The work in this paper is supported by the National Research Foundation, Prime Ministers Office, Singapore, under its Competitive Research Programme (CRP Award No. NRF-CRP8-2011-08). The work is also in part supported by the Tencent-NUS Collaborative Research Grant. We would like to thank the anonymous reviewers for their insightful feedback that helped improve the paper.
References

1. Agrawal, R., Jagadish, H.: Recovery algorithms for database machines with non-volatile main memory. In: Database Machines, pp. 269–285 (1989)
2. Agrawal, R., Carey, M.J., Livny, M.: Concurrency control performance modeling: alternatives and implications. TODS 12(4), 609–654 (1987)
3. Arulraj, J., Perron, M., Pavlo, A.: Write-behind logging. PVLDB 10(4), 337–348 (2016)
4. Bernstein, P.A., Hadzilacos, V., Goodman, N.: Concurrency Control and Recovery in Database Systems. Addison-Wesley, Reading (1987)
5. Cooper, B.F., Silberschatz, A., Tam, E., Ramakrishnan, R., Sears, R.: Benchmarking cloud serving systems with YCSB. In: SoCC, pp. 143–154 (2010)
6. DeWitt, D.J., Katz, R.H., Olken, F., Shapiro, L.D., Stonebraker, M.R., Wood, D.A.: Implementation techniques for main memory database systems. SIGMOD 14(2), 1–8 (1984)
7. Diaconu, C., Freedman, C., Ismert, E., Larson, P.A., Mittal, P., Stonecipher, R., Verma, N., Zwilling, M.: Hekaton: SQL server’s memory-optimized OLTP engine. In: SIGMOD, pp. 1243–1254 (2013)
8. Eswaran, K.P., Gray, J.N., Lorie, R.A., Traiger, I.L.: The notions of consistency and predicate locks in a database system. Commun. ACM 19(11), 624–633 (1976)
9. Faleiro, J.M., Abadi, D.J.: Rethinking serializable multiversion concurrency control. PVLDB 8(11), 1190–1201 (2015)
10. Gray, J.: The transaction concept: virtues and limitations. VLDB 81, 144–154 (1981)
11. Hagmann, R.B.: Reimplementing the cedar file system using logging and group commit. In: SOSP, pp. 155–162 (1987)
12. Harizopoulos, S., Abadi, D.J., Madden, S., Stonebraker, M.: OLTP through the looking glass, and what we found there. In: SIGMOD, pp. 981–992 (2008)
13. Jagadish, H., Silberschatz, A., Sudarshan, S.: Recovering from main-memory lapses. VLDB 93, 391–404 (1993)
14. Jagadish, H.V., Lieuwen, D., Rastogi, R., Silberschatz, A., Sudarshan, S.: Dali: a high performance main memory storage manager. In: VLDB, pp. 48–59 (1994)
15. Johnson, R., Pandis, I., Stoica, R., Athanassoulis, M., Ailamaki, A.: Aether: a scalable approach to logging. PVLDB 3(1–2), 681–692 (2010)
16. Kallman, R., Kimura, H., Natkins, J., Pavlo, A., Rasin, A., Zdonik, S., Jones, E.P., Madden, S., Stonebraker, M., Zhang, Y., et al.: H-store: a high-performance, distributed main memory transaction processing system. VLDB 1(2), 1496–1499 (2008)
17. Kemper, A., Neumann, T.: Hyper: a hybrid OLTP and OLAP main memory database system based on virtual memory snapshots. In: ICDE, pp. 195–206 (2011)
18. Kernighan, B.W., Robinson, J.T.: On optimistic methods for concurrency control. TODS 6(2), 213–226 (1981)
19. Kim, K., Wang, T., Johnson, R., Pandis, I.: Ermia: fast memory-optimized database system for heterogeneous workloads. In: SIGMOD, pp. 1675–1687 (2016)
20. Kimura, H.: Foedus: OLTP engine for a thousand cores and NVRAM. In: SIGMOD, pp. 691–706 (2015)
21. Kimura, H., Graefe, G., Kuno, H.A.: Efficient locking techniques for databases on modern hardware. In: International Workshop on Accelerating Data Management Systems Using Modern Processor and Storage Architectures, pp. 1–12 (2012)
22. Kung, H.T., Robinson, J.T.: On optimistic methods for concurrency control. TODS 6(2), 213–226 (1981)
23. Larson, P.Á., Blanis, S., Diaconu, C., Freedman, C., Patil, J.M., Zwilling, M.: High-performance concurrency control mechanisms for main-memory databases. PVLDB 5(4), 298–309 (2011)
24. Lehman, T.J., Carey, M.J.: A concurrency control algorithm for memory-resident database systems. In: Foundations of Data Organization and Algorithms, pp. 489–504 (1989)
25. Li, X., Eich, M.H.: Post-crash log processing for fuzzy checkpointing main memory databases. In: ICDE, pp. 117–124 (1993)
26. Lomet, D., Tzoumas, K., Zwilling, M.: Implementing performance competitive logical recovery. PVLDB 4(7), 430–439 (2011)
27. Louie, R.H.Y., Li, Y., Vucetic, B.: Practical physical layer network coding for two-way relay channels: performance analysis and comparison. IEEE Trans. Wirel. Commun. 9(2), 764–777 (2010)
28. Malviya, N., Weisberg, A., Madden, S., Stonebraker, M.: Rethinking main memory OLTP recovery. In: ICDE, pp. 604–615 (2014)
29. Mohan, C., Haderle, D., Lindsay, B., Pirahesh, H., Schwartz, P.: Aries: a transaction recovery method supporting fine-granularity locking and partial rollbacks using write-ahead logging. TODS 17(1), 94–162 (1992a)
30. Mohan, C., Pirahesh, H., Lorie, R.: Efficient and flexible methods for transient versioning of records to avoid locking by read-only transactions. SIGMOD 21(2), 124–133 (1992b)
31. Mu, S., Cui, Y., Zhang, Y., Lloyd, W., Li, J.: Extracting more concurrency from distributed transactions. In: OSDI, pp. 479–494 (2014)
32. Neumann, T., Mühlbauer, T., Kemper, A.: Fast serializable multiversion concurrency control for main-memory database systems. In: SIGMOD, pp. 677–689 (2015)
33. Ongaro, D., Rumble, S.M., Stutsman, R., Ousterhout, J., Rosenblum, M.: Fast crash recovery in ramcloud. In: SOSP, pp. 29–41 (2011)
34. Pelley, S., Wenisch, T.F., Gold, B.T., Bridge, B.: Storage management in the NVRAM era. PVLDB 7(2), 121–132 (2013)
35. Pinheiro, E., Weber, W.D., Barroso, L.A.: Failure trends in a large disk drive population. FAST 7, 17–23 (2007)
36. Ren, K., Thomson, A., Abadi, D.J.: Lightweight locking for main memory database systems. PVLDB 6(2), 145–156 (2012)
37. Ren, K., Faleiro, J.M., Abadi, D.J.: Design principles for scaling multi-core OLTP under high contention. In: SIGMOD, pp. 1583–1598 (2016)
38. Roos, F., Lindah, S.: Distribution system component failure rates and repair times—an overview. In: NORDAC, pp. 23–24 (2004)
39. Schroeder, B., Gibson, G.: A large-scale study of failures in high-performance computing systems. TDS2 7(4), 337–350 (2010)
40. Stonebraker, M., Abadi, D.J., Batkin, A., Chen, X., Cherniack, M., Freireira, M., Lau, E., Lin, A., Madden, S., O’Neil, E., et al.: C-store: a column-oriented dbms. In: VLDB, pp. 553–564 (2005)
41. Tan, K., Cai, Q., Ooi, B.C., Wong, W., Yao, C., Zhang, H.: In-memory databases: challenges and opportunities from software and hardware perspectives. SIGMOD Rec. 44(2), 35–40 (2015)
42. Tu, S., Zheng, W., Kohler, E., Liskov, B., Madden, S.: Speedy transactions in multicore in-memory databases. In: SOSP, pp. 18–32 (2013)
43. Wang, T., Johnson, R.: Scalable logging through emerging non-volatile memory. PVLDB 7(10), 865–876 (2014)
44. Wu, Y., Chan, C.Y., Tan, K.L.: Transaction healing: scaling optimistic concurrency control on multicores. In: SIGMOD, pp. 1689–1704 (2016)
45. Yao, C., Agrawal, D., Chen, G., Lin, Q., Ooi, B.C., Wong, W.F., Zhang, M.: Exploiting single-threaded model in multi-core in-memory systems. TKDE 28(10), 2635–2650 (2016a)
46. Yao, C., Agrawal, D., Chen, G., Ooi, B.C., Wu, S.: Adaptive logging: optimizing logging and recovery costs in distributed in-memory databases. In: SIGMOD, pp. 1119–1134 (2016b)
47. Yu, X., Bezerra, G., Pavlo, A., Devadas, S., Stonebraker, M.: Staring into the abyss: an evaluation of concurrency control with one thousand cores. PVLDB 8(3), 209–220 (2014)
48. Yu, X., Pavlo, A., Sanchez, D., Devadas, S.: Tictoc: time traveling optimistic concurrency control. SIGMOD 8, 209–220 (2016)
49. Yuan, Y., Wang, K., Lee, R., Ding, X., Xing, J., Blanas, S., Zhang, X.: Bcc: reducing false aborts in optimistic concurrency control with low cost for in-memory databases. PVLDB 9(6), 504–515 (2016)
50. Zhang, H., Chen, G., Ooi, B.C., Tan, K.L., Zhang, M.: In-memory big data management and processing: a survey. TKDE 27(7), 1920–1948 (2015)
51. Zheng, W., Tu, S., Kohler, E., Liskov, B.: Fast databases with fast durability and recovery through multicore parallelism. In: OSDI, pp. 465–477 (2014)