RIOT: Replicated Independently-Ordered Transactions

Jim Webber
jim.webber@neo4j.com
Neo4j
London, UK

Georgios Theodorakis
gtheodorakis@nvidia.com
Nvidia
London, UK

Hugo Firth
hugo.firth@neo4j.com
Neo4j
London, UK

Natacha Crooks
ncrooks@berkeley.edu
University of California, Berkeley
Berkeley, CA, USA

Abstract

Consensus protocols such as Raft and Paxos implement state machine replication through a single leader that enforces a totally ordered log. While this simplifies correctness, it introduces sequential bottlenecks that restrict scalability. We present RIOT, a generalized consensus protocol that eliminates centralized leadership and log replication in favor of decentralized coordination over a directed acyclic graph (DAG) of entries. RIOT guarantees that all servers maintain a logically identical DAG that enforces ordering under contention, while allowing commutative operations to execute concurrently.

RIOT is motivated by our work on distributed graph databases, which must guarantee reciprocal consistency for edges that span shards. Unlike specialized transaction protocols, RIOT makes no assumptions about concurrency control or transaction models. It provides a replicated state machine abstraction that integrates cleanly with transactional databases, treating DAG entries as transaction placeholders. Both single-phase and two-phase variants are supported, ensuring atomic agreement on entries and their ordering constraints.

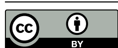
We integrate RIOT with Neo4j and evaluate it against Neo4j's production Raft implementation. For common workloads, RIOT delivers up to 2.5× higher throughput and 2.3× lower tail latency while matching exiting consistency guarantees. In doing so, RIOT demonstrates how consensus can be generalized to unlock scalability for transactional databases at scale.

CCS Concepts

• **Computing methodologies → Distributed algorithms**; • **Information systems → Data management systems**; *Network data models*; **Database transaction processing**; **Distributed database transactions**; **Parallel and distributed DBMSs**; • **Computer systems organization → Reliability**; **Redundancy**; **Availability**; *Fault-tolerant network topologies*; **Reliability**; *Availability*.

Keywords

Leaderless consensus, distributed transactions, graph databases



This work is licensed under a Creative Commons Attribution 4.0 International License.
SIGMOD Companion '26, Bengaluru, India
© 2026 Copyright held by the owner/author(s).
ACM ISBN 979-8-4007-2450-3/2026/05
https://doi.org/10.1145/3788853.3803094

ACM Reference Format:
Jim Webber, Georgios Theodorakis, Hugo Firth, and Natacha Crooks. 2026. RIOT: Replicated Independently-Ordered Transactions. In *Companion of the International Conference on Management of Data (SIGMOD Companion '26), May 31-June 05, 2026, Bengaluru, India.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3788853.3803094

1 Introduction

Graph databases [1, 27, 30, 40] have been widely adopted across industry and research. The labeled property graph (LPG) model [36] has become especially popular, offering a flexible representation that supports transactional as well as analytical workloads. Consequently, graph databases have become common in many domains, including transportation and route planning [45], social network analysis [4, 42], and AI applications that use retrieval-augmented generation (GraphRAG) [26].

Graph Model. The labeled property graph (LPG) represents data as nodes connected by named, directed relationships. Both nodes and relationships can store arbitrary key–value properties. These simple building blocks support the construction of high-fidelity networks to model complex application domains. Query languages such as Cypher [17] and the emerging ISO standard GQL [12] provide expressive and efficient mechanisms for graph querying.

Every relationship in the LPG model must have both a start node and an end node, and queries should be able to freely traverse relationships in either direction at equal cost. These invariants contribute significantly to the usability of the model. They also introduce a difficult correctness challenge for system implementers, specifically when data is partitioned across shards.

Leader-based protocols and scalability. Our experience operating a cloud-hosted graph database service called AuraDB [16] shows that graph workloads are read-dominant. Even update operations usually require graph traversal to identify the target subgraph before modifications are applied. As a result, graph workloads benefit substantially from vertical scaling on a single server, where the locality advantages of a unified memory space improve both traversal and update performance.

Single-server deployments, while efficient for graphs, lack the fault-tolerance required for high availability. A common next step is to replicate the single server using a leader-based consensus protocol such as Raft [34]. This is the approach employed by Neo4j today. With this approach, all updates are coordinated through a leader that enforces a global order of operations, thereby preserving graph invariants. Replication provides fault-tolerance by

maintaining multiple copies of the data and improves performance, but limits for the overall volume of stored data to the capacity of a single machine.

Leader-based replication protocols remain an active area of research, with advances spanning both software techniques [5, 7, 9, 19, 21, 33, 34], and systems that combine specialized hardware deployments [6]. While these developments improve performance and reliability, they do not resolve the inherent architectural limitation: the leader is a systemic bottleneck during normal operation and even more so under failure and recovery.

Leaderless protocols. The growing adoption of graph databases has driven demand for deployments at very large scale, with graphs containing trillions of relationships and requiring high transactional throughput [31, 41]. However, even with efficient algorithms and carefully engineered implementations, such workloads eventually exceed the capacity of single-leader architectures.

Leaderless designs aim to improve throughput by granting servers greater autonomy in processing operations. Several protocols have been proposed in this space, ranging from generalized crash-tolerant consensus protocols such as EPaxos [3, 28, 37, 38] to database-oriented approaches that integrate replication with concurrency control, such as Tapir [46], Janus [29], and Basil [39]. The common strategy is to reason about data access patterns: operations are analyzed in terms of their read and write sets to determine dependencies. Commutative operations may execute concurrently, while potentially conflicting writes are serialized to preserve correctness.

Unfortunately, this method is not well suited to graph databases which build their read and write sets dynamically during query execution as they traverse the graph. Knowing the full extent of a key set is therefore only possible *after* a query has completed its execution. This unpredictability is further compounded by significant variability: a graph query which encounters a dense subgraph can see an exponential rise in the number of keys accessed.

The Accord protocol [38] appears to offer a promising direction by representing dependencies between transactions as a DAG, thereby reasoning at a coarser granularity key accesses and reducing processing overheads. Unlike our approach, however, Accord still constructs the transaction DAG by analyzing key accesses prior to execution.

Sharding for graphs. Cloud hyperscalers have demonstrated that managing very large datasets and sustaining high-throughput workloads ultimately requires horizontal scalability, where data is partitioned across shards and all servers participate in processing both reads and writes. Partitioning graphs across shards, however, presents unique challenges.

The core difficulty arises when relationships span shards. Updates or deletions must be applied consistently across all affected partitions. For example, if Alice follows Bob, then Bob must also reflect that Alice follows him; if Alice unfollows Bob, the connection must be removed from both sides. This property, referred to as *reciprocal consistency* [8], is fundamental to correctness for graph databases. Systems that fail to enforce it risk partial or dangling relationships, a known data corruption issue in eventually-consistent graph stores such as JanusGraph [18]. Violating invariants not only renders queries non-deterministic, but also allow write-after-read operations to propagate corruption through the graph.

Our contribution. In this work, we propose RIOT, a replicated state machine protocol that departs from traditional leader-based designs. The protocol has four defining properties: (i) each server reasons about transaction dependencies and leverages its local history to contribute to a globally consistent ordering; (ii) leadership is decentralized, both within and across shards; (iii) consensus and replication are unified in a single mechanism; and (iv) coordination is minimized wherever possible.

We implement RIOT in Neo4j, where it manages large-scale OLTP graph workloads while enforcing reciprocal consistency. The system achieves redundancy with low coordination overhead and provides strong transactional guarantees, including serializability. Although motivated by the scalability and correctness challenges of distributed graph databases, RIOT generalizes naturally to a replicated state machine protocol suitable for a broad class of transactional systems, with the following beneficial characteristics:

(i) RIOT is a multi-shard leaderless consensus protocol, which provides high-level concurrency control, fault-tolerance, and scalability. Servers in RIOT curate a Directed Acyclic Graph (DAG) of entries that represent transactions and their histories, called the TxDAG. Any server in RIOT can act as a coordinator at any time providing scalability through concurrency. A coordinator attempts to form a majority with other participant servers to append a new entry into their TxDAGs. Compatible, safe orderings are computed by determining whether the inbound TxDAg entry's *leading edge* (the committed leaf nodes of the TxDAG at the originating server) are also present in the participant's local history.

By definition, a leading edge contains only entries that are committed by a majority. When a prepare message arrives at a participant, it can immediately commit any locally matching transactions that happen to be in the prepared state with no further coordination. Then, if the incoming leading edge is wholly present within the local history, the entry is considered compatible. Practically, when RIOT is integrated database systems, the final decision to prepare an entry whose leadign edge is compatible will be made by the underlying resource manager, where constraints in the data model and so forth will be verified.

When responding affirmatively to the coordinator, the participant provides its own leading edge, qualifying its vote to ensure the histories are mutually compatible (since their TxDAGs evolve independently). The bilateral exchange of qualified votes ensures safe convergence on a consistent TxDAG.

(ii) Implementation, integration, and benchmark of RIOT integrated with Neo4j to enable leaderless, multi-shard execution. RIOT replicates and orders entries in the TxDAG, where each entry represents database transactions consumed by Neo4j. Compared with Neo4j's existing Raft-based replication, RIOT achieves up to 2.5× higher throughput and significantly lower average latency under representative workloads. Furthermore, by eliminating leader elections, RIOT improves availability under failures, avoiding pauses and removing a major source of performance jitter.

The remainder of this paper is organized as follows. Sec. 2 provides background on consensus protocols and the challenges of graph data storage. Sec. 3 introduces the design of our protocol, including its ordering, consensus, and replication mechanisms. Sec. 4 describes integration with Neo4j, a well-known production graph

database. Sec. 5 presents an experimental evaluation comparing our implementation against a Raft-based baseline. Sec. 6 discusses related work, and Sec. 7 concludes and offers some thoughts on future work.

2 Labeled Property Graph: Data Model and Operations

RIOT was originally designed to improve the scalability and availability of Neo4j. Its design is informed by Neo4j's labeled property graph data model(LPG) model [36] that Neo4j upholds, which presents distinct challenges compared to traditional data models.

In particular, edges in the LPG model are bidirectionally traversable (at the same cost), and must be connected to a node at both ends. This is advantageous for users. It helps them to create high-fidelity models. However, it complicates distributed implementations particularly when edges connect nodes across different shards.

Graph data often follows power-law distributions, which means most operations exhibit low contention. Empirical analysis from Neo4j's cloud service (AuraDB [16]) confirms this: writes constitute less than 30% of daily queries, and write conflicts leading to aborts occur in only 1 out of every 700 transactions. Such characteristics suggest that protocols optimized for low-conflict execution can achieve substantial performance gains.

Contemporary Graph Database Architecture

Managing graph data requires a balance between performance and maintaining structural integrity. Existing approaches generally fall into three categories, each with distinct limitations for large-scale LPG workloads:

Leader-based Consensus. Graph databases, including Neo4j [30] and Amazon Neptune [1], use leader-based protocols over single partitions [43, 44]. Those systems ensure a safe total order, but the leader can become a bottleneck for updates and also causes availability stalls during elections.

Graph-NoSQL Systems. Systems such as JanusGraph [18] atop Cassandra [20] or HBase [13] potentially offer high throughput, but they fail to uphold reciprocal consistency across partitions [8]. This deficiency leads to dangling edges and irreparable data corruption in no-fault operation from write-after-read anomalies.

Distributed Relational Databases. Spanner Graph (built atop Spanner [6]) encodes graphs as relational tables, that are partitioned across shards. It coordinates updates with a mixture of (hardware-optimized) two-phase commit across shards and leader-based replication using MultiPaxos [22] within shards. Reciprocal consistency *is* upheld with this approach, but since 30% of relationships typically cross shards [10], the expensive two-phase commit path is commonly used.

3 RIOT Protocol

To address the challenges outlined in Sec. 2, we present RIOT, a novel leaderless consensus protocol originally developed as an extension to Neo4j [30]. Although it borrows familiar database terminology such as prepare and commit, RIOT is a general consensus protocol. When integrated with a transactional database, RIOT establishes a safe partial order of transactions on each server. When integrated with database systems, RIOT allows each Resource Manager (RM) to participate in decision making, using the RM's concurrency control and isolation policies to bias transaction outcomes.

Conceptually, RIOT shares some aspects of Raft [34] but eliminates centralized leadership, decides agreement on a DAG (called a *TxDAG*) rather than a sequential log, and allows replication operations to fail safely under fault or race conditions. Using a DAG preserves ordering where necessary while executing independent operations concurrently when possible. Accordingly, each server's history is *compatible* but not always strictly *identical*. RIOT is also able to support cross-partition coordination to maintain reciprocal consistency for graph edges that span shards.

Each server's *TxDAG* records its local history of *committed* and *prepared* transactions. Committed transactions appear as either leaf or internal nodes and are immutable. Prepared transactions are appended as leaf nodes which may be subsequently changed to committed or removed. We define the *leading edge* of the TxDAG as the set of committed nodes with no committed children. Intuitively, the leading edge represents the current version of the system.

Consensus begins when a client submits a transaction to any server. That server becomes the coordinator for that transaction. The coordinator binds the transaction to its current leading edge - transactions which have by definition been committed by a majority *before* the current transaction. It then sends a *PREPARE* message to other servers containing both the transaction payload (a Cypher query in our case) and the leading edge of the coordinator.

Upon receiving a *PREPARE* message, a participant first processes the incoming leading edge. Transactions in a leading edge have been committed by a majority, and so any locally prepared transactions that are present in the incoming leading edge can be immediately committed without further coordination. The participant then verifies that all transactions in the incoming edge exist in its own local TxDAG. If verification fails, the participant sends an *ABORT* message to the coordinator, preventing commits on top of divergent histories.

If the incoming edge passes verification, we say that it is *compatible*. The transaction is then added to the local TxDAG as a *prepared* leaf node with the current leading edge as its immediate ancestors. The participant then computes the set difference between its leading edge and that of the incoming transaction, known as the *qualifier*, formally defined as: qualifier $= \{tx \mid tx \in$ localhistory $\wedge\ tx \notin$ incomingleadingedge$\}$. The participant logic then connects the preparing transaction to the qualifier's transactions in the local TxDAG, making explicit the happened-before relationship that would otherwise be implicit. In a system where the coordinator and participant happened to have identical committed state, the qualifier will be empty.

When RIOT is used as part of a database system, the participant then invokes its local resource manager to validate the transaction. If no conflicts or constraint violations are detected, the prepare phase can continue, otherwise the participant can immediately undo its preparatory work and send and *abort* message back to the coordinator.

Should the participant pass the incoming leading edge check and the RM level checks, then it can vote to commit the transaction. It

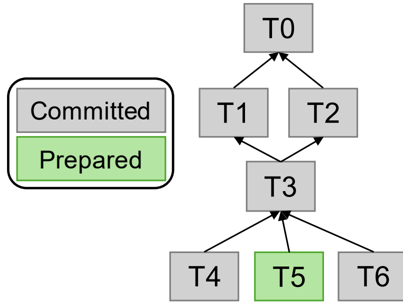


Figure 1: TxDAG example

sends a *PREPARED* message back to the coordinator, injecting the qualifier as its leading edge.

The coordinator receives the *PREPARED* message and performs the same checks as the participant on the incoming qualifier. First the coordinator checks the qualifier transactions exist in its history. It then attempts to link all entries in the qualifier to entries in its TxDAG to verify a mutually compatible history with the participant.

Once enough prepared messages have been validated, the coordinator either observes a supermajority of $\lfloor \frac{3n}{4} \rfloor + 1$ servers with no further work needed, or a simple majority of $\frac{n}{2} + 1$ servers. A simple majority triggers a second phase to inform participants of the outcome, via a *COMMIT* message with the majority-agreed set of ancestors or an *ABORT* message where the coordinator determines it is too divergentor lagging to act safely. The possible actions on receiving an outcome message from a coordinator are shown in Table 1.

	Global Commit	**Global Abort**
Local Commit	Commit transaction	Abort transaction
Local Abort	Potential divergence, consider recovery from other servers	Do nothing

Table 1: Local vs Global Transaction Outcomes

We now discuss the assumptions and operational model for RIOT (Sec. 3.1), followed by an explanation of its intended behavior both under contention and in the uncontended fast path (Sec. 3.3). We then explain aborts and retries (Sec. 3.4), catch-up (Sec. 3.5) and recovery (Sec. 3.6), support for multi-shard transactions (Sec. 3.7), and correctness (Sec. 3.8). Finally we explain how RIOT can support long-lived systems where cluster membership adapts to failures over time (Sec. 3.9).

3.1 System Model and Preliminaries

RIOT assumes a non-Byzantine system [25] consisting of a set of N identical servers, each with a unique identifier, communicating via asynchronous message passing. Messages may be delayed, and servers may progress at different speeds. The protocol makes no assumptions about shared memory or a global clock.

Servers may fail by crashing, but we assume that a majority $\frac{n}{2} + 1$ remains operational. For fast-path decisions (i.e., single round-trip consensus), a larger quorum of size $\lceil \frac{3n}{4} \rceil + 1$ is required to ensure that the decision taken as part of a fast quorum is reproducible such that all future decisions are consistent.

Each server maintains a local view of the system state, composed of a TxDAG and two sets: P and LE. Each TxDAG stores the history of transactions for a single server. Part of the TxDAG is the set $LE = \{i, j, k\}$ that represents the current leading edge. It consists of committed transactions that have no committed descendants. This set defines a frontier for accepting new transactions. Another set, $P = \{a \mapsto \{b, c, d\}, \ldots\}$, represents *prepared transactions*, where each transaction a depends on a set of ancestor transactions that are committed in the TxDAG.

Fig. 1 shows an example of a TxDAG. Each node represents a transaction, and edges indicate dependency relationships. In this example, `T0` was the earliest committed transaction. It became the leading edge for concurrent transactions `T1` and `T2` which both committed before `T3` began. `T3` then prepared and committed with `T1` and `T2` as ancestors. It subsequently became the leading edge for concurrent transactions `T4`, `T5`, and `T6`. `T4` and `T6` have been committed and constitute the current leading edge. `T5` remains in the prepared state (for now).

Servers exchange commands in the form ⟨`VERB`, `TID`, [`PREDICATE`]⟩, where `VERB` indicates the operation type – such as `PREPARE`, `COMMIT`, `ABORT`, or their corresponding responses (e.g., `PREPARED`). The `TID` is the transaction identifier,[1] and the optional `PREDICATE` encodes conditions such as required ancestors. For example, a prepare request might take the form `PREPARE` $\{3\},\ LE = \{0, 1, 2\}$, indicating that transaction 3 depends on transactions 0, 1, and 2. A successful response would be `PREPARED` $\{3\},\ LE = \{0, 1, 2\}$. Two commands are considered *non-commutative* (i.e., conflicting) if their results depend on the execution order; otherwise, they are commutative and can be executed in any order or even concurrently. The TxDAG captures the dependencies of commutative and non-commutative operations: if they commute, they are allowed to share the same ancestors, if not a majority will form to allow one to proceed while the other aborts.

RIOT aligns with the principles of Generalized Consensus [23]. It applies transactions in a globally safe order, satisfies *non-triviality* (only proposed transactions are committed), *stability* (committed transactions persist and extend prior decisions), *liveness* (proposed transactions are eventually decided), and *consistency* (servers converge on compatible histories).

3.2 Mutual History Compatibility Proof

We now show that when two servers exchange compatible leading edges, their committed histories must be identical. The reasoning follows from quorum intersection and the deterministic way servers build their TxDAGs.

Each server S maintains a TxDAG $G_S = (V_S, E_S)$, where edges record ancestor relationships declared at prepare time. The *leading edge* LE_S is the set of committed transactions with no committed children. New transactions are prepared against the coordinator's

[1] For clarity, we omit the details of read-write operations associated with the transaction, though in reality they are included in the command payloads.

leading edge. A participant accepts a new transaction only if that transaction's dependencies are present its own history. When a participant receives a leading edge, it can immediately commit any corresponding prepared records in its own TxDAG without further coordination since they must have been committed by a quorum elsewhere.

Lemma 1 – Quorum agreement

If a transaction commits at one server, then a majority of servers voted for it. Any other majority intersects this one, so no conflicting decision is possible. Fast-path quorums are larger still, intersecting with all majorities, and so their outcomes are also stable.

Lemma 2 – Closure under ancestry

We first define a *down-closure* as follows:

Definition 3.1. For a TxDAG $G = (V, E)$ and a set of transactions $X \subseteq V$, the *down-closure* of X is

$$\downarrow X \ = \ \{\, v \in V \mid \exists x \in X \text{ such that there is a path } v \rightsquigarrow x \,\},$$

All transactions v that are ancestors of some $x \in X$, together with X itself.

Now supposing servers A and B have mutually compatible leading edges (each leading edge appears in the other's history). Then the committed down-closures of those edges are the same at both servers. That is, if a transaction lies on a path to some $x \in \mathrm{LE}_A$, then the quorum that committed x also ensured those ancestors were present, so they must also appear at B. Repeating along the path gives equality of the closures.

Lemma 3 – Determinism of structure

Through the exchange of *PREPARE* and *PREPARED* messages, the coordinator and participating servers establish the necessary precondition for an entry to be inserted into the TxDAG: the ancestors in the transaction's leading edge must be present in the participant's TxDAG, and the transactions in the participant's leading edge must be present in the coordinator's TxDAG.

The parents of a transaction in a TxDAG act as a kind of version vector. All the dependencies — *happens-before* relationships — must be preserved across all servers. Since the TxDAG only concerns itself happens-before relationships between committed transactions, commutative transactions may be prepared and committed in different orders across different servers. They are logically concurrent and appear as siblings in the TxDAG's structure.

For each transaction, a majority agrees upon and persists the same happens-before relationships. Minorities detect divergence and catch up with the majority. In this RIOT guarantees an isomorphic TxDAG across all servers, and an *equivalent* (though not necessarily equal) execution order for transactions.

Theorem – Compatibility implies identical histories

If two servers have mutually compatible leading edges, then the induced subgraphs on the down-closures of those edges are identical. The mapping between them is just the identity on transaction identifiers. Thus servers not only agree on which transactions are present, but also on how those transactions depend on each other.

Intuitively, each server's committed history can be seen as the shape of its TxDAG. Mutual compatibility guarantees the shapes contain the same transactions, and determinism guarantees the common connections. Therefore the shapes must be identical in each server. Should compatibility fail, owing to failures or lag, the system aborts rather than attempting to extend shapes into inconsistent forms.

Quorum intersection ensures that the declared ancestors for each transaction is the same across all servers such that if T3 depends on T1 and T2, then all servers will record that fact. If T2 never declares a dependency on T1, then no server will record T1 → T2. Though servers see transactions in different physical orders, their TxDAGs converge to the equivalent partial orders.

3.3 Protocol Details

In RIOT, progress is tracked using the `TransactionStatus` enum. In addition to familiar states such as `ABORTED`, `COMMITTED`, `PREPARED`, and `UNKNOWN` (i.e., not yet seen), RIOT includes `INCOMPATIBLE` as a more explicit kind of `ABORTED` that signals incompatible leading edges that would otherwise lead to divergence, `HEURISTIC` for transactions whose outcome does not respect the majority of votes, and `ABORTED_AFTER_REPLAY` to prevent transactions that failed during recovery being committed again (which could alter their commit order).

An `Update` sent from a client to initiate coordination contains a unique client-generated transaction ID and a payload (such as a Cypher query). A `Transaction` combines the submitted update, its current status, and identifiers its parent transactions, given by the leading edge of the coordinator when coordination begins. Client-visible outcomes are represented by the `Outcome` type which contains the original transaction ID and the final status of the consensus (i.e., committed or aborted), as well as any values produced by resource managers during execution (i.e., query results).

We shall now describe the common *two-phase* execution before describing how *single-phase* consensus works in RIOT.

3.3.1 Two-Phase Consensus.

Prepare Phase. The two-phase path begins with the prepare phase, shown in Alg. 1 (lines 1-15) and Alg. 2.

(1) The Coordinator binds the transaction to its current leading edge (`LE`), providing its ancestors(line 3).
(2) The Coordinator sends prepare messages to all RMs, including its local one. In our prototype, remote RMs are contacted via RPC while the local RM is invoked directly on a separate thread. The leading edge is sent along with the transaction data (line 5).
(3) Each Participant first checks whether the transaction has already been seen. If it already exists in the `TxDAG`, its current status and ancestors are immediately returned (Alg. 2, lines 4-5), skipping the rest of the prepare logic. If the transaction is new, the Participant enters the prepare phase.
(4) Each Participant then verifies that all declared ancestors are either `PREPARED` or `COMMITTED` locally (Alg. 2, lines 7–10).

Algorithm 1: Coordinator logic using two-phase consensus

```
Local Variables: localRM ⟵ initialize local RM
                 remoteRMs ⟵ initialize connections
1  submitTransaction(workload):
2  begin
3      tx ⟵ { workload, UNKNOWN, localRM.LE, 0, null }
4      // Prepare phase: send prepare to all RMs
5      prepares ⟵ prepare(tx)
6      // Abort if no majority or incompatible LE
7      if !checkValidLeadingEdge (prepares) or
        !hasMajority (prepares) then
8          tx.status ⟵ ABORTED
9          return abort(tx.tid)
10     tx.status ⟵ PREPARED
11     // Commit phase: send commit to all RMs
12     commits ⟵ commit(tx)
13     if !hasMajority (commits) then
14         tx.status ⟵ HEURISTIC
15     return (tx.status, tx.tid)
16 status(tid):
17 begin
18     return (localRM.status(tid), tid)
19 prepare(tx):
20 begin
21     prepares ⟵ {}
22     foreach rm ∈ localRM ∪ remoteRMs do
23         prepares ⟵ prepares ∪ rm.prepare(tx.tid)
24     return prepares
25 commit(tx):
26 begin
27     commits ⟵ {}
28     foreach rm ∈ localRM ∪ remoteRMs do
29         commits ⟵ commits ∪ rm.commit(tx.tid)
30     return commits
31 abort(tid):
32 begin
33     foreach rm ∈ remoteRMs do
34         rm.abort(tid)
35     return (tid, ABORTED)
```

(5) Any PREPARED ancestors are committed without further coordination, since they have already been committed by a majority elsewhere (Alg. 2, lines 12-16). For each successfully committed ancestor, a counter is incremented to prevent in-use entries being garbage collected. Counters are later decremented when dependent transactions are committed and entries in the TxDAG outside the leading edge with a zero-count can be safely garbage collected.

(6) The RM then attempts to prepare the transaction. If successful, the transaction is added to the TxDAG with status PREPARED and a qualifier computed and applied to the TxDAG,]. The transaction handle for the database is then stored to allow a subsequent commit (or abort) later(Alg. 2, lines 20–22). Otherwise, the transaction is marked as ABORTED.

(7) Each RM responds with the result of its prepare and its qualifier. The Coordinator collects all responses and checks:
   (a) **Quorum:** Whether there is a majority of successful responses that also contain compatible qualifiers (i.e., $\lceil \frac{N}{2} \rceil + 1$), if so the Coordinator proceeds to the commit phase (line 12).
   (b) **Divergence:** If the coordinator fails to obtain a commit quorum, the transaction is aborted. The checkValidLeadingEdge(), which checks whether the qualifiers' transactions exist in the coordinator's TxDAG and will logically convert PREPARED messages into ABORT messages if required transactions are not present.

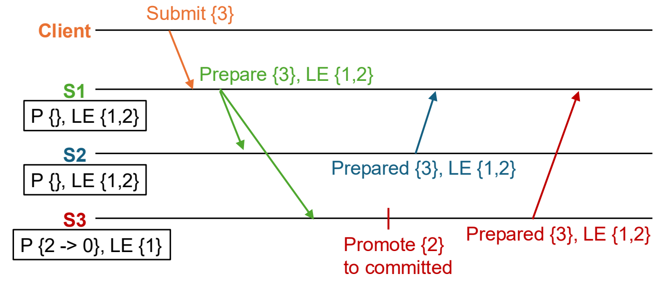


Figure 2: Prepare phase with compatible histories

Fig. 2 illustrates the prepare phase with three servers (S1, S2, and S3), using natural numbers to denote TIDs for clarity. Color coding highlights the participants and their actions: the client in orange, and the three servers in green, blue, and red, respectively. At this point, S1 and S2 share the same state: neither holds transactions in the PREPARED state and both maintain $LE = \{1, 2\}$. In contrast, S3 has not yet committed transaction 2, likely due to a delayed or lost commit message.

When a client submits a new transaction to S1, the transaction inherits S1's *LE* as its ancestry, and PREPARE messages are issued to S2 and S3. Server S2 responds immediately since its history matches that of S1. Server S3, however, must first promote transaction 2 to COMMITTED (see lines 12–16 in Alg. 2). This promotion is safe because inclusion in another server's *LE* means a majority has committed the entry. Once S3 reconciles its state, it becomes consistent with S1 and S2, allowing it to process the new transaction and respond affirmatively.

Commit Phase. The Coordinator initiates the commit phase by sending commit messages containing the transaction ID and the ancestors of the transaction from its local TxDAG to all RMs (not just those who voted to commit). This is shown in Alg. 1, lines 26-30, and Alg. 3, lines 26-40.

(1) Each RM first checks whether the transaction is in the PREPARED state. If it is not, the RM returns its current status (line 30). If a transaction is marked as COMMITTED by a remote server but has not been observed locally, the server realizes that it has become stale or diverged and must catch up.
(2) If the transaction is valid (i.e., PREPARED), the RM proceeds with the commit.

(3) If the transaction does not have compatible ancestry with the metadata received on the *COMMIT* message, the server has become stale or diverged and must catch up before proceeding (line 31).
(4) On success, the transaction and its dependencies are recorded durably. The transaction's ancestors are removed from the leading edge, the newly committed transaction is added, and the reference counters of its ancestors are decremented (lines 36-37) for later TXDAG garbage collection purposes.
(5) If the commit fails, the RM aborts it (lines 47-53).
(6) The Coordinator collects all commit responses. If a majority have voted to commit, the transaction is finalized. If not, the transaction is marked HEURISTIC, signaling that the protocol has been violated and the system may require user reconciliation.

Algorithm 2: Resource Manager – Prepare

```
Local Variables: localDB ← initialize database
                 LE ← {} // Leading Edge
                 txDAG ← {} // Transaction history
                 WAL ← {} // Persisted to disk
1  prepare(tx):
2  begin
3      if txDAG.contains(tx.tid) then
4          txHolder ← txDAG.get(tx.tid)
5          return (txHolder.status, txHolder.ancestors)
6      // Ensure all ancestors are safe to proceed
7      foreach ancestor ∈ tx.ancestors do
8          ancestorStatus ←
            txDAG.get(ancestor.tid).status
9          if ancestorStatus ∉ {PREPARED, COMMITTED}
            then
10             return (INCOMPATIBLE, LE)
11     // Attempt to commit any PREPARED ancestors
12     foreach ancestor ∈ tx.ancestors do
13         txHolder ← txDAG.get(ancestor.tid)
14         txHolder.status ← commit(txHolder.tid)
15         if txHolder.status == COMMITTED then
16             txHolder.counter ← txHolder.counter+1
17         else
18             return (abort(txHolder.tid),
                txHolder.ancestors)
19     // Attempt to prepare the transaction
20     txHandler ← localDB.beginTx()
21     if txHandler.prepare(tx.workload) then
22         tx.localTxHandler ← txHandler
23         tx.status ← PREPARED
24         txDAG ← txDAG ∪ {tx}
25     else
26         txHandler.abort()
27         tx.status ← ABORTED
28     return (tx.status, LE)
```

Algorithm 3: Resource Manager – Commit and Abort

```
26 commit(tid, coord_leading_edge):
27 begin
28     txHolder ← txDAG.get(tid)
29     if txHolder.status ≠ PREPARED then
30         return txHolder.status
31     if txHolder.ancestors ⊈ coord_leading_edge
        then
32         catchup(txDAG.leadingEdge())
33     txHandler ← txHolder.localTxHandler
34     if txHandler.commit() then
35         txHolder.status ← COMMITTED
36         WAL ← WAL ∪ {txHolder.tx}
37         cleanupLeadingEdge(txHolder)
38         foreach ancestor ∈ txHolder.ancestors do
39             txDAG.get(ancestor.tid).counter ←
                txDAG.get(ancestor.tid).counter - 1
40     else
41         txHolder.status ← abort(txHolder.tid)
42     return txHolder.status
43 cleanupLeadingEdge(tx):
44 begin
45     foreach ancestor ∈ tx.ancestors do
46         LE ← LE \{ancestor}
47     LE ← LE ∪ {tx}
48 abort(tid):
49 begin
50     txHolder ← txDAG.get(tid)
51     if txHolder.status ≠ PREPARED then
52         return txHolder.status
53     txHolder.localTxHandler.abort()
54     txHolder.status ← ABORTED
55     return txHolder.status
```

3.3.2 Single-Phase Consensus. Like other leaderless protocols, RIOT can reach decisions with a single round-trip when failures, conflicts, and divergence are limited. To enable this, the coordinator (line 7 of Alg. 1) checks for a fast quorum of size $\lceil \frac{3N}{4} \rceil + 1$. If such a quorum is obtained, the coordinator may reply to the client immediately. Otherwise, the protocol defaults to two-phase coordination described earlier.

Unlike other protocols, RIOT does not reorder conflicting transactions in the two-phase path. This choice simplifies both execution and recovery but modestly increases the likelihood of aborts. To mitigate this, RIOT requires a larger supermajority of $\lceil \frac{3N}{4} \rceil + 1$ for fast-path decisions, which is stricter than the quorum in protocols like Fast Paxos [24] where reordering is allowed during recovery.

3.4 Aborts and Retries

In RIOT, ABORT and INCOMPATIBLE responses are fundamental components of the protocol rather than outright error states. These statuses serve to signal dissent among servers and prevent the system from entering a state of permanent divergence. The INCOMPATIBLE

status occurs when a coordinator or participant is lagging, and attempt to use stale history to initiate or participate in consensus. Detecting and acting on incompatibility is cheap, requiring only a comparison between leading edges.

Conversely, a resource manager may trigger a local abort even after a transaction's leading edge is verified as compatible by RIOT. These outcomes typically arise from underlying database constraints, such as deadlocks or transient resource conflicts, which may not manifest uniformly across all participating servers. This multi-level approach is safe because the final transaction outcome is still strictly decided by a majority.

Although local aborts are acceptable within the design, high abort rates can ultimately degrade throughput. In practice, their frequency is shaped by workload characteristics and the structure of the stored graph which may spread or concentrate contention. Pathological workloads that concentrate updates on a small set of highly contended elements are possible, but atypical. Real-world graph workloads tend to distribute access broadly, and even transactions involving high-degree nodes often touch distinct relationships or attributes, limiting direct contention. The primary exceptions are operations such as node deletions, which have inherently wider scope. Empirical evidence from Neo4j's commercial cloud service, AuraDB [16], shows commit-to-abort ratios of approximately 700:1 across the entire fleet, demonstrating that contention-induced aborts are rare in production workloads.

RIOT does not automatically retry aborted transactions as a way of reducing spurious aborts. Instead, when a client receives an `INCOMPATIBLE` or `HEURISTIC` outcome, it can choose to resubmit the transaction or not, depending on its context.

3.5 Catch Up

RIOT does not reorder transactions, and so servers may temporarily diverge due to significant out of order message arrival or unavailability. Divergence is detected via mismatched leading edges, arising when (i) a coordinator receives a remote edge containing unseen commits, or (ii) a participant receives a commit for an unprepared transaction or one with incompatible ancestry.

A divergent server initiates catch-up by sending its current leading edge to peers within the shard. If all the transactions in the divergent server's leading edge are locally resolvable on the recipient server, it responds with a sub-graph of its own `TxDAG` containing the supplied leading edge, any concurrent transactions and all descendants of either set, up to and including the its own current leading edge (the leaves of its `TxDAG`).

The divergent server then applies all transactions in the returned sub-graph, in breadth-first order from root to leaves. If the divergence is too large, or some transactions cannot be locally resolved on any recipient (e.g. because of garbage collection in the txDAG), then the divergent server must copy a full snapshot to catch up.

3.6 Recovery

To handle coordinator failures, all servers keep an immutable copy of their `PREPARED` responses (including leading edge metadata), until the associated transactions are completed.

Once a server S_1 suspects a coordinator S_2 of having failed, it sends a `RECOVER_PREPARED` message to every other server in the shard, containing its original `PREPARED` response, along with its associated leading edge metadata. The server cannot make a unilateral decision, it must wait until enough servers have responded that a majority outcome can be safely determined.

Other servers that receive `RECOVER_PREPARED` for a transaction which they have prepared (that they are not coordinating) must reply to the sender with the original `PREPARED` response sent to the coordinator. For example, S_3 would send S_1 a copy of the exact `PREPARED` response that it originally sent to S_2, including S_3's leading edge at the time. If the transaction is unknown by S_3, then it responds `ABORTED` and remembers that fact permanently. S_3 cannot subsequently prepare that transaction. Conversely, if the transaction has already been processed by S_3, then it simply replies with its existing decision: `ABORTED` or `COMMITTED`.

Once servers begin exchanging `RECOVER_PREPARED` messages, three outcomes are possible:

(1) Servers straightforwardly learn about a pre-existing majority `COMMITTED` or `ABORTED` status for the transaction.
(2) Servers gather a majority of compatible `PREPARED` responses. The outcome of the transaction is inferred and its state updated to `COMMITTED`.
(3) Servers collect enough **incompatible** `PREPARED` responses such that no compatible majority can form without the failed coordinator. The outcome of the transaction is inferred and its state updated to `ABORTED`.

All servers act independently in the recovery phase, and no new coordinator is established. However, the original coordinator might itself recover during or after recovery of a transaction and retransmit `PREPARE` messages. In response it will receive `PREPARED`, `ABORTED`, or `COMMITTED` messages from the other servers it contacts. It may even receive `RECOVER_PREPARED` messages. A recovered coordinator processes these messages in good faith, committing or aborting as appropriate. If the recovered coordinator is particularly lagging, it will observe incompatible transaction states and execute the catch up protocol from Sec. 3.5.

3.7 Multi-Shard Transactions

RIOT uses a two-layer approach for multi-shard transaction whose aim is to minimize network traffic between shards. When a client submits a multi-shard transaction, the receiving server acts as the *primary coordinator* which then enlists *interposed coordinators* on remote shards to act as participants. The original coordinator executes consensus on its local shard, propagating the remote shards' details to local participants for recovery purposes. Meanwhile each interposed coordinator executes consensus on its local shard. The original coordinator then ensures that every shard reaches a commit decision in order for the transaction to proceed. We term this a *conjunction of majorities*: a transaction succeeds only if all shards vote to commit, but requiring only a local majority within each shard.

This design enables RIOT to scale by allowing any server to act as a coordinator while minimizing wide-area communication. Interposed coordinators serve as shard-local proxies, eliminating the need for the primary to coordinate directly with every server affected by the transaction while preserving safety guarantees. We now provide a detailed explanation of the steps involved:

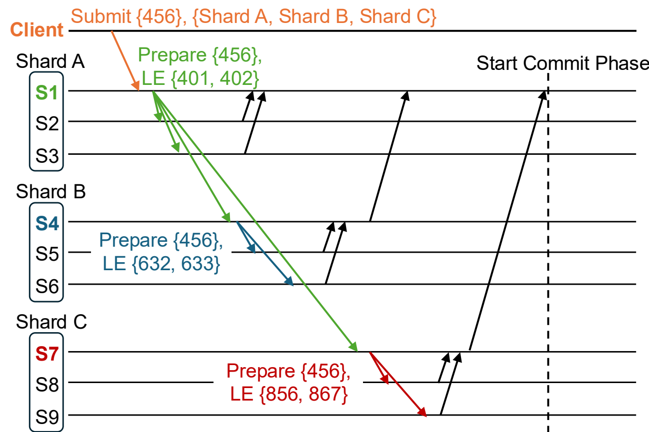


Figure 3: Prepare phase with interposed coordinators

(1) **Prepare Phase:**
  (a) The primary coordinator sends prepare requests to all shard-local RMs and one to each interposed coordinator for participating remote shards.
  (b) Each interposed coordinator executes the prepare phase within its shard and returns the result to the primary coordinator. They can use single-phase or two-phase execution to achieve this.
(2) **Consensus Check:**
  (a) The primary coordinator waits for responses to its prepare message from all interposed coordinators.
  (b) It also verifies that a majority of its own shard-local RMs have responded positively (with either supermajority or simple majority).
(3) **Commit or Abort:**
  (a) If all shards vote to prepare, the primary coordinator proceeds with the commit phase.
  (b) During commit, the primary coordinator sends commit requests to all interposed coordinators and its shard-local RMs.
  (c) Interposed coordinators within each shard may then execute the second phase of consensus, if they have only achieved a simple majority.

Fig. 3 shows the prepare phase of a multi-shard transaction involving three shards. Messages are color-coded for clarity: orange represents the client, green indicates the primary coordinator (server S1), and blue and red denote the interposed coordinators in other shards (S4 and S7). Server S1 is the primary coordinator and initiates the prepare phase by sending messages within shard A. At the same time, it designates interposed coordinators in shards B (server S4) and C (server S7) to manage the transaction locally within their respective shards, which return their responses to S1. Once all responses are received, the primary coordinator can proceed to the commit phase.

Recovery for remote shards extends the single-shard case. A server in a remote shard that has stalled and suspects its interposed coordinator has failed is permitted to send RECOVER_PREPARED messages not only to its local peers but also to members of the originating shard to request the global outcome. Other shard-local servers can then learn about the global decision via local RECOVER_PREPARED messages.

3.8 Correctness

RIOT ensures serializability, safety, and correctness through a combination of dependency-aware coordination, quorum-based decision-making, and well-defined local execution semantics. Each server maintains a local *TxDAG* that records the ancestry of all committed and prepared transactions, which is enriched during successive rounds of coordination. A transaction can only be prepared if all of its ancestors are known to be committed, and committed transactions form a consistent prefix of the global execution history in the *leading edge* of the TxDAG. This design ensures that all servers converge on compatible histories, even in the presence of failures or message delays. Transactions are accepted only if a majority agrees on their compatibility in aggregate. Multi-shard commits require unanimous agreement across all involved shards, to preserve atomicity. Accordingly, RIOT establishes a globally consistent order that is equivalent to some serial execution. Other work [15] has formally established the safety and correctness of RIOT using TLA+.

By exchanging leading edges of the TxDAG during the prepare *and* commit phases, RIOT prevents lagging servers or diverged servers from participating in transactions where they do not have the appropriate history. It also defends against subtle corner cases where messages have been pathologically delayed while the system is at its fault-tolerant limits. When a server detects that it is lagging or diverged (typically by disagreeing with transaction outcomes), it runs a catch-up protocol which synchronizes its local TxDAG with others in the system so that it can properly participate in processing future transactions.

3.9 Cluster Membership

To address changes in cluster membership caused by scaling, maintenance, or failure, RIOT supports dynamic cluster reconfiguration through consensus similar to Raft, but modified for our leaderless design. During reconfiguration, both the old and new configurations are recognized, and a topology change commits only if it achieves majority agreement in both. This overlap preserves safety by ensuring every committed transaction is visible to a majority in each configuration, eliminating the risk of split-brain.

Membership changes are encoded as TxDAG entries and handled like normal transactions. Servers leaving the cluster continue to participate until the joint configuration commits, while new servers first join as non-voting learners, synchronize to a consistent snapshot, and are then promoted to full members. This design reflects the FLP impossibility [11]: progress cannot be guaranteed under faults, but when quorums exist, RIOT reconfigures safely and consistently, without the stalls associated with leader re-election.

4 RIOT Architecture

In this section, we describe our implementation of RIOT in Neo4j, which extends the database to support multiple writers and sharding. More generally, we demonstrate how a transactional graph database can be built atop our generalized consensus protocol. We

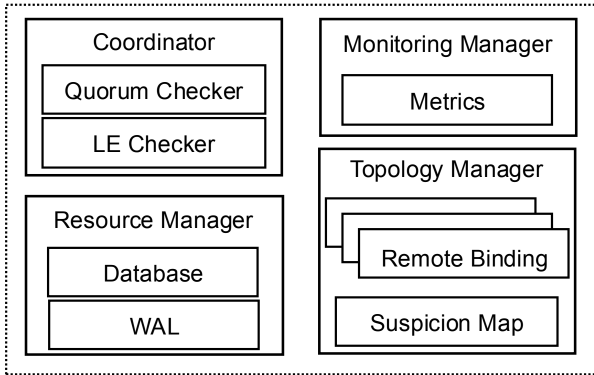


Figure 4: RIOT server architecture

first outline the design constraints of RIOT in Sec. 4.1, and then present the architecture of our prototype in Fig. 4.

4.1 Composition with existing systems

Neo4j is a mature graph database, and making major architectural changes to it would be impractical. Our implementation of RIOT extends Neo4j rather than seeking to change it, and so we avoid protocols from Related Work that would require extensive modifications to database internals. Instead, we follow the design of the existing Neo4j Raft implementation, and RIOT is integrated via the same Neo4j commit hooks. Correctness depends on the database (acting as an RM in RIOT) aborting any conflicting transactions, which standard mechanisms such as pessimistic locking or MVCC with write intents provide. This lets RIOT handle consensus and ordering while concurrency control remains a function of the database. However, we did make a small modification to Neo4j's *Forseti* lock manager [32], so that it aborts immediately on any conflict. This gives us fine-grained control over generating contention during experiments. This does not affect systems with the unmodified Neo4j implementation.

All inter-server communication is implemented via gRPC [14], optimized with load balancing, compression, and aggressive keepalive settings to minimize runtime overhead. These choices lead to a practical design that is well-suited to our experimental needs.

4.2 Running RIOT

To initiate a transaction, a client sends a *BEGIN* message to any available RIOT server. The message includes a unique transaction identifier (TID) and a Cypher query to execute. Because our prototype does not incorporate a full query planner, the client must also specify the target shards for the transaction—a limitation of the prototype rather than of RIOT itself.

When a RIOT server receives a *BEGIN* message from a client, it assumes the role of *Coordinator* for that transaction – both within the origin shard, and across shards if required. It determines the set of participating servers, represented as *Remote Bindings*, using the query metadata and its local *Topology Manager* data.

Once the coordinator has established membership, it initiates the prepare phase. Using its *Quorum* and *Leading Edge* checkers, the coordinator determines whether there are enough servers with a consistent history to execute the fast path, fall back to the slow path, or abort. It also detects whether other participants have diverged and might require catch-up. During the prepare and commit phases, each server's local *Resource Manager* records transaction metadata (including status and dependencies for crash recovery) and attempts to apply changes to the database. Recall that resource managers are the ultimate arbiters: even if the RIOT layer indicates that an ordering is safe, the prepare and commit decisions ultimately reside with the underlying database.

For practical deployments, a *Topology Manager* maintains cluster membership and a *Suspicion Map* records any incompatible leading edges or missed replies. Such metadata is piggybacked on regular protocol messages and is used to bias the choice of interposed coordinators and trigger catch-up for lagging servers. For experimentation we used a lightweight *Monitoring Manager* to gather performance metrics such as latency, throughput, leading-edge size, and commit/abort counts. Inter-server communication is implemented over gRPC with standard optimizations (load balancing, compression, keepalives) to minimize overhead.

5 Evaluation

We now evaluate the performance of RIOT using Neo4j's Raft implementation as a baseline (Sec. 5.2). Then, we analyze the impact of failures on the performance of both systems, showing how throughput and availability are affected differently (Sec. 5.3).

5.1 Experimental Setup

All experiments ran on a cluster of `m5.2xlarge` AWS EC2 instances (8 vCPUs, 32 GiB RAM), using Amazon Linux 2023 (kernel v6.1). We run Neo4j Enterprise Edition v5.26 using its recommended Corretto OpenJDK17. We run our experimental version of Neo4j integrated with RIOT using OpenJDK 23 in order to make use of Java's virtual threads[35].

Database Systems and Workloads. To ensure a fair comparison, we disable the fast path in RIOT— though it improves performance up to 40% — to be more consistent with Raft's propose-notify model. In all experiments, the dataset fits entirely in memory to eliminate disk IO as a performance bottleneck, though data is still persisted to disk for durability.

We use a synthetic write load on both systems using following pattern: `MERGE (person:Person id: $id)`. An index is created on the `id` property to enable efficient lookups, and we generate queries that repeatedly access the same node (identified by `id`) to introduce contention between concurrent transactions. Many clients then issue transactions over the network from separate machines, each waiting for an outcome response before sending the next transaction. This mirrors the usage of the system from a typical multi-user application (e.g. mobile app).

Metrics. The primary performance metrics in our benchmarks are throughput, measured in transactions per second (tx/s), and end-to-end tail latency, measured in milliseconds (ms) at the 99th percentile (p99). In the plots, candlesticks represent the 5th, 25th, 50th, 75th, and 99th latency percentiles.

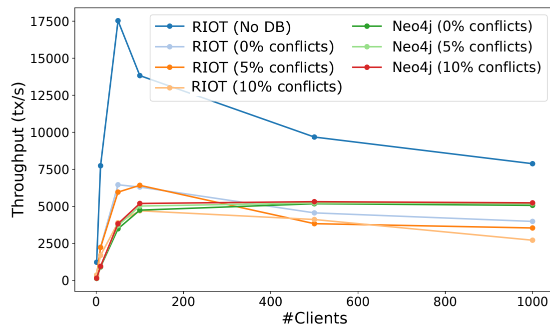


Figure 5: Throughput with a 3-node cluster

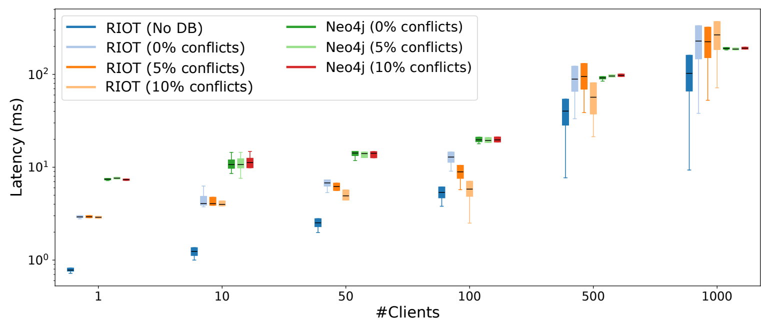


Figure 6: Latency with a 3-node cluster

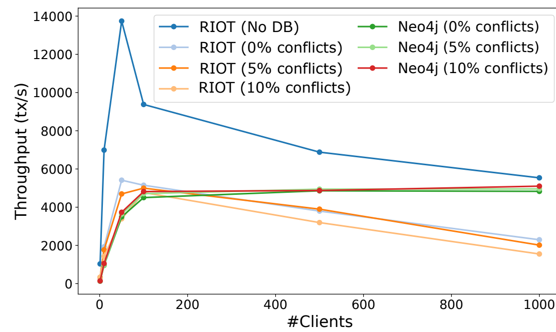


Figure 7: Throughput with a 5-node cluster

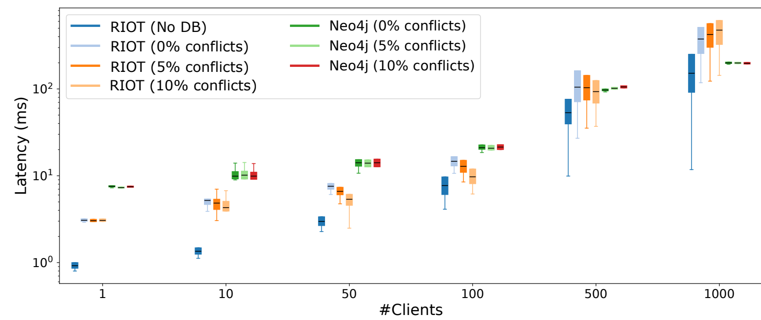


Figure 8: Latency with a 5-node cluster

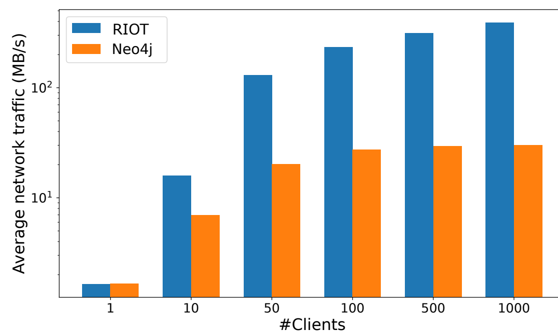


Figure 9: Average network traffic per machine

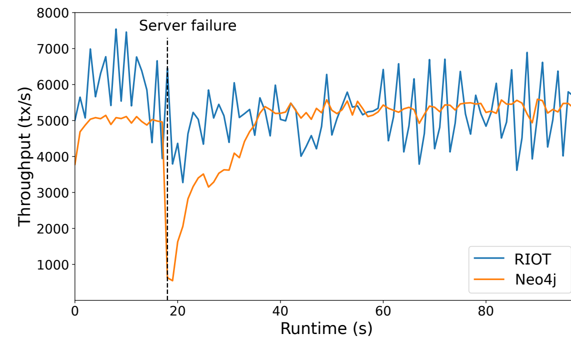


Figure 10: Throughput when a server fails

5.2 Comparison with Raft

In this experiment, we compare the performance of RIOT integrated with Neo4j against regular Neo4j with its mature Raft implementation. Using the synthetic workload we vary the number of concurrent clients across six levels: 1, 10, 50, 100, 500, and 1000. We also inject different levels of contention by adjusting the percentage of conflicting transactions to 0%, 5%, and 10%, realistic values drawn from our experience running Neo4j's cloud service, AuraDB. We also measured results for RIOT without Neo4j as a backend – labeled *RIOT (No DB)* – to isolate protocol performance from any bottlenecks in the database such as locking.

3-node cluster. Our first experiment uses a 3-server cluster with a single shard—a commonplace configuration for enterprise Neo4j deployments. Fig. 5 and Fig. 6 present throughput and latency percentiles, respectively.

Metrics AuraDB show that the vast majority of instances have fewer than 100 client connections. In this range, RIOT outperforms Neo4j, achieving 32–250% higher throughput and up to 2.3× lower p99 latency (2.5× lower on average). However as the number of clients increases, this advantage reduces: at 500–1000 clients, RIOT throughput drops by up to 25% relative to Neo4j with Raft, and tail latency increases by as much as 2.3×, though average latency remains similar. This degradation stems from larger *leading edges* exchanged between servers under high concurrency—with active transactions from roughly 60% of clients—which inflates message size and processing cost. In our current implementation, the size of leading edges exchanged during operations grows linearly with

the number of active concurrent transactions. Larger message sizes create processing and transmission bottlenecks. Practical implementations will require mechanisms such as compressing the leading edge, or even backpressure to reduce concurrency to efficient levels which has the side-effect of shortening the leading edge.

Fig. 9 shows the average per-node network traffic as client counts rise. With up to 10 clients, when leading edges remain small, RIOT matches Raft's traffic. Unlike Raft, where the leader handles approximately 6× more traffic than followers, RIOT balances load evenly across nodes. Beyond 10 clients, traffic grows substantially, reaching up to 13× higher (MB/s) than Raft with large numbers of active clients. This overhead reduces performance by increasing latency and lowering throughput, though it is not intrinsic: leading edges can be compressed in simple workloads, and in practice, systems often use connection pooling in their middle-tiers which bounds concurrency and so keeps leading edge sizes small enough to yield good performance.

Under contention (5–10% conflict rates yielding comparable abort ratios), RIOT performance degrades proportionally to the conflict rate, yet it still outperforms Neo4j in throughput and latency up to 100 clients. Finally, in the *RIOT (No DB)* configuration, which isolates protocol performance by removing the database backend completely, RIOT scales with client count until large leading edges again create significant overheads, though latency remains lower overall as is expected in the absence of real database operations.

5-node cluster. In Fig. 7 and Fig. 8, we repeat the experiment using a 5-node cluster. The results follow a similar curve to the 3-node setup: RIOT achieves approximately 15% to 240% higher throughput and over 2× lower average and tail latency compared to Neo4j with Raft when the number of clients is at most 100. Beyond that point, performance begins to degrade, particularly in the presence of conflicts.

Intuitively, more servers should improve concurrency and throughput. However, in practice we find that adding more servers amplifies network traffic, both in terms of larger leading edges because more concurrent work can occur and because the majorities required to process a consistent outcome are also larger. These cause performance to degrade more quickly with respect to number of concurrent clients than in the 3-node cluster.

Discussion. RIOT scales efficiently up to around 100 clients, outperforming Neo4j's Raft implementation. The improvement comes at the cost of higher network utilization due to parallel writes (and transmitting larger leading edges), compared to Neo4j's leader-based approach. Both systems exhibit asymptotic limits with a single-shard configuration, however RIOT supports multi-shard deployments, which Raft does not. Accordingly RIOT offers greater scalability by partitioning the underlying graph across shards, within which a great deal of independent, parallel, work is possible.

5.3 Performance under failure conditions

To study failures and leader elections in Raft, we reused the previous workload on a 3-node cluster for RIOT and Neo4j with 100 clients and no contention. Fig. 10 presents the throughput over time for both. At 19 seconds, we simulate a failure by stopping the Raft leader in Neo4j. Empircally this happens on average between 1.47-1.77 times per day across the AuraDB fleet of many tens of thousands of Raft-based clusters. We clearly observer that write performance drops precipitously for 17 seconds as a new leader is elected and clients rebind. With RIOT we kill a randomly selected server, but the system continues processing transactions with no unavailability (though clearly with reduced fault-tolerance and processing power).

6 Related Work

RIOT benefits from a rich heritage of leaderless consensus protocols. Works such as EPaxos [28], CAESAR [2], Accord [38], and TAPIR [46] all offer fast paths (single round-trip) in the uncontended case and fall back to multi-round coordination under conflicts or failures. Their common approach is to track dependencies at the level of keys or objects, deriving serialization orders from declared read/write sets.

EPaxos and CAESAR improve performance by optimizing quorum intersections or timestamp assignment, but both depend on predeclared read/write sets. TAPIR reduces replication cost via inconsistent logs but also relies on key-level conflict detection. Accord is closest in spirit to RIOT, as it uses a DAG of dependencies to determine order, but it builds the DAG via keyset analysis. Key-level analysis is not well suited to graph databases since queries generate key sets dynamically and key sets can expand explosively in dense subgraphs

RIOT reasons at the granularity of transactions: each entry in the TxDAG can encapsulate many key accesses, which reduces the dependency-tracking overhead. Our coarser-grained approach trades off fewer opportunities for commutativity against much lower metadata cost, making it well-suited to graph workloads where fine-grained dependency tracking is impractical.

7 Conclusion

In this work, we present RIOT, a generalized consensus algorithm motivated by our work on extending Neo4j for mutliple writers and multiple shards. Each server in RIOT tracks dependencies between entries in its TxDAG and the protocol ensures that each server's TxDAG has a safe partial ordering which is compatible with the others. RIOT provides low-latency execution, even where replicas diverge or failures happen. Our evaluation shows that RIOT outperforms Raft in Neo4j by up to 2.5× in throughput and 2.3× in tail latency under low contention while remaining competitive under higher loads and conflict ratios. Although RIOT was designed to extend Neo4j, it places minimal requirements on an underlying system and is straightforward to integrate with other transactional database systems.

8 Acknowledgments

We thank Tobias Lindåker (Relational AI) for his work on the original TxDAG structure and coordination-free learning via leading edges. We would also like to thank Junhao Hu, Michael Cahill, and Alan Fekete (The University of Sydney, Australia) for their work on formal modeling of RIOT, and their perceptive insights into the protocol's subtle edge cases and n^{th} order side-effects.

References

[1] Amazon Neptune. 2024. https://aws.amazon.com/neptune/. Last access: March 27, 2026.

[2] Balaji Arun, Sebastiano Peluso, Roberto Palmieri, Giuliano Losa, and Binoy Ravindran. 2017. Speeding up consensus by chasing fast decisions. In *DSN*.

[3] Balaji Arun, Sebastiano Peluso, Roberto Palmieri, Giuliano Losa, and Binoy Ravindran. 2021. Taming the Contention in Consensus-Based Distributed Systems. *IEEE Transactions on Dependable and Secure Computing* 18, 6 (2021), 2907–2925. doi:10.1109/TDSC.2020.2970186

[4] Lars Backstrom, Dan Huttenlocher, Jon Kleinberg, and Xiangyang Lan. 2006. Group formation in large social networks: membership, growth, and evolution. In *SIGKDD*.

[5] Mahesh Balakrishnan, Dahlia Malkhi, John D. Davis, Vijayan Prabhakaran, Michael Wei, and Ted Wobber. 2013. CORFU: A distributed shared log. *ACM Trans. Comput. Syst.* 31, 4, Article 10 (Dec. 2013), 24 pages. doi:10.1145/2535930

[6] James C Corbett, Jeffrey Dean, Michael Epstein, Andrew Fikes, Christopher Frost, Jeffrey John Furman, Sanjay Ghemawat, Andrey Gubarev, Christopher Heiser, Peter Hochschild, et al. 2013. Spanner: Google's globally distributed database. *TOCS* (2013).

[7] Cong Ding, David Chu, Evan Zhao, Xiang Li, Lorenzo Alvisi, and Robbert Van Renesse. 2020. Scalog: Seamless reconfiguration and total order in a scalable shared log. In *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*. 325–338.

[8] Paul Ezhilchelvan, Isi Mitrani, and Jim Webber. 2020. Modeling the Gradual Degradation of Eventually-Consistent Distributed Graph Databases. *Queueing Models and Service Management* (2020).

[9] Hua Fan, Hao Tan, Wenchao Zhou, and Feifei Li. 2025. FLEET: High-Performance Durable Replicated State Machines using Scattered and Coordinated Log Entries. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*. Proceedings of the VLDB Endowment, 1522–1535. https://www.vldb.org/pvldb/vol18/p1522-fan.pdf

[10] Hugo Firth and Paolo Missier. 2017. TAPER: query-aware, partition-enhancement for large, heterogenous graphs. *Distributed and Parallel Databases* (2017).

[11] Michael J. Fischer, Nancy A. Lynch, and Michael S. Paterson. 1985. Impossibility of Distributed Consensus with One Faulty Process. *Journal of the Association for Computing Machinery (JACM)* 32, 2 (1985), 374–382. doi:10.1145/3149.214121

[12] International Organization for Standardization (ISO). 2024. Information Technology - Database Lanuages - GQL, ISO/IEC 39075:2024. https://www.iso.org/standard/76120.html Accessed on April 3, 2025.

[13] Lars George. 2011. HBase: The Definitive Guide. In *O'Reilly Media*.

[14] gRPC Authors. 2023. gRPC - GitHub Repository. https://github.com/grpc/grpc. Accessed: 2024-04-03.

[15] Junhao Hu. 2025. TLA+ Proof for RIOT Protocol. https://github.com/hyhjh211/TwoPhaseProtocol. Accessed: 2025-08-14.

[16] Neo4j Inc. 2024. Neo4j AuraDB: Fully Managed Graph Database as a Service. https://neo4j.com/cloud/aura/. Accessed: 2024-04-05.

[17] Neo4j Inc. 2025. Neo4j Cypher Manual. https://neo4j.com/docs/cypher-manual/current/introduction/. Accessed: 2025-03-31.

[18] JanusGraph Project. 2023. JanusGraph: Distributed graph database. https://janusgraph.org. Accessed: 2025-08-27.

[19] Tian Jiang, Xiangdong Huang, Shaoxu Song, Chen Wang, Jianmin Wang, Ruibo Li, and Jincheng Sun. 2023. Non-blocking raft for high throughput IoT data. In *2023 IEEE 39th International Conference on Data Engineering (ICDE)*. IEEE, 1140–1152.

[20] Avinash Lakshman and Prashant Malik. 2010. Cassandra: a decentralized structured storage system. *SIGOPS* (2010).

[21] Leslie Lamport. 1998. The part-time parliament. *ACM Transactions on Computer Systems* 16, 2 (May 1998), 133–169. doi:10.1145/279227.279229

[22] Leslie Lamport. 2001. Paxos Made Simple. *SIGACT News* (2001).

[23] Leslie Lamport. 2005. Generalized consensus and Paxos. (2005).

[24] Leslie Lamport. 2006. Fast Paxos. *Distributed Computing* 19, 2 (2006), 79–103. doi:10.1007/s00446-006-0005-x

[25] Leslie Lamport, Robert Shostak, and Marshall Pease. 1982. The Byzantine Generals Problem. *TOPLAS* (1982).

[26] Patrick Lewis, Ethan Perez, Aleksandra Piktus, Fabio Petroni, Vladimir Karpukhin, Naman Goyal, Heinrich Küttler, Mike Lewis, Wen-tau Yih, Tim Rocktäschel, et al. 2020. Retrieval-augmented generation for knowledge-intensive nlp tasks. *NeurIPS* (2020).

[27] MemGraph. 2024. https://memgraph.com/. Last access: March 27, 2026.

[28] Iulian Moraru, David G. Andersen, and Michael Kaminsky. 2013. There is more consensus in Egalitarian parliaments. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles* (Farminton, Pennsylvania) *(SOSP '13)*. Association for Computing Machinery, New York, NY, USA, 358–372. doi:10.1145/2517349.2517350

[29] Shuai Mu, Lamont Nelson, Wyatt Lloyd, and Jinyang Li. 2016. Consolidating Concurrency Control and Consensus for Commits under Conflicts. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*. USENIX Association, Savannah, GA, 517–532. https://www.usenix.org/conference/osdi16/technical-sessions/presentation/mu

[30] Neo4j. 2010. Neo4j Graph Database. https://neo4j.com/. Last access: March 27, 2026.

[31] Neo4j. 2021. Neo4j Breaks Scale Barrier with Trillion+ Relationship Graph. https://neo4j.com/press-releases/neo4j-scales-trillion-plus-relationship-graph/. Last access: March 27, 2026.

[32] Neo4j. 2024. Forseti Lock Manager. https://github.com/neo4j/neo4j/blob/5.16/community/lock/src/main/java/org/neo4j/kernel/impl/locking/forseti/ForsetiLockManager.java. Last access: March 27, 2026.

[33] Brian M. Oki and Barbara H. Liskov. 1988. Viewstamped Replication: A New Primary Copy Method to Support Highly-Available Distributed Systems. In *Proceedings of the Seventh Annual ACM Symposium on Principles of Distributed Computing* (Toronto, Ontario, Canada) *(PODC '88)*. Association for Computing Machinery, New York, NY, USA, 8–17. doi:10.1145/62546.62549

[34] Diago Ongarro and John Ousterhout. 2014. In Search of an Understandable Consensus Algorithm. In *Proceedings of USENIX Annual Technical Conference*.

[35] Ron Pressler and Alan Bateman. 2023. *JEP 436: Virtual Threads (Second Preview)*. Technical Report JEP 436. OpenJDK. https://openjdk.org/jeps/436.

[36] Ian Robinson, Jim Webber, and Emil Eifrem. 2015. *Graph databases: new opportunities for connected data*.

[37] Fedor Ryabinin, Alexey Gotsman, and Pierre Sutra. 2024. SwiftPaxos: Fast Geo-Replicated State Machines. In *21st USENIX Symposium on Networked Systems Design and Implementation (NSDI 24)*. USENIX Association, Santa Clara, CA, 345–369. https://www.usenix.org/conference/nsdi24/presentation/ryabinin

[38] Benedict Elliott Smith, Tony Zhang, Blake Eggleston, and Scott Andreas. 2021. *CEP-15: Fast General Purpose Transactions*. Technical Report. Apache Software Foundation.

[39] Florian Suri-Payer, Matthew Burke, Zheng Wang, Yunhao Zhang, Lorenzo Alvisi, and Natacha Crooks. 2021. Basil: Breaking up BFT with ACID (transactions). In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles* (Virtual Event, Germany) *(SOSP '21)*. Association for Computing Machinery, New York, NY, USA, 1–17. doi:10.1145/3477132.3483552

[40] TigerGraph. 2024. https://www.tigergraph.com/. Last access: March 27, 2026.

[41] Bing Tong, Yan Zhou, Chen Zhang, Jianheng Tang, Jing Tang, Leihong Yang, Qiye Li, Manwu Lin, Zhongxin Bao, Jia Li, et al. 2024. Galaxybase: A High Performance Native Distributed Graph Database for HTAP. *Proc. VLDB Endow.* (2024).

[42] Johan Ugander and Lars Backstrom. 2013. Balanced label propagation for partitioning massive graphs. In *WSDM*.

[43] Alexandre Verbitski, Anurag Gupta, Debanjan Saha, Murali Brahmadesam, Kamal Gupta, Raman Mittal, Sailesh Krishnamurthy, Sandor Maurice, Tengiz Kharatishvili, and Xiaofeng Bao. 2017. Amazon aurora: Design considerations for high throughput cloud-native relational databases. In *SIGMOD*.

[44] Alexandre Verbitski, Anurag Gupta, Debanjan Saha, Murali Brahmadesam, Kamal Gupta, Raman Mittal, Sailesh Krishnamurthy, Sandor Maurice, Tengiz Kharatishvili, and Xiaofeng Bao. 2018. Amazon Aurora: On Avoiding Distributed Consensus for I/Os, Commits, and Membership Changes. In *Proceedings of the 2018 International Conference on Management of Data (SIGMOD '18)*. 1041–1052. doi:10.1145/3183713.3196937

[45] Huanhuan Wu, James Cheng, Silu Huang, Yiping Ke, Yi Lu, and Yanyan Xu. 2014. Path problems in temporal graphs. *Proc. VLDB Endow.* (2014).

[46] Irene Zhang, Naveen Kr Sharma, Adriana Szekeres, Arvind Krishnamurthy, and Dan RK Ports. 2018. Building consistent transactions with inconsistent replication. *TOCS* (2018).

Received 17 Nov 2025